# Computer Architecture

## Lecture 14: Prefetching II

Rahul Bera
Prof. Onur Mutlu

ETH Zürich
Fall 2023
Nov 10 2023

# Recall: 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)

# Recall: 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
- Prefetcher throttling
- Issues in multi-core, multiprocessor, multithreaded systems
- …

# Recall: 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
- Prefetcher throttling
- Issues in multi-core, multiprocessor, multithreaded systems
- …

# Agenda for Today

- Prefetch-aware system design
  - Memory controller
- Advanced prefetching techniques
  - RL-based prefetching
  - Hiding latency via off-chip prediction
  - Execution-based prefetching

# Prefetch-Aware DRAM Controller

# Prefetch-Aware DRAM Controllers

Chang Joo Lee
Onur Mutlu*
Veynu Narasiman
Yale N. Patt

**Electrical and Computer Engineering**
**The University of Texas at Austin**

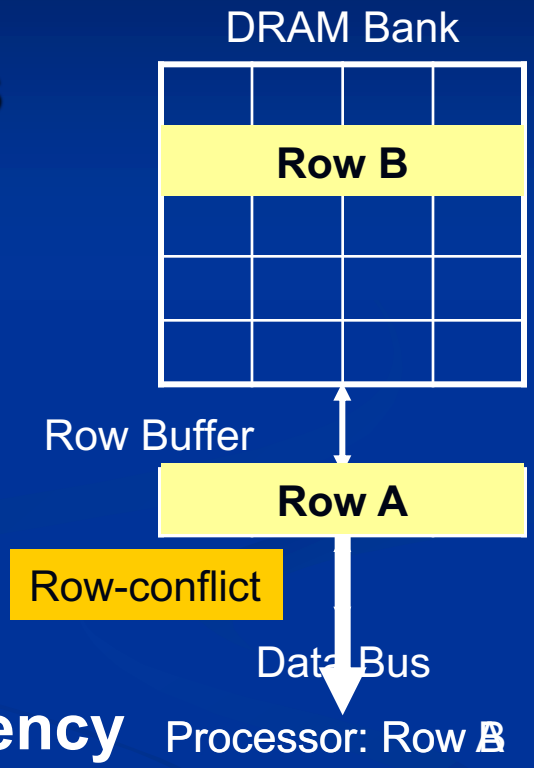
***Microsoft Research and Carnegie Mellon University**

UT ECE

Microsoft Research

# Modern DRAM Systems

- Rows and columns of DRAM cells
- A *row buffer* in each bank
- Non-uniform access latency:
  - Row-hit:
    - Data is in the row buffer
  - Row-conflict:
    - Data is not in the row buffer
    - Needs to access the DRAM cells
  - **Row-hit latency < Row-conflict latency**




**Prioritize row-hit accesses to increase DRAM throughput [Rixner et al. ISCA2000]**

# 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**

# Key Mechanism

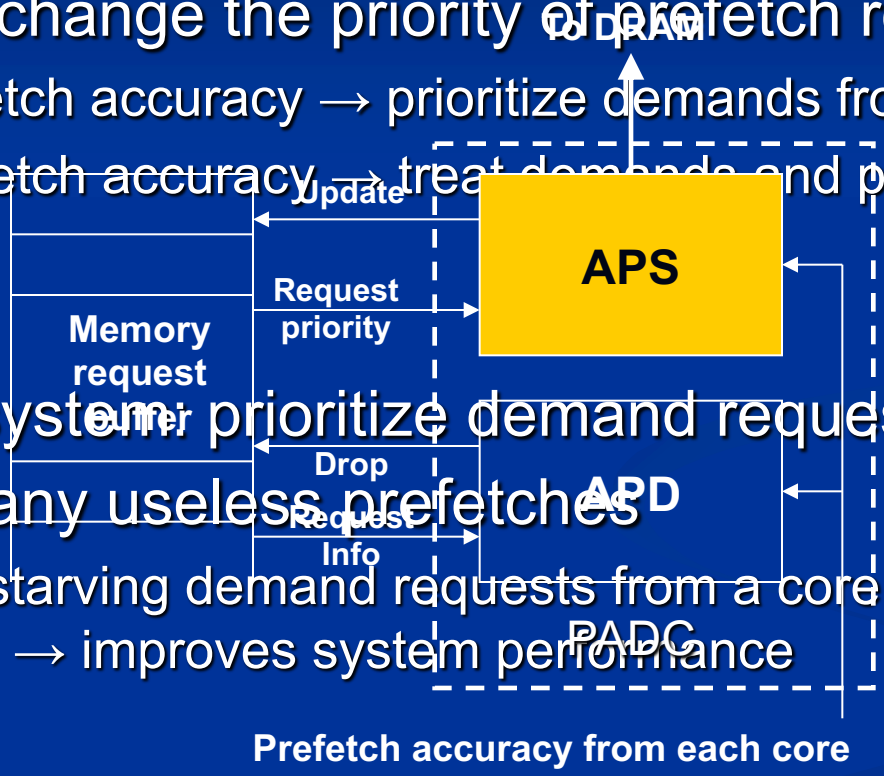
- Prefetch-Aware DRAM Controllers (PADC)
  - Adaptive Prefetch Scheduling
    - Increase DRAM throughput by exploiting row-buffer locality when prefetches are useful
    - Delay service of prefetches when they are useless
  - Adaptive Prefetch Dropping
    - With APS, remove useless prefetches effectively while keeping the benefits of useful prefetches
  - Improve performance and bandwidth efficiency for both single-core and CMP systems
  - Low cost and easily implementable

# Prefetch Accuracy Estimation

- Prefetch accuracy = $\frac{\#\text{Prefetches used}}{\#\text{Prefetches sent}}$
- Hardware support:
  - Prefetch bit (per L2 cache line, MSHR entry): Indicates whether it is a prefetch or demand
  - Prefetch sent counter (per core)
  - Prefetch used counter (per core)
  - Prefetch accuracy register (per core)
    - Estimated every 100K cycles

# Adaptive Prefetch Scheduling (APS)

1. Adaptively change the priority of prefetch requests
   - Low prefetch accuracy → prioritize demands from the core
   - High prefetch accuracy → treat demands and prefetches equally
2. In a CMP system, prioritize demand requests from a core that has many useless prefetches
   - To avoid starving demand requests from a core with low prefetch accuracy → improves system performance



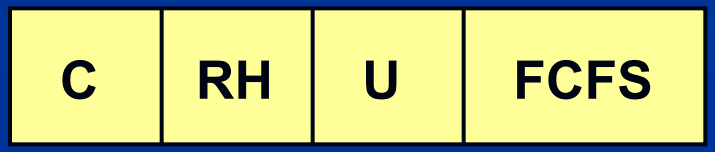
# Adaptive Prefetch Scheduling (APS)

1. Critical requests
   - All demand requests
   - Prefetch requests from cores whose prefetch accuracy ≥ promotion threshold

2. Urgent requests
   - Demand requests from cores whose prefetch accuracy < promotion threshold

# Adaptive Prefetch Scheduling (APS)

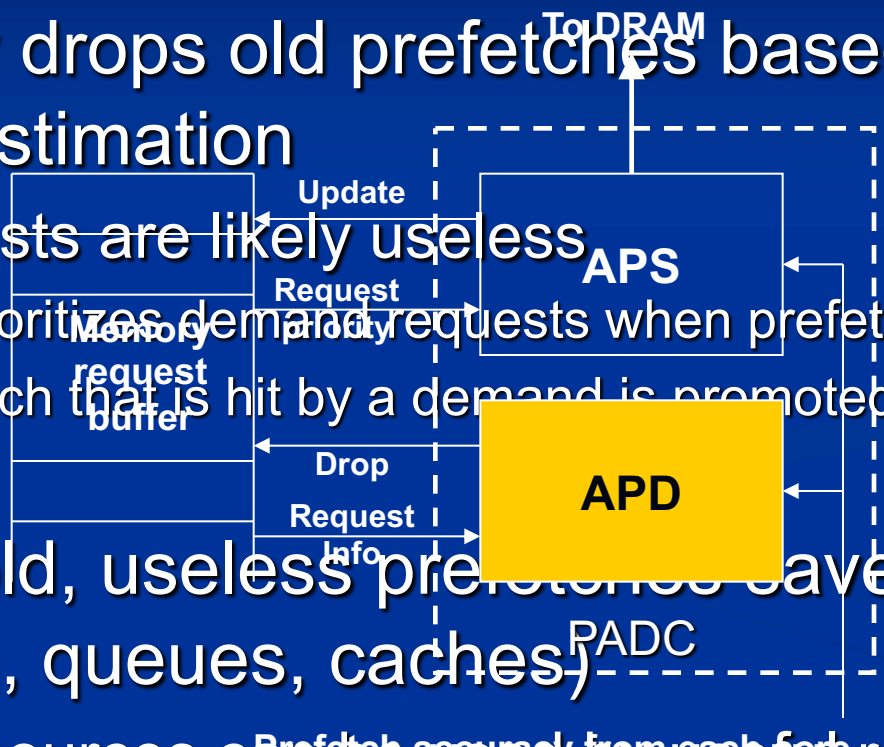
- Each memory request buffer entry: priority fields




- Prioritization order:
  1. Critical request (C)
  2. Row-hit request (RH)
  3. Urgent request (U)
  4. Oldest request (FCFS)

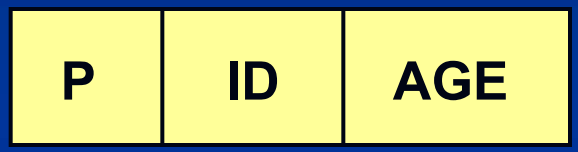
# Adaptive Prefetch Dropping (APD)

- Proactively drops old prefetches based on prefetch accuracy estimation
  - Old requests are likely useless
    - APS prioritizes demand requests when prefetch accuracy is low
    - A prefetch that is hit by a demand is promoted to a demand
- Dropping old, useless prefetches saves resources (bandwidth, queues, caches)
  - Saved resources can be used by useful requests



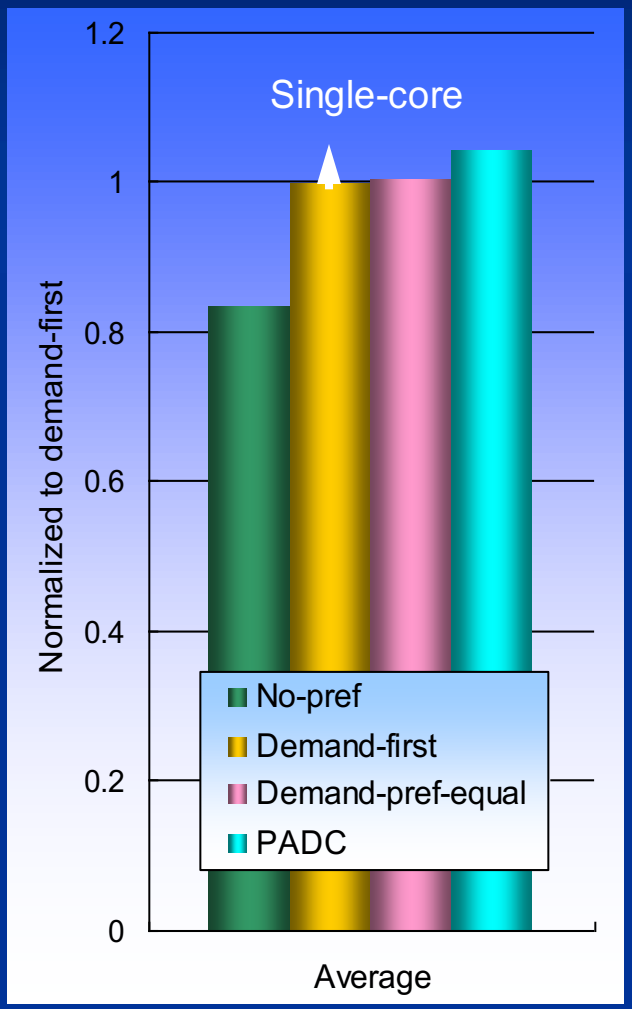
# Adaptive Prefetch Dropping (APD)

- Each memory request buffer entry: drop information

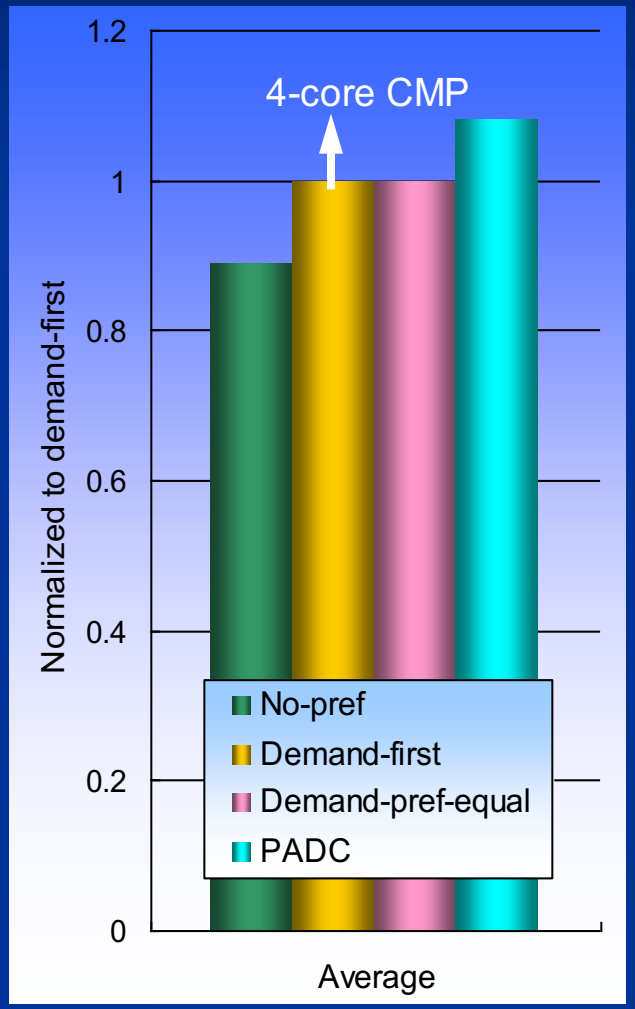



  - Prefetch bit (P)
  - Core ID field (ID)
  - Age field (AGE)
- Drop prefetch requests whose AGE > Drop threshold
- Drop threshold is dynamically determined based on prefetch accuracy estimation
  - Lower accuracy → Lower threshold

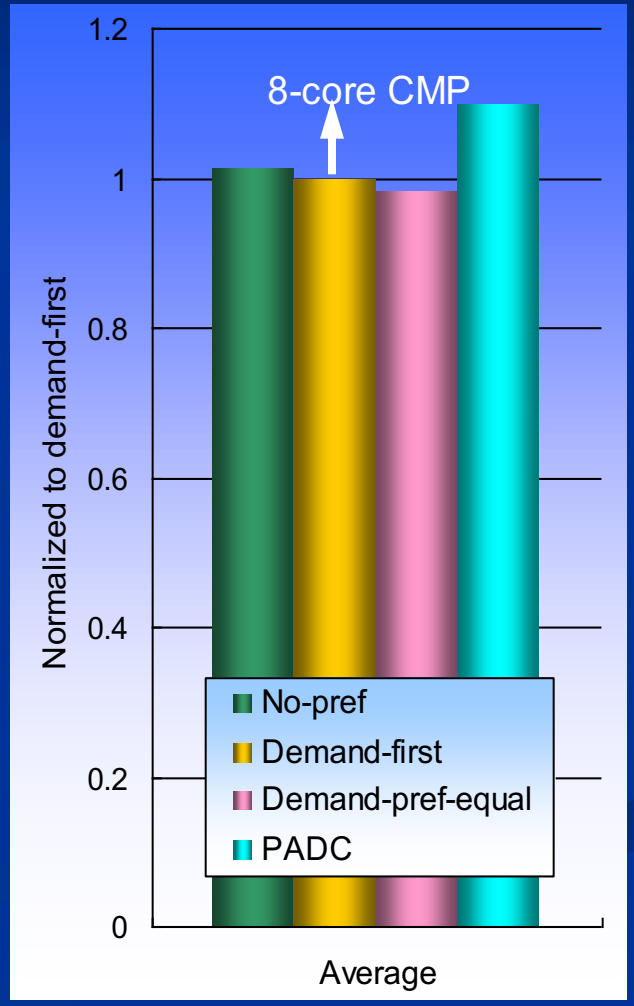
# Performance of PADC




4.3%




8.2%




9.9%

# Prefetch-Aware DRAM Controllers

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)
An extended version as HPS Technical Report, TR-HPS-2008-002, University of Texas at Austin, September 2008.

**Prefetch-Aware DRAM Controllers**

Chang Joo Lee† Onur Mutlu§ Veynu Narasiman† Yale N. Patt†

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

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

# Advanced Prefetching: Self-Optimizing Prefetcher

# 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**


# Pythia

## A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori,
Taha Shahroodi, Sreenivas Subramoney, Onur Mutlu

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

SAFARI Research Group
safari.ethz.ch

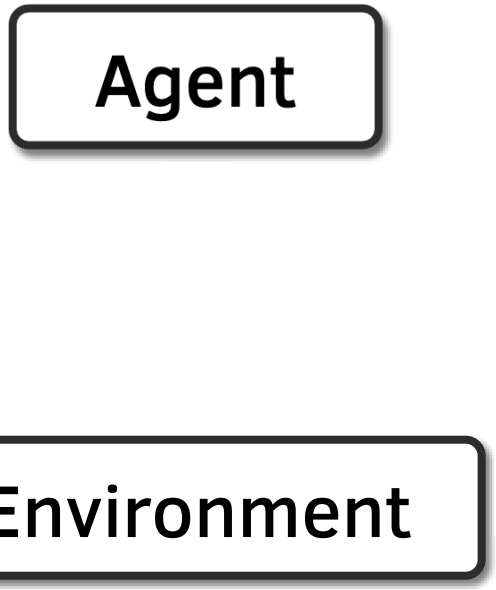
ETH zürich

intel

TU Delft

# 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

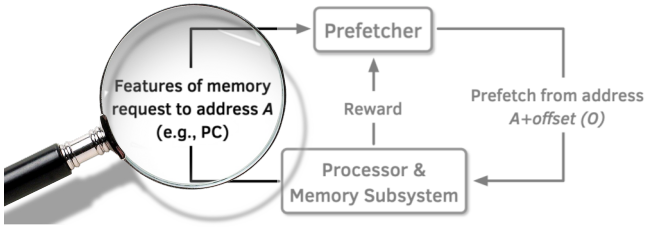
Pythia formulates prefetching as a **reinforcement learning** problem

# 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

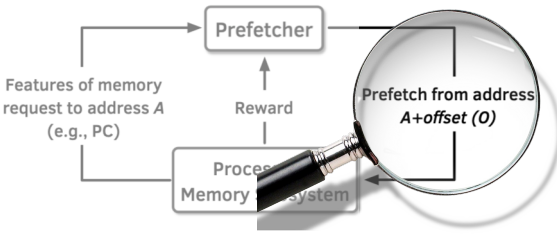



- **Control-flow examples**
  - PC
  - Branch PC
  - Last-3 PCs, …
- **Data-flow examples**
  - Cacheline address
  - Physical page number
  - Delta between two cacheline addresses
  - Last 4 deltas, …

# What is Action?

Given a demand access to address A

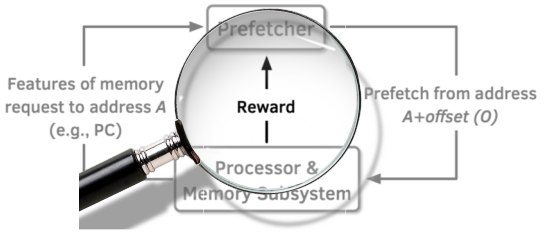
the action is to **select prefetch offset "O"**




- **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

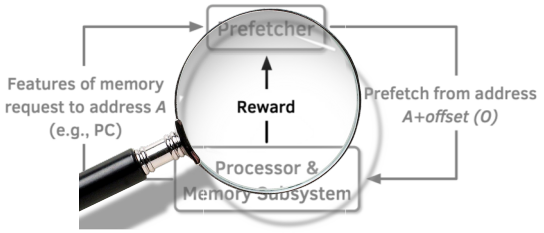



- Encapsulates two metrics:
  - **Prefetch usefulness** (e.g., accurate, late, out-of-page, …)
  - **System-level feedback** (e.g., mem. b/w usage, cache pollution, energy, …)

- We demonstrate Pythia with **memory bandwidth usage** as the system-level feedback

# What is Reward?

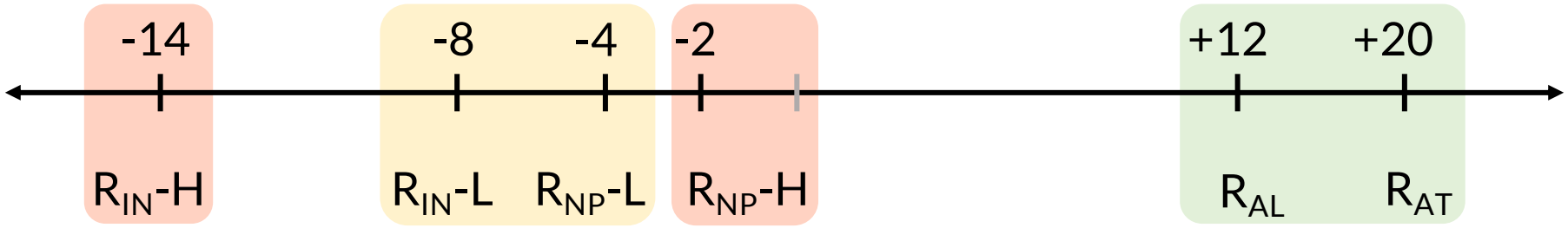
- **Five** distinct reward levels
  - *Accurate and timely* ($R_{AT}$)
  - *Accurate but late* ($R_{AL}$)
  - *Loss of coverage* ($R_{CL}$)
  - *Inaccurate*
    - With low memory b/w usage ($R_{IN}$-L)
    - With high memory b/w usage ($R_{IN}$-H)
  - *No-prefetch*
    - With low memory b/w usage ($R_{NP}$-L)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

# Steering Pythia's Objective via Reward Values

- Example reward configuration for
  - Generating accurate prefetches
  - Taking bandwidth-aware prefetch decisions

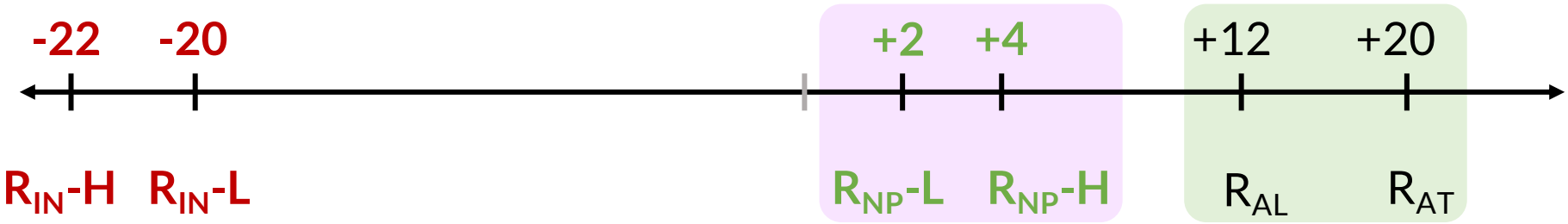



**AT** = *Accurate & timely*; **AL** = *Accurate & late*; **NP** = *No-prefetching*; **IN** = *Inaccurate*;
**H** = *High mem. b/w*; **L** = *Low mem. b/w*

1. **Highly prefers to generate accurate prefetches**
2. **Prefers not to prefetch if memory bandwidth usage is low**
3. **Strongly prefers not to prefetch if memory bandwidth usage is high**

# Steering Pythia's Objective via Reward Values

- Customizing reward values to make Pythia **conservative** towards prefetching




**AT** = *Accurate & timely*; **AL** = *Accurate & late*; **NP** = *No-prefetching*; **IN** = *Inaccurate*;
**H** = *High mem. b/w*; **L** = *Low mem. b/w*

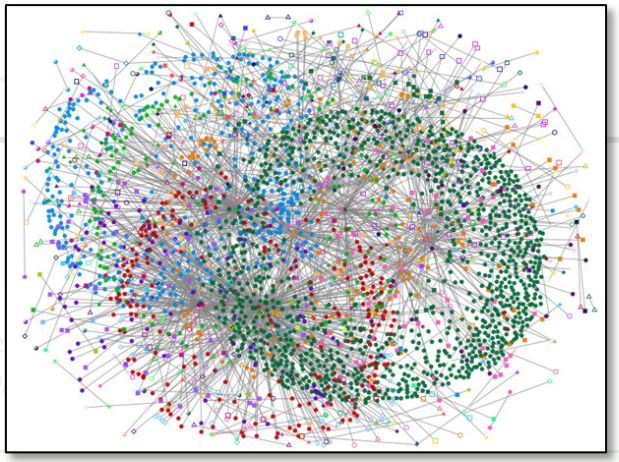
1 **Highly prefers to generate accurate prefetches**

2 **Otherwise prefers not to prefetch**

# Steering Pythia's Objective via Reward Values

- Customizing reward values to make Pythia conservative towards prefetching

Server-class processors



Bandwidth-sensitive workloads

# Steering Pythia's Objective via Reward Values

- Customizing reward values to make Pythia conservative towards prefetching

-22 -20 +2 +4 +12 +20

**Pythia's objective can be easily customized in silicon without changing the hardware**

1 Highly prefers to generate accurate prefetches

2 Otherwise prefers not to prefetch

# Basic Pythia Configuration

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

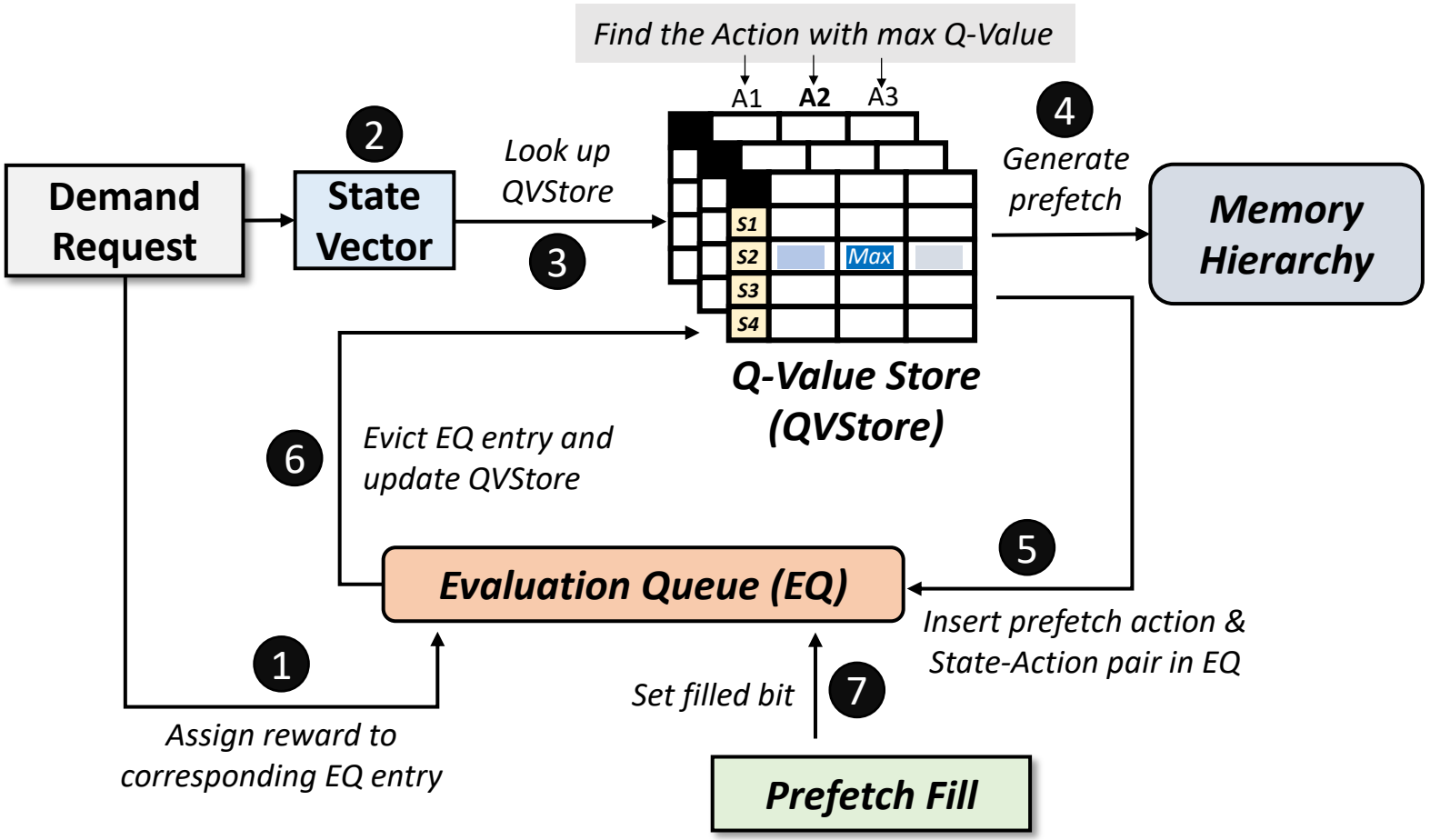
- **State:** 2 features
  - PC+Delta
  - Sequence of last-4 deltas

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

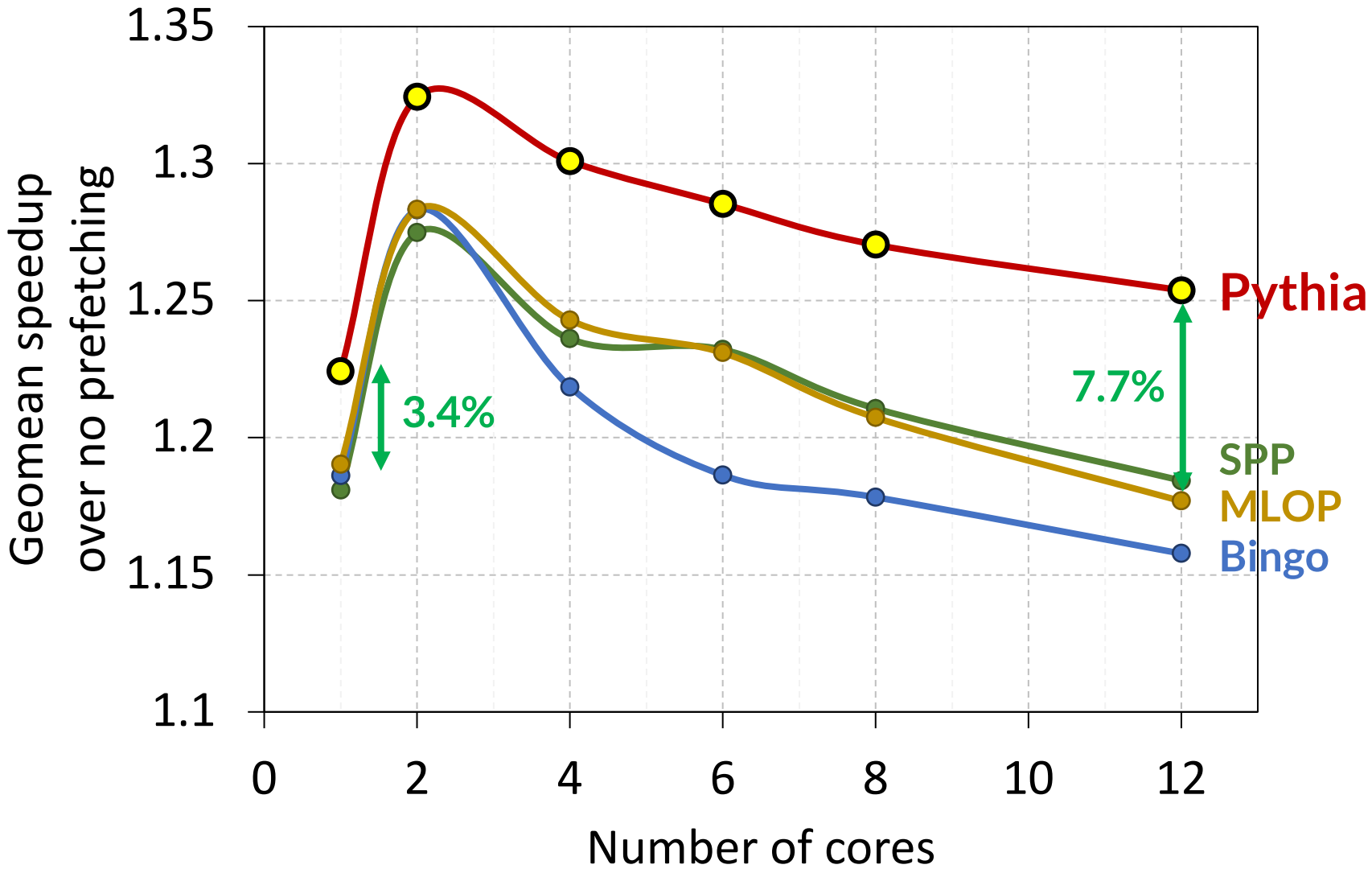
- **Rewards:**
  - $R_{AT}$ = +20; $R_{AL}$ = +12; $R_{NP}$-H=-2; $R_{NP}$-L=-4;
  - $R_{IN}$-H=-14; $R_{IN}$-L=-8; $R_{CL}$=-12

# More Detailed Pythia Overview

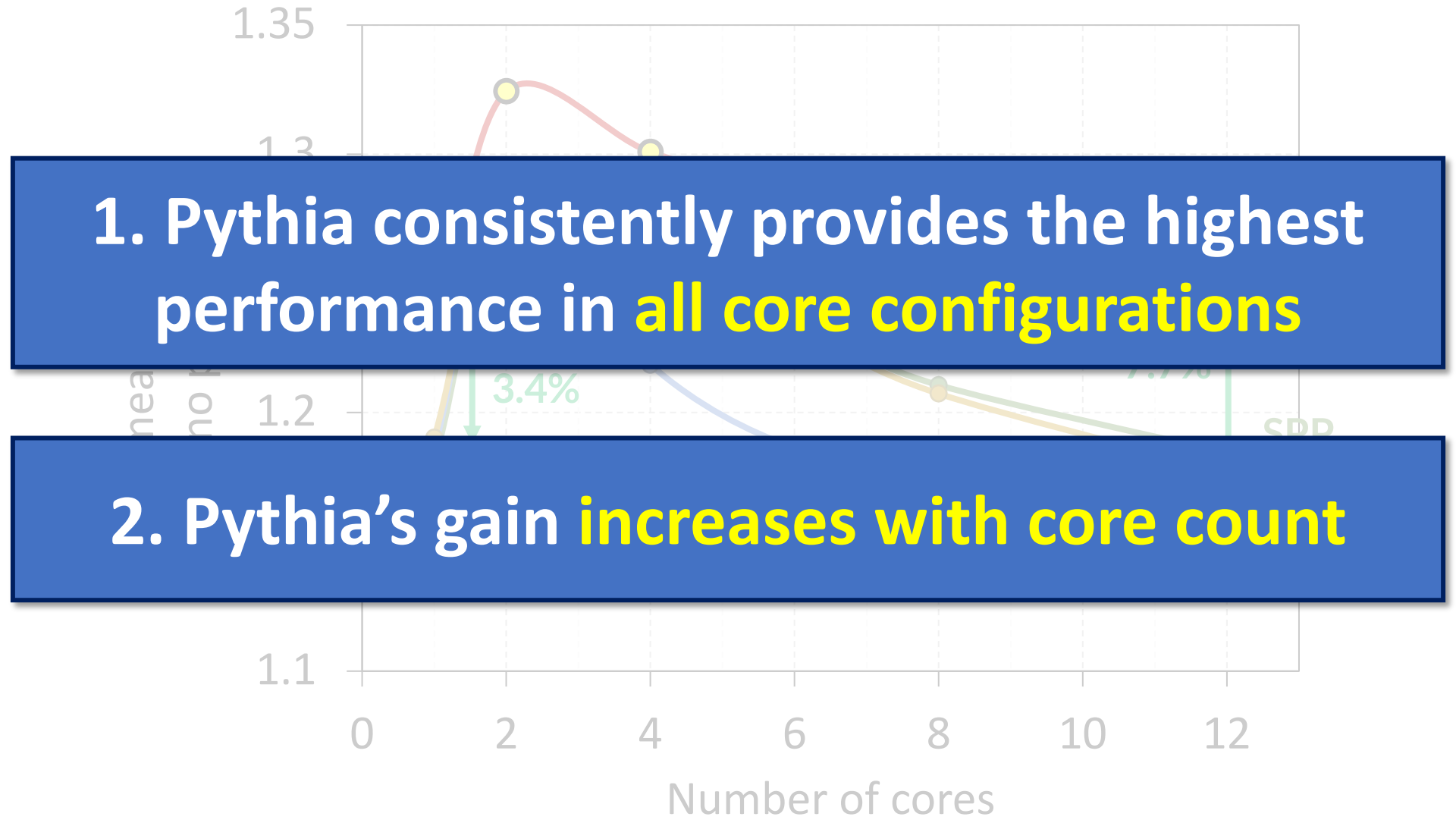
- **Q-Value Store**: Records Q-values for *all* state-action pairs
- **Evaluation Queue**: A FIFO queue of recently-taken actions



# Performance with Varying Core Count



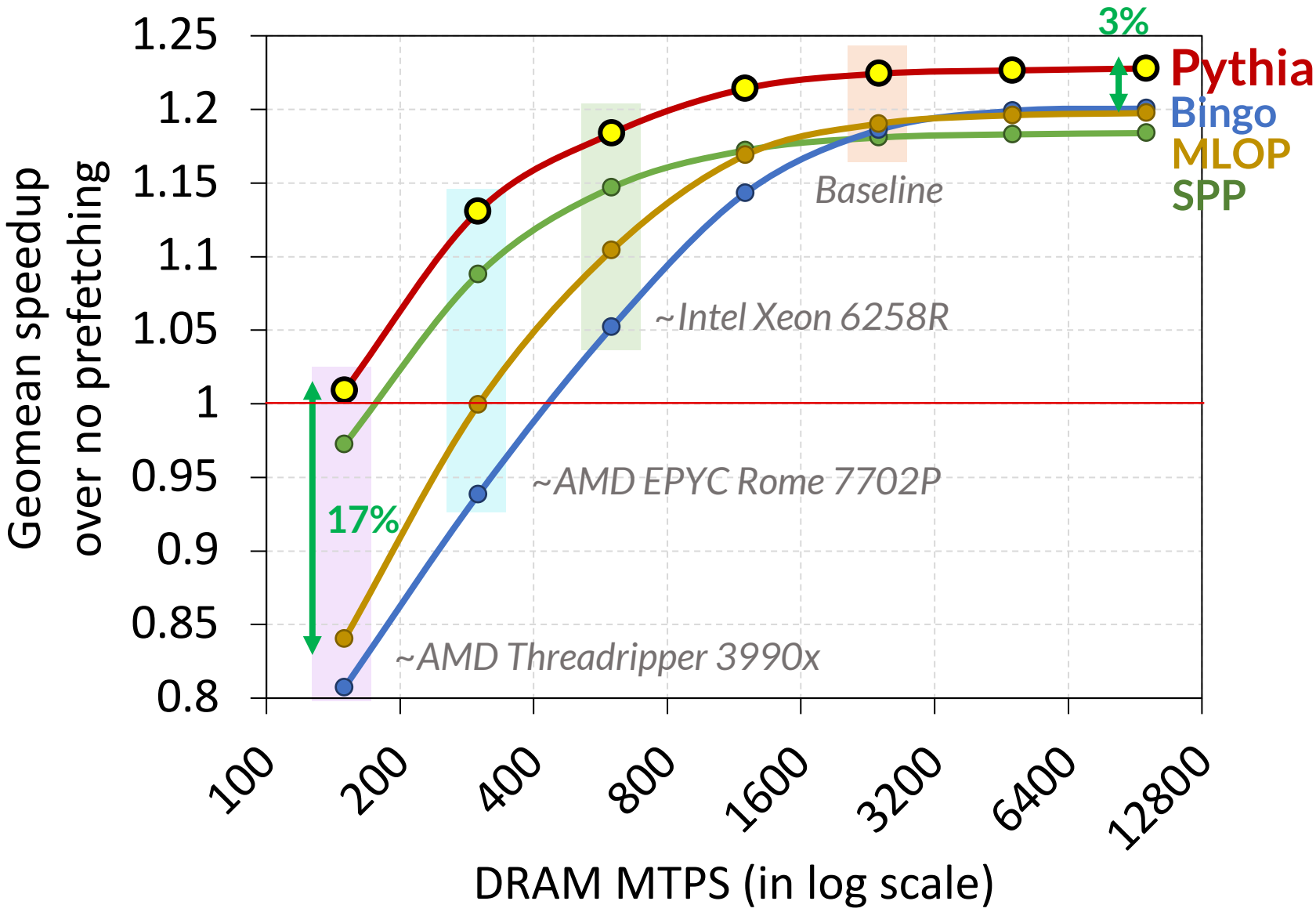
# Performance with Varying Core Count



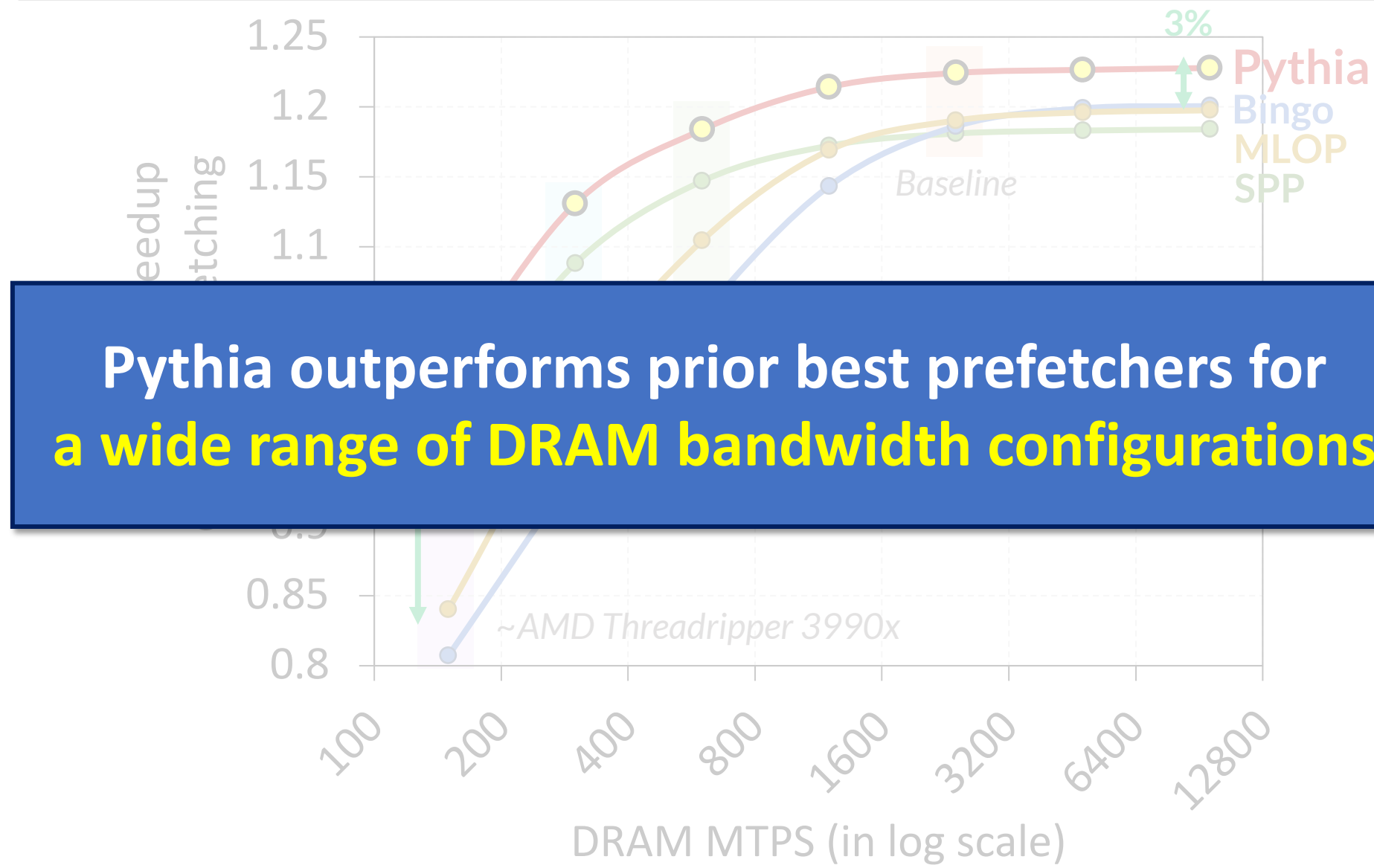

1. Pythia consistently provides the highest performance in all core configurations

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

CMU-SAFARI / Pythia Public

Unwatch 3 | Star 7 | Fork 2

Code | Issues | Pull requests | Actions | Projects | Wiki | Security | Insights | Settings

master | 1 branch | 5 tags | Go to file | Add file | Code

rahulbera Updated README f96dee9 2 days ago 38 commits

| | | |
|---|---|---|
| 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 |

**About**

A customizable hardware prefetching framework using online reinforcement learning as described in the MICRO 2021 paper by Bera and Kanellopoulos et al.

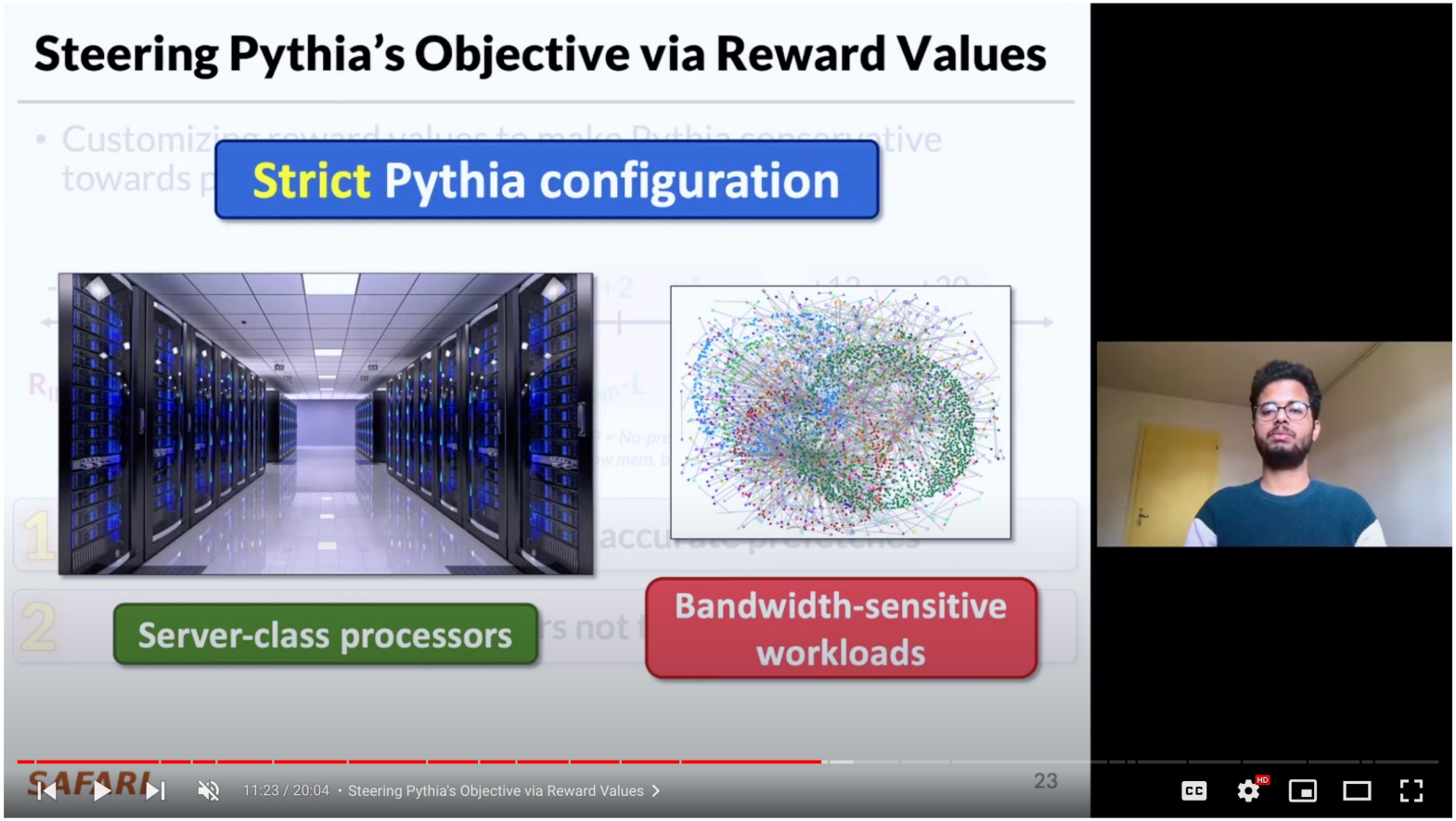
arxiv.org/pdf/2109.12021.pdf

machine-learning | reinforcement-learning | computer-architecture | prefetcher | microarchitecture | cache-replacement | branch-predictor | champsim-simulator | champsim-tracer

Readme | View license | Cite this repository

**Releases** 5

# Pythia Talk Video




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



https://www.youtube.com/watch?v=6UMFRW3VFPo&list=PL5Q2soXY2Zi--0LrXSQ9sST3N0k0bXp51&index=8

# 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[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**


# Pythia

## A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori,
Taha Shahroodi, Sreenivas Subramoney, Onur Mutlu

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

SAFARI Research Group
safari.ethz.ch

ETH zürich

intel

TU Delft

# Recommended Book on Reinforcement Learning




http://incompleteideas.net/book/RLbook2020.pdf

# Can We Do Better?

- Prefetching and caching are latency hiding techniques
- Pythia saves ~50% memory requests from going to main memory
- What about the remaining 50%? Can we do something to hide their latency too?

# Off-Chip Prediction

Rahul Bera, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadati, and Onur Mutlu,
**"Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction"**
*Proceedings of the 55th International Symposium on Microarchitecture* (***MICRO***), Chicago, IL, USA, October 2022.
[Slides (pptx) (pdf)]
[Longer Lecture Slides (pptx) (pdf)]
[Talk Video (12 minutes)]
[Lecture Video (25 minutes)]
[arXiv version]
[Source Code (Officially Artifact Evaluated with All Badges)]
***Officially artifact evaluated as available, reusable and reproducible.***
***Best paper award at MICRO 2022.***


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

Rahul Bera[1] Konstantinos Kanellopoulos[1] Shankar Balachandran[2] David Novo[3]
Ataberk Olgun[1] Mohammad Sadrosadati[1] Onur Mutlu[1]

[1]ETH Zürich [2]Intel Processor Architecture Research Lab [3]LIRMM, Univ. Montpellier, CNRS

https://arxiv.org/pdf/2209.00188.pdf


# HERMES

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

**Rahul Bera**, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadati, Onur Mutlu

https://github.com/CMU-SAFARI/Hermes




# The Key Problem

**Long-latency off-chip load requests**

Often **stall** processor by
**blocking instruction retirement** from
Reorder Buffer (ROB)

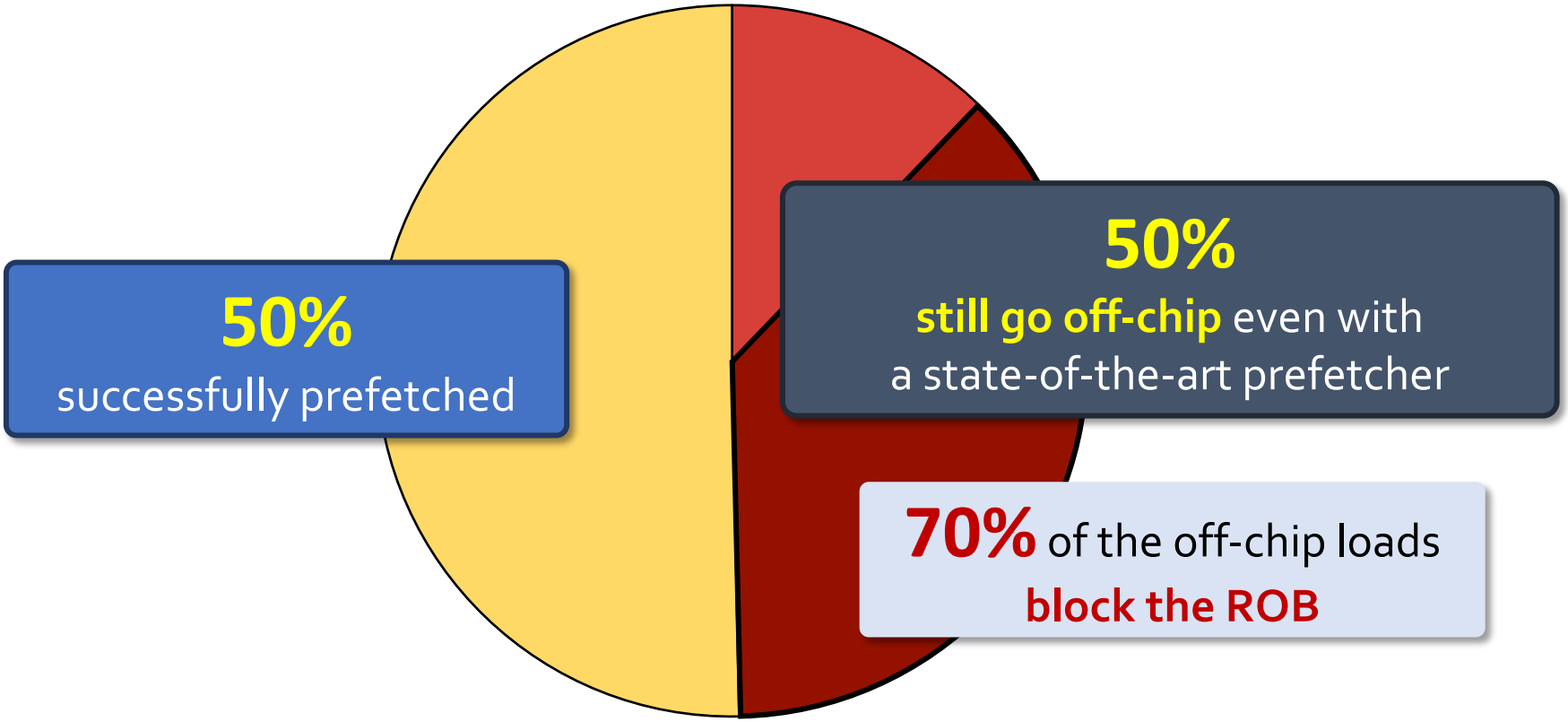
**Limit performance**

# Traditional Solutions

1. Employ sophisticated **prefetchers**
2. **Increase size** of on-chip caches

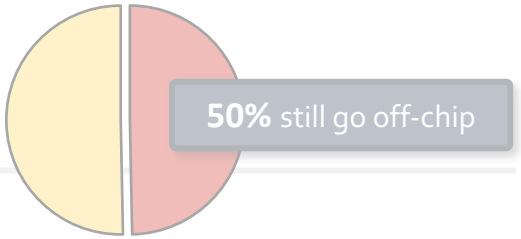
# Key Observation 1

**Many loads still go off-chip**

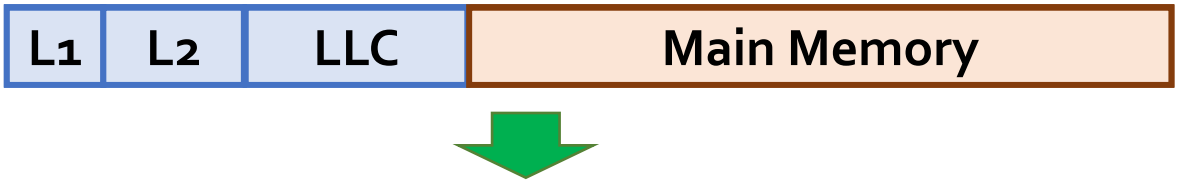



*# off-chip loads without any prefetcher*

# Key Observation 2



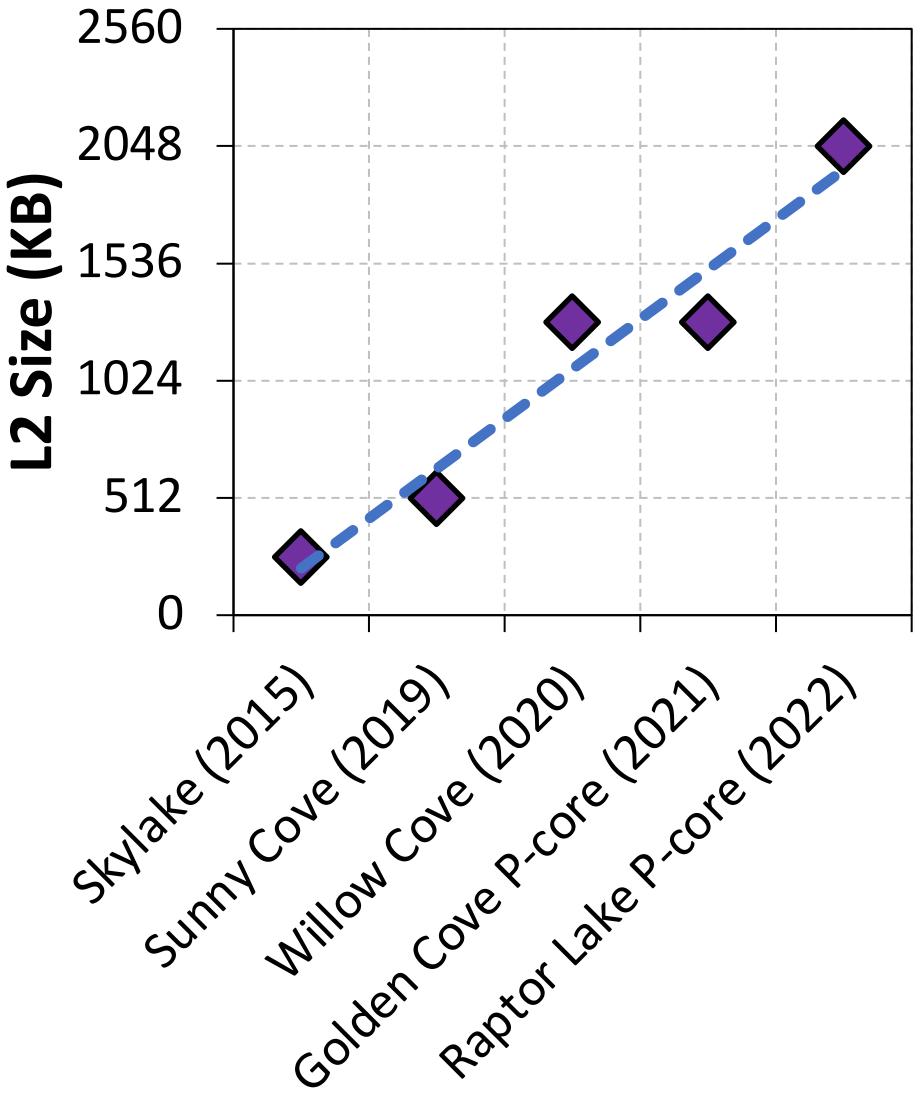

**On-chip cache access latency**
significantly contributes to off-chip load latency

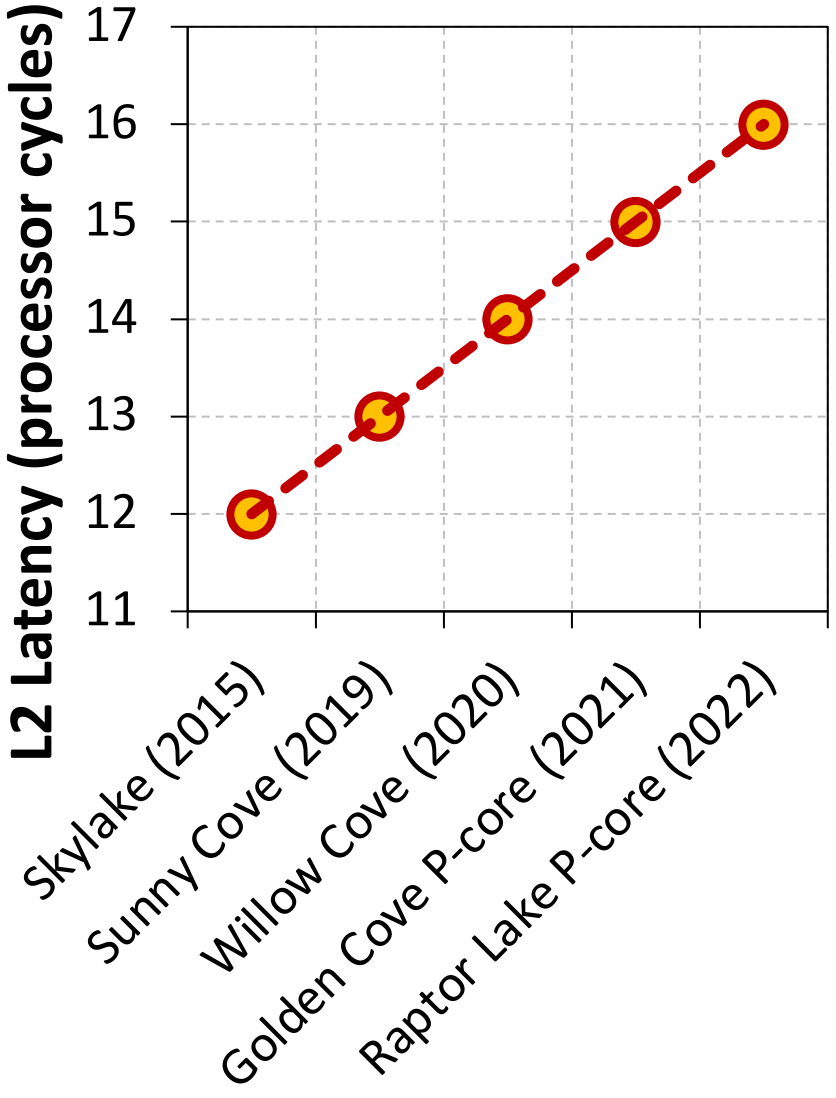



*Saved cycles*

**40%** of the **stalls** can be eliminated by **removing on-chip cache access latency from critical path**

# Caches are Getting Bigger and Slower…



# Our Goal

Improve processor performance
by **removing on-chip cache access latency**
from the **critical path of off-chip loads**

# HERMES

**Predicts** which load requests
are likely to **go off-chip**

Starts **fetching** data **directly** from **main memory**
while concurrently accessing the cache hierarchy

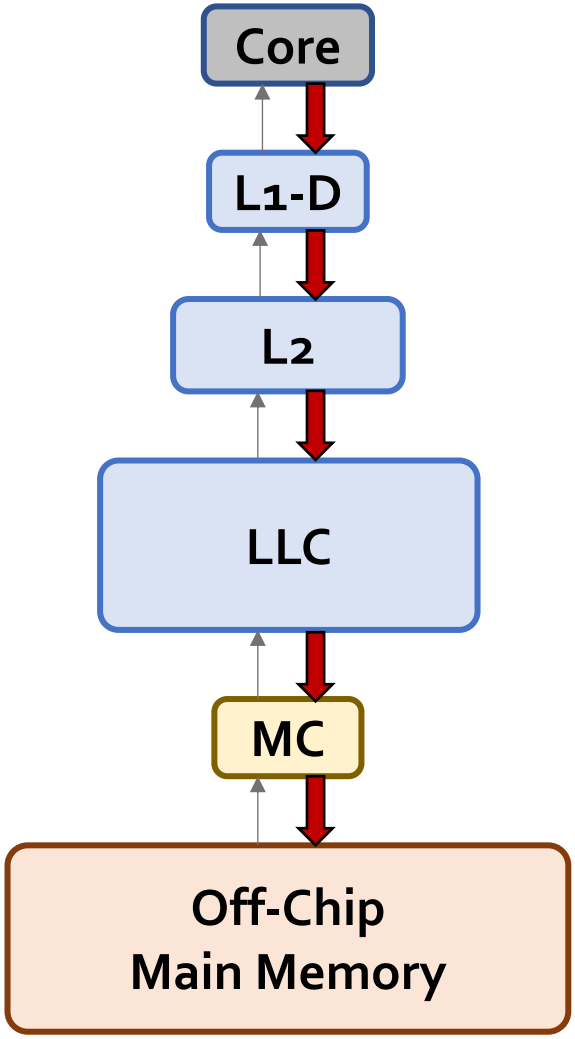
# Key Contribution

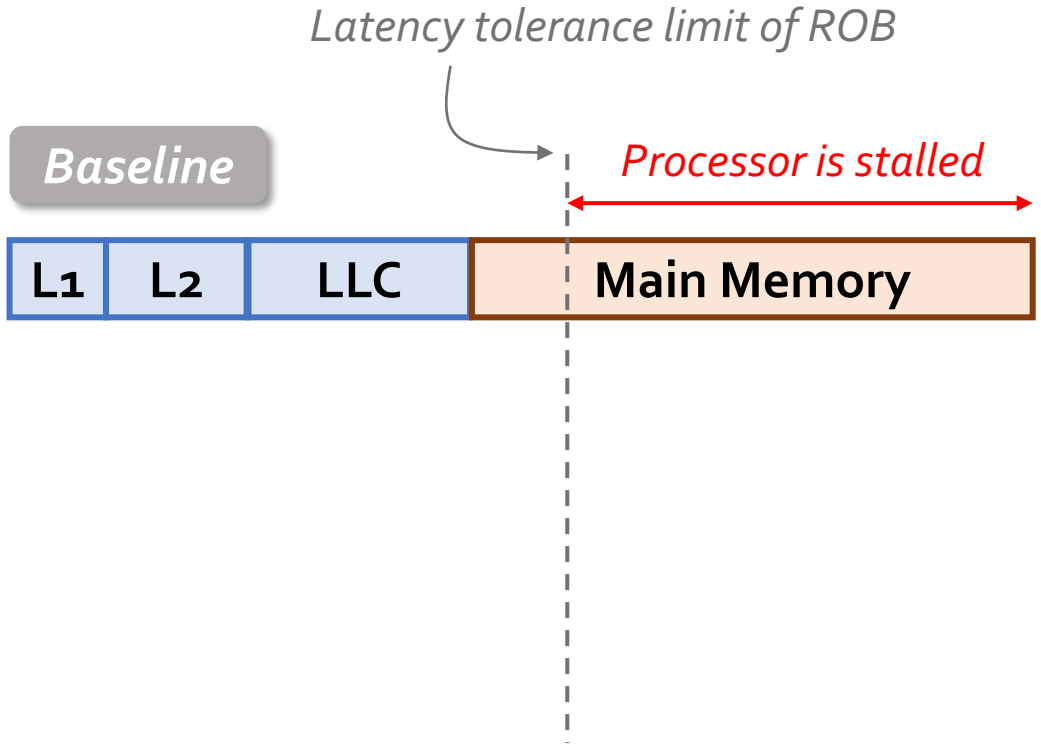
Hermes employs **the first perceptron-based** off-chip load predictor

That predicts which loads are likely to **go off-chip**

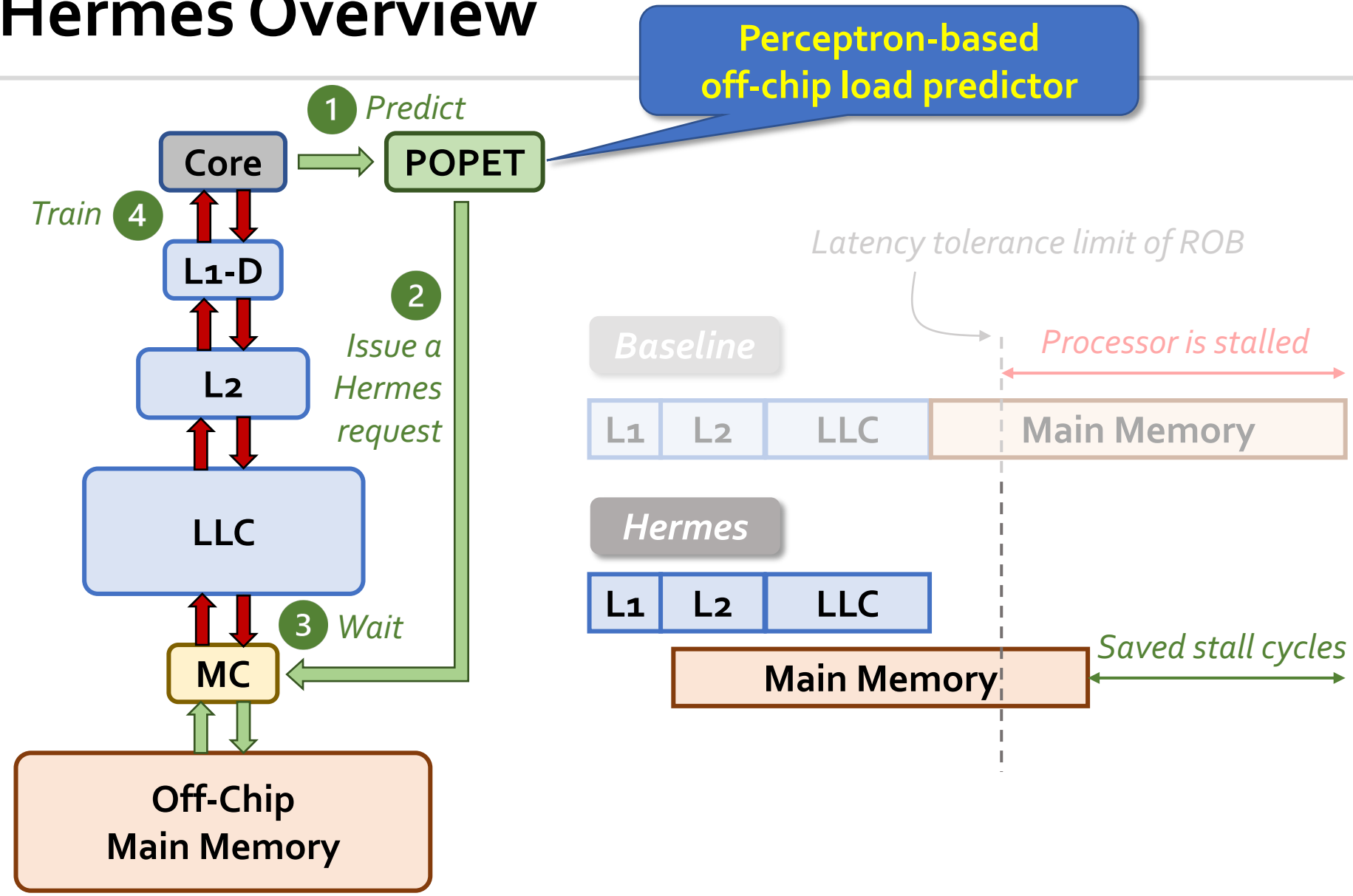
By **learning** from **multiple program context information**

# Hermes Overview



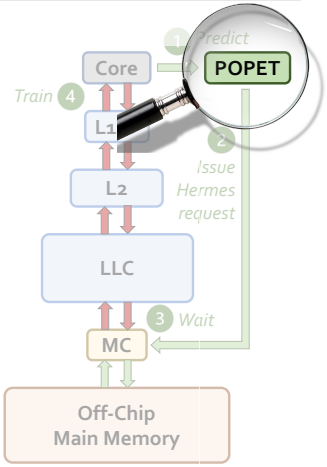



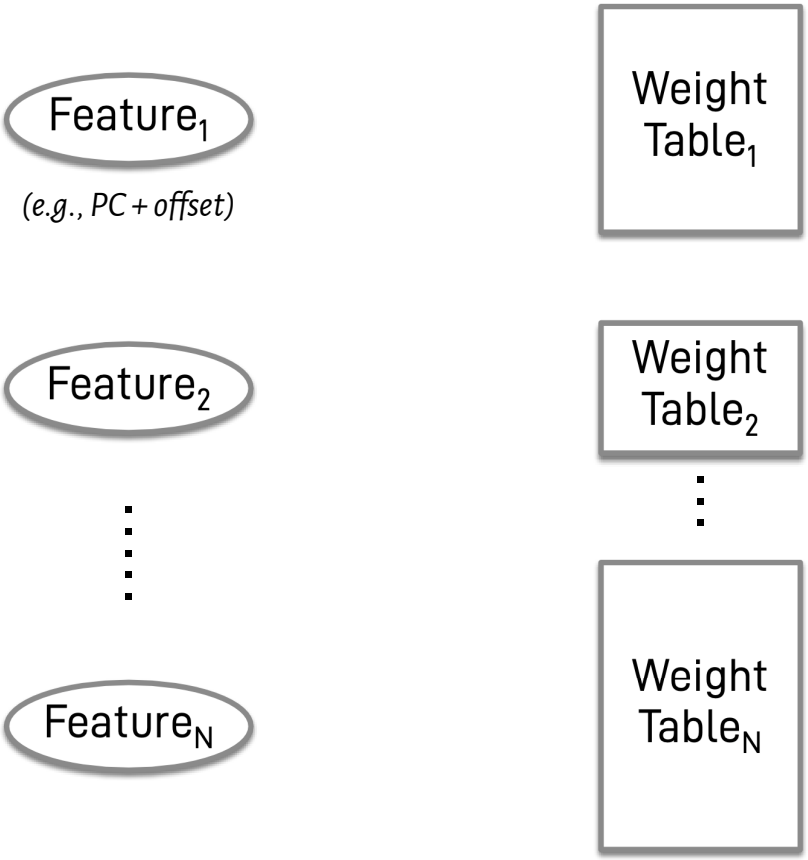
# Hermes Overview



# POPET: Perceptron-Based Off-Chip Predictor

- Multi-feature hashed perceptron model[1]
  - Each feature has its own *weight table*
    - Stores correlation between feature value and off-chip prediction



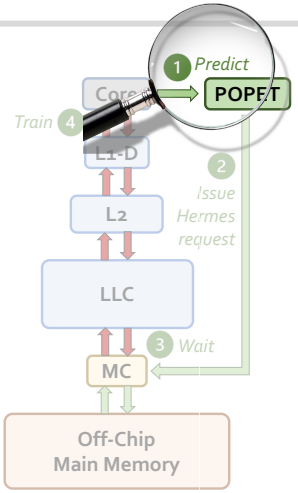



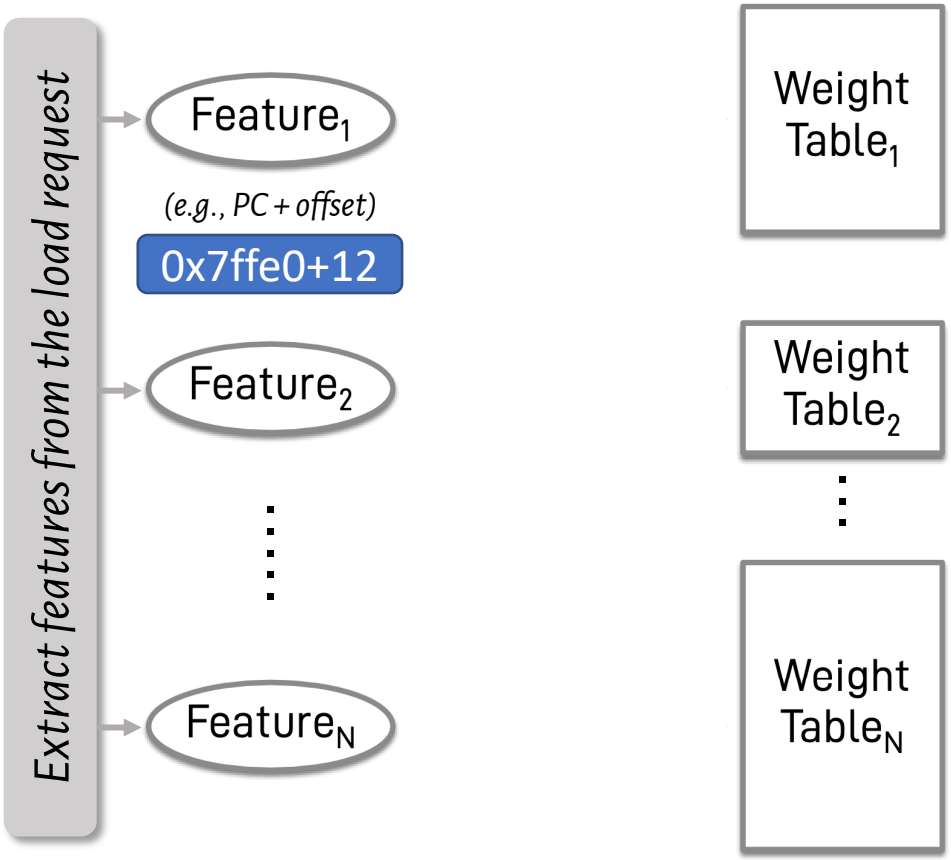


[1] D. Tarjan and K. Skadron, "Merging Path and Gshare Indexing in Perceptron Branch Prediction," TACO, 2005

# Predicting using POPET

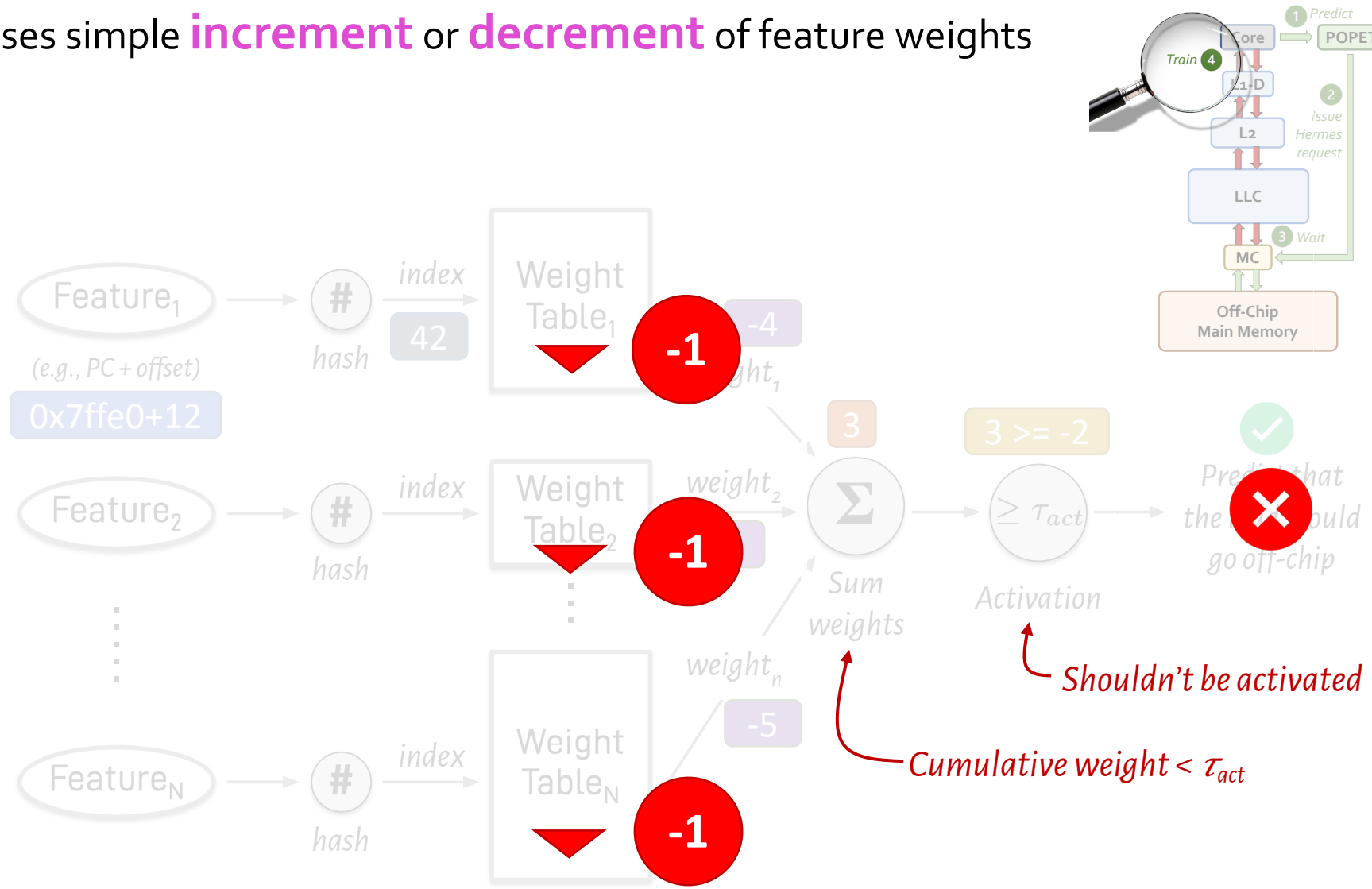
- Uses simple **table lookups**, **addition**, and **comparison**



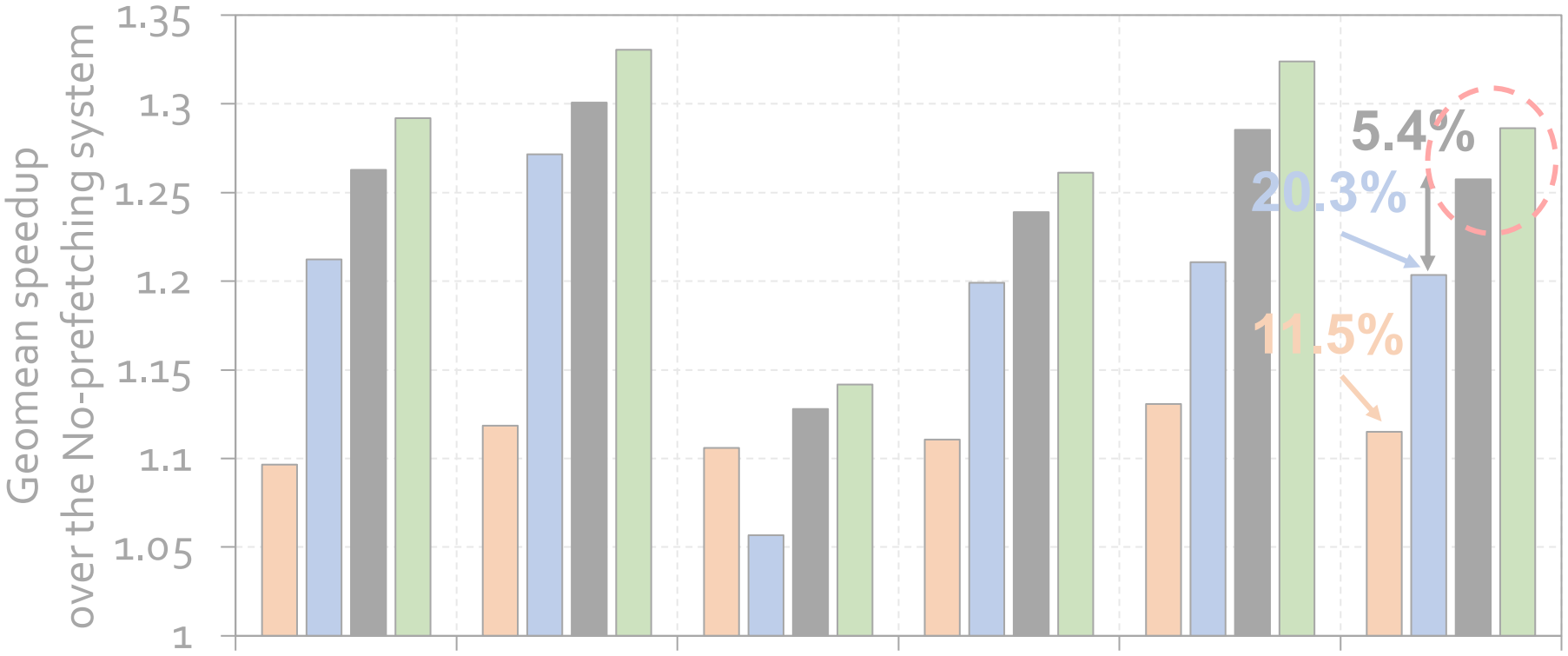



# Training POPET

- Uses simple **increment** or **decrement** of feature weights



# Single-Core Performance Improvement

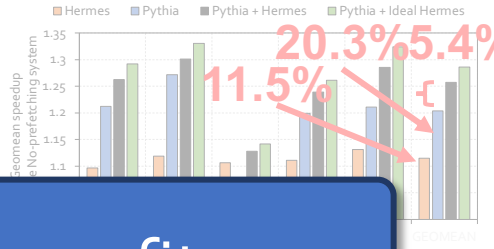



**Hermes alone** provides nearly

Hermes provides nearly **90%** performance benefit of **Ideal Hermes** that has an **ideal off-chip load predictor**

with only **2.5%** storage overhead

# Increase in Main Memory Requests

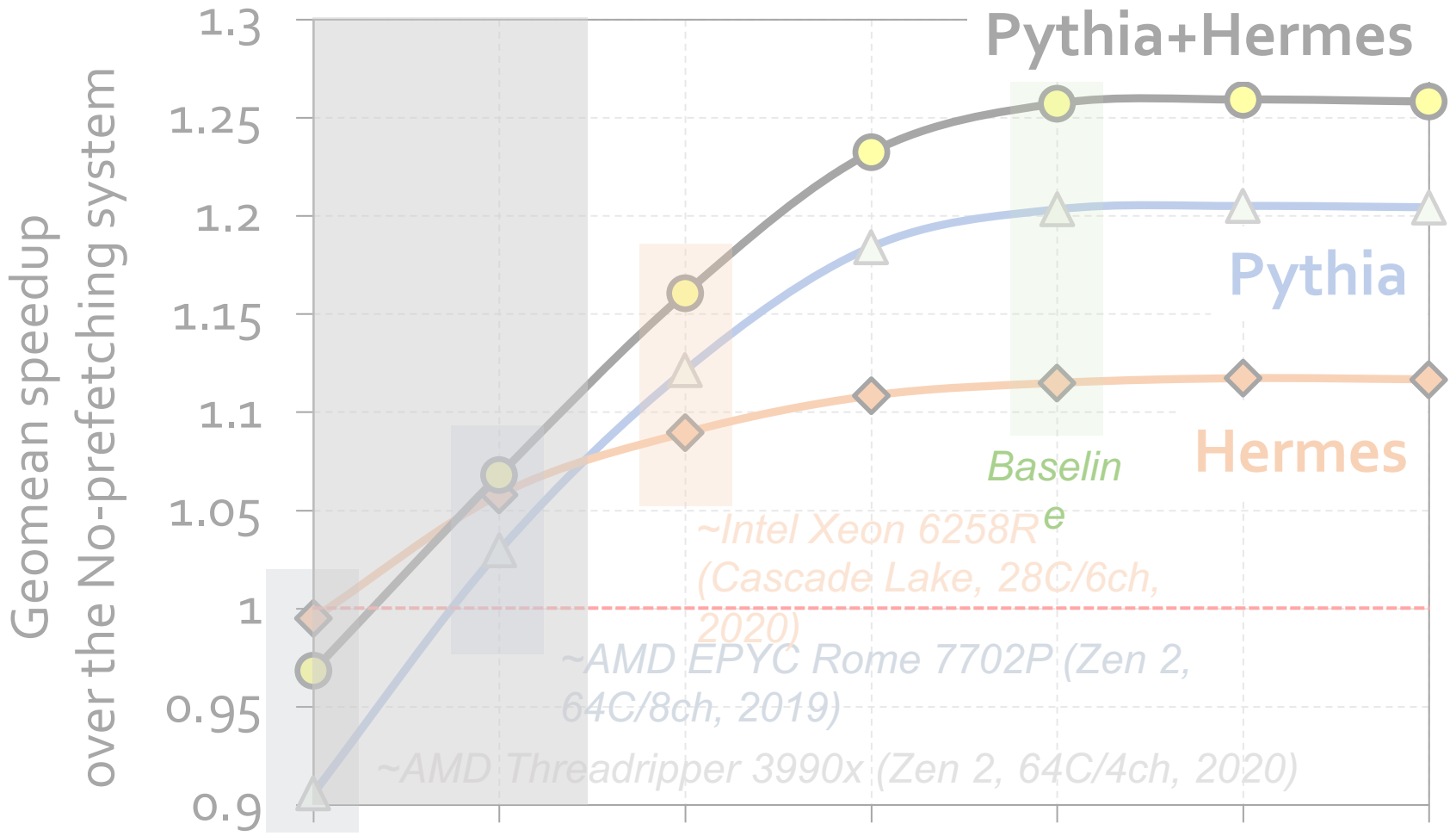



For **every 1% performance** benefit,
**increase in main memory requests**

| | |
|---|---|
| Pythia | **2%** |
| Hermes on top of Pythia | **1%** |
| Hermes alone | **0.5%** |

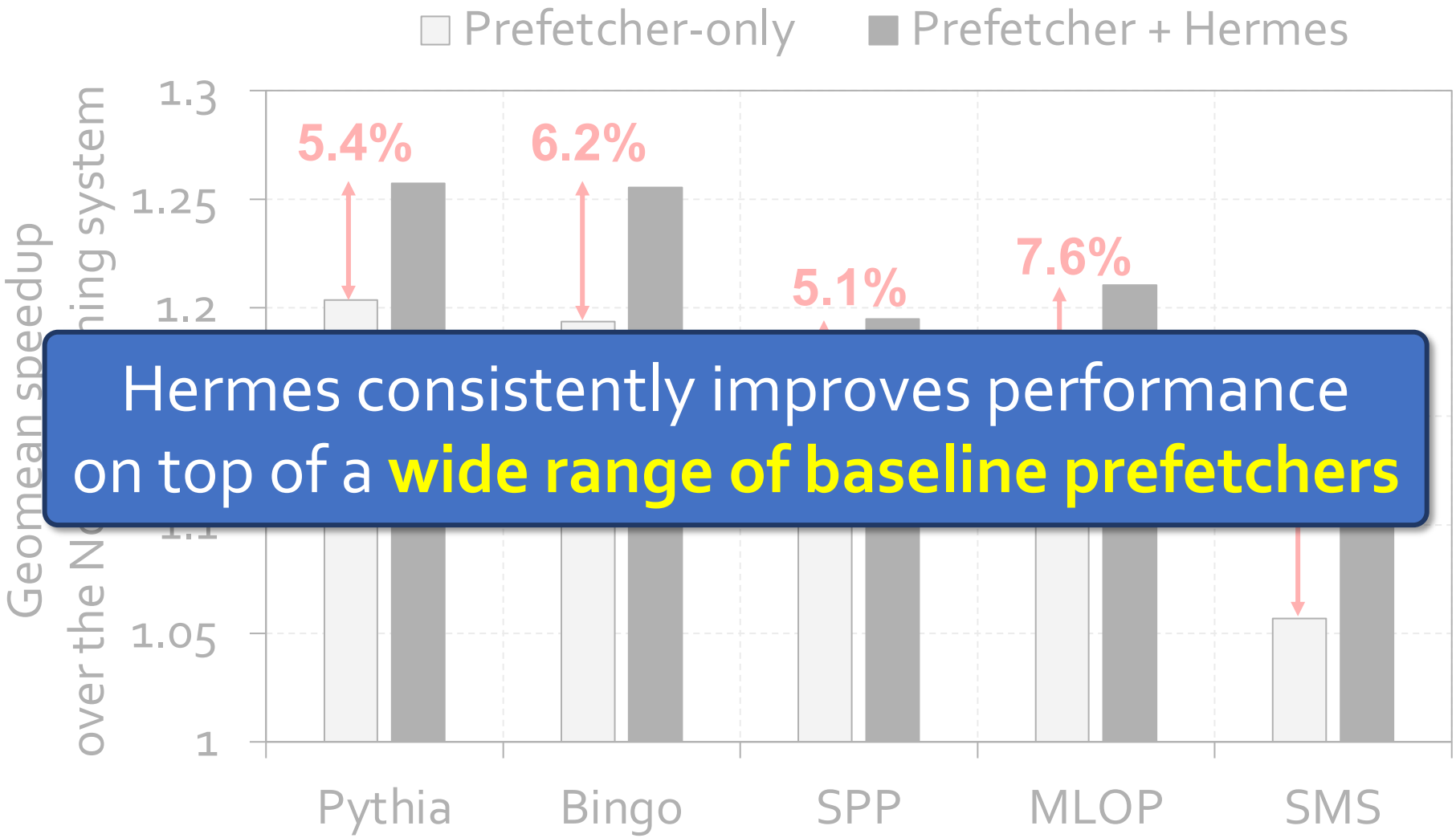
Hermes is more **bandwidth-efficient**
than even an efficient prefetcher like Pythia

# Performance with Varying Memory Bandwidth




Hermes+Pythia outperforms Pythia
**across all bandwidth configurations**

# Performance with Varying Baseline Prefetcher




Hermes consistently improves performance on top of a **wide range of baseline prefetchers**

# Overhead of Hermes

**4 KB** storage overhead

**1.5%** power overhead*

**On top of an Intel Alder Lake-like performance-core [2] configuration*



[2] https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3

# To Summarize…

# Summary

Hermes advocates for **off-chip load prediction**, a **different** form of speculation than **load address prediction** employed by prefetchers

**Off-chip load prediction** can be applied **by itself** or **combined with load address prediction** to provide performance improvement

# Summary

Hermes employs **the first perceptron-based** off-chip load predictor

**High accuracy**
(77%)

**High coverage**
(74%)

**Low storage overhead**
(4KB/core)

**High performance improvement**
over best prior baseline
(5.4%)

**High performance per bandwidth**

# Hermes is Open Sourced




All workload traces

**13** prefetchers

- Stride [Fu+, MICRO'92]
- Streamer [Chen and Baer, IEEE TC'95]
- SMS [Somogyi+, ISCA'06]
- AMPM [Ishii+, ICS'09]
- Sandbox [Pugsley+, HPCA'14]
- BOP [Michaud, HPCA'16]
- SPP [Kim+, MICRO'16]
- Bingo [Bakshalipour+, HPCA'19]
- SPP+PPF [Bhatia+, ISCA'19]
- DSPatch [Bera+, MICRO'19]
- MLOP [Shakerinava+, DPC-3'19]
- IPCP [Pakalapati+, ISCA'20]
- Pythia [Bera+, MICRO'21]

**9** off-chip predictors

| Predictor type | Description |
|---|---|
| Base | Always NO |
| Basic | Simple confidence counter-based threshold |
| Random | Random Hit-miss predictor with a given positive probability |
| HMP-Local | Hit-miss predictor [Yoaz+, ISCA'99] with local prediction |
| HMP-GShare | Hit-miss predictor with GShare prediction |
| HMP-GSkew | Hit-miss predictor with GSkew prediction |
| HMP-Ensemble | Hit-miss predictor with all three types combined |
| TTP | Tag-tracking based predictor |
| Perc | Perceptron-based OCP used in this paper |



https://github.com/CMU-SAFARI/Hermes


# HERMES

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

**Rahul Bera**, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadati, Onur Mutlu

https://github.com/CMU-SAFARI/Hermes




# 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"

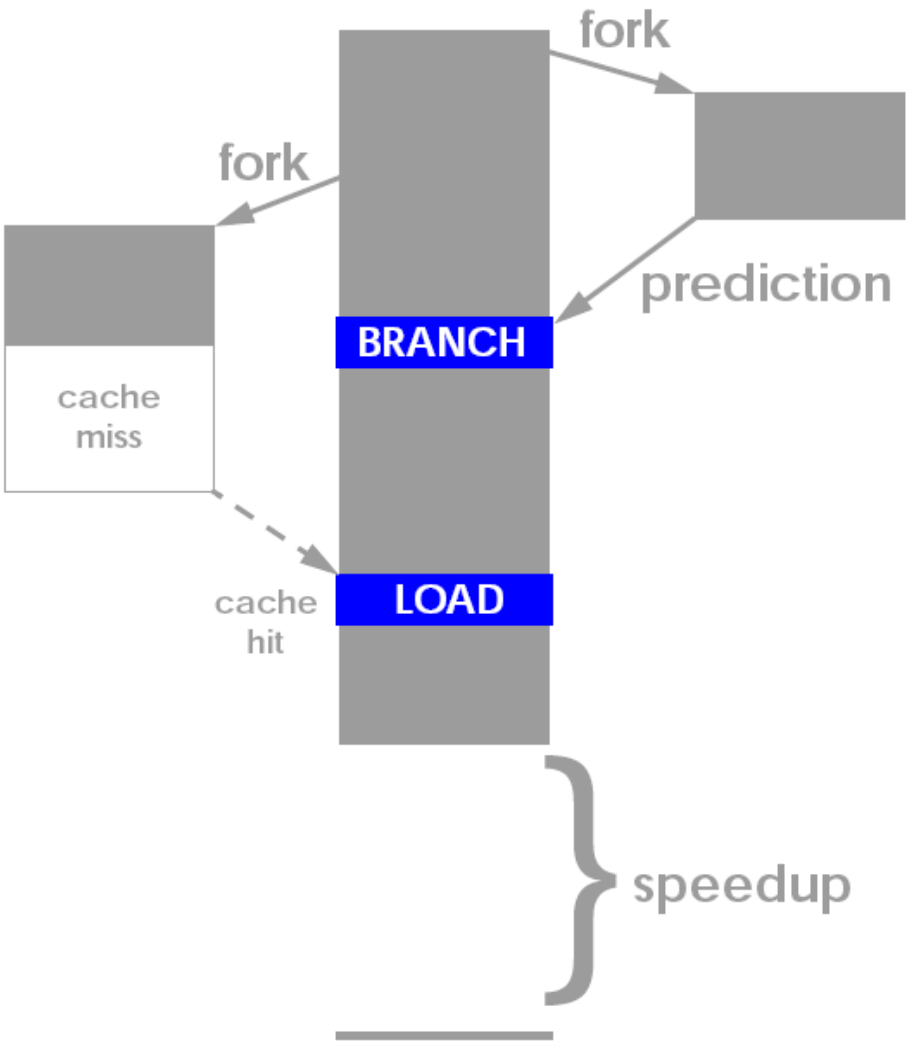
- Speculative thread can be executed
  - On a separate processor/core
  - On a separate hardware thread context (think fine-grained multithreading)
  - On the same thread context in idle cycles (during cache misses)

# 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



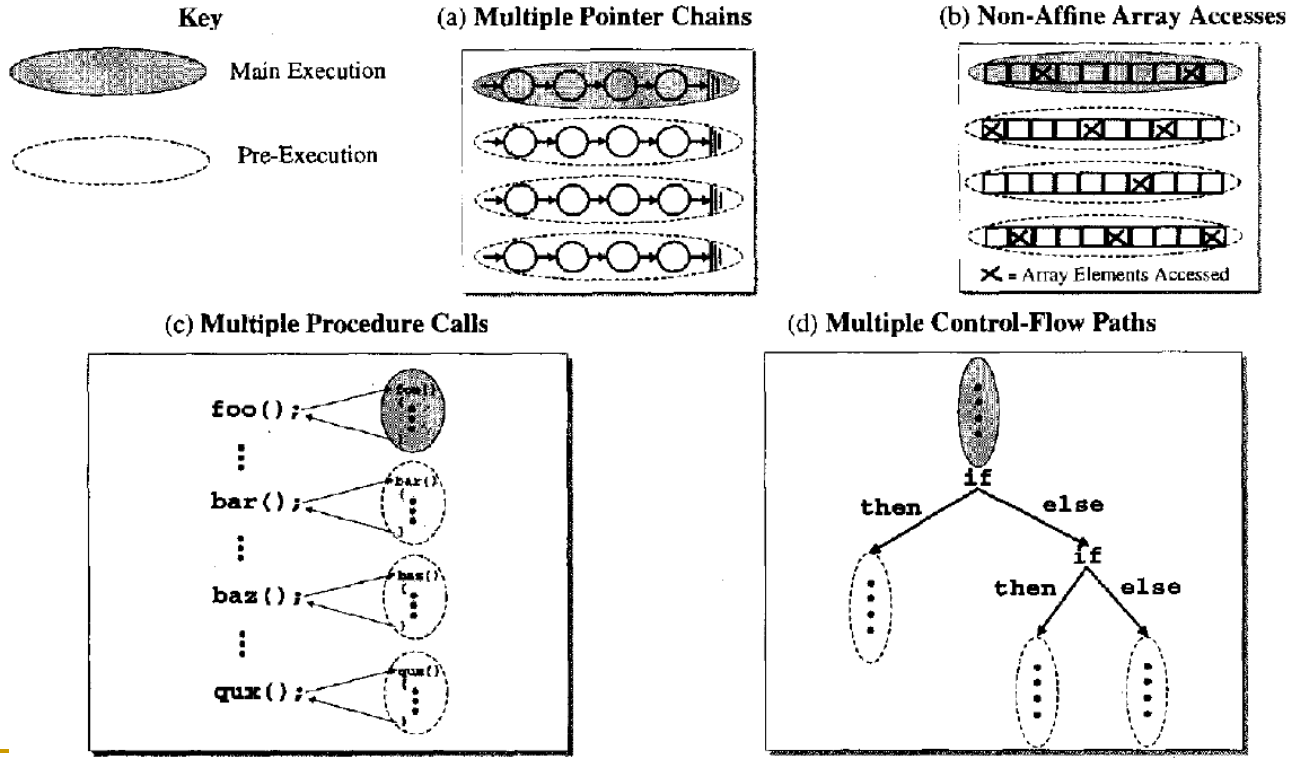

- Dubois and Song, "Assisted Execution," USC Tech Report 1998.
- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.
- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.

# 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();
```

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

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

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

# Example ISA Extensions

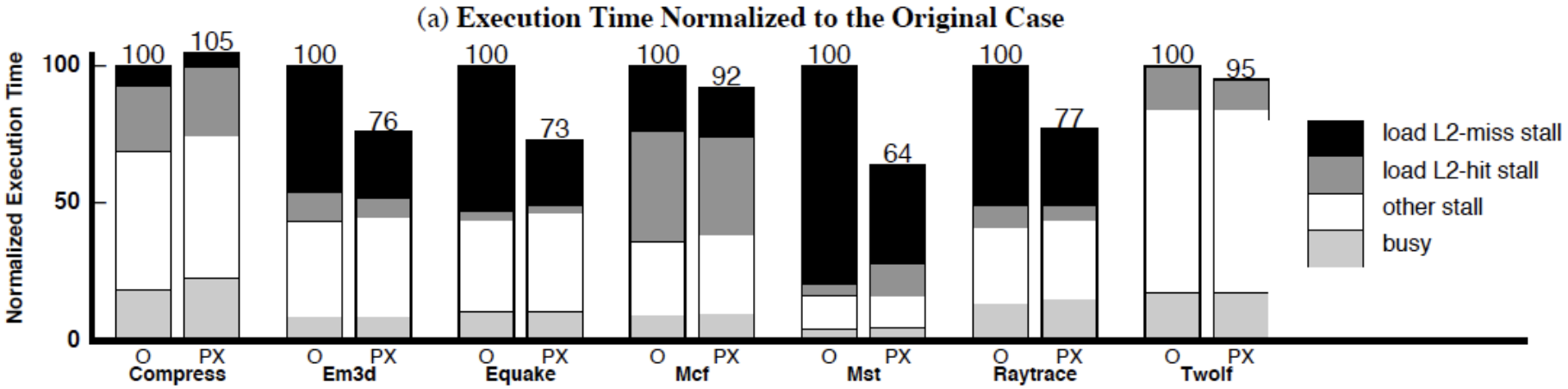
$Thread\_ID$ = **PreExecute_Start**($Start\_PC$, $Max\_Insts$):
Request for an idle context to start pre-execution at $Start\_PC$ and stop when $Max\_Insts$ instructions have been executed; $Thread\_ID$ holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction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

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

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

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

# Results on a Multithreaded Processor




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

cache miss

# Fork Point for Prefetching Thread

**Figure 3.** *The* **`node_to_heap`** *function, which serves as the fork point for the slice that covers* **`add_to_heap`**.

```
void node_to_heap (..., float cost, ...) {
    struct s_heap *hptr;   <-- fork point
    ...
    hptr = alloc_heap_data();
    hptr->cost = cost;
    ...
    add_to_heap (hptr);
}
```

# Pre-execution Thread Construction

**Figure 4.** *Alpha assembly for the* ***add_to_heap*** *function. The instructions are annotated with the number of the line in Figure 2 to which they correspond. The problem instructions are in bold and the shaded instructions comprise the un-optimized slice.*

```
node_to_heap:
     ... /* skips ~40 instructions */
2    lda     s1, 252(gp)     # &heap_tail
2    ldl     t2, 0(s1)       # ifrom = heap_tail
1    ldq     t5, -76(s1)     # &heap[0]
3    cmplt   t2, 0, t4       # see note
4    addl    t2, 0x1, t6     # heap_tail ++
1    s8addq  t2, t5, t3      # &heap[heap_tail]
4    stl     t6, 0(s1)       # store heap_tail
1    stq     s0, 0(t3)       # heap[heap_tail]
3    addl    t2, t4, t4      # see note
3    sra     t4, 0x1, t4     # ito = ifrom/2
5    ble     t4, return      # (ito < 1)
loop:
6    s8addq  t2, t5, a0      # &heap[ifrom]
6    s8addq  t4, t5, t7      # &heap[ito]
11   cmplt   t4, 0, t9       # see note
10   move    t4, t2          # ifrom = ito
6    ldq     a2, 0(a0)       # heap[ifrom]
6    ldq     a4, 0(t7)       # heap[ito]
11   addl    t4, t9, t9      # see note
11   sra     t9, 0x1, t4     # ito = ifrom/2
6    lds     $f0, 4(a2)      # heap[ifrom]->cost
6    lds     $f1, 4(a4)      # heap[ito]->cost
6    cmptlt  $f0,$f1,$f0     # (heap[ifrom]->cost
6    fbeq    $f0, return     #  < heap[ito]->cost)
8    stq     a2, 0(t7)       # heap[ito]
9    stq     a4, 0(a0)       # heap[ifrom]
5    bgt     t4, loop        # (ito >= 1)
return:
     ... /* register restore code & return */
```

*note: the divide by 2 operation is implemented by a 3 instruction sequence described in the strength reduction optimization.*

**Figure 5.** *Slice constructed for example problem instructions. Much smaller than the original code, the slice contains a loop that mimics the loop in the original code.*

```
slice:
1    ldq     $6, 328(gp)     # &heap
2    ldl     $3, 252(gp)     # ito = heap_tail
slice_loop:
3,11 sra     $3, 0x1, $3     # ito /= 2
6    s8addq  $3, $6, $16     # &heap[ito]
6    ldq     $18, 0($16)     # heap[ito]
6    lds     $f1, 4($18)     # heap[ito]->cost
6    cmptle  $f1,$f17,$f31   # (heap[ito]->cost
                             #  < cost) PRED
     br      slice_loop

## Annotations
fork: on first instruction of node_to_heap
live-in: $f17<cost>, gp
max loop iterations: 4
```

# An Execution-Based Prefetcher: Runahead Execution

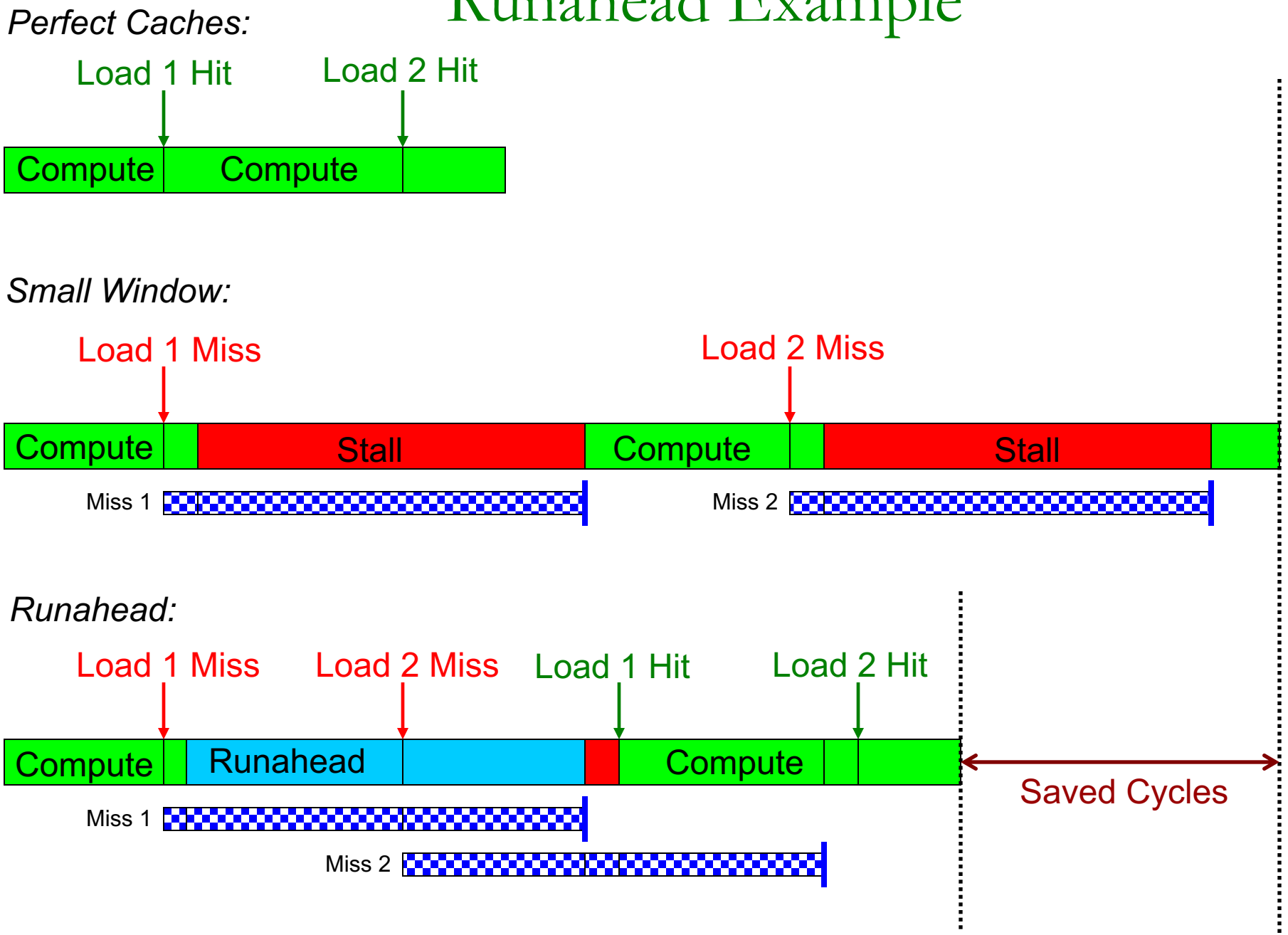
# Runahead Execution

- 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 (with minimal stalling)
  - The purpose of pre-execution is to generate prefetches
  - Long-latency instructions are marked INV and **dropped**
    - **Enables room for later instructions in the window**
- 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

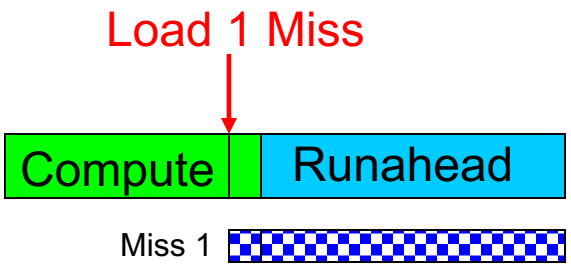
Instead of stalling during an L2 cache miss:

- Processor speculative pre-executes the program far ahead into the instruction stream without stalling for long-latency instructions
- 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

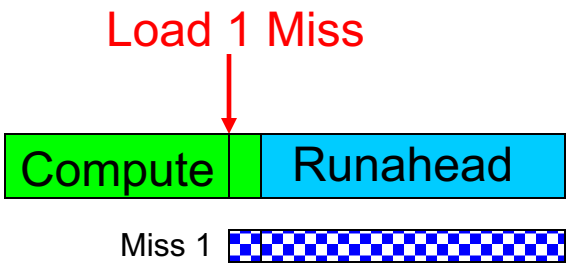
# Instruction Processing in Runahead Mode




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

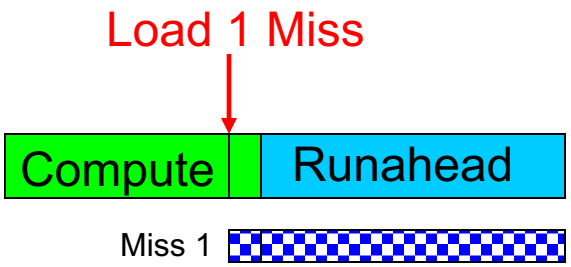
- It is purely speculative: Architectural (software-visible) register/memory state is NOT updated in runahead mode.
- L2-miss dependent instructions are identified and treated specially.
  - They are quickly removed from the instruction window.
  - Their results are not trusted.

# L2-Miss Dependent Instructions



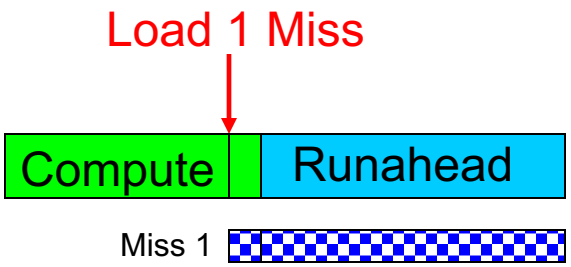

- Two types of results produced: INV and VALID
- INV = Dependent on an L2 miss
- INV results are marked using INV bits in the register file and store buffer.
- INV values are not used for prefetching/branch resolution.

# Removal of Instructions from Window



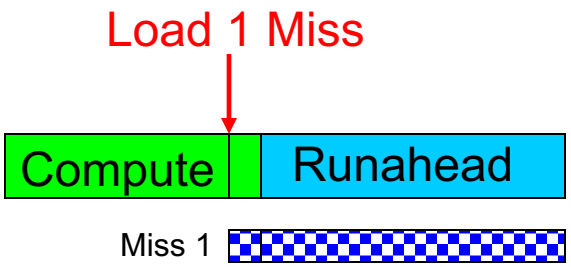

- Oldest instruction is examined for pseudo-retirement
  - An INV instruction is removed from window immediately.
  - A VALID instruction is removed when it completes execution.
- Pseudo-retired instructions free their allocated resources.
  - This allows the processing of later instructions.
- Pseudo-retired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

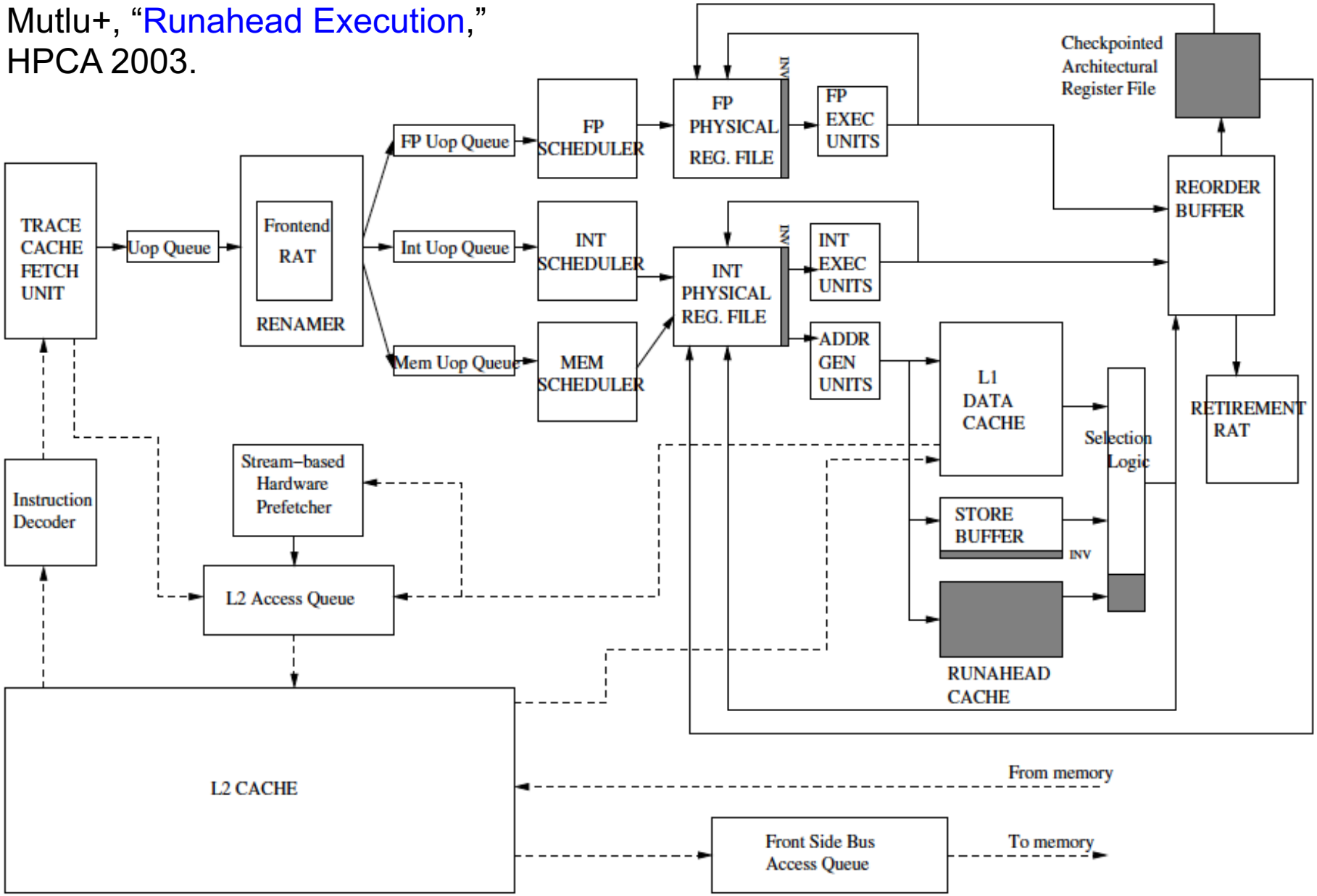
# Branch Handling in Runahead Mode




- 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.



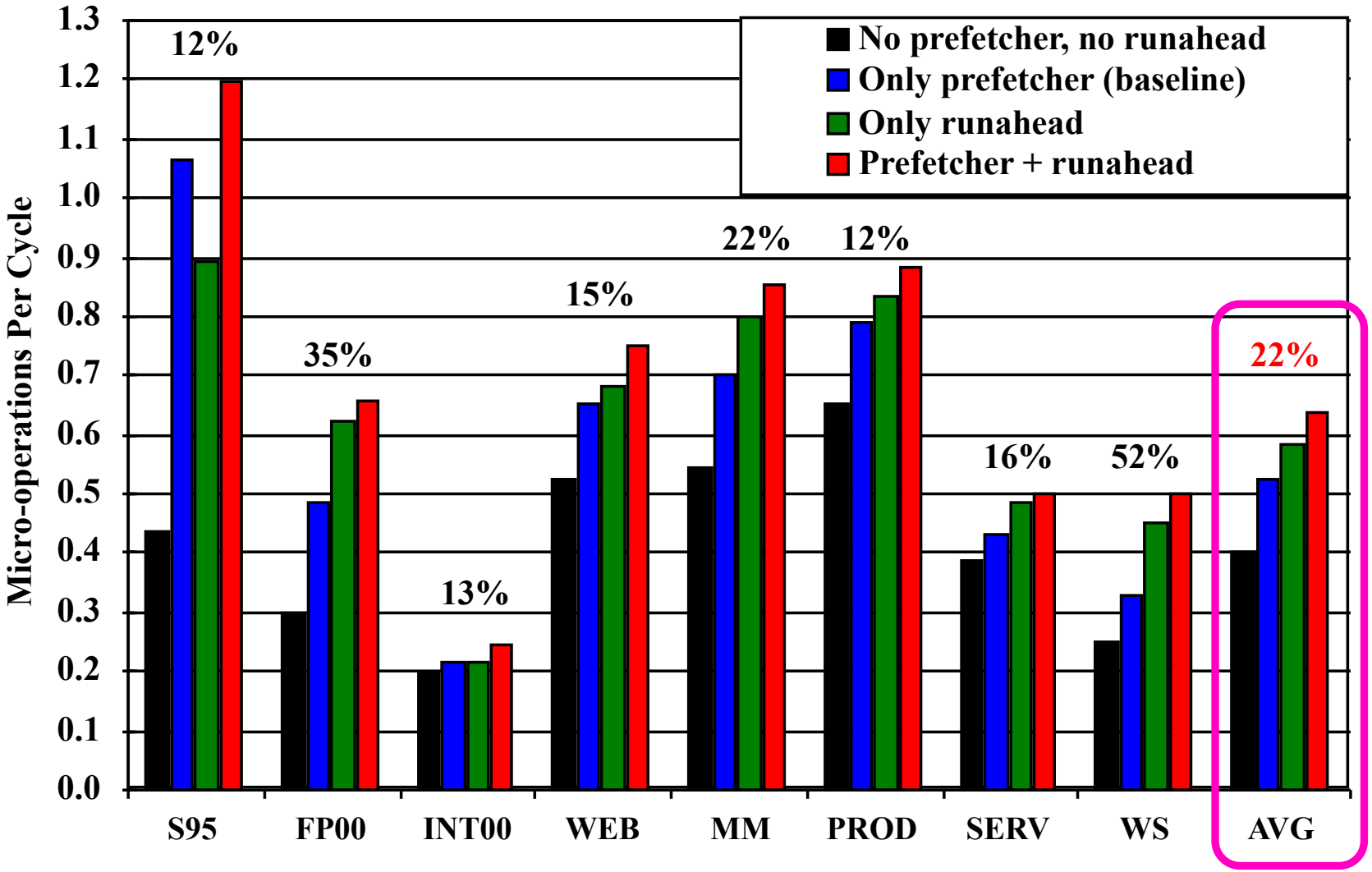
# Runahead Execution Pros and Cons

- 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

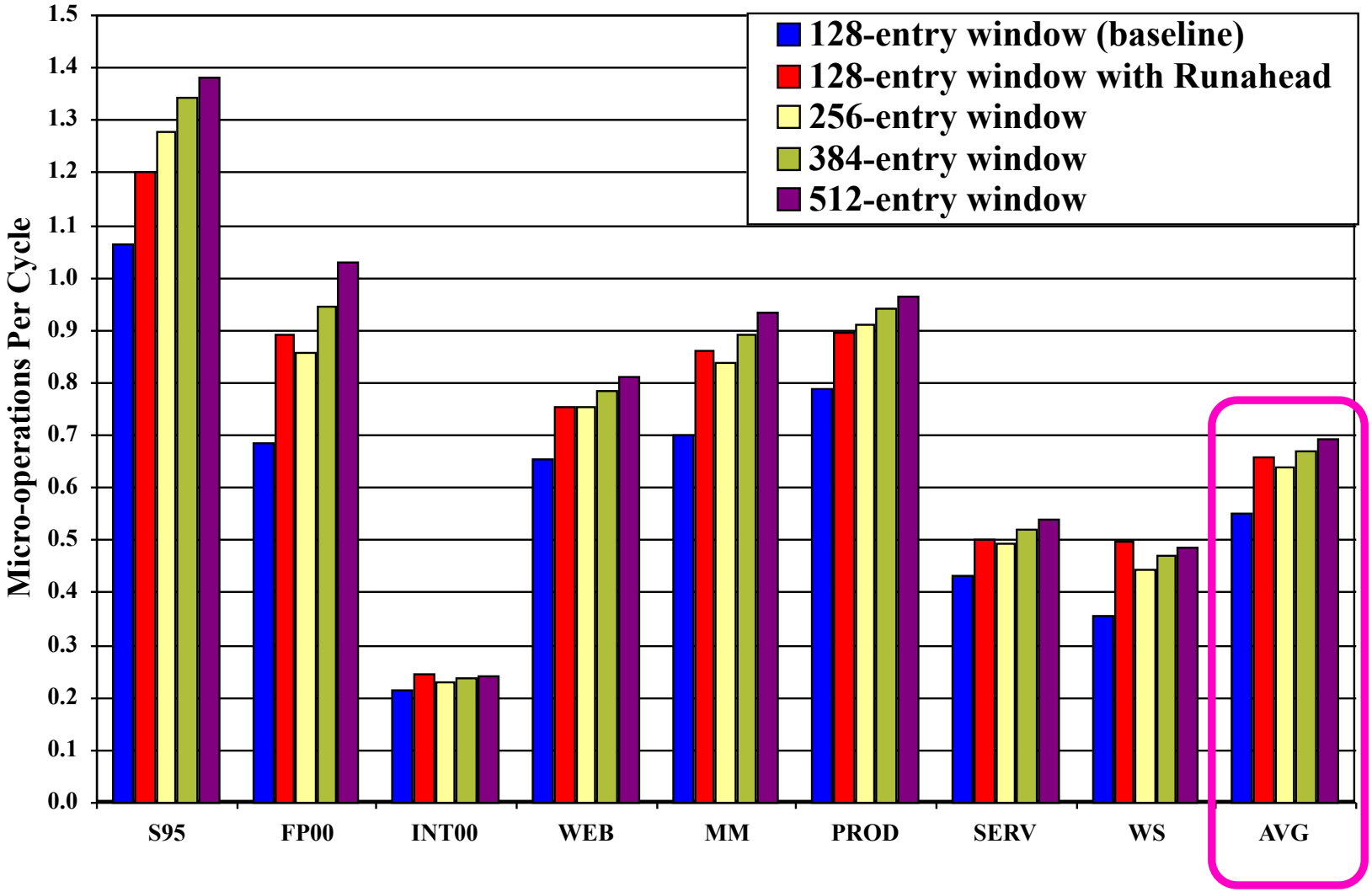
- Disadvantages/Limitations
  - -- Extra executed instructions
  - -- Limited by branch prediction accuracy
  - -- Cannot prefetch dependent cache misses
  - -- Effectiveness limited by available "memory-level parallelism" (MLP)
  - -- Prefetch distance (how far ahead to prefetch) limited by memory latency

- Implemented in Sun ROCK, IBM POWER6, NVIDIA Denver

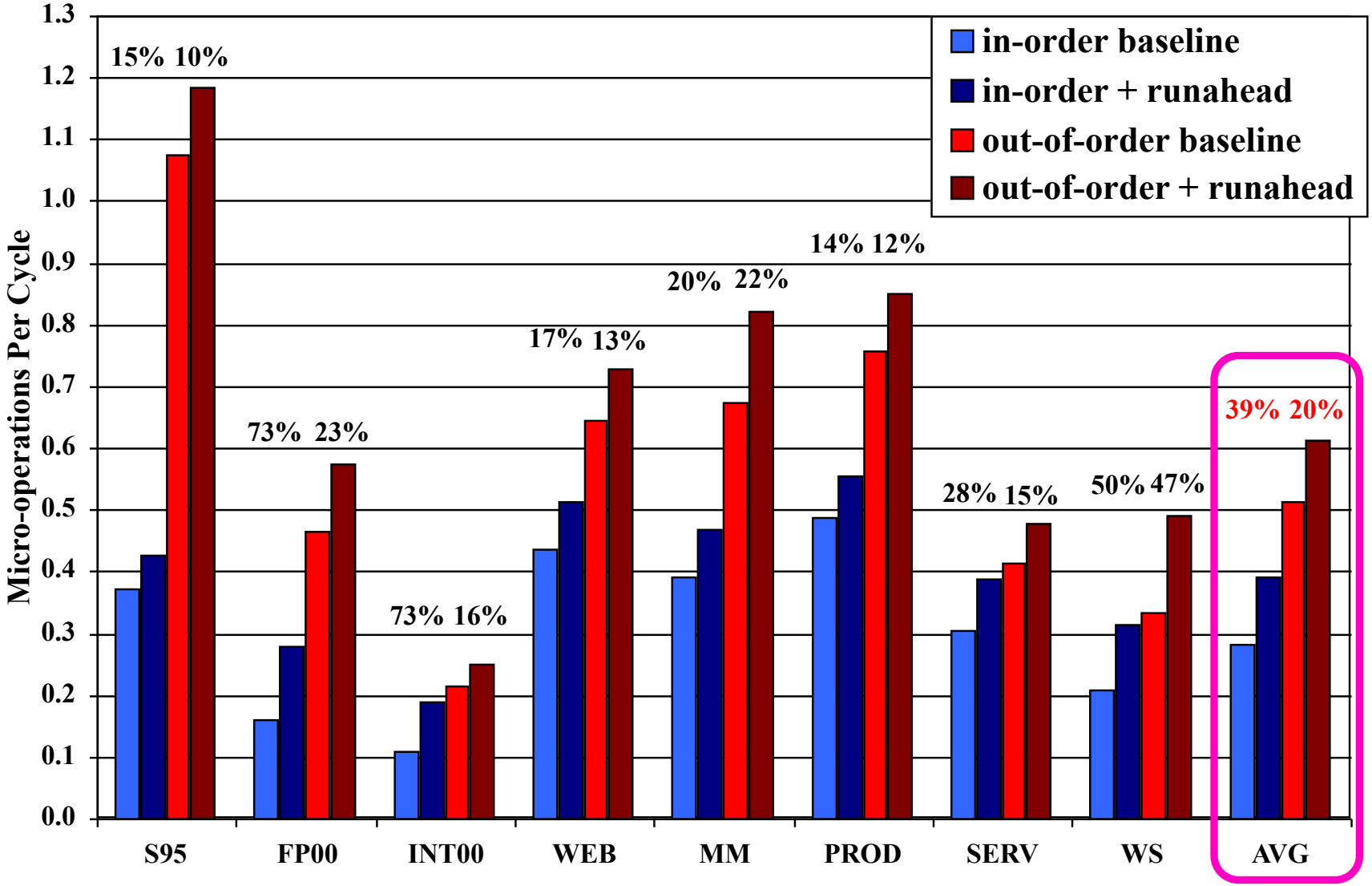
# Performance of Runahead Execution



# Runahead Execution vs. Large Windows

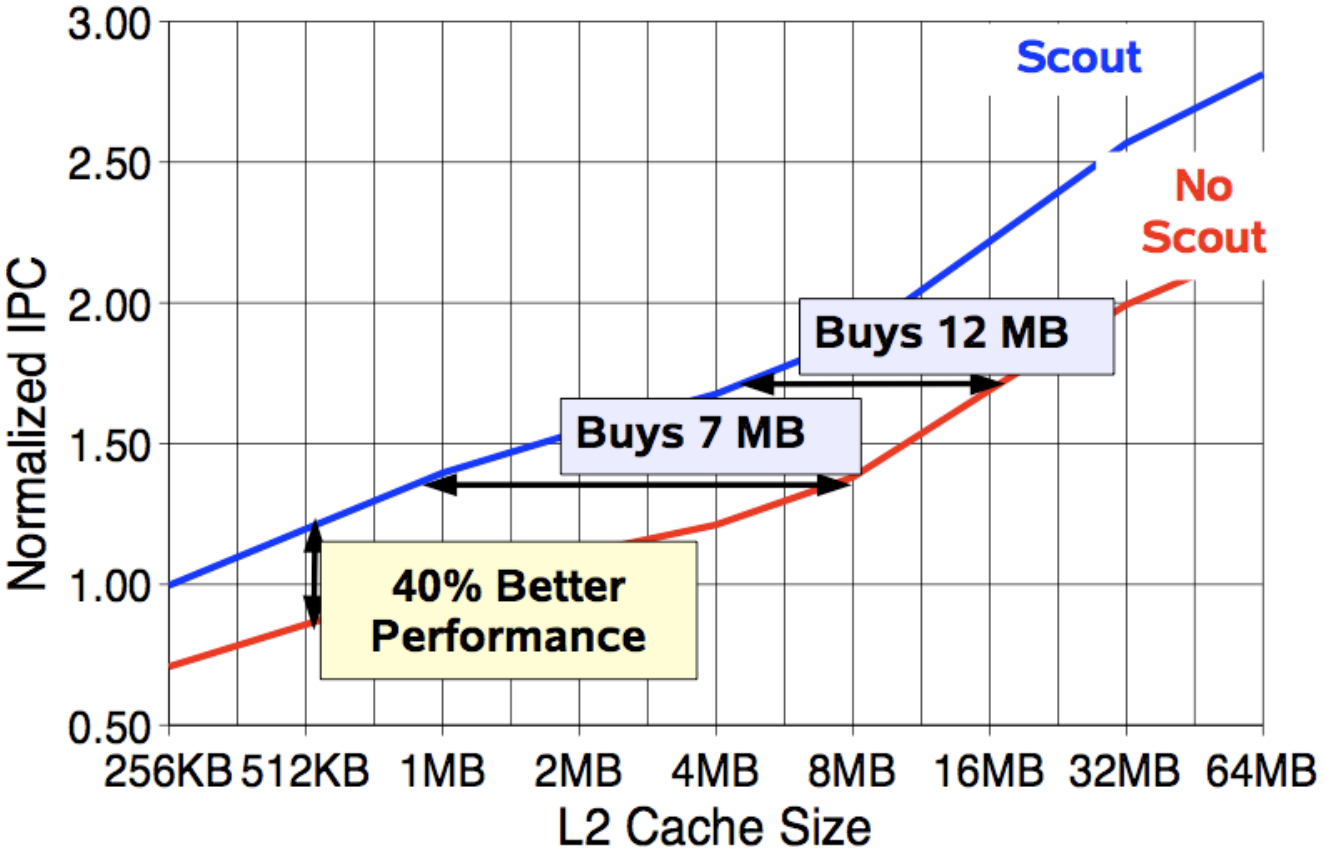


# Runahead on In-order vs. Out-of-order



# Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.




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

# 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**

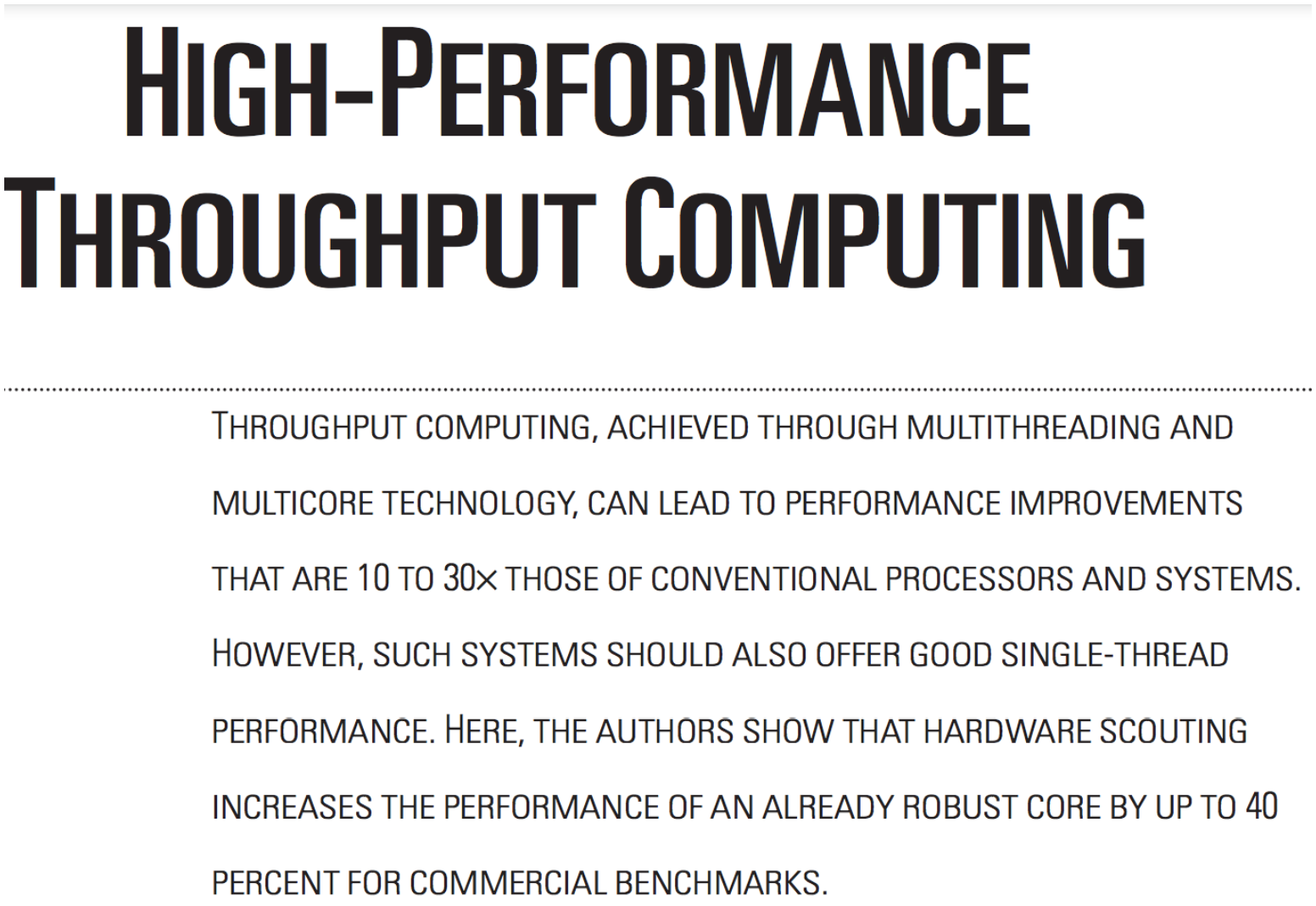
Onur Mutlu § Jared Stark † Chris Wilkerson ‡ Yale N. Patt §

§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

# More on Runahead in Sun ROCK



HIGH-PERFORMANCE THROUGHPUT COMPUTING

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

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

# More on Runahead in Sun ROCK

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

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

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

# 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).*

# Runahead Execution in NVIDIA Denver

## 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 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.[1] These prefetch requests warm up the data cache and DTLB well before the actual execution of the instructions that require the data. Runahead 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).

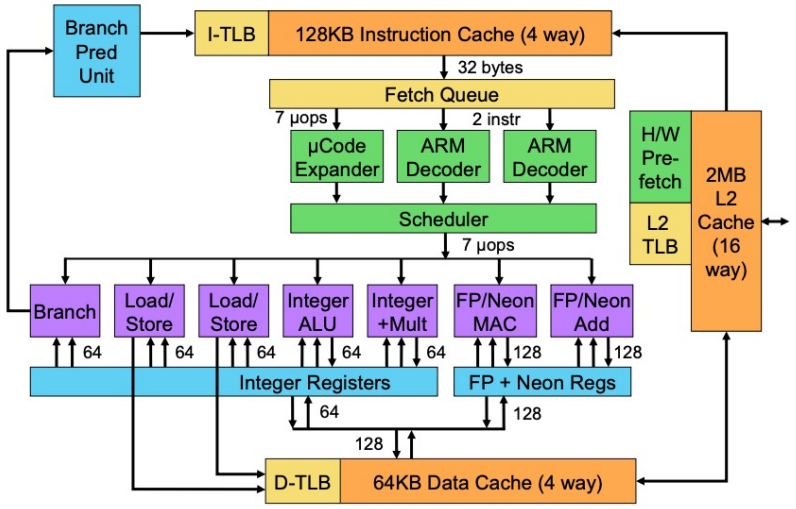



**Figure 3. Denver CPU microarchitecture.** This design combines a fairly

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

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

# Runahead Enhancements

# Runahead Enhancements

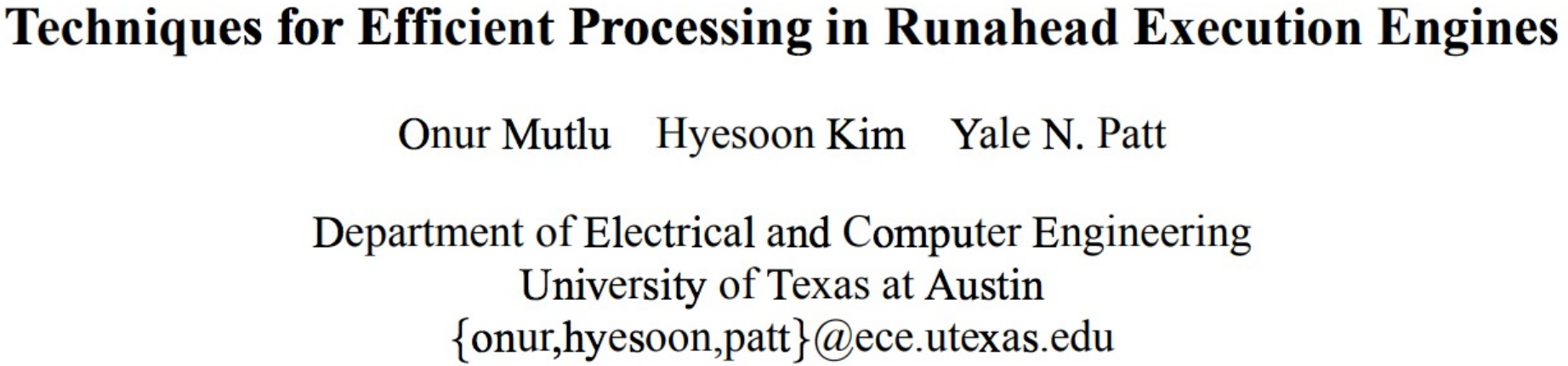
- Mutlu et al., "Techniques for Efficient Processing in Runahead Execution Engines," ISCA 2005, IEEE Micro Top Picks 2006.
- Mutlu et al., "Address-Value Delta (AVD) Prediction," MICRO 2005.
- Armstrong et al., "Wrong Path Events," MICRO 2004.
- Mutlu et al., "An Analysis of the Performance Impact of Wrong-Path Memory References on Out-of-Order and Runahead Execution Processors," IEEE TC 2005.
- Hashemi et al., "Continuous Runahead," MICRO 2016.

# 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]
- Limited runahead distance
  - Runahead ends when the miss that caused it returns
  - Continuous Runahead [MICRO'16]

# 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

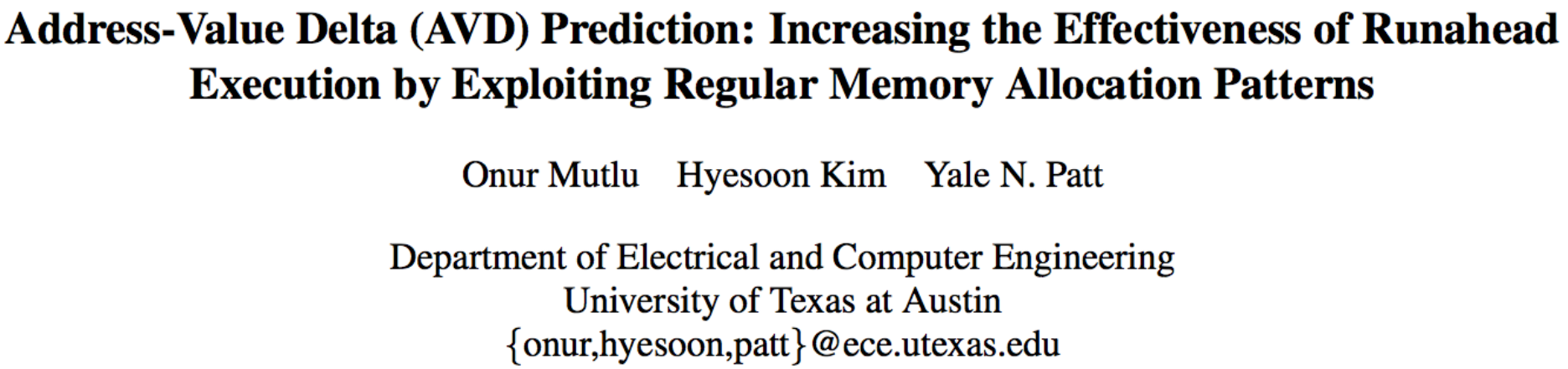
Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu

# 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.

# 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.***



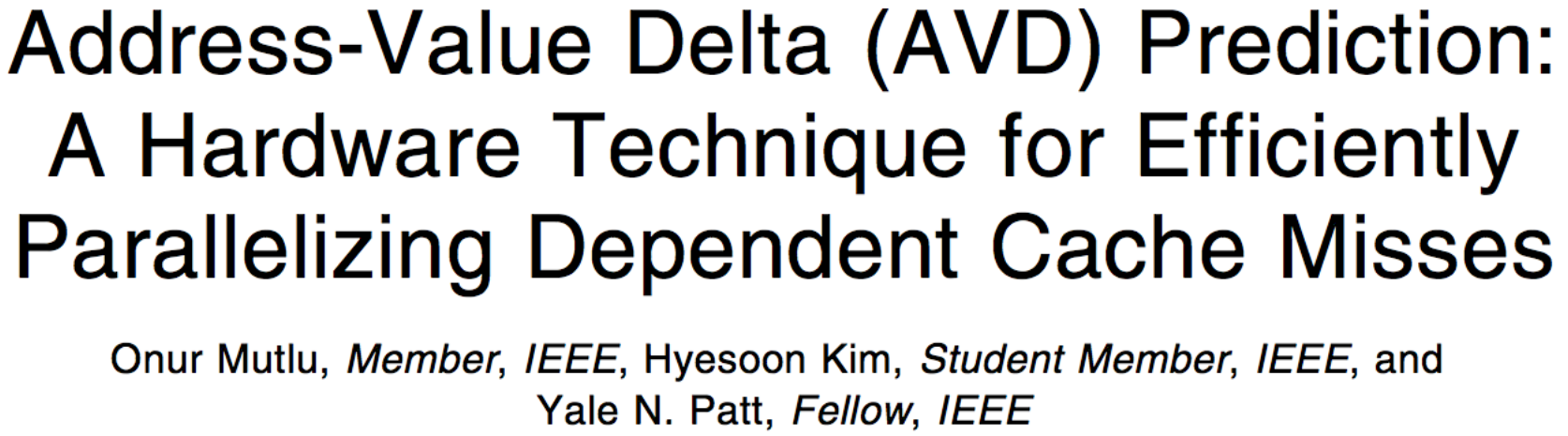
**Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns**

Onur Mutlu Hyesoon Kim Yale N. Patt

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

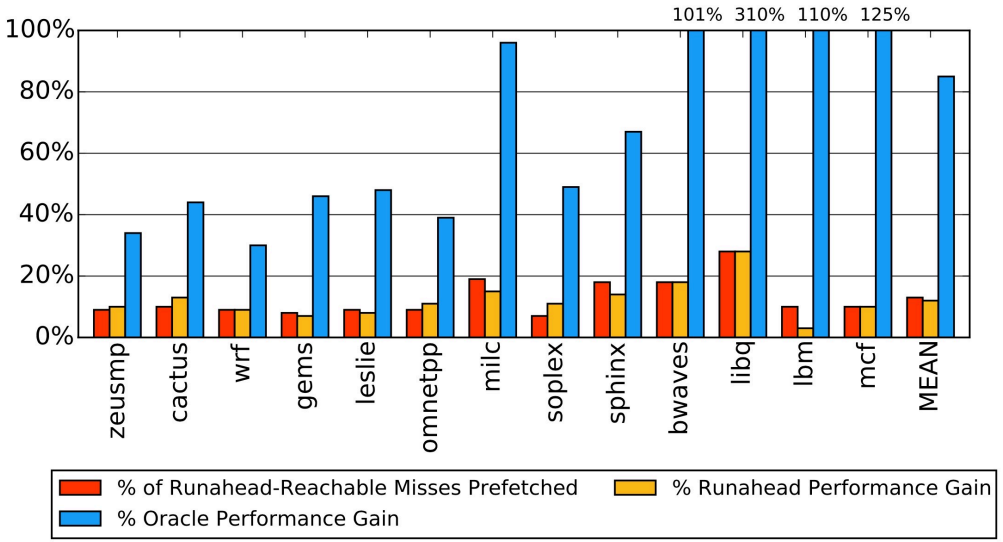
# More on Efficient Runahead Execution

- 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.

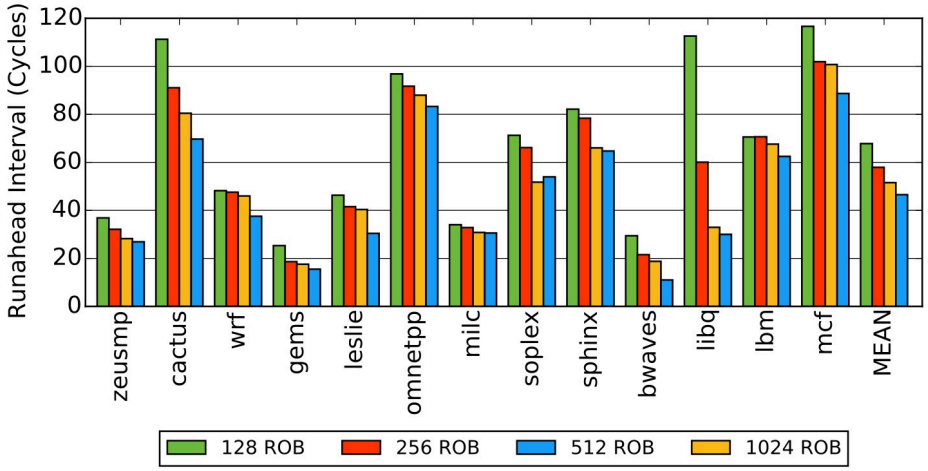


# Continuous Runahead (I)

- **Key Observations:**
  - Runahead covers only 13% of all runahead-reachable misses




  - Why? Because runahead execution interval is very short (60 cycles on average)
  - Runahead ends after the miss that caused it is serviced



# Continuous Runahead (II)

- **Key Idea:** Keep runahead going even after a long-latency cache miss is serviced
  - Identify chains of instructions that lead to LL cache misses
  - Keep executing each chain of instructions in a loop in runahead mode using specialized runahead execution hardware (CRE)
    - to continue generating long-latency cache misses
    - CRE is located in the memory controller
  - Stop executing the chain after a fixed number of instructions

- **Key Results:**
  - 70% coverage of runahead-reachable misses (up from 13%)
  - 22-24% performance gain over best runahead implementation in single-core systems

# Continuous Runahead (III)

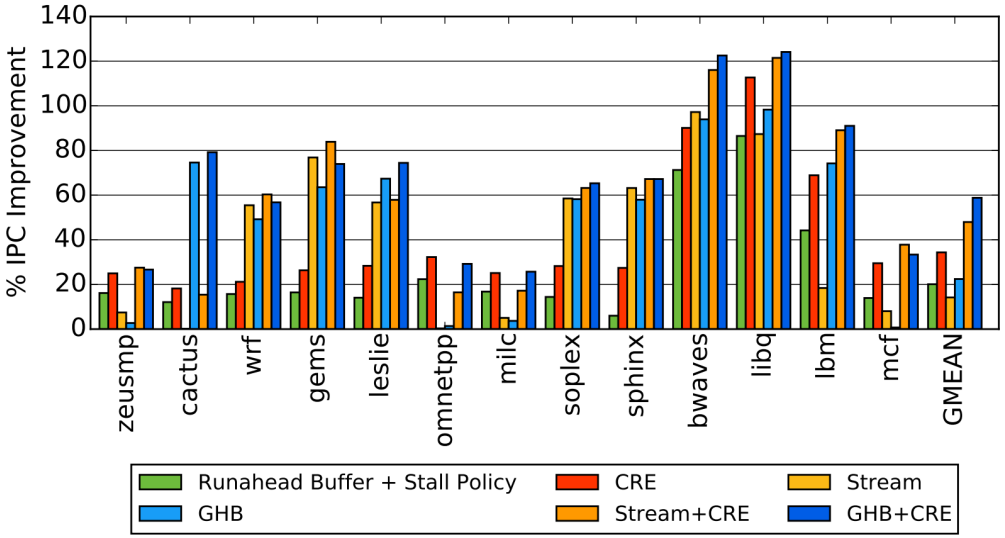



**Figure 11: Single-core performance.**

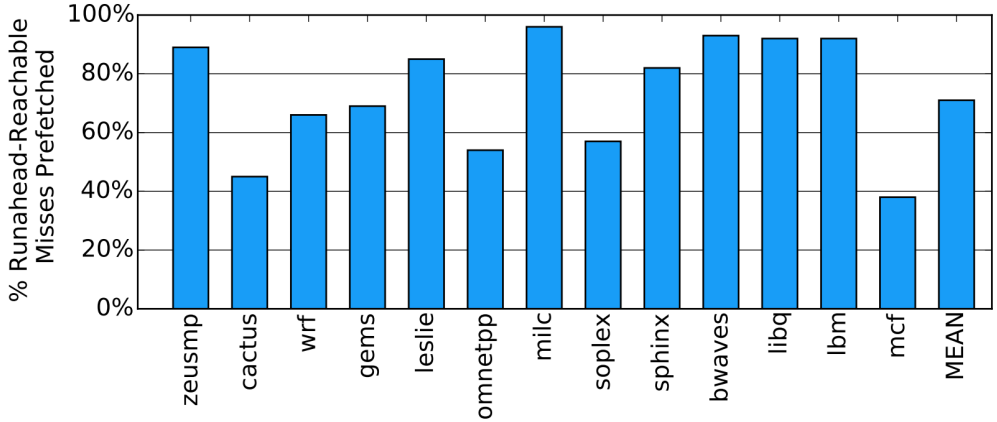
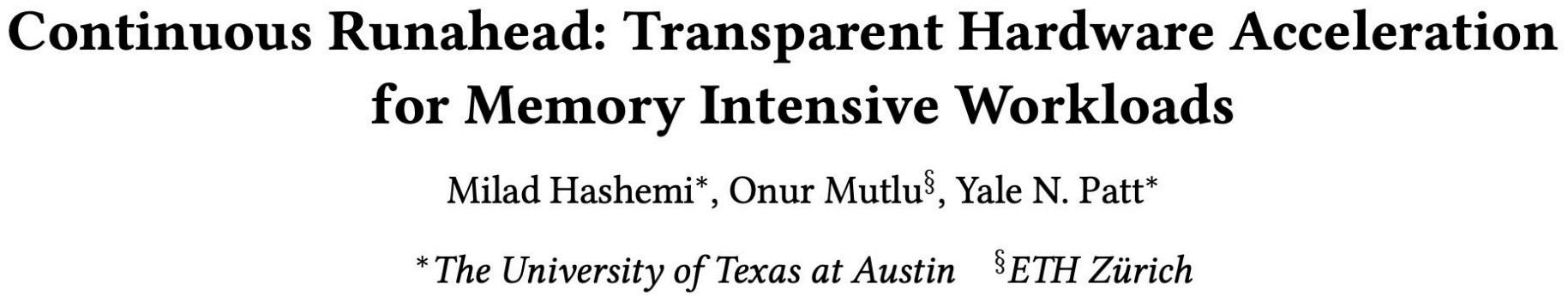



**Figure 12: Percentage of runahead-reachable misses prefetched by the CRE.**

# 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§, Yale N. Patt*

*The University of Texas at Austin §ETH Zürich

# Decoupled Vector Runahead

- **Key Idea**: (1) Vectorizes scalar operations inside a loop to generate prefetches to many independent loads at once, (2) Proactively prefetches even when core has not yet stalled

**Decoupled Vector Runahead**


Ajeya Naithani
Ghent University
Belgium

Jaime Roelandts
Ghent University
Belgium

Sam Ainsworth
University of Edinburgh
United Kingdom

Timothy M. Jones
University of Cambridge
United Kingdom

Lieven Eeckhout
Ghent University
Belgium


Best Paper Award, MICRO 2023

https://ajnaithani.github.io/docs/micro23.pdf

# 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


Dundas+, "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss," ICS 1997.

# 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.



Glew, "MLP yes! ILP no!," ASPLOS WACI 1998.

# 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.

# Suggestion to Researchers: Principle: Passion

Follow Your Passion

(Do not get derailed by naysayers)

# Suggestion to Researchers: Principle: Resilience

Be Resilient

# Principle: Learning and Scholarship

Focus on

learning and scholarship

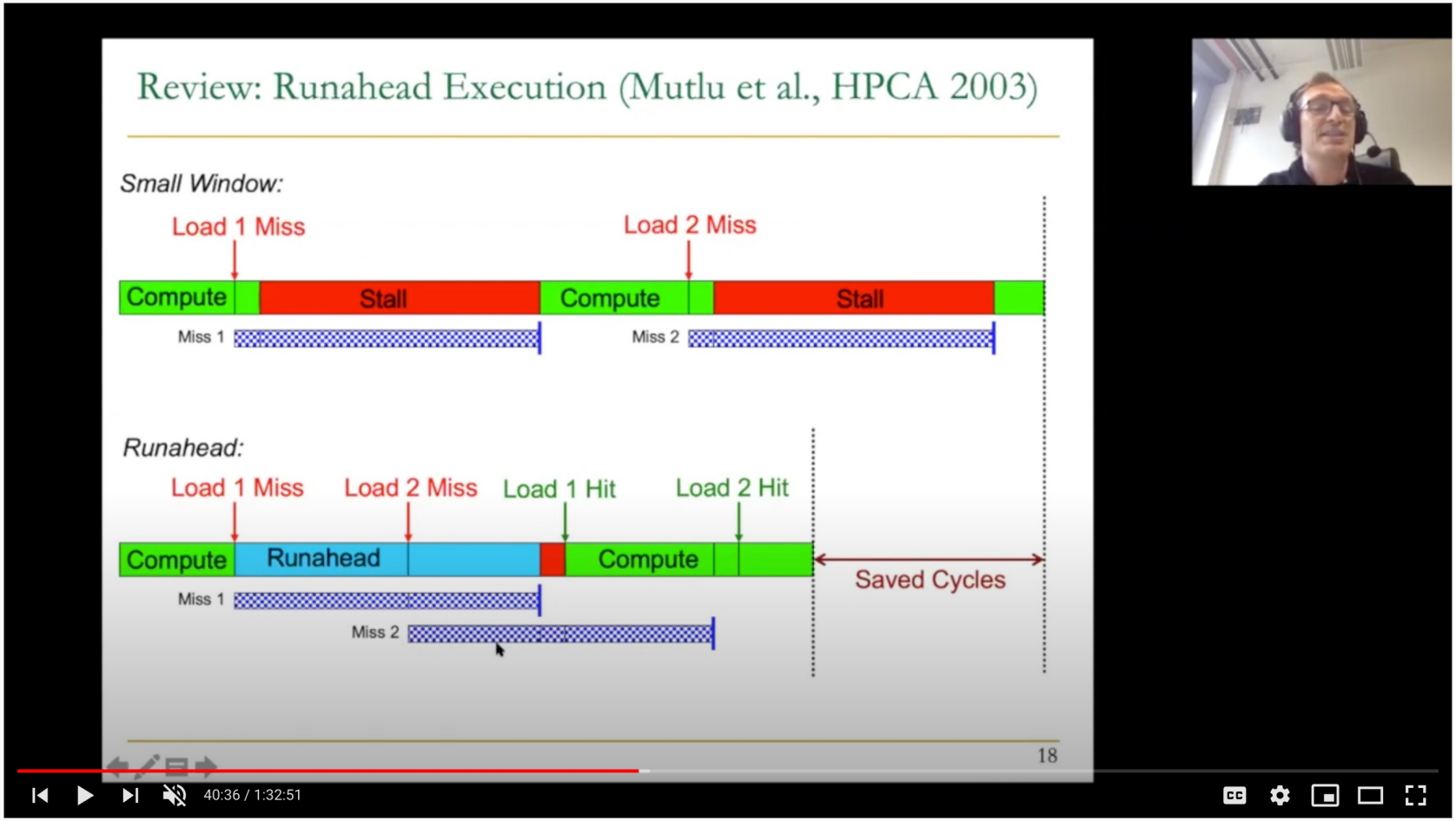
# Principle: Learning and Scholarship

The quality of your work defines your impact

# More on Runahead Execution

- Lecture video from Fall 2020, Computer Architecture:
  - 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

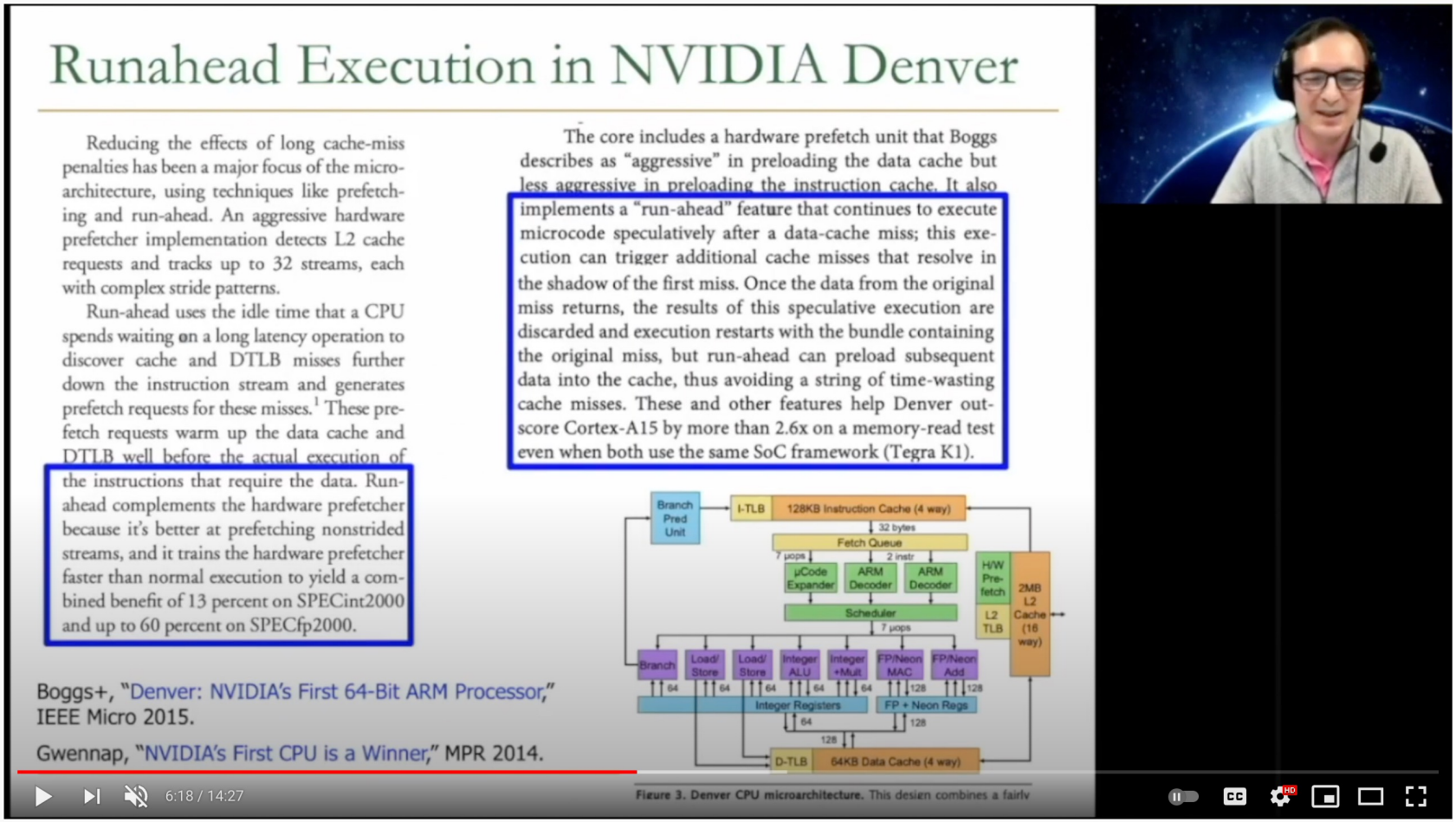
14 0 SHARE SAVE

Onur Mutlu Lectures
16.5K subscribers

ANALYTICS EDIT VIDEO

https://www.youtube.com/watch?v=zPewo6IaJ_8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=34

# More on Runahead Execution (II)




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

1,162 views • Premiered Mar 6, 2021

50 0 SHARE SAVE

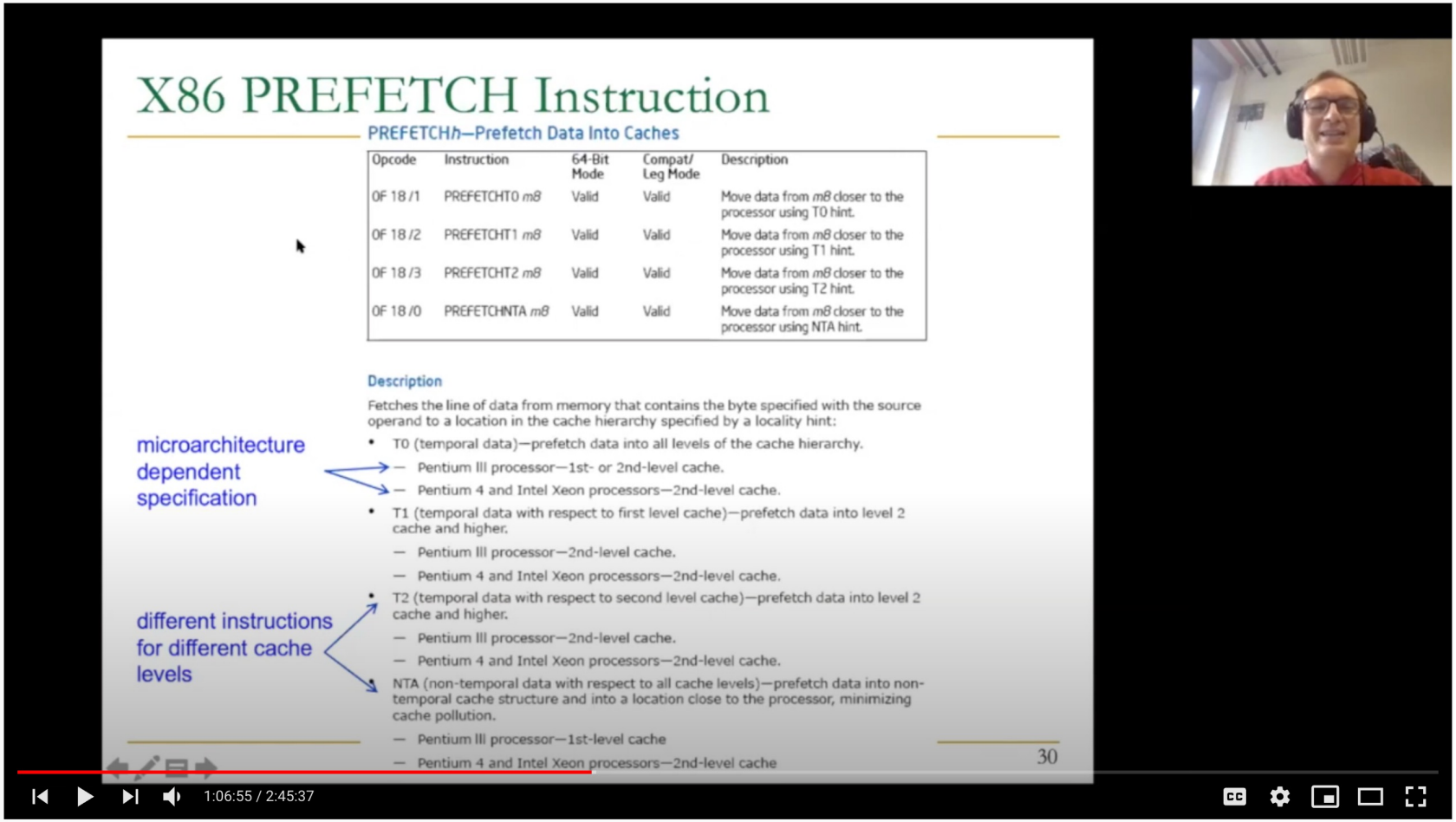
Onur Mutlu Lectures
16.5K subscribers

ANALYTICS EDIT VIDEO

https://www.youtube.com/watch?v=KFCOecRQTIc

# More Recommended Material on Prefetching

# Lectures on Prefetching (I)




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

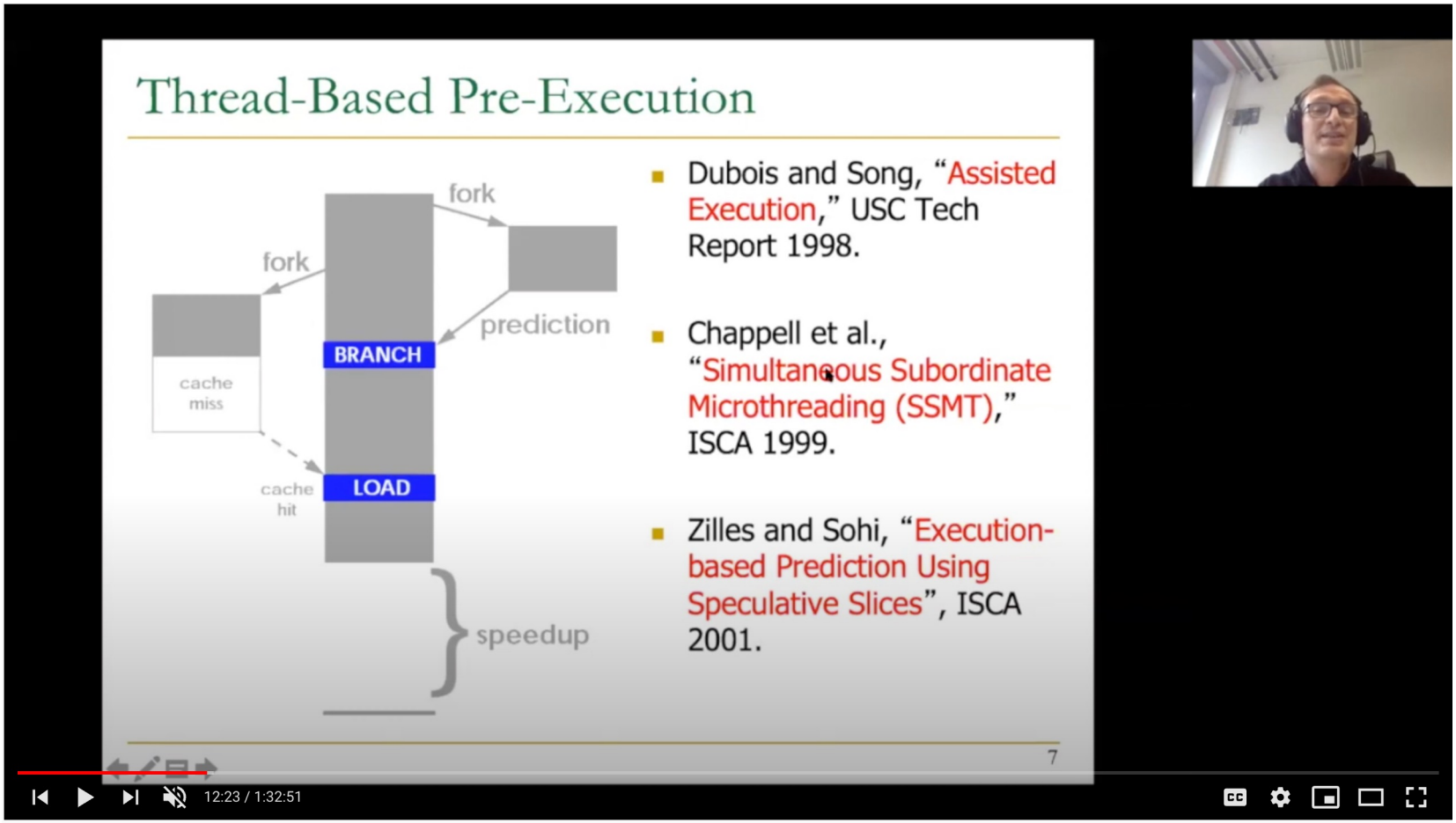
1,203 views • Nov 29, 2020 26 0 SHARE SAVE

Onur Mutlu Lectures
16.5K subscribers

ANALYTICS EDIT VIDEO

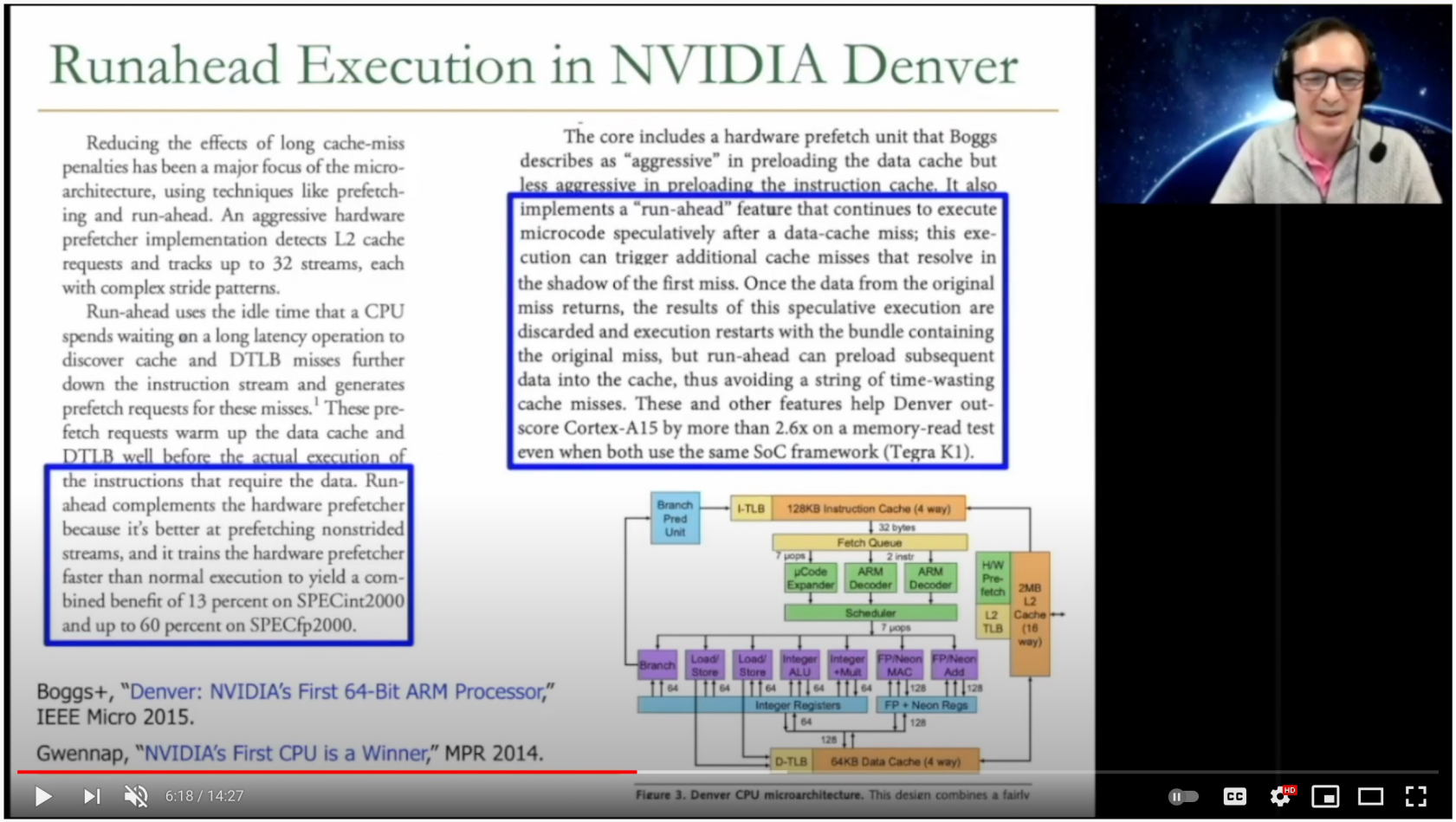
https://www.youtube.com/watch?v=xZmDyj0g3Pw&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=33

# Lectures on Prefetching (II)




https://www.youtube.com/watch?v=zPewo6IaJ_8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=34

# Lectures on Prefetching (III)




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

1,162 views • Premiered Mar 6, 2021

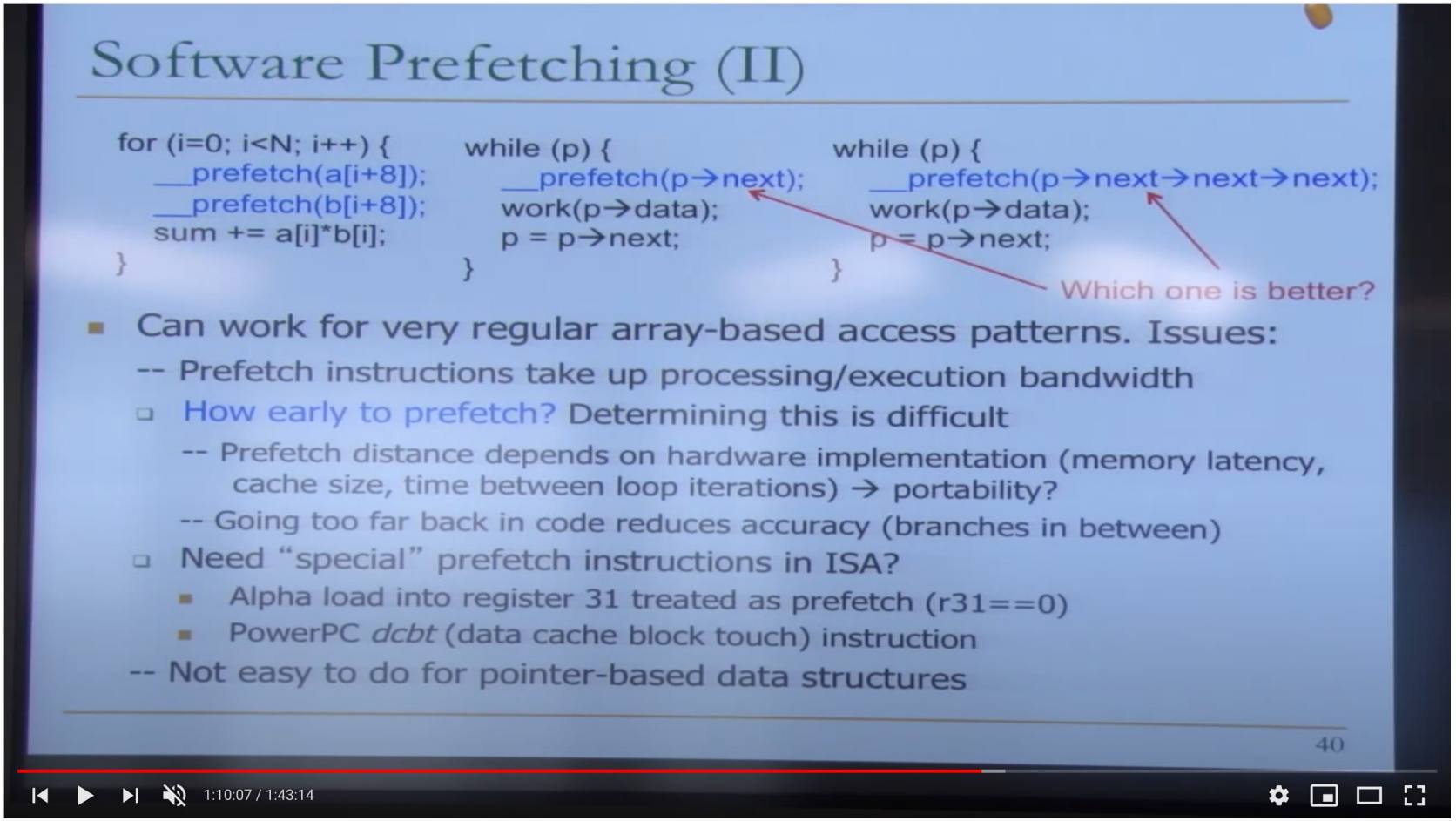
50 0 SHARE SAVE ...

Onur Mutlu Lectures
16.5K subscribers

ANALYTICS EDIT VIDEO

https://www.youtube.com/watch?v=KFCOecRQTIc

# Lectures on Prefetching (IV)



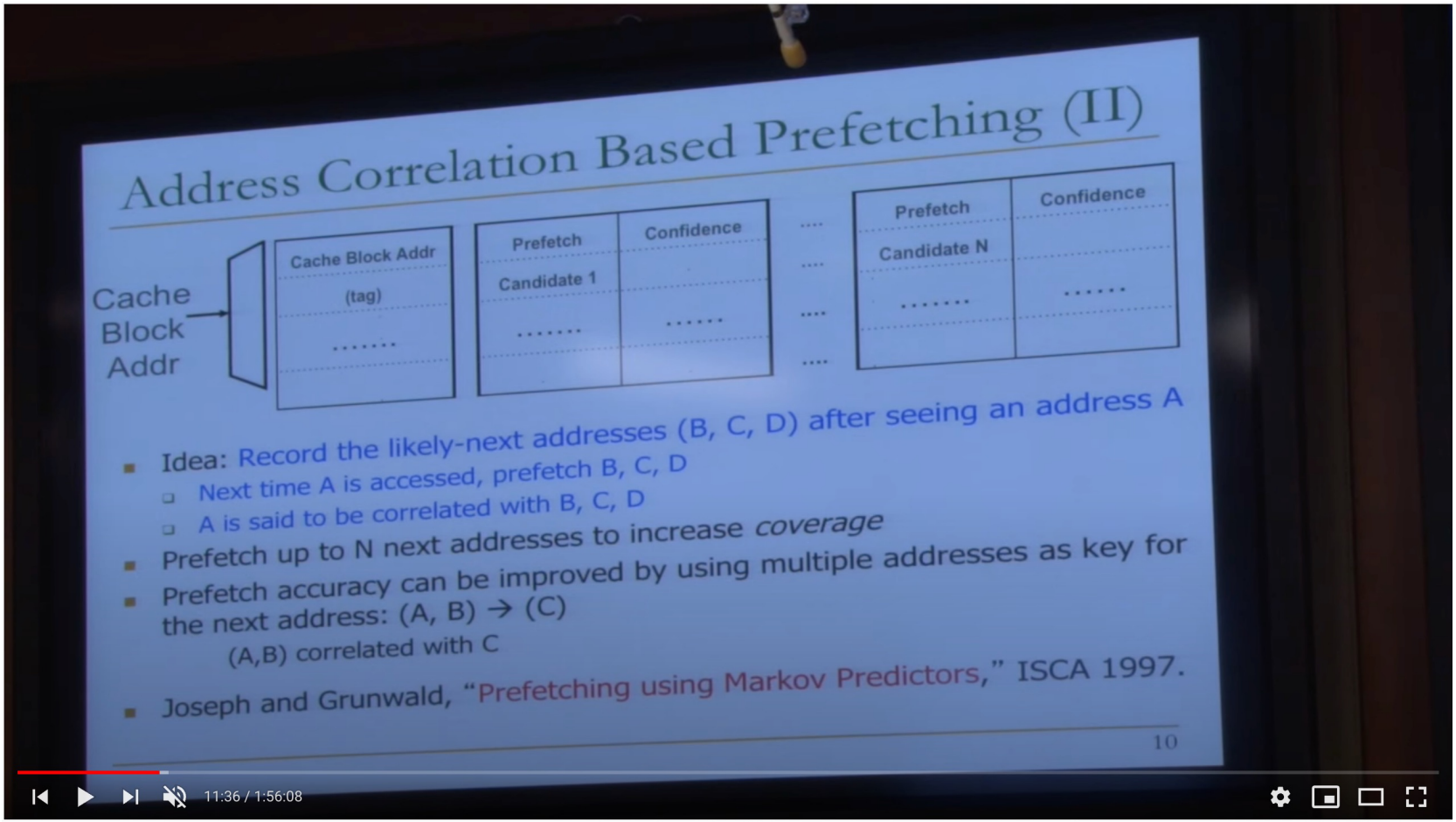

Lecture 25: Prefetching - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

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=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=29

# Lectures on Prefetching (V)




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

https://www.youtube.com/watch?v=TUFins4z6o4&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=30

# 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
- 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 14: Prefetching II

Rahul Bera

Prof. Onur Mutlu

ETH Zürich

Fall 2023

Nov 10 2023

# Backup Slides:
# More on Runahead Execution

# Recommended Readings on Prefetching

- 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

# Recommended Readings on Prefetching

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt, **"Runahead Execution: An Effective Alternative to Large Instruction Windows"** *IEEE Micro, Special Issue: Micro's Top Picks from Microarchitecture Conferences* (***MICRO TOP PICKS***), Vol. 23, No. 6, pages 20-25, November/December 2003.

# Recommended Readings on Prefetching

- Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt, **"Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers"**
  *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (**HPCA**)*, pages 63-74, Phoenix, AZ, February 2007. Slides (ppt)
  ***One of the five papers nominated for the Best Paper Award by the Program Committee.***

**Feedback Directed Prefetching:**
**Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers**

Santhosh Srinath†‡ Onur Mutlu§ Hyesoon Kim‡ Yale N. Patt‡

†Microsoft
ssri@microsoft.com

§Microsoft Research
onur@microsoft.com

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

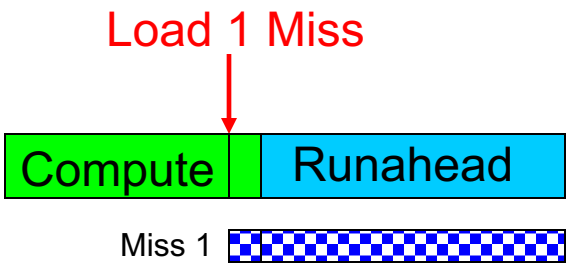
# Recommended Readings on Prefetching

- 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.

# 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

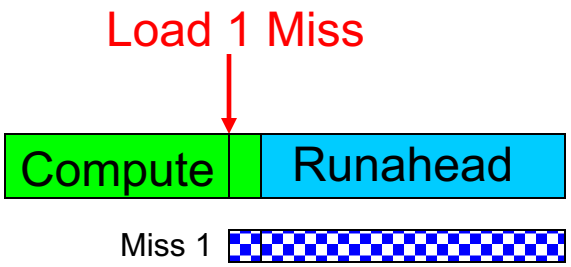
# Instruction Processing in Runahead Mode




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

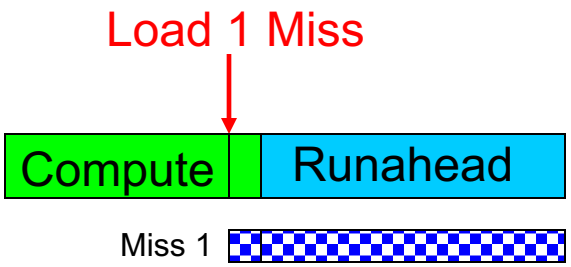
- It is purely speculative: Architectural (software-visible) register/memory state is NOT updated in runahead mode.
- L2-miss dependent instructions are identified and treated specially.
  - They are quickly removed from the instruction window.
  - Their results are not trusted.

# L2-Miss Dependent Instructions



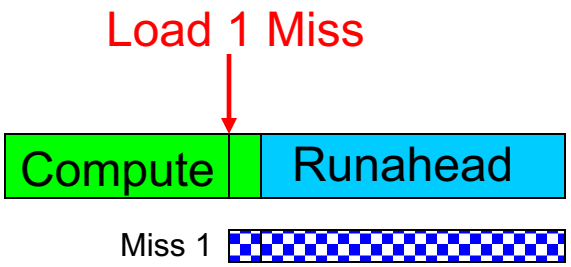

- Two types of results produced: INV and VALID
- INV = Dependent on an L2 miss
- INV results are marked using INV bits in the register file and store buffer.
- INV values are not used for prefetching/branch resolution.

# Removal of Instructions from Window



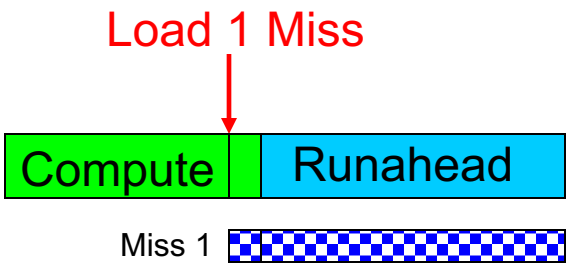

- Oldest instruction is examined for pseudo-retirement
  - An INV instruction is removed from window immediately.
  - A VALID instruction is removed when it completes execution.
- Pseudo-retired instructions free their allocated resources.
  - This allows the processing of later instructions.
- Pseudo-retired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

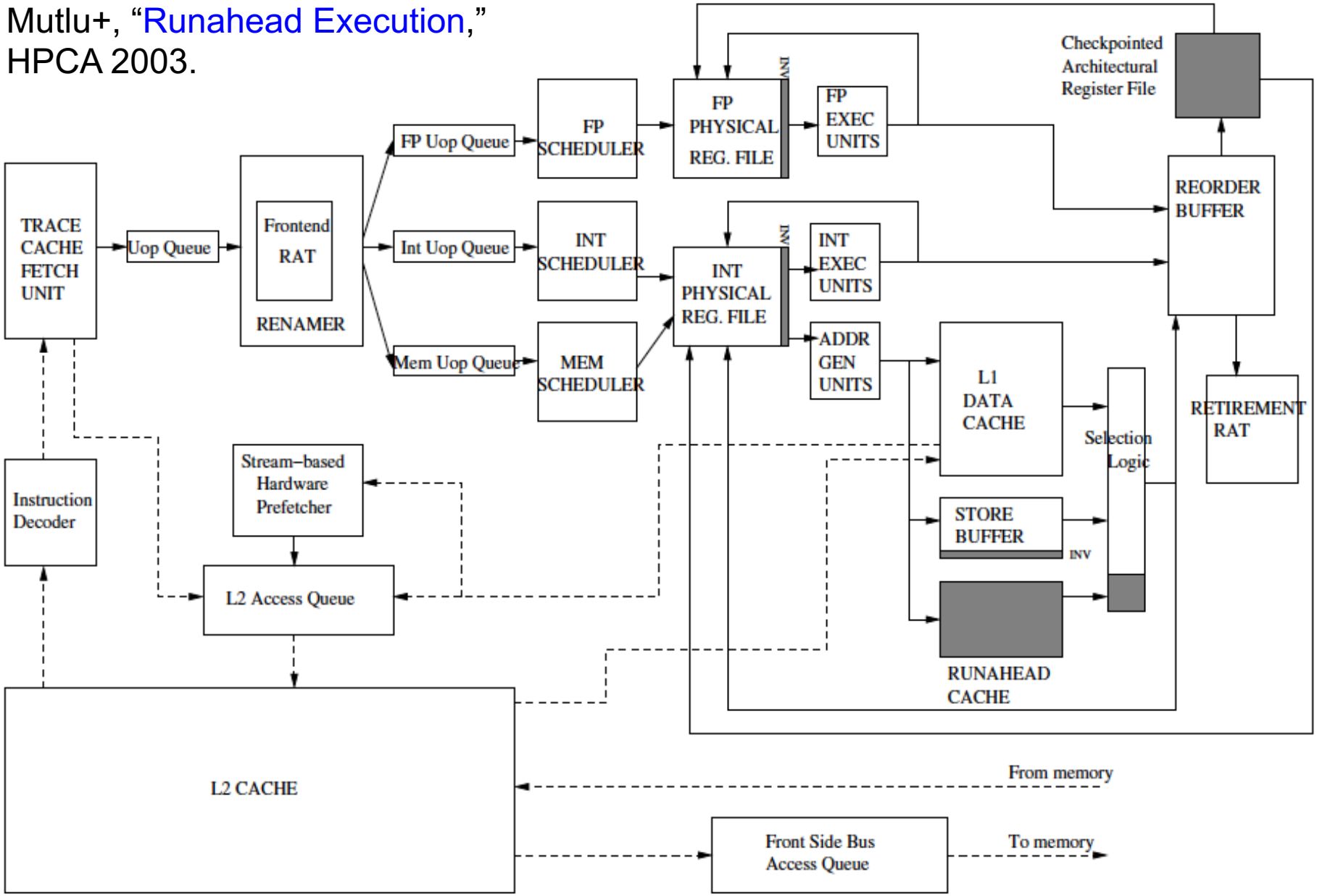
# Branch Handling in Runahead Mode




- 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

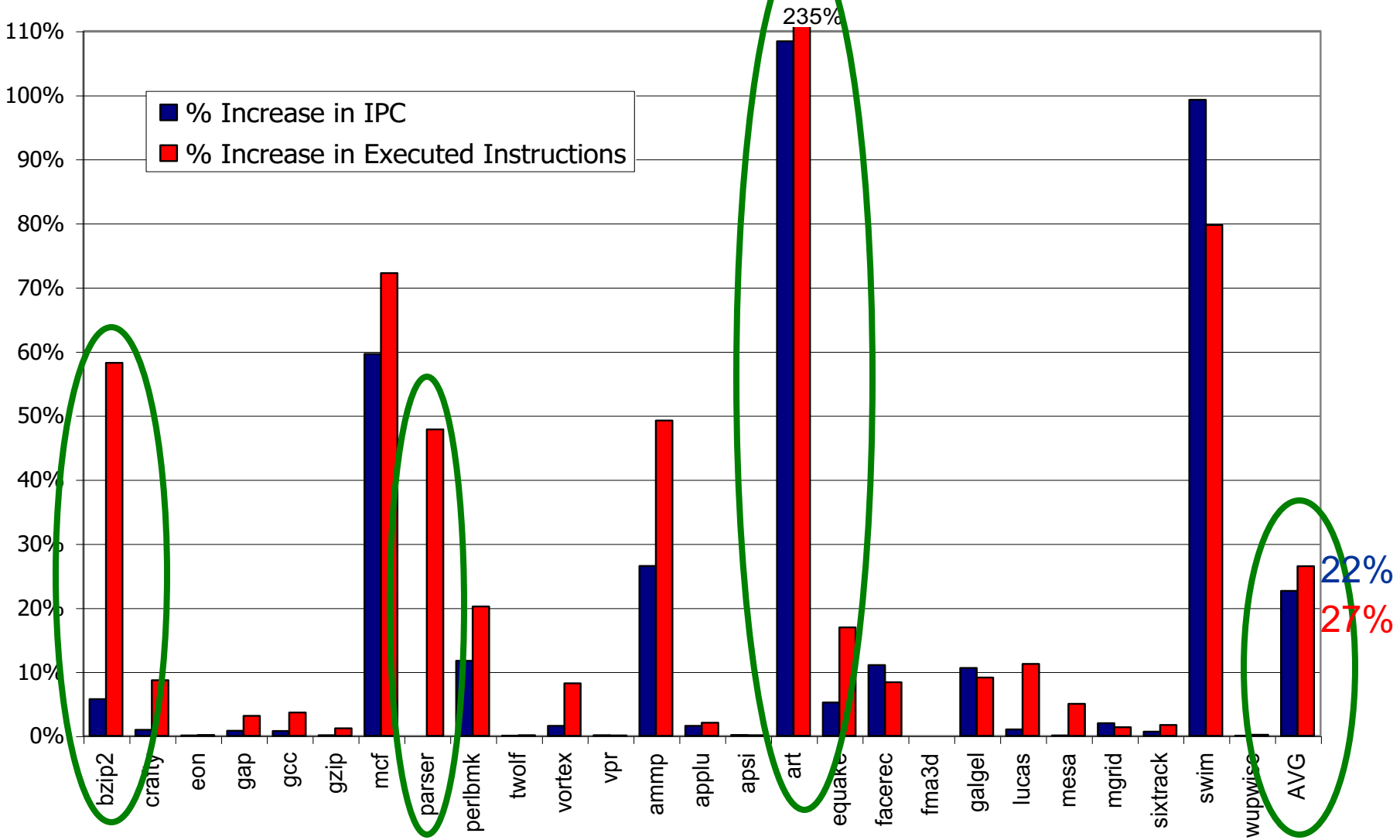
Mutlu+, "Runahead Execution," HPCA 2003.



# 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]

# 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

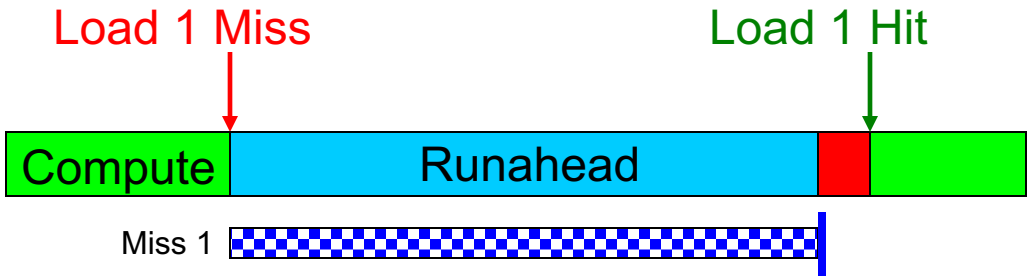
# Overlapping Runahead Periods

- Two runahead periods that execute the same instructions


- Second period is inefficient

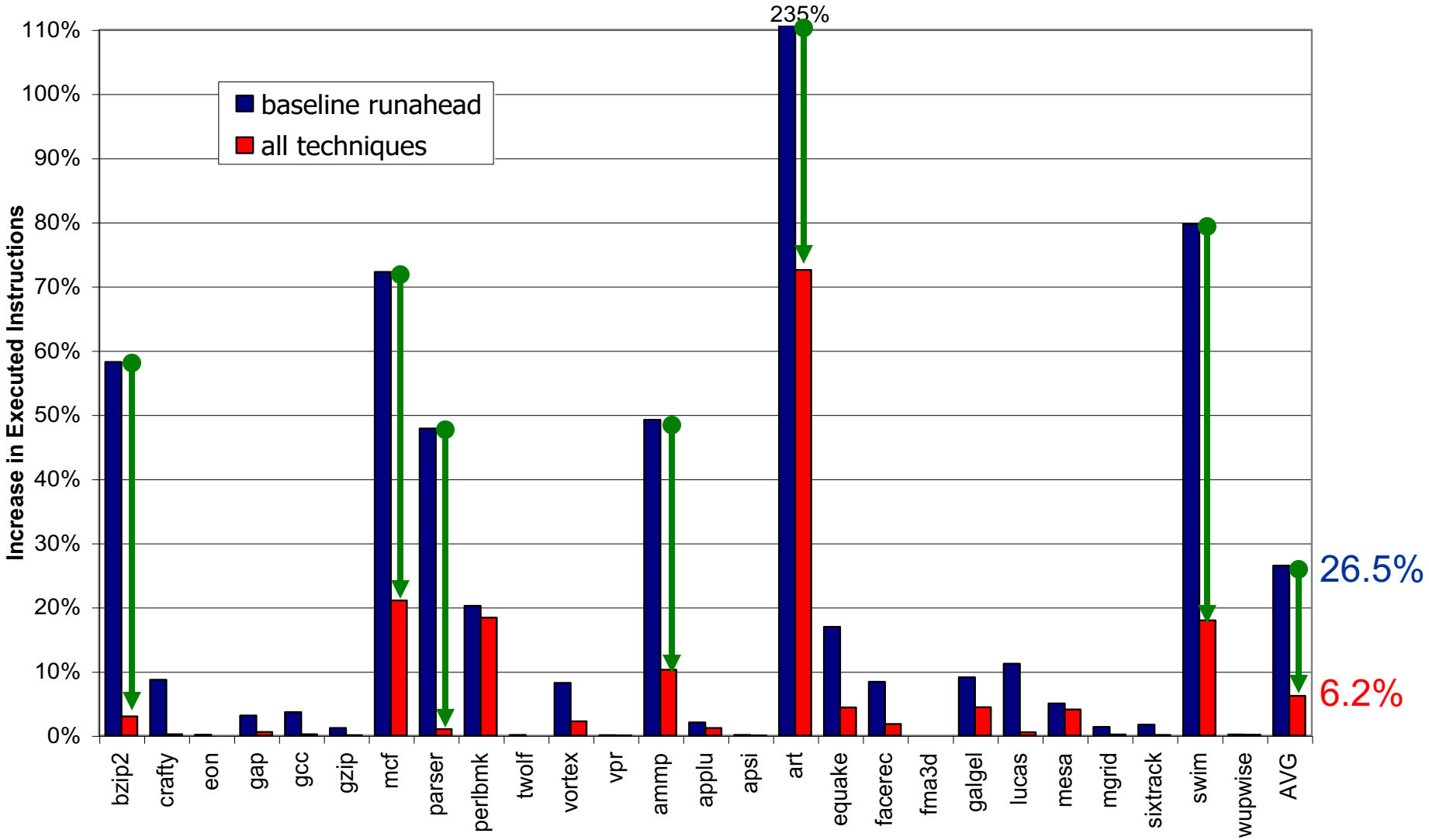
# Useless Runahead Periods

- Periods that do not result in prefetches for normal mode

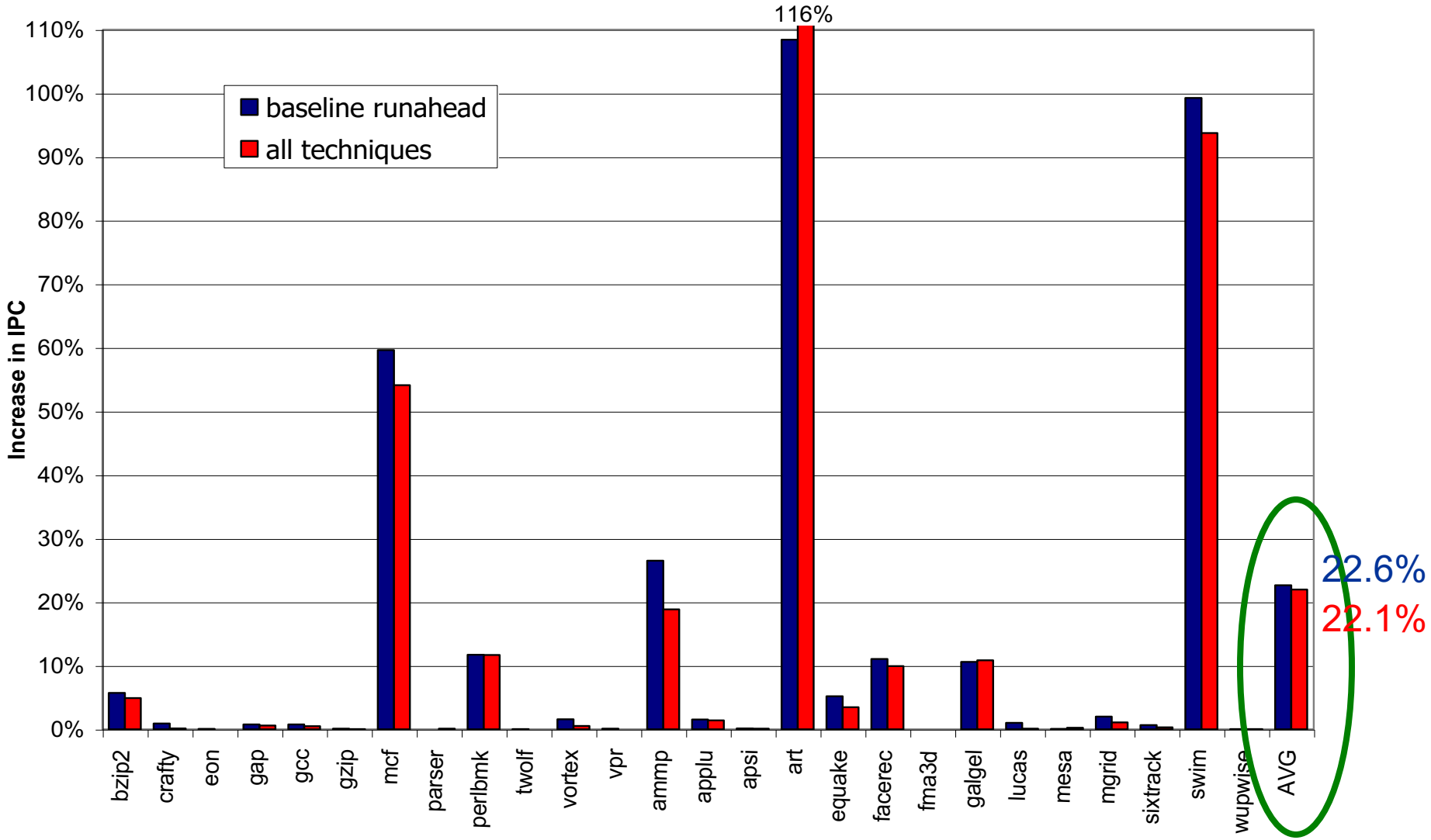



- They exist due to the lack of memory-level parallelism
- Mechanism to eliminate useless periods:
  - Predict if a period will generate useful L2 misses
  - Estimate a period to be useful if it generated an L2 miss that cannot be captured by the instruction window
    - Useless period predictors are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

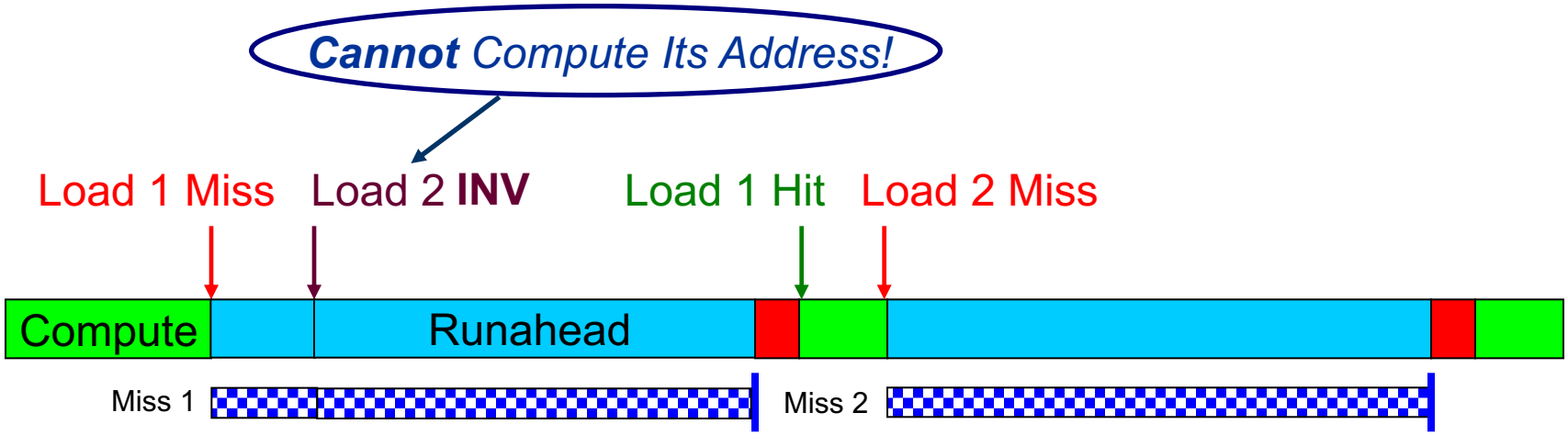
- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
  **"Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance"**
  *IEEE Micro, Special Issue: Micro's Top Picks from Microarchitecture Conferences* (***MICRO TOP PICKS***), Vol. 26, No. 1, pages 10-20, January/February 2006.

# 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]

# 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

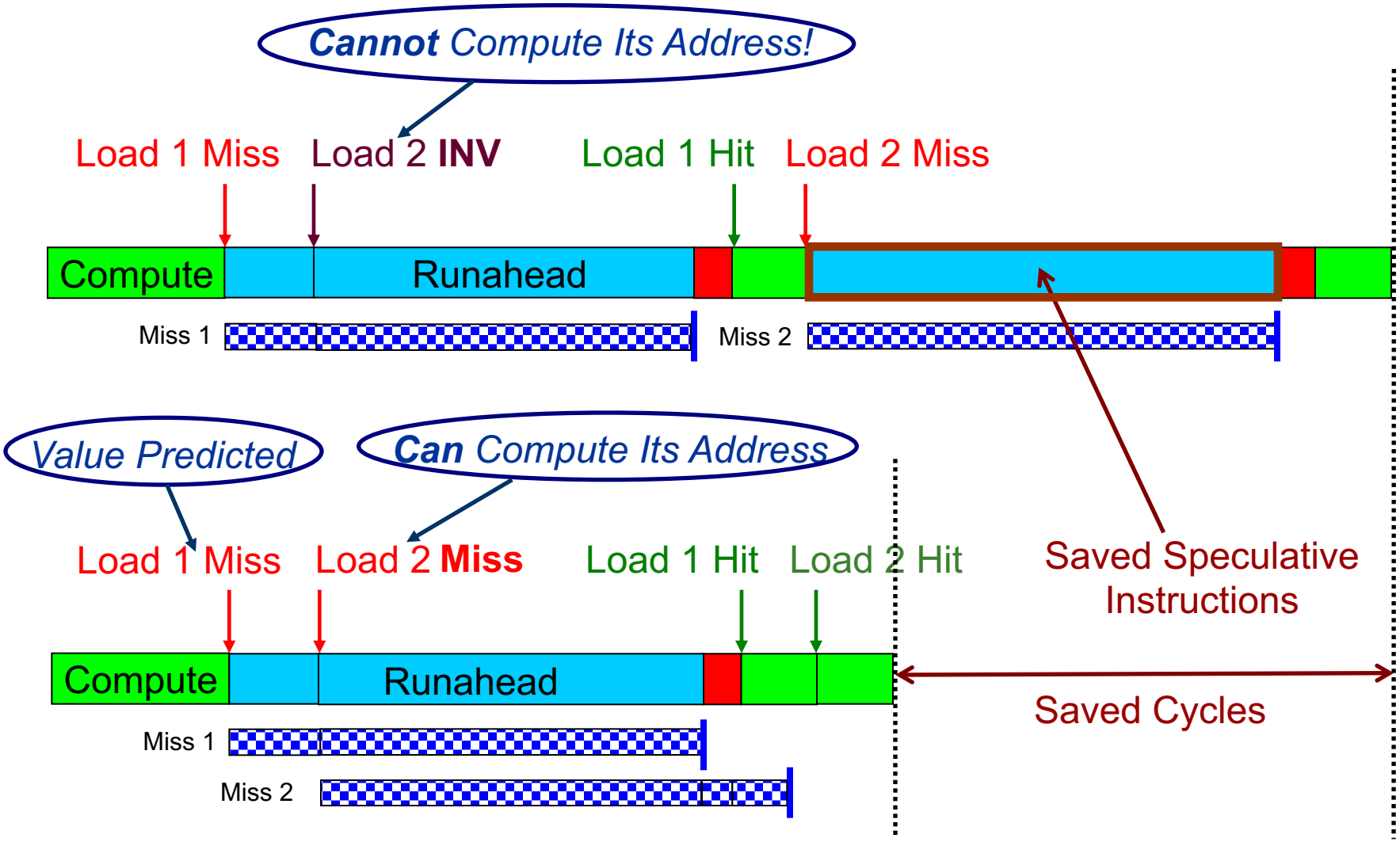
# Parallelizing Dependent Cache Misses

- 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.***

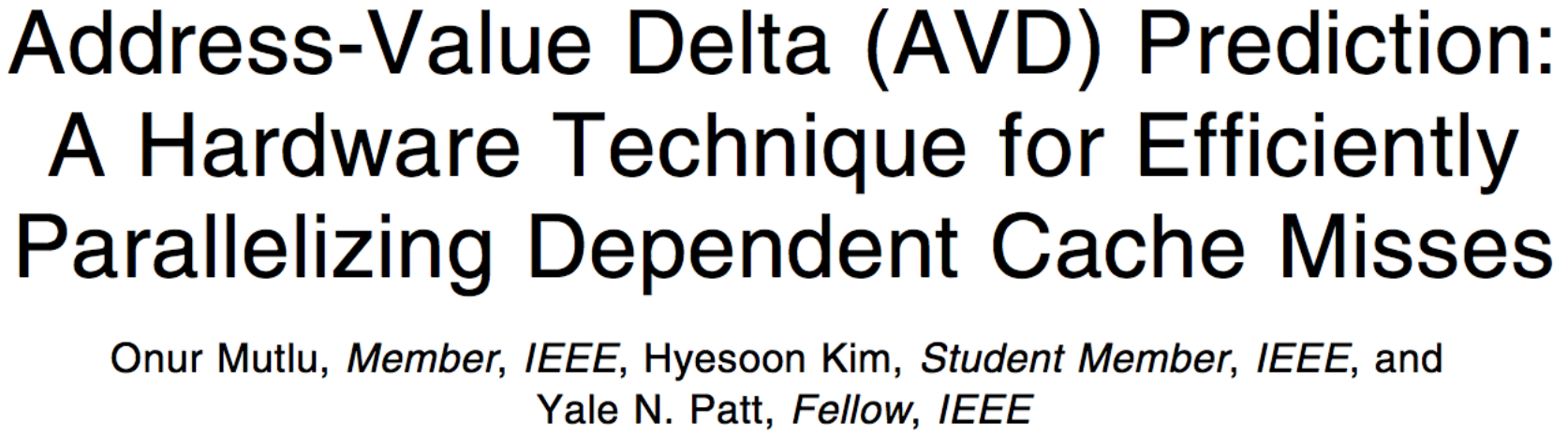
**Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns**

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

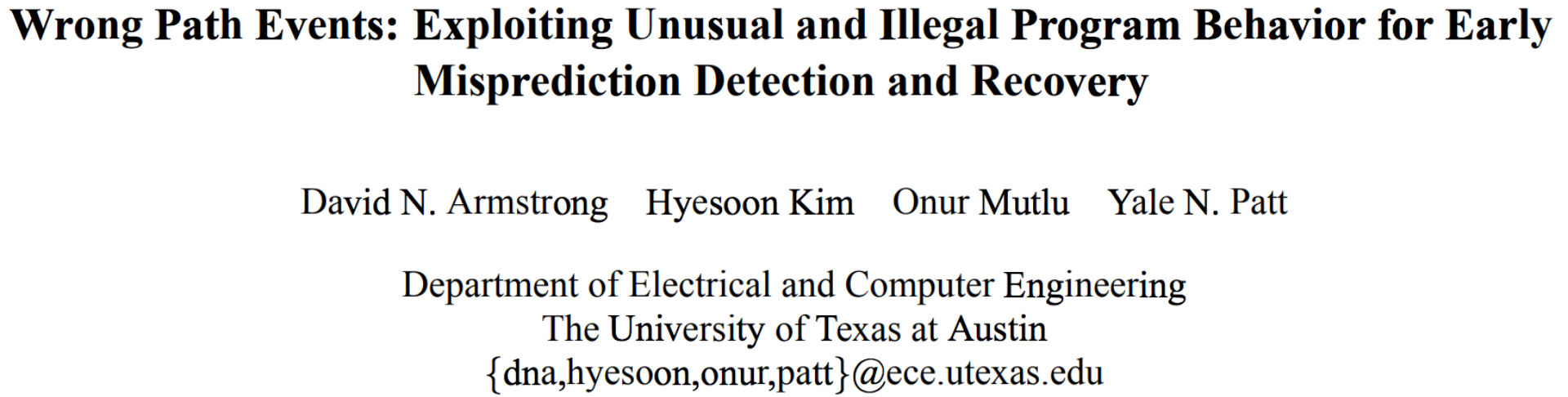
Onur Mutlu, *Member*, *IEEE*, Hyesoon Kim, *Student Member*, *IEEE*, and Yale N. Patt, *Fellow*, *IEEE*

# 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]

# 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"** *Proceeedings 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

# Effects of 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

# Effects of 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

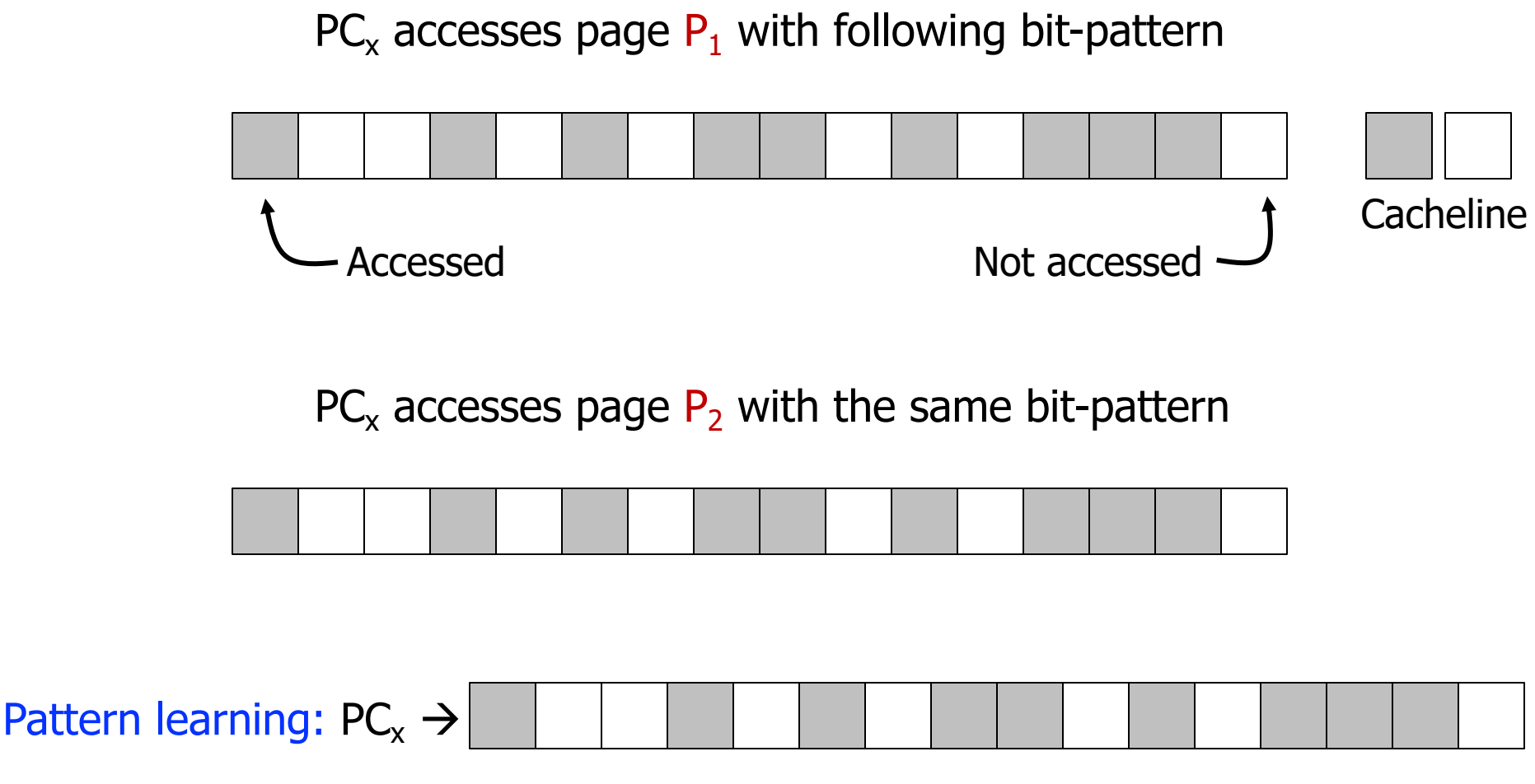
Onur Mutlu, *Student Member*, *IEEE*, Hyesoon Kim, *Student Member*, *IEEE*,
David N. Armstrong, and Yale N. Patt, *Fellow*, *IEEE*

# Another Example Prefetcher: Spatial Memory Streaming

# 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)



# 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

# More on Spatial Prefetching

- Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney, **"DSPatch: Dual Spatial Pattern Prefetcher"**
  *Proceedings of the 52nd International Symposium on Microarchitecture (**MICRO**)*, Columbus, OH, USA, October 2019.
  [Slides (pptx) (pdf)]
  [Lightning Talk Slides (pptx) (pdf)]
  [Poster (pptx) (pdf)]
  [Lightning Talk Video (90 seconds)]

**DSPatch: Dual Spatial Pattern Prefetcher**

Rahul Bera[1] Anant V. Nori[1] Onur Mutlu[2] Sreenivas Subramoney[1]

[1]Processor Architecture Research Lab, Intel Labs [2]ETH Zürich