Mitigating the Memory Bottleneck with Machine Learning-Driven and Data-Aware Microarchitectural Techniques

Rahul Bera

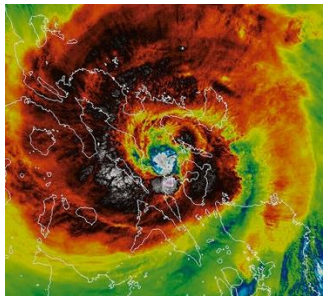
Doctoral Examination
01.10.2025

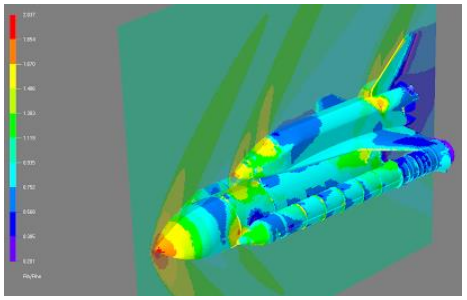
Advisor:
Onur Mutlu (ETH Zurich)

Co-Examiners:
Aamer Jaleel (Nvidia Research)
Boris Grot (University of Edinburgh)
Chris Wilkerson (Intel)
Daniel Jiménez (Texas A&M University)
Pradip Bose (IBM T. J. Watson Research Center)

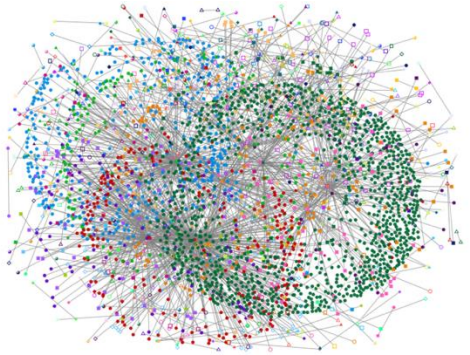
SAFARI ETH zürich

Modern Workloads Exhibit Massive Data Footprints





High-performance computing



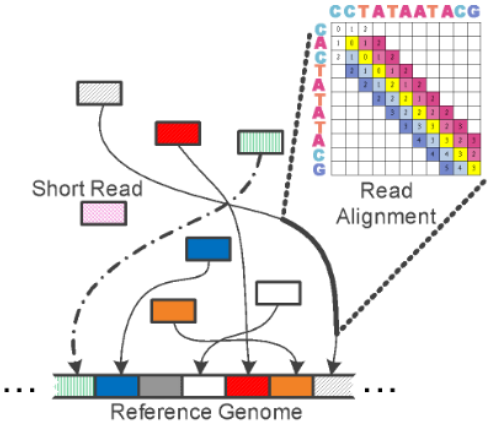
Graph analytics

Llama 2

Whisper

BioGPT

Large-scale artificial intelligence



Genome analysis

Modern Workloads Exhibit Massive Data Footprints

Massive data footprints
overwhelm state-of-the-art memory systems

Memory is *(and likely will be)*
the key performance & energy bottleneck


Large-scale artificial intelligence

Genome read mapping

Numerous Microarchitectural Techniques to Address the Memory Bottleneck

Cache management policies

Hit/miss prediction

Dead block prediction

Thread-based precomputation

Cache bypassing

Memory renaming

Runahead execution

Out-of-order execution

Load value prediction

Spatial prefetching

Memoization

Software prefetching

Coarse-grained multithreading

Temporal prefetching

Simultaneous multithreading

Load address speculation

Memory dependence prediction

Load address precomputation

Dead instruction elimination

Key Observation

Even though microarchitectural techniques observe a large amount of

Application data

(e.g., program counter value, memory addresses, memory values)

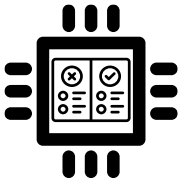
System data

(e.g., bandwidth usage, cache utilization and pollution)

Decisions they make online are often agnostic to the data they observe

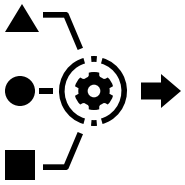
Key Observation

Decisions they make online are often agnostic to the data they observe



Limited adaptation to observed data

Make decisions based on **rigid, and often myopic, human-crafted heuristics**, disregarding the large volume of data present at runtime



Insufficient exploitation of data characteristics

Do not fully exploit or often disregard **diverse properties of application data** (e.g., criticality, value locality, compressibility)

Data-agnostic decision making limits effectiveness

Thesis Statement

By enabling microarchitecture to

1. Adapt its policies by **continuously learning** from application data and system-generated data

2. Tailor its decisions **by exploiting characteristics** of application data

We can significantly **improve performance and energy efficiency** of state-of-the-art processors

Our Approach

Exploit lightweight and practical
machine learning methods

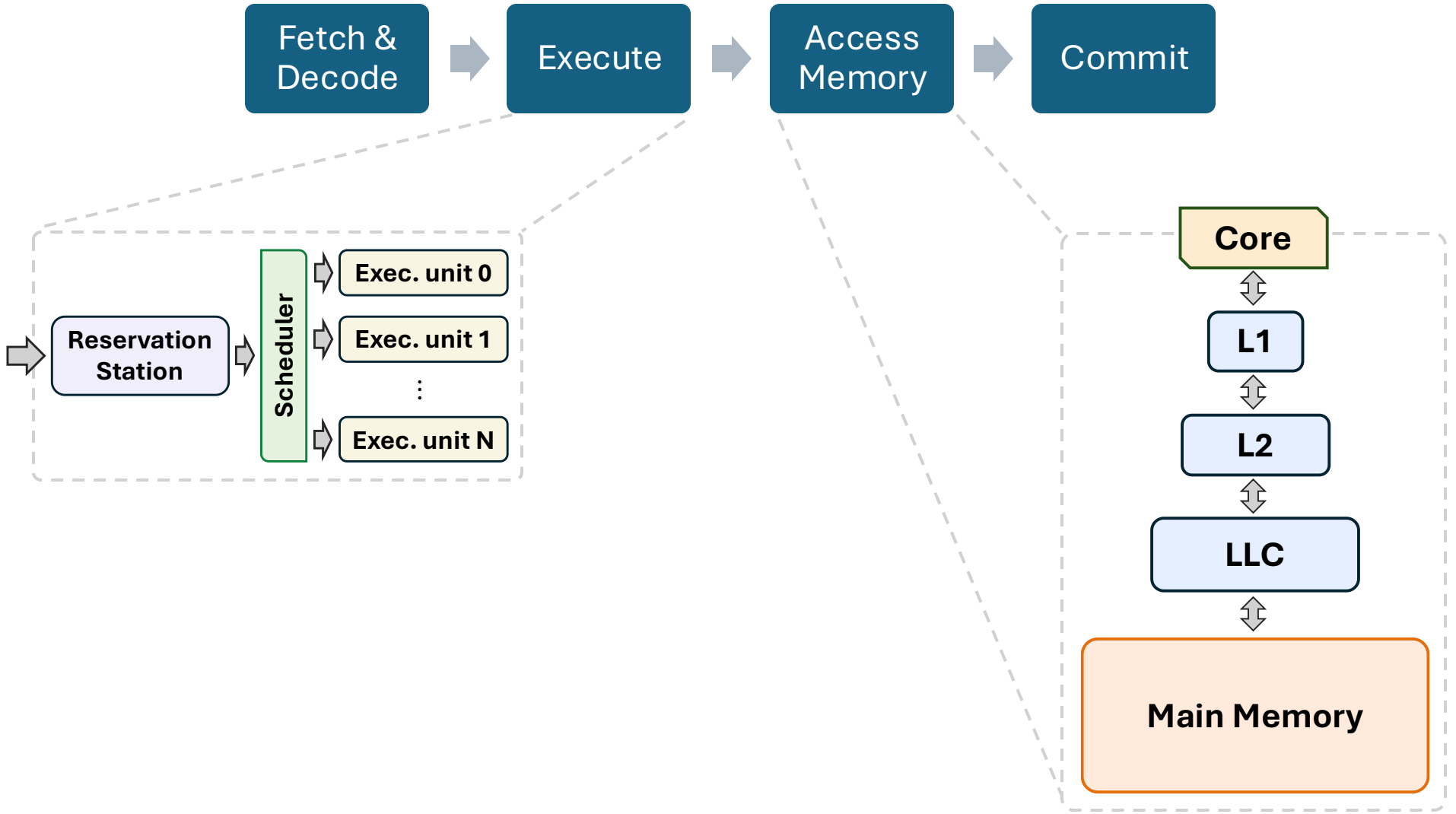
1

Adapt its policies by **continuously learning**
from application data and system-generated data

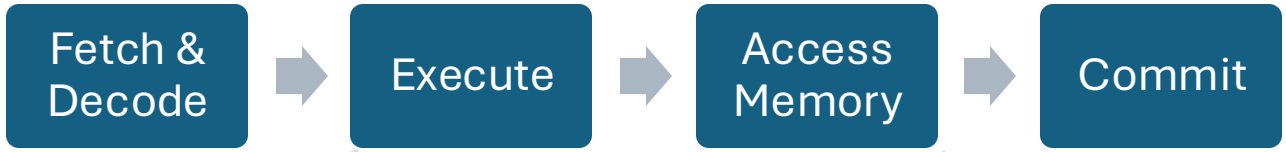
2

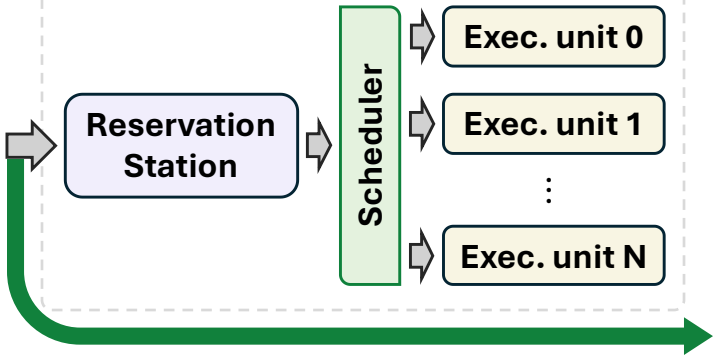
Tailor its decisions **by exploiting characteristics**
of the application data

Study and exploit
underexplored data characteristics


Fetch & Decode
Execute
Access Memory
Commit
Reservation Station
Scheduler
Exec. unit 0
Exec. unit 1
Exec. unit N
Core
L1
L2
LLC
Main Memory

Key Contributions





Constable [ISCA'24] 4

Safely eliminating load instruction execution by exploiting load address-value stability

2 **Hermes** [MICRO'22]

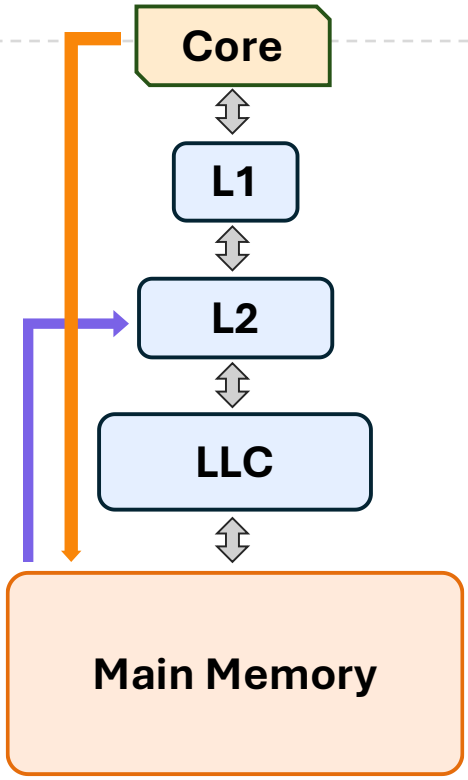
Off-chip prediction (OCP) using perceptron learning

3 **Athena** [under review]

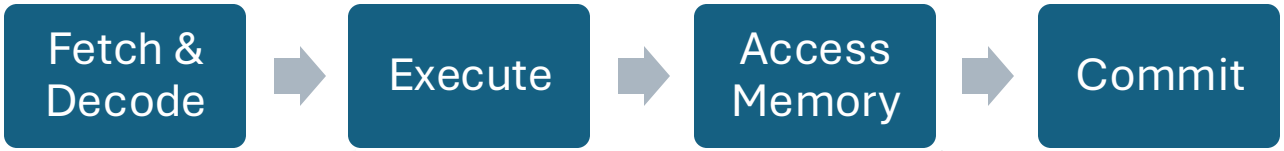
Prefetcher-OCP coordination using reinforcement learning

1 **Pythia** [MICRO'21]

Hardware data prefetching using reinforcement learning



Key Contributions



Constable [ISCA'24] 4

Safely eliminating load instruction execution by exploiting load address-value stability

2 Hermes [MICRO'22]

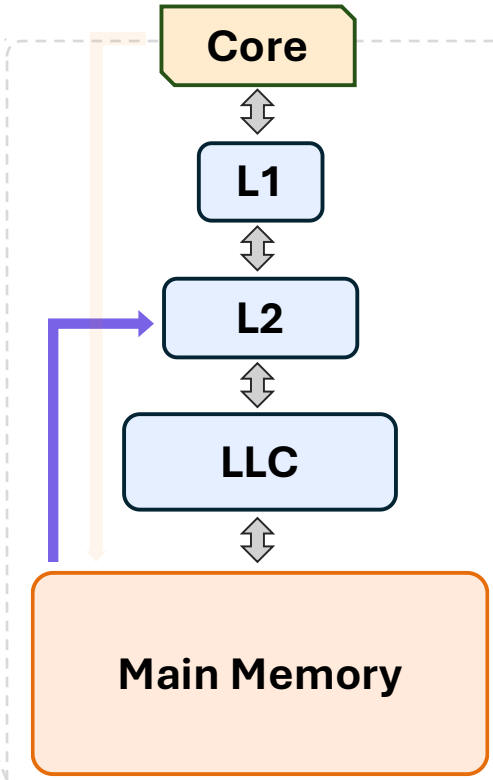
Off-chip prediction (OCP) using perceptron learning

3 Athena [under review]

Prefetcher-OCP coordination using reinforcement learning

1 Pythia [MICRO'21]

Hardware data prefetching using reinforcement learning



What is a Hardware Data Prefetcher?

- **Predicts address** of future memory requests and **fetches** their data **before the program needs them**

- Typically correlates access patterns in past memory requests with program feature

Program feature → access pattern

(e.g., load PC)

When the **program feature repeats**, the prefetcher anticipates the **repetition of the same access pattern** and predicts future addresses

Three Limitations in Prior Prefetchers

Reliance on a single, fixed program feature for prefetch prediction

Rigidity limits dynamic adaptation in workloads where the selected feature provides poor correlation with access pattern

Lack of inherent system awareness

Neglecting a prefetcher's undesirable effects on the system (e.g., bandwidth usage, cache pollution) often harms performance

Lack of in-silicon customizability

Rigid hardware structure provides no customizability to select new program features and/or prefetching objective at runtime

Our Goal

Design a **holistic prefetching framework** that can

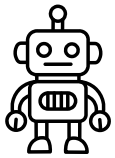
1. **Learn** to prefetch using **multiple program features** and **inherent system-level feedback** information

2. Be **easily customized in silicon** to change program feature and/or prefetching objective on-the-fly

PYTHIA

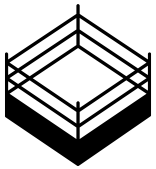
Key Idea: Formulate prefetching as **reinforcement learning**

RL agent



Pythia

Autonomously *learn what to prefetch*

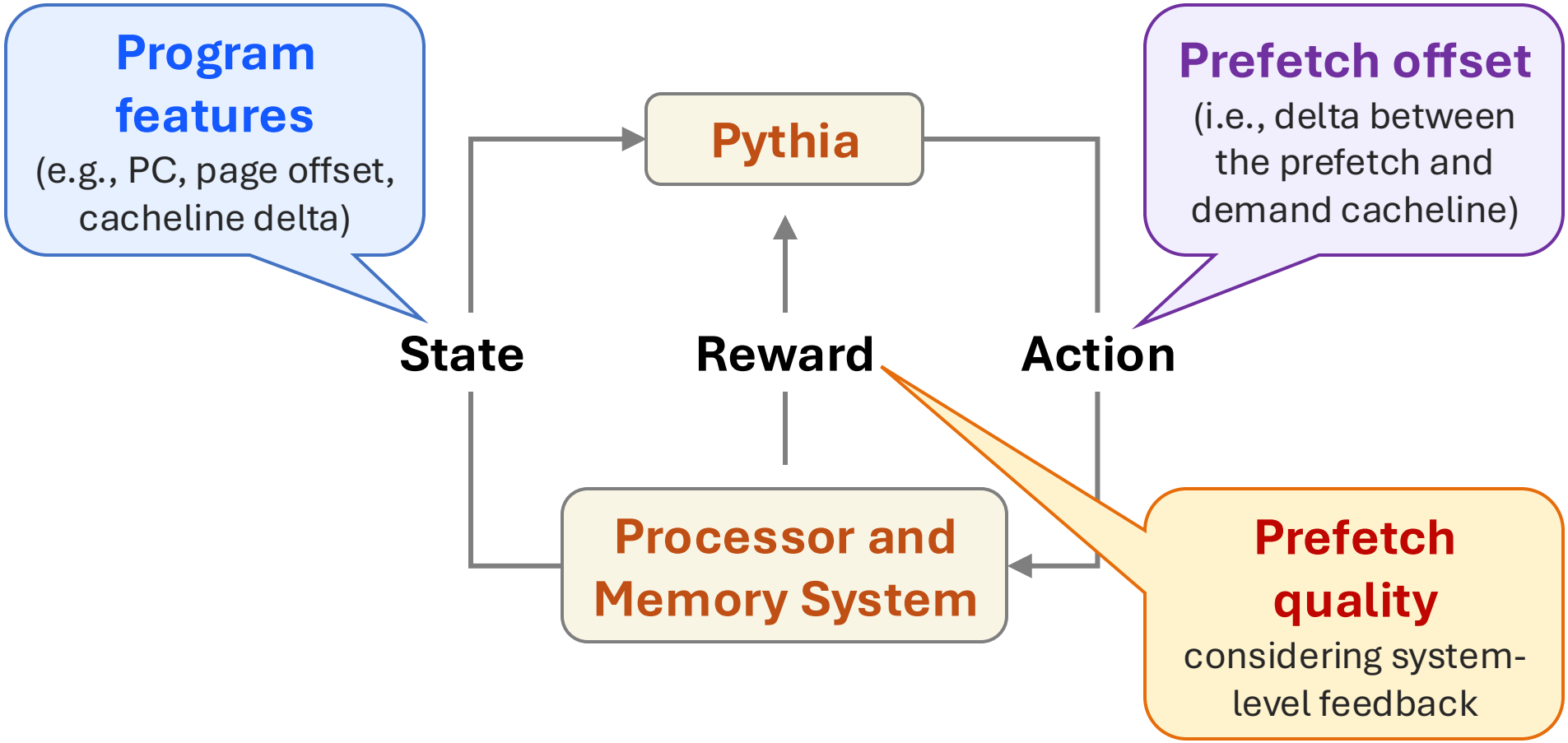


Processor and Memory System

Environment

PYTHIA

Key Idea: Formulate prefetching as **reinforcement learning**



Trace-driven simulation using **ChampSim**

Two-part evaluation

Hardware prototyping using **Chisel**

Trace-driven simulation using **ChampSim**

Wide range of prefetchers

- **SPP** [Kim+, MICRO'16]
- **Bingo** [Bakshalipour+, HPCA'19]
- **MLOP** [Shakerinava+, DPC3'19]
- **SPP+DSPatch** [Bera+, MICRO'19]
- **SPP+ PPF** [Bhatia+, ISCA'20]

Wide range of workloads

- **150 single-core** workloads from SPEC CPU 2006, 2017, PARSEC, and Ligra
- **270 multi-core** workload mixes

Wide range of system-configurations

- **#cores**: 1 – 12 cores
- **Main memory bandwidth**: 150 – 9600 MTPS
- **LLC size**: 256KB – 4 MB
- **Number of prefetchers** at different cache level

Pythia Public

Edit Pins | Unwatch 8 | Fork 47 | Starred 149

ma... | 4 Branches | 5 Tags | Go to file t | + | Code

rahulbera Merge pull request #19 from CM... 555cd59 · 6 months ago 50 Commits

branch	Initial commit for MICRO'21 artifact eval...	4 years ago
config	Initial commit for MICRO'21 artifact eval...	4 years ago
docs	Github pages documentation	4 years ago
experiments	Added chart visualization in Excel temp...	4 years ago
inc	Srand initilization with deterministic se...	3 years ago
patches	[17-cannot-compile-libbf] Adding a pat...	6 months ago

About

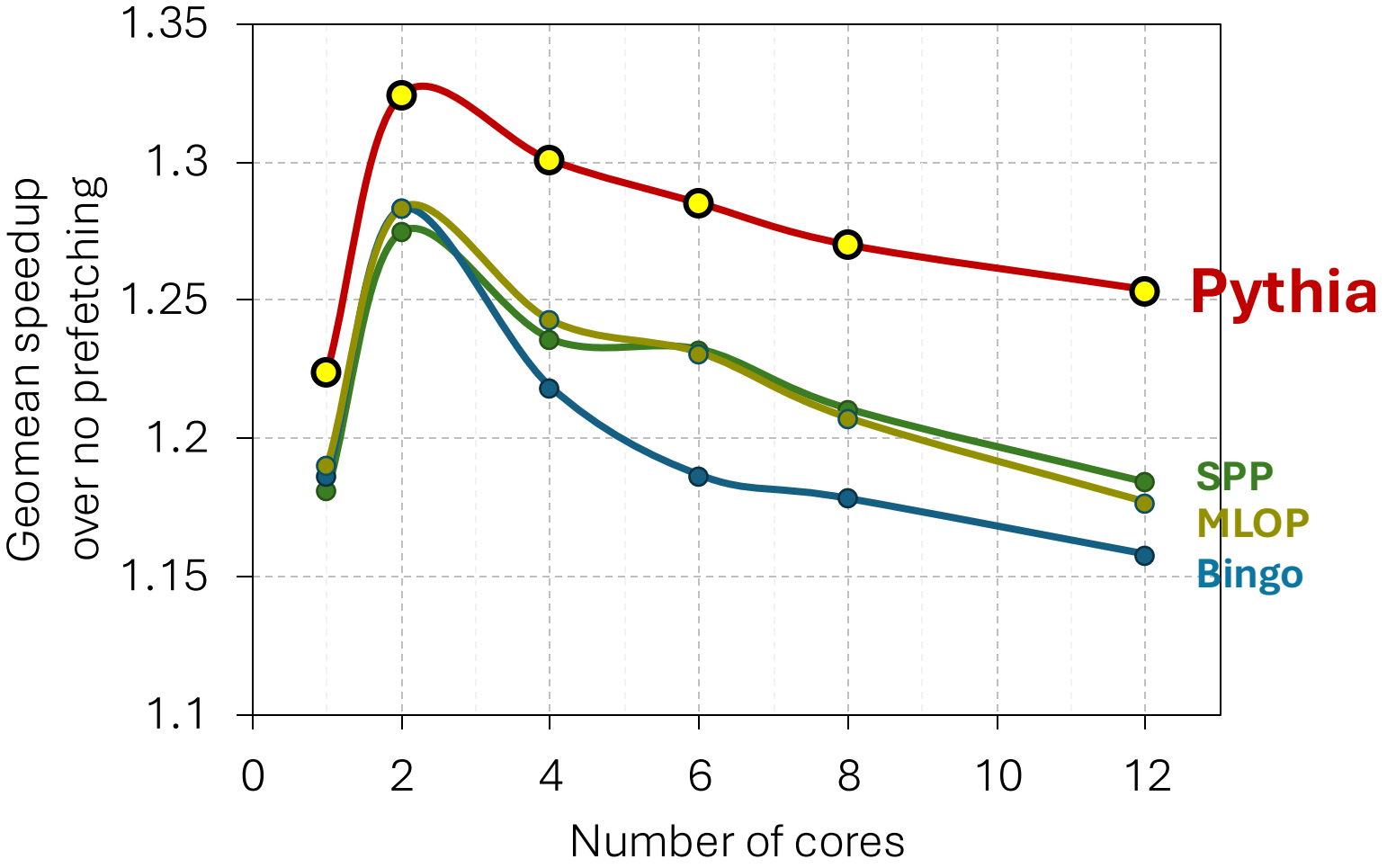
A customizable hardware prefetching framework using online reinforcement learning as described in the MICRO 2021 paper by Bera et al. (https://arxiv.org/pdf/2109.12021.pdf).

arxiv.org/pdf/2109.12021.pdf

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

Open-sourced and artifact-evaluated with all three ACM badges

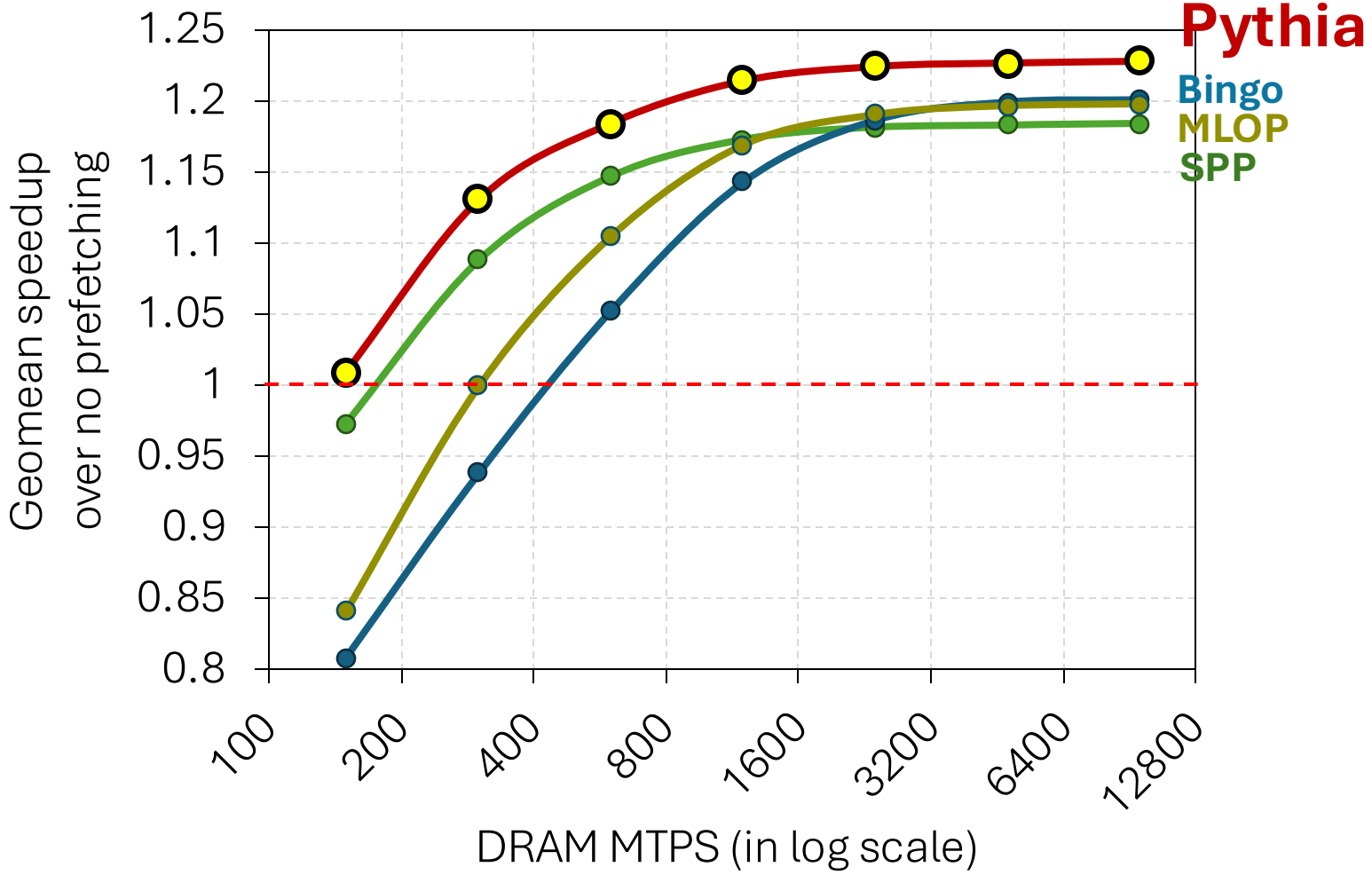
Key Results



Outperforms prior prefetchers

in wide range of configurations varying core count

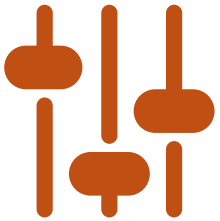
Key Results



Outperforms prior prefetchers

in wide range of memory bandwidth configurations

Key Results



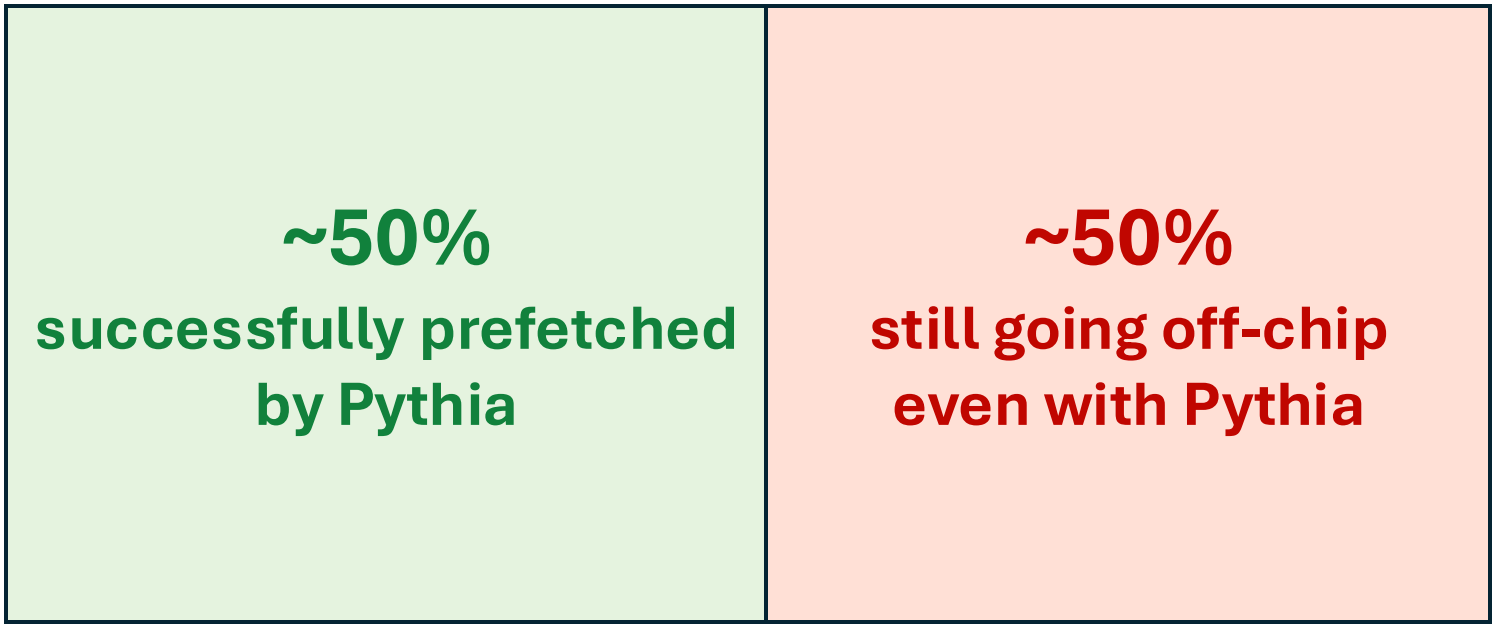
More performance gains with customization

Uto 7.8% perf. gain than baseline Pythia in Ligra by reward customization

Low-overhead, low-latency, and high-throughput predictions

1.03% area and 0.4% power overhead
Meets latency and throughput of an L2 prefetcher

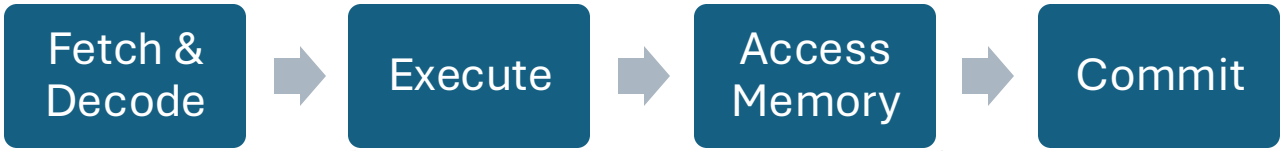
Does Pythia Solve the Memory Bottleneck?



demand loads going to off-chip without Pythia

What to do with these loads that are still going off-chip?

Key Contributions



Constable [ISCA'24] 4

Safely eliminating load instruction execution by exploiting load address-value stability

2 Hermes [MICRO'22]

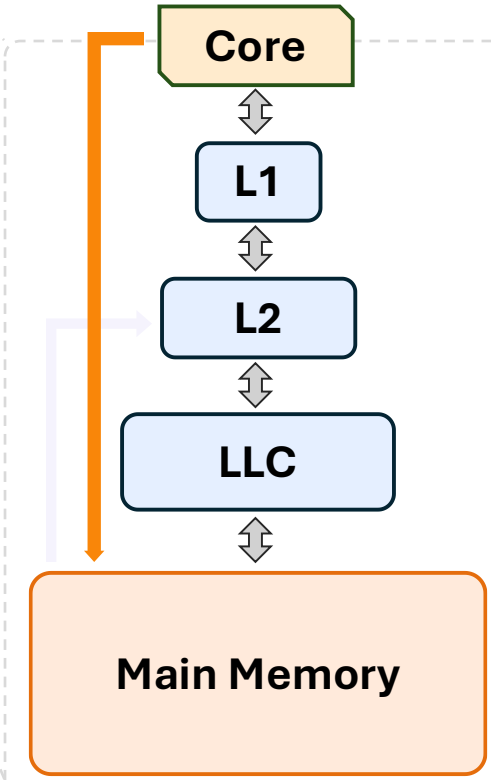
Off-chip prediction (OCP) using perceptron learning

3 Athena [under review]

Prefetcher-OCP coordination using reinforcement learning

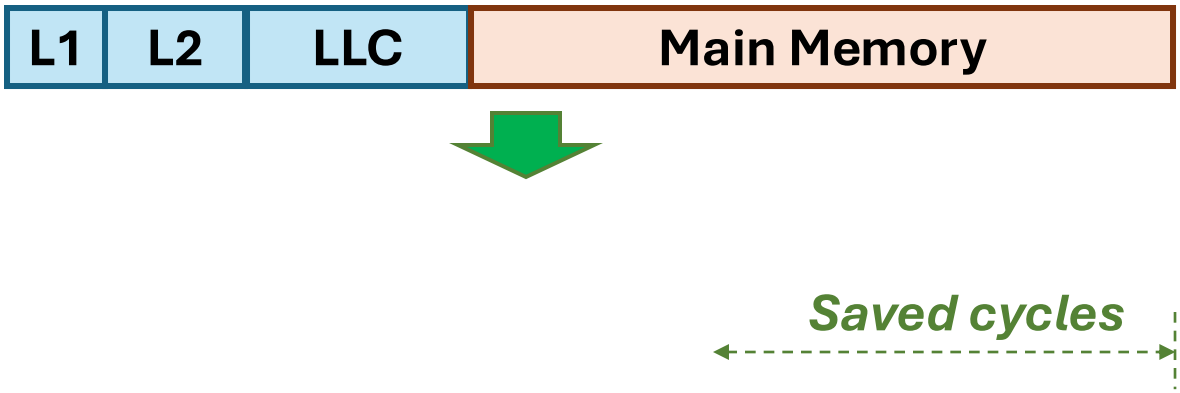
1 Pythia [MICRO'21]

Hardware data prefetching using reinforcement learning



Key Observation for Off-Chip Loads

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

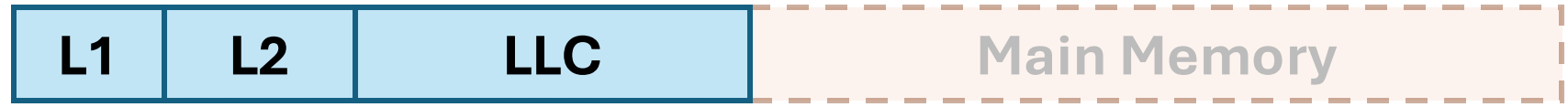


40% of the **stalls** can be eliminated by **removing on-chip cache access latency from critical path**

Our Goal

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

Predicts which loads are likely to go off-chip



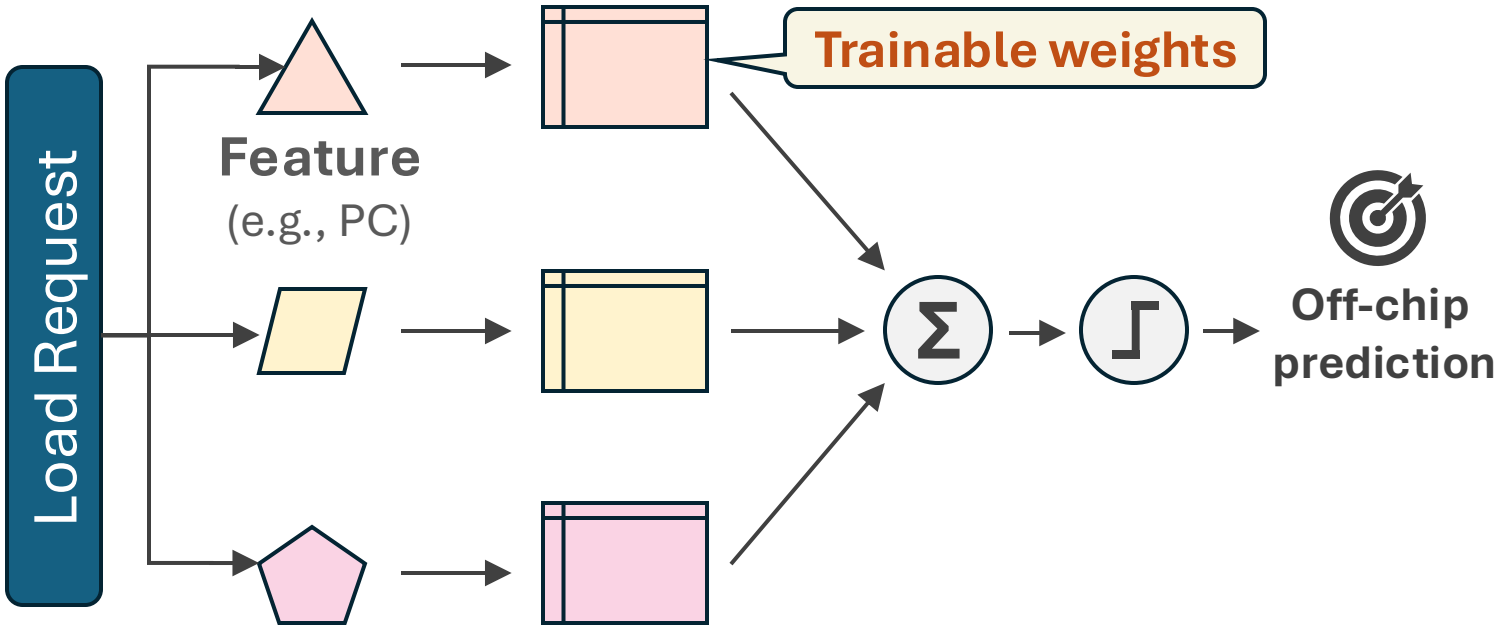
Starts **fetching** data **directly** from main memory

HERMES

Predicts which loads are likely to go off-chip

The first perceptron-based off-chip load predictor

Adaptively learns to identify off-chip loads from multiple program features



Trace-Driven Simulation Methodology

Prefetchers

- **Pythia** [Bera+, MICRO'21]
- **Bingo** [Bakshalipour+, HPCA'19]
- **MLOP** [Shakerinava+, DPC3'19]
- **SPP+ PPF** [Bhatia+, ISCA'20]
- **SMS** [Somogyi+, ISCA'06]

Off-chip Predictors

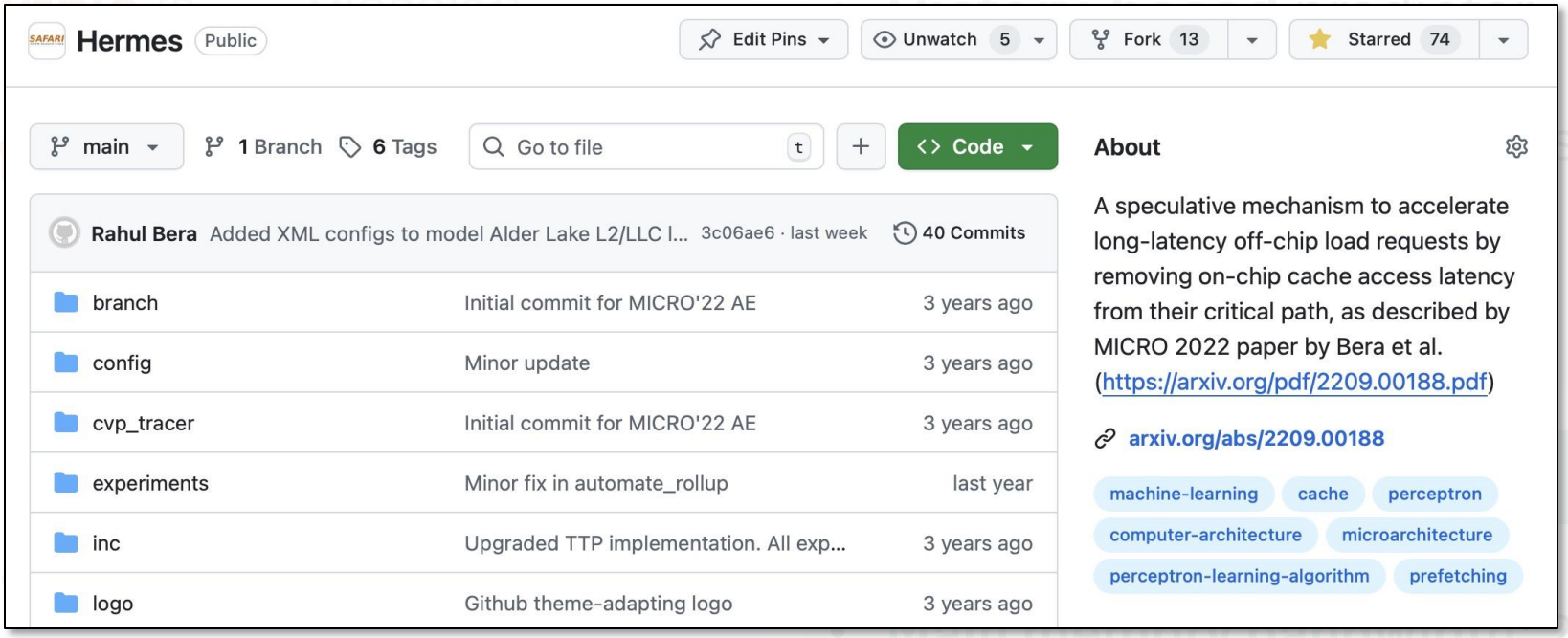
- **History-based** predictor [Yoaz+, ISCA'99]
- **Address Tag-Tracking** based predictor
- **Ideal** off-chip predictor

Workloads

- **110 single-core** workloads from SPEC CPU 2006, 2017, PARSEC, and Ligra, and more
- **220 four-core and eight-core** workload mixes

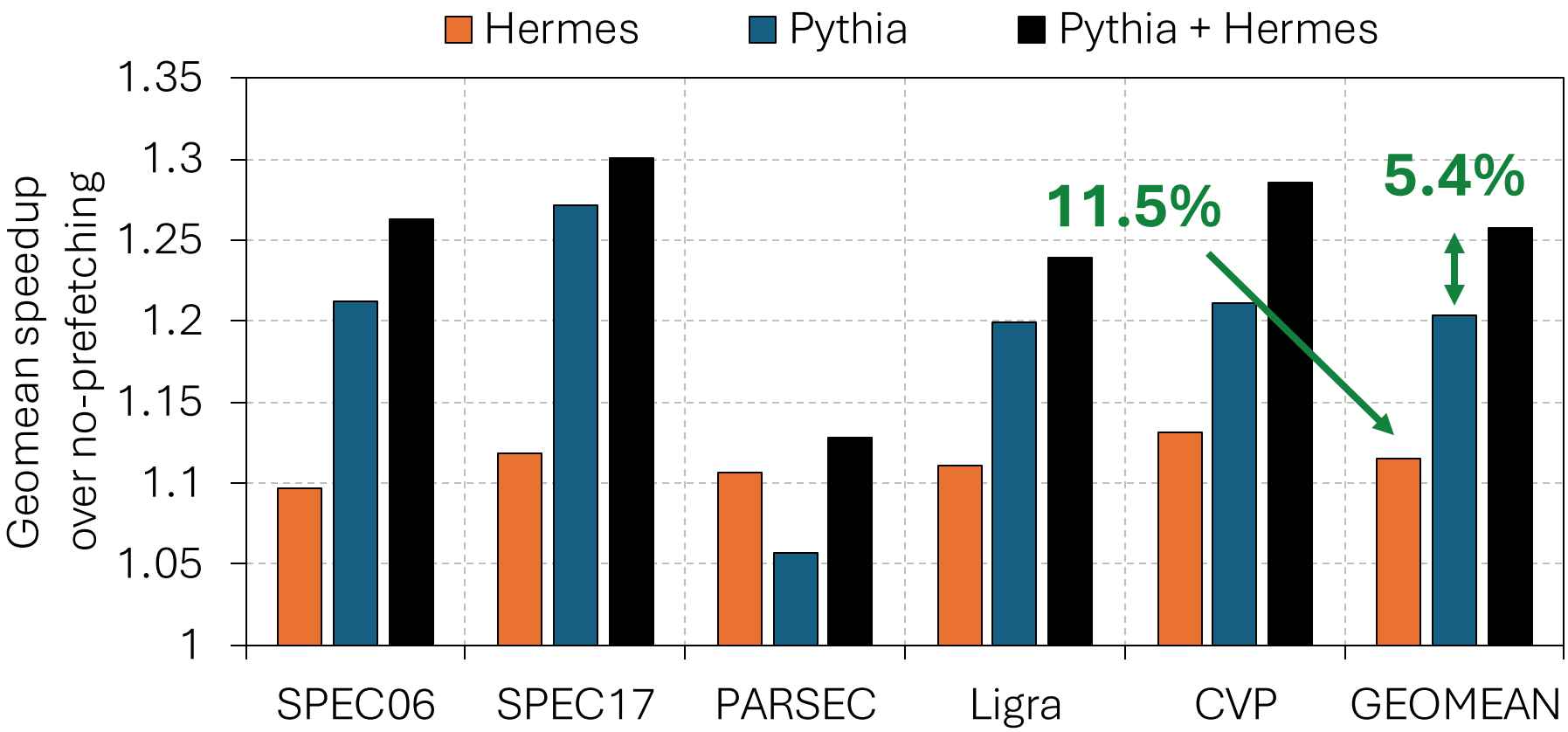
System Configurations

- **# cores**: 1- 8 cores
- **Main memory bandwidth**: 200-12800 MTPS
- **Cache access latency**
- **Interconnect latency**
- …



Open-sourced and artifact-evaluated with all three ACM badges

Key Results



Provides performance gains by itself

Key Results

High-accuracy and high-coverage

Off-chip prediction with 77% accuracy & 74% coverage

Low storage and low power overhead

Only 4 KB storage per core
1.5% increase in power over Pythia

One Problem: Two Fundamentally-Different Approaches

PYTHIA

HERMES

Data prefetcher

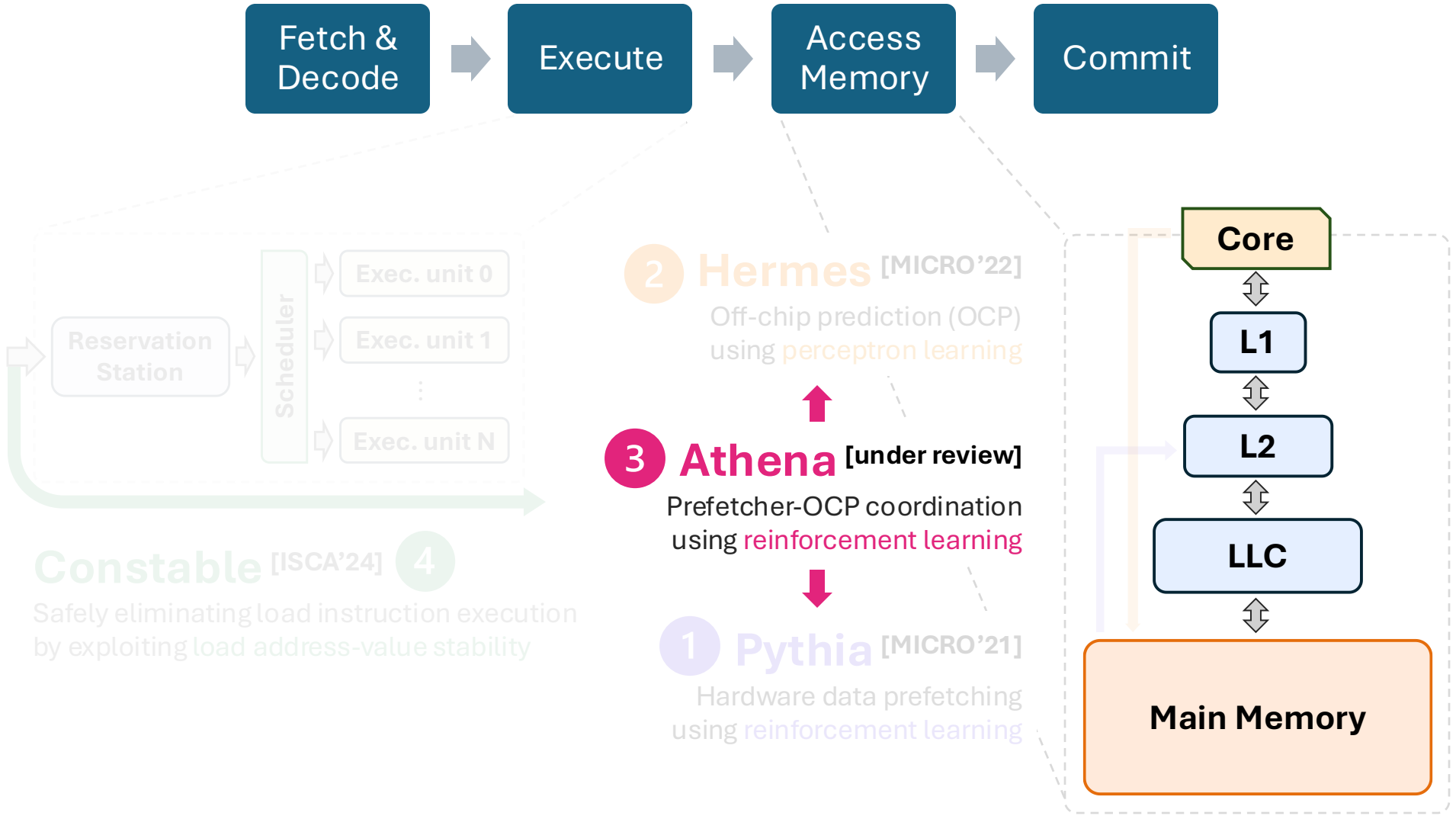
Predicts address of future memory requests

Off-chip predictor

Predicts whether or not a given memory **request would go off-chip**

How do they behave together?

Key Contributions



Key Observations

1

Off-chip prediction (OCP) and data prefetching individually provide complementary performance gains

Especially in memory bandwidth-constrained systems, where prefetcher often degrades performance, but an OCP improves performance

2

Naively combination is not beneficial

Enabling both mechanisms often negates performance benefits of OCP

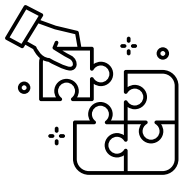
3

Existing mechanisms lack flexibility or have significant performance headroom

TLP [Jamet+, HPCA'24] lacks flexibility.
HPAC [Ebrahimi+, MICRO'09] and MAB [Gerogiannis+, MICRO'23] are not optimal

Our Goal

Design a **holistic framework** that



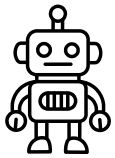
Autonomously synergizes OCP with multiple prefetchers throughout the cache hierarchy

To deliver **consistent performance benefits**, regardless of workloads and system configuration

Athena

Key Idea: Formulate OCP-prefetcher coordination as a **reinforcement learning** problem

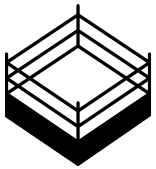
Athena



RL agent

Athena

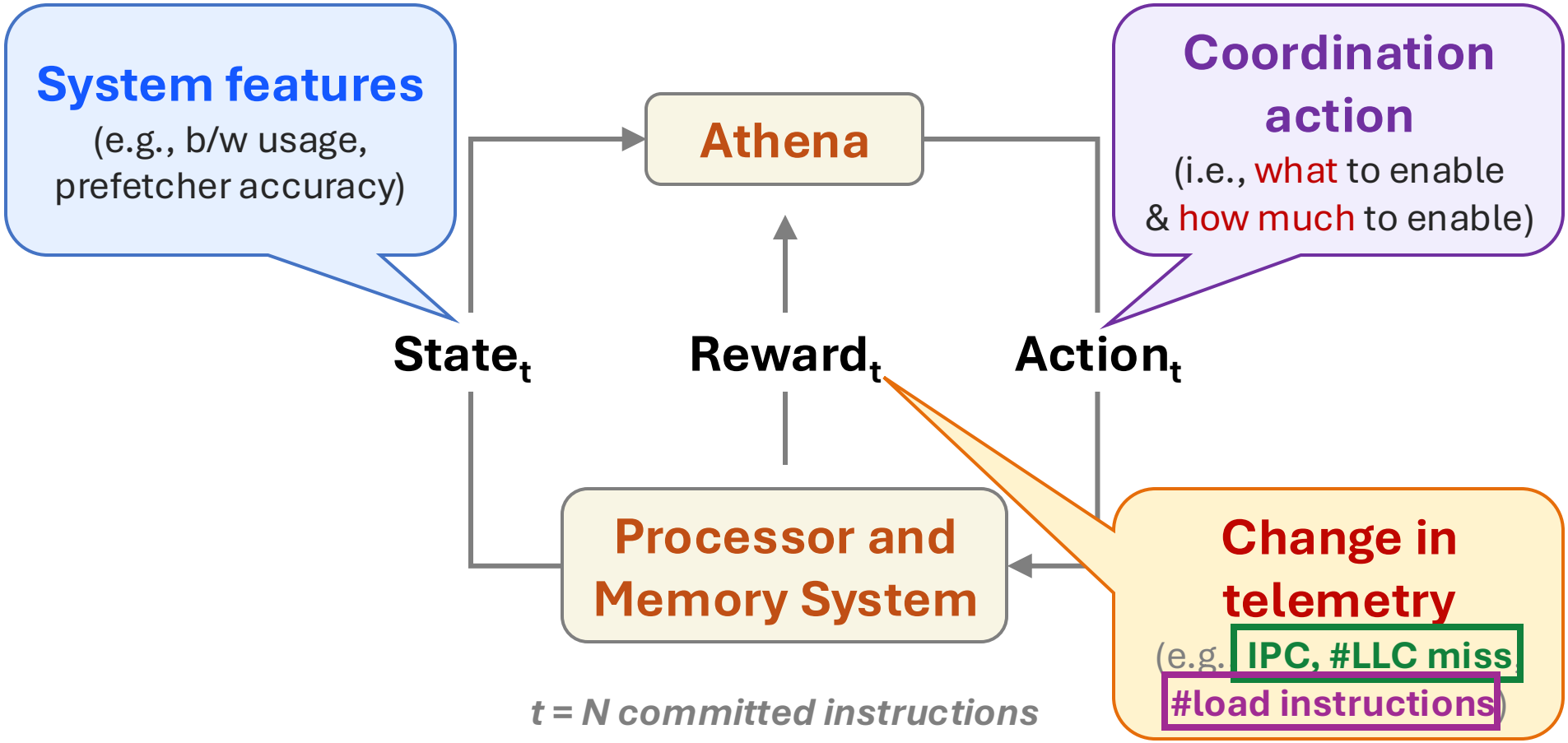
Autonomously *learn how to coordinate*



Processor and Memory System

Environment

Athena


System features
(e.g., b/w usage, prefetcher accuracy)
Athena
Coordination action
(i.e., what to enable & how much to enable)
State$_t$
Reward$_t$
Action$_t$
Processor and Memory System
Change in telemetry
(e.g., IPC, #LLC miss, #load instructions)
t = N committed instructions

Trace-Driven Simulation Methodology

Prefetchers

- **Stride** [Baer, SC'91]
- **IPCP** [Pakalapati+, ISCA'20]
- **Pythia** [Bera+, MICRO'21]
- **SPP+ PPF** [Bhatia+, ISCA'20]
- **SMS** [Somogyi+, ISCA'06]
- **MLOP** [Shakerinava+, DPC3'19]

Off-chip Predictors

- **Hermes** [Bera+, MICRO'22]
- **HMP** [Yoaz+, ISCA'99]
- **TTP** [Jalili+, HPCA'22]

Coordination Policies

- **HPAC** [Ebrahimi+, MICRO'09]
- **MAB** [Yoaz+, ISCA'99]
- **TLP** [Jamet+, HPCA'24]

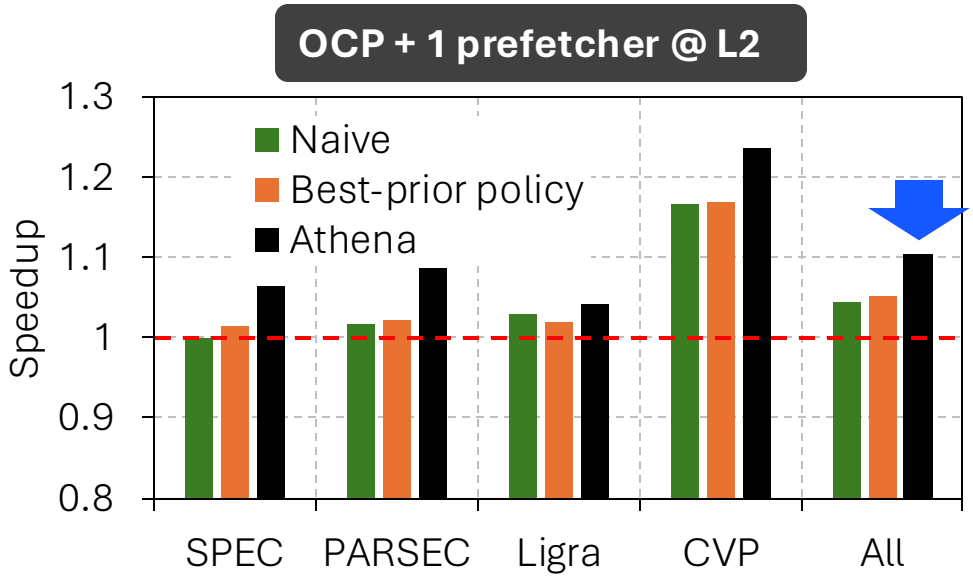
System Configurations

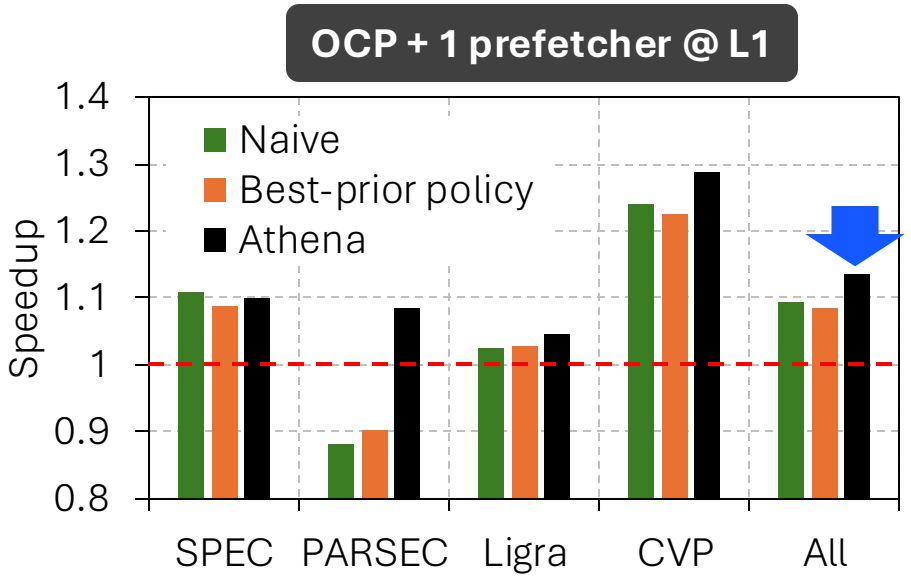
- **# cores**: 1- 8 cores
- **Main memory bandwidth**: 200 - 1600 MTPS
- 4 cache designs with various prefetchers at different cache levels

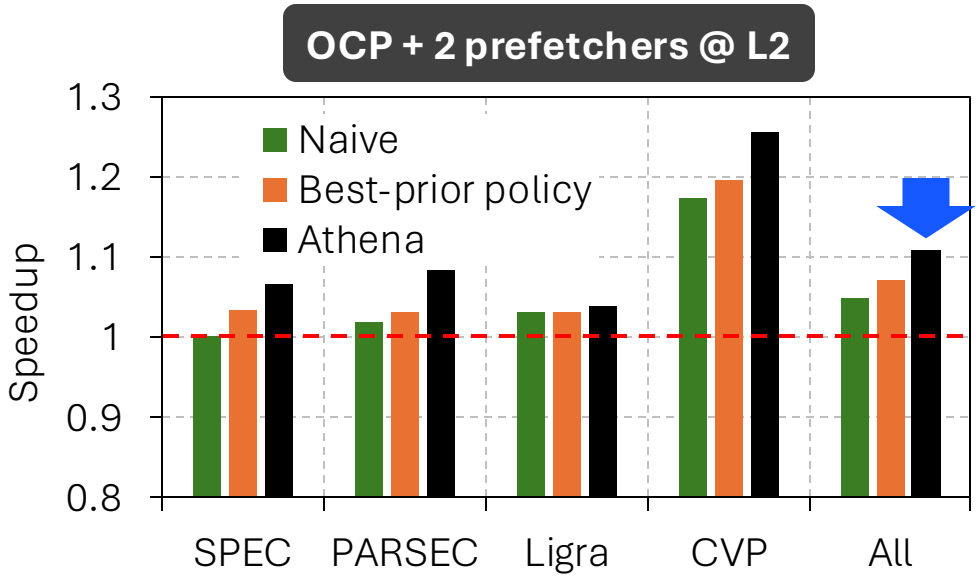
Workloads

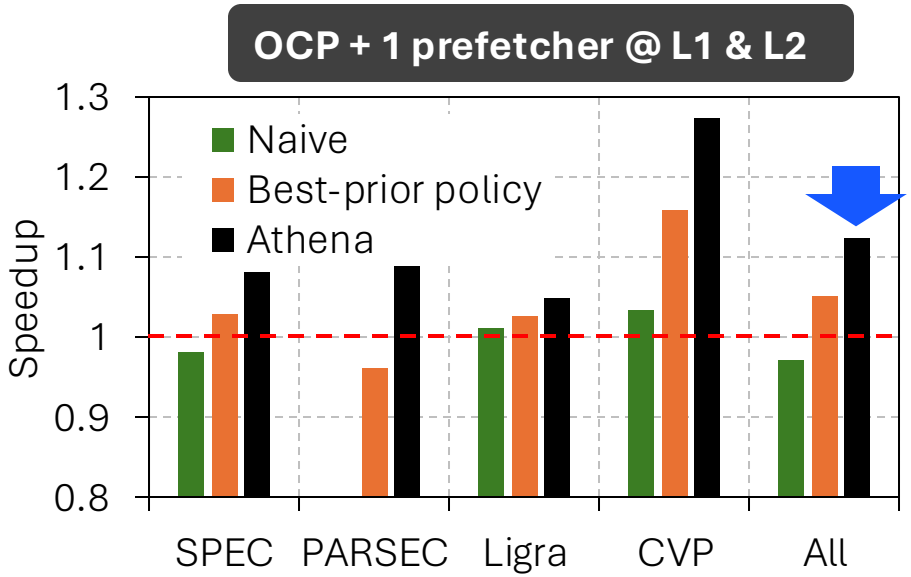
- **100 single-core** workloads
- **180 four-core and eight-core** workload mixes

Key Results

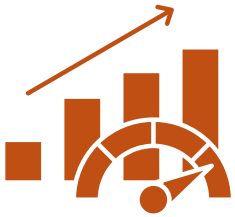

OCP + 1 prefetcher @ L2
Speedup
1.3
1.2
1.1
1
0.9
0.8
Naive
Best-prior policy
Athena
SPEC
PARSEC
Ligra
CVP
All


OCP + 1 prefetcher @ L1
Speedup
1.4
1.3
1.2
1.1
1
0.9
0.8
Naive
Best-prior policy
Athena
SPEC
PARSEC
Ligra
CVP
All


OCP + 2 prefetchers @ L2
Speedup
1.3
1.2
1.1
1
0.9
0.8
Naive
Best-prior policy
Athena
SPEC
PARSEC
Ligra
CVP
All


OCP + 1 prefetcher @ L1 & L2
Speedup
1.3
1.2
1.1
1
0.9
0.8
Naive
Best-prior policy
Athena
SPEC
PARSEC
Ligra
CVP
All

Key Results



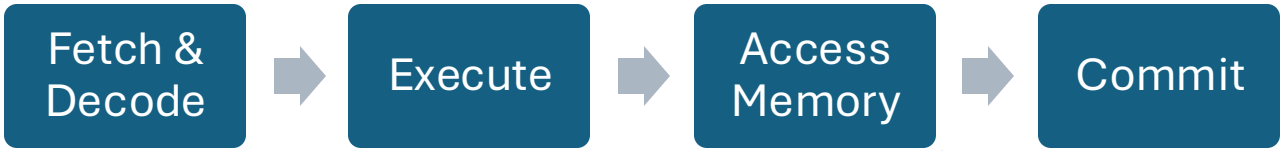
Consistent performance benefit

In coordinating **four types of prefetchers** and **three types of off-chip predictors**

Low storage overhead

Only 3 KB per core

Key Contributions: Recap



Constable [ISCA'24] 4

Safely eliminating load instruction execution by exploiting load address-value stability

2 **Hermes** [MICRO'22]

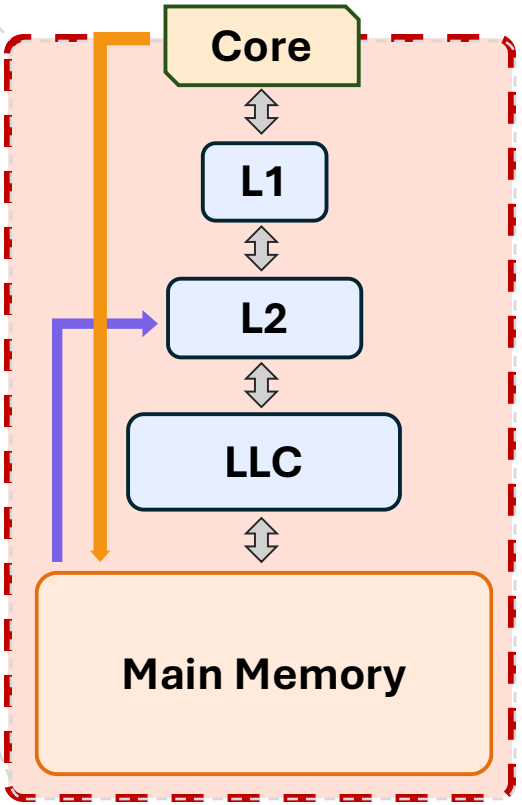
Off-chip prediction (OCP) using perceptron learning

3 **Athena** [under review]

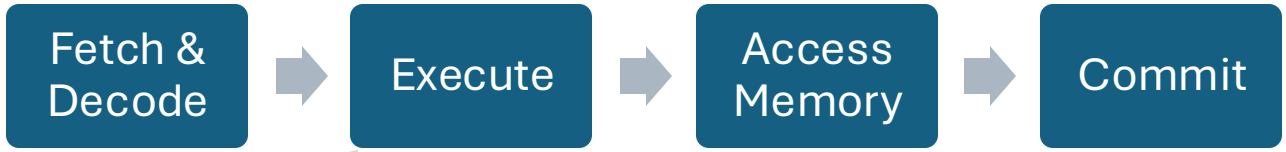
Prefetcher-OCP coordination using reinforcement learning

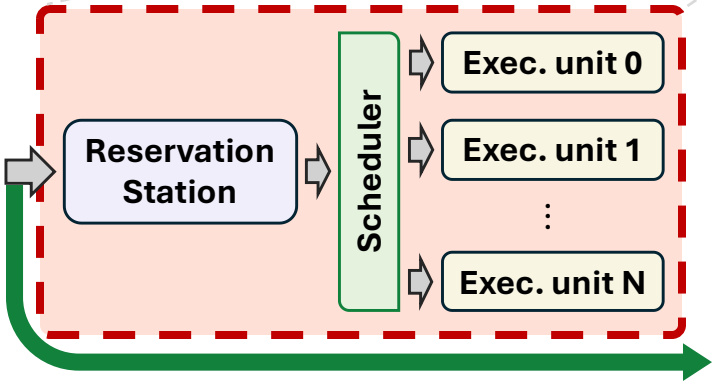
1 **Pythia** [MICRO'21]

Hardware data prefetching using reinforcement learning



Key Contributions





Constable [ISCA'24] 4

Safely eliminating load instruction execution by exploiting load address-value stability

2 Hermes [MICRO'22]

Off-chip prediction (OCP) using perceptron learning

3 Athena [under review]

Prefetcher-OCP coordination using reinforcement learning

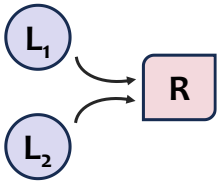
1 Pythia [MICRO'21]

Hardware data prefetching using reinforcement learning

Key Observations

A large fraction (34%) of dynamic loads fetch the same data from the same address throughout the entire workload

Techniques like load value prediction (LVP) and memory renaming (MRN) do not fully exploit this data characteristic



These loads cause instruction-level parallelism loss due to resource dependence

i.e., blocking other loads from getting executed due to limited resources. LVP and MRN fail to mitigate resource dependence

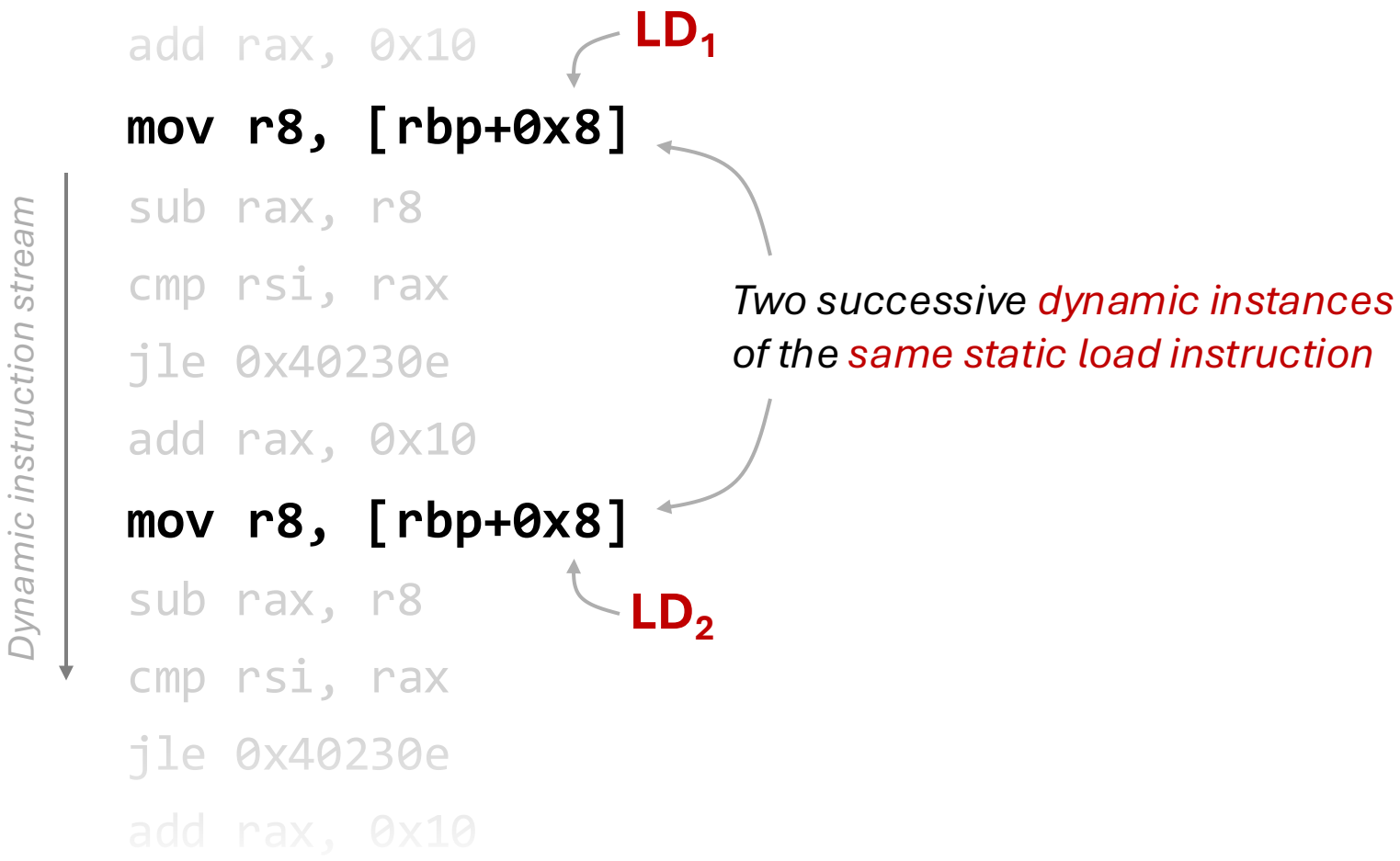
Mitigating both data and resource dependence has higher potential performance improvement

Eliminating execution of such loads provides more than 2X potential performance benefit than just breaking their data dependence

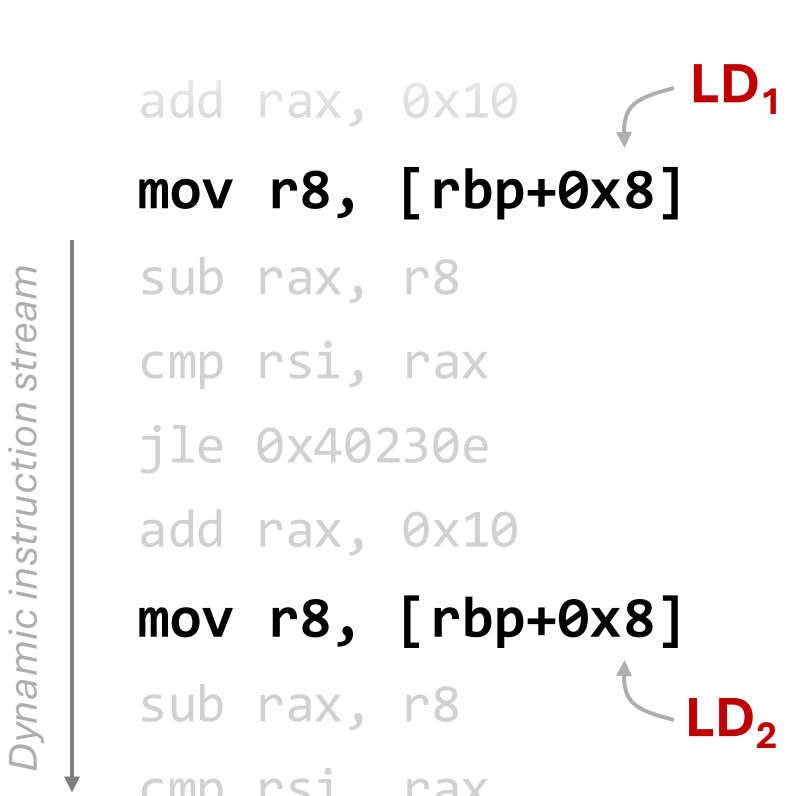
Our Goal

Improve instruction-level parallelism by mitigating **both load data and resource dependence**

Constable: Key Insight



Constable: Key Insight



If the source register (**rbp**) has not been modified

LD_2 would have the same address as LD_1

Address generation of LD_2 can be eliminated

If no store or snoop request to address **[rbp+0x8]**

LD_2 would fetch the same data as LD_1

Data fetching of LD_2 can be eliminated

Constable: Key Steps

Dynamically identify load instructions
that have historically fetched
the same data from the same load address
(i.e., *likely-stable*)

Eliminate execution of likely-stable loads
by **tracking modifications** to
their source registers and their load addresses

Evaluation with Industry-Grade Simulator

Strong Baseline

- **Aggressive** OoO processor
- With **memory renaming, zero/constant/move elimination, branch folding**
- **Five** data prefetchers

Compared Against

- **EVES,** the state-of-the-art load value predictor [Seznec+, CVP'18]
- **Early load address resolution** [Berkerman+, ISCA'00]
- **Register file prefetching** [Shukla+, ISCA'22]

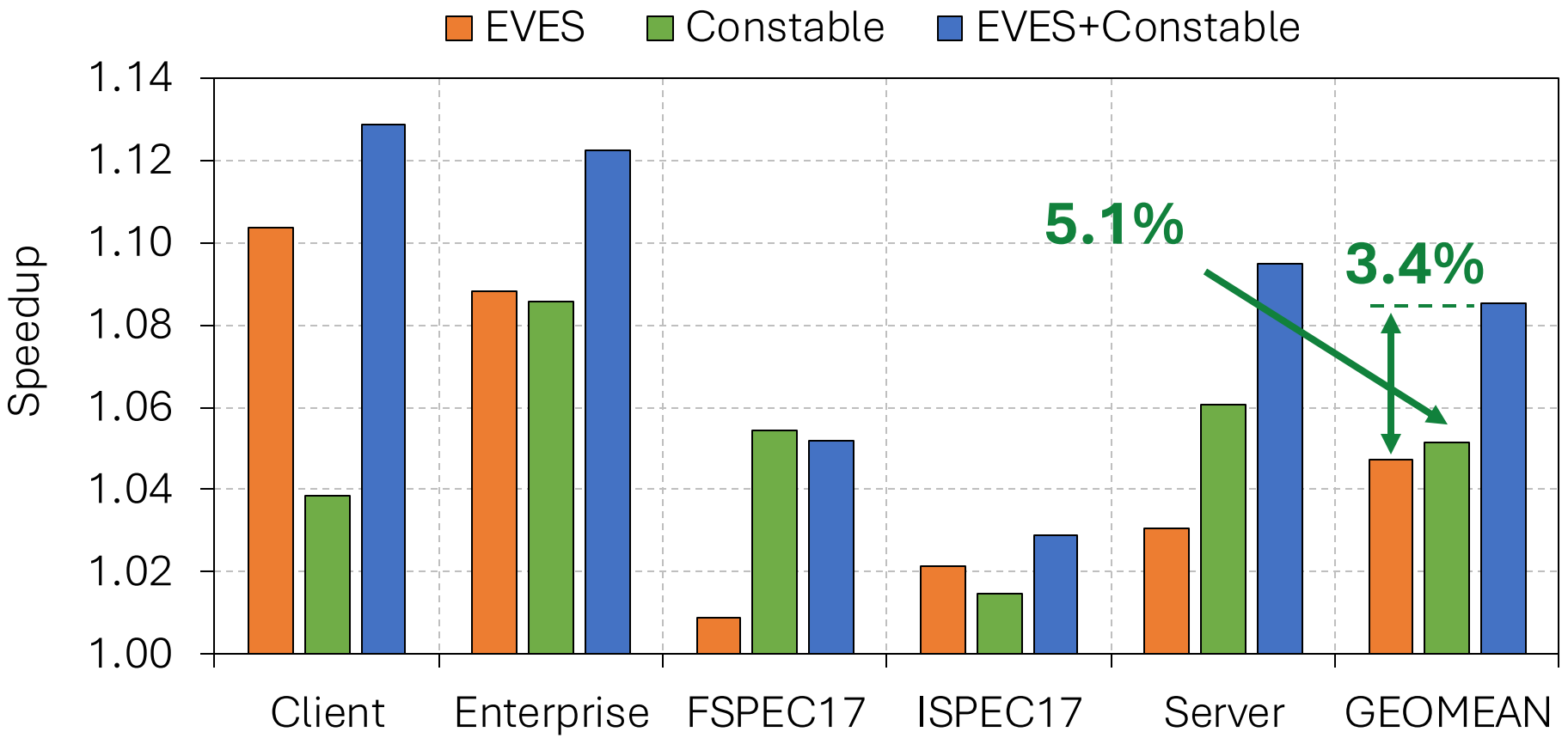
Workloads

- **90 workloads** from variety of application domains
  - **SPEC CPU 2017**
  - **Client**
  - **Enterprise**
  - **Server**

System Configurations

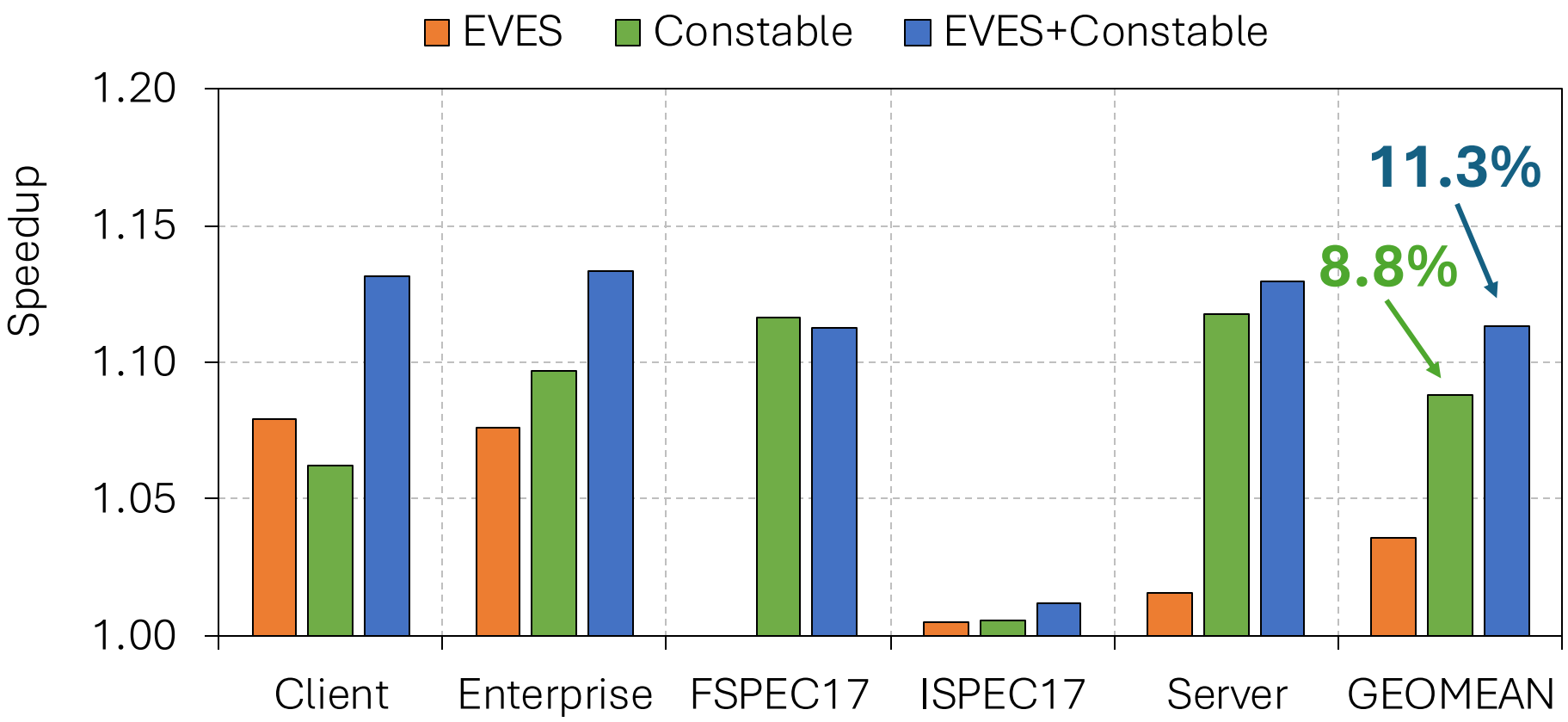
- Without **simultaneous multithreading**
- With **2-way simultaneous multithreading**

Key Results – Without SMT



Provides performance gains by itself

Key Results – With 2-way SMT



Higher performance benefits

in a 2-way SMT processor

Key Results

Improved resource efficiency

26% L1-D access reduction,
8.8% RS allocation reduction

Reduction in dynamic power

9.1% L1-D power reduction, 5.1% RS power reduction

Low storage and area overhead

12.4 KB per core, 0.232 mm2 area overhead

Putting It All Together

Recall: Thesis Statement

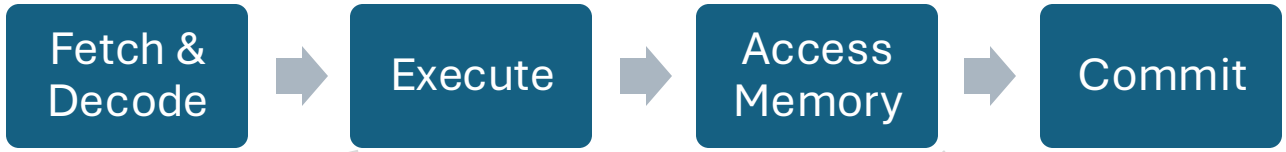
By enabling microarchitecture to

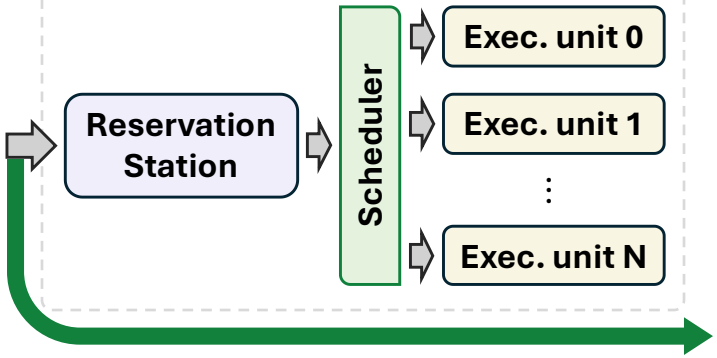
1. Adapt its policies by **continuously learning** from application data and system-generated data

2. Tailor its decisions **by exploiting characteristics** of application data

We can significantly **improve performance and energy efficiency** of state-of-the-art processors

Key Contributions





Constable [ISCA'24] 4

Safely eliminating load instruction execution by exploiting load address-value stability

2 **Hermes** [MICRO'22]

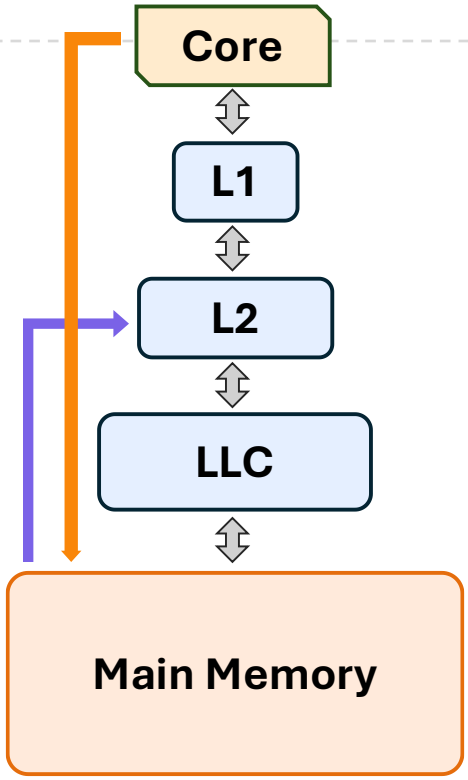
Off-chip prediction (OCP) using perceptron learning

3 **Athena** [under review]

Prefetcher-OCP coordination using reinforcement learning

1 **Pythia** [MICRO'21]

Hardware data prefetching using reinforcement learning



Putting it All Together

ML-driven techniques

1 **Pythia** [MICRO'21]
Hardware data prefetching using reinforcement learning

2 **Hermes** [MICRO'22]
Off-chip prediction (OCP) using perceptron learning

3 **Athena** [under review]
Prefetcher-OCP coordination using reinforcement learning

1 Adapt their policies by **continuously learning** from application data and system-generated data

2 Tailor its decisions **by exploiting characteristics** of the application data

4 **Constable** [ISCA'24]
Safely eliminating load instruction execution by exploiting load address-value stability

Data-aware technique

Putting it All Together

1. **Pythia** [MICRO'21]
   Hardware data prefetching using reinforcement learning
2. **Hermes** [MICRO'22]
   Off-chip prediction (OCP) using perceptron learning
3. **Athena** [under review]
   Prefetcher-OCP coordination using reinforcement learning

4. **Constable** [ISCA'24]
   Safely eliminating load instruction execution by exploiting load address-value stability

Significantly improve
performance and/or energy efficiency
over best-prior mechanisms

Recall: Thesis Statement

By enabling microarchitecture to

1. Adapt its policies by **continuously learning** from application data and system-generated data

2. Tailor its decisions **by exploiting characteristics** of application data

We can significantly **improve performance and energy efficiency** of state-of-the-art processors

Future Directions

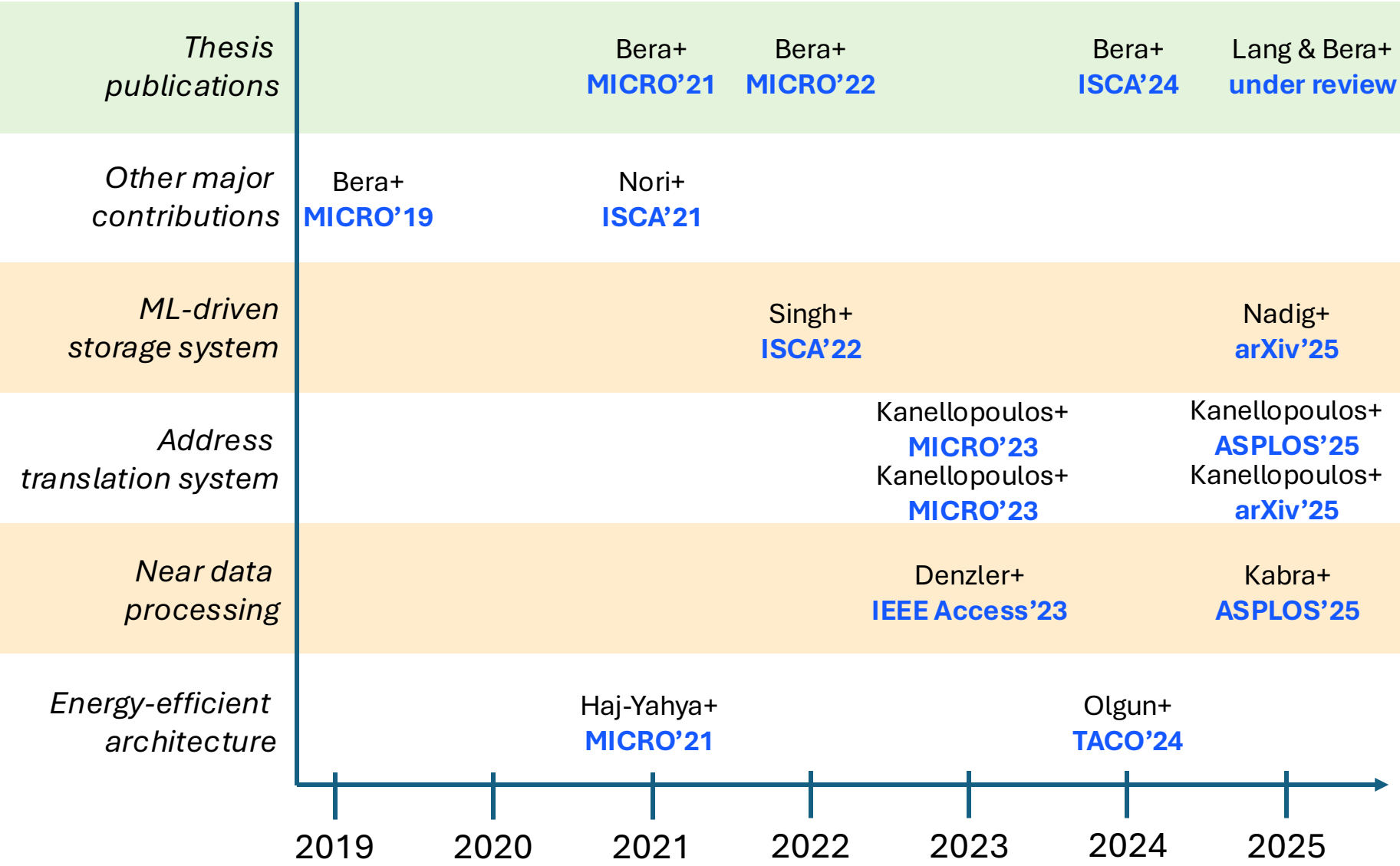
Extend

- RL-driven online **instruction criticality prediction**
- RL-driven **shared resource partitioning** in multi-core systems
- **Multi-agent RL** for coordinated caching, prefetching & memory scheduling
- **Compiler-microarchitecture co-design** to eliminate redundant loads
- Near-accurate speculation-aware **lazy instruction execution**

Expand

- **Off-chip prediction in GPUs**/accelerators
- Accurate **memory location prediction** for disaggregated memory systems
- Off-chip prediction for **transparent identification of PIM regions**
- Learning-based **data partitioning/placement** for PIM
- Learning-based self-adaptive **prefetcher for PIM**

My Involvements During Ph.D.



Acknowledgements

- Onur Mutlu
- Anant V. Nori, Sree Subramoney
- Aamer Jaleel, Boris Grot, Chris Wilkerson, Daniel Jiménez, Pradip Bose
- Kaveh Razavi
- Amazingly-talented mentees: Zhenrong, Sgouras, Clément, and Liana
- All SAFARI group members (especially Big K., Nika, Rakesh, and Mohammad)
- Industrial partners & sponsors
- Friends and colleagues all over the world
- My mother, my wife, and my aunt

Acknowledgements

My beloved baba,

Mr. R. R. Bera

(1960 - 2020)

Mitigating the Memory Bottleneck with Machine Learning-Driven and Data-Aware Microarchitectural Techniques

Rahul Bera

Doctoral Examination
01.10.2025

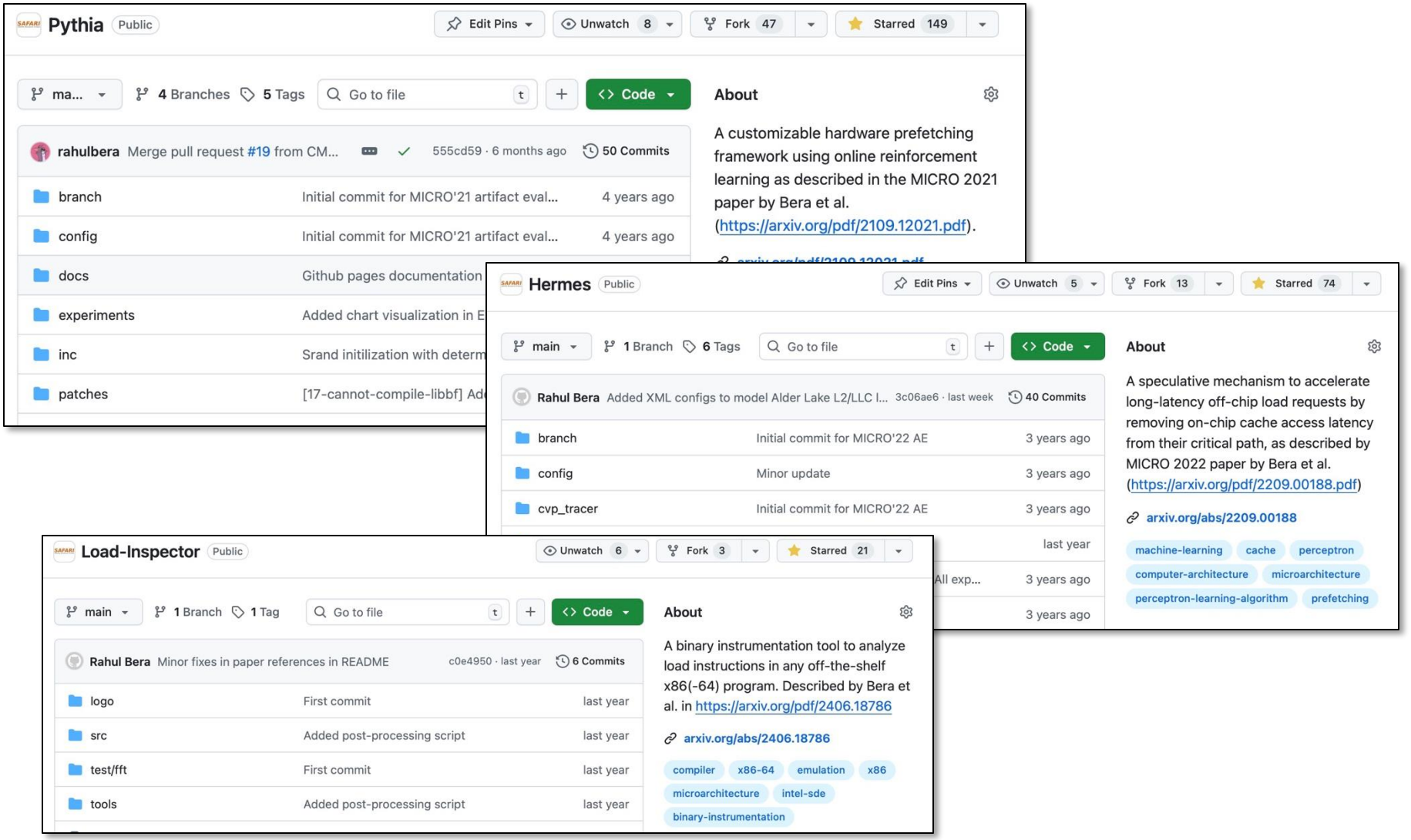
Advisor:
Onur Mutlu (ETH Zurich)

Co-Examiners:
Aamer Jaleel (Nvidia Research)
Boris Grot (University of Edinburgh)
Chris Wilkerson (Intel)
Daniel Jiménez (Texas A&M University)
Pradip Bose (IBM T. J. Watson Research Center)

SAFARI ETH zürich

Influence on the Community

Open-Sourced Artifacts



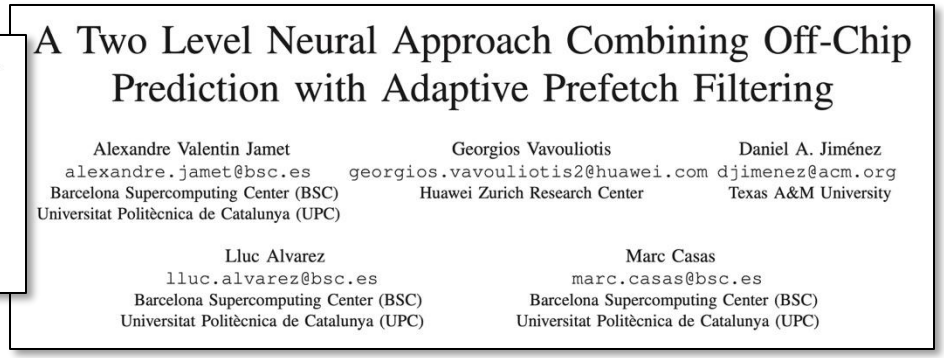
Impact on the Community

- Pythia, Hermes, and Constable – all have served as the state-of-the-art in many subsequent works

Micro-MAMA: Multi-Agent Reinforcement Learning for Multicore Prefetching

Charles Block, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, coblock2@illinois.edu
Gerasimos Gerogiannis, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, gg24@illinois.edu
Josep Torrellas, University of Illinois Urbana-Champaign, Urbana, Illinois, USA, torrella@illinois.edu

MICRO'25



A Two Level Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering

Alexandre Valentin Jamet, alexandre.jamet@bsc.es, Barcelona Supercomputing Center (BSC), Universitat Politècnica de Catalunya (UPC)
Georgios Vavouliotis, georgios.vavouliotis2@huawei.com, Huawei Zurich Research Center
Daniel A. Jiménez, djimenez@acm.org, Texas A&M University
Lluc Alvarez, lluc.alvarez@bsc.es, Barcelona Supercomputing Center (BSC), Universitat Politècnica de Catalunya (UPC)
Marc Casas, marc.casas@bsc.es, Barcelona Supercomputing Center (BSC), Universitat Politècnica de Catalunya (UPC)

HPCA'24

Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making

Gerasimos Gerogiannis, University of Illinois at Urbana-Champaign, USA, gg24@illinois.edu
Josep Torrellas, University of Illinois at Urbana-Champaign, USA, torrella@illinois.edu

MICRO'23

CLIP: Load Criticality based Data Prefetching for Bandwidth-constrained Many-core Systems

Biswabandan Panda, biswa@cse.iitb.ac.in, Indian Institute of Technology Bombay, Mumbai, India

MICRO'23

Merging Similar Patterns for Hardware Prefetching

Shizhi Jiang*, Qiusong Yang✉†, and Yiwei Ci‡
*University of Chinese Academy of Sciences
*†‡Institute of Software, Chinese Academy of Sciences
Beijing, China
Email: *shizhi@nfs.iscas.ac.cn, †qiusong@iscas.ac.cn, ‡yiwei@iscas.ac.cn

MICRO'22

The Curious Case of Global Stable Loads

Shagnik Pal, The University of Texas at Austin, Austin, USA, shagnik@utexas.edu
Jeeho Ryoo, Fairleigh Dickinson University, Vancouver, Canada, j.ryoo@fdu.edu
Lizy K. John, The University of Texas at Austin, Austin, USA, ljohn@ece.utexas.edu

IISWC'25

Backup

Pythia

Hermes

Athena

Constable

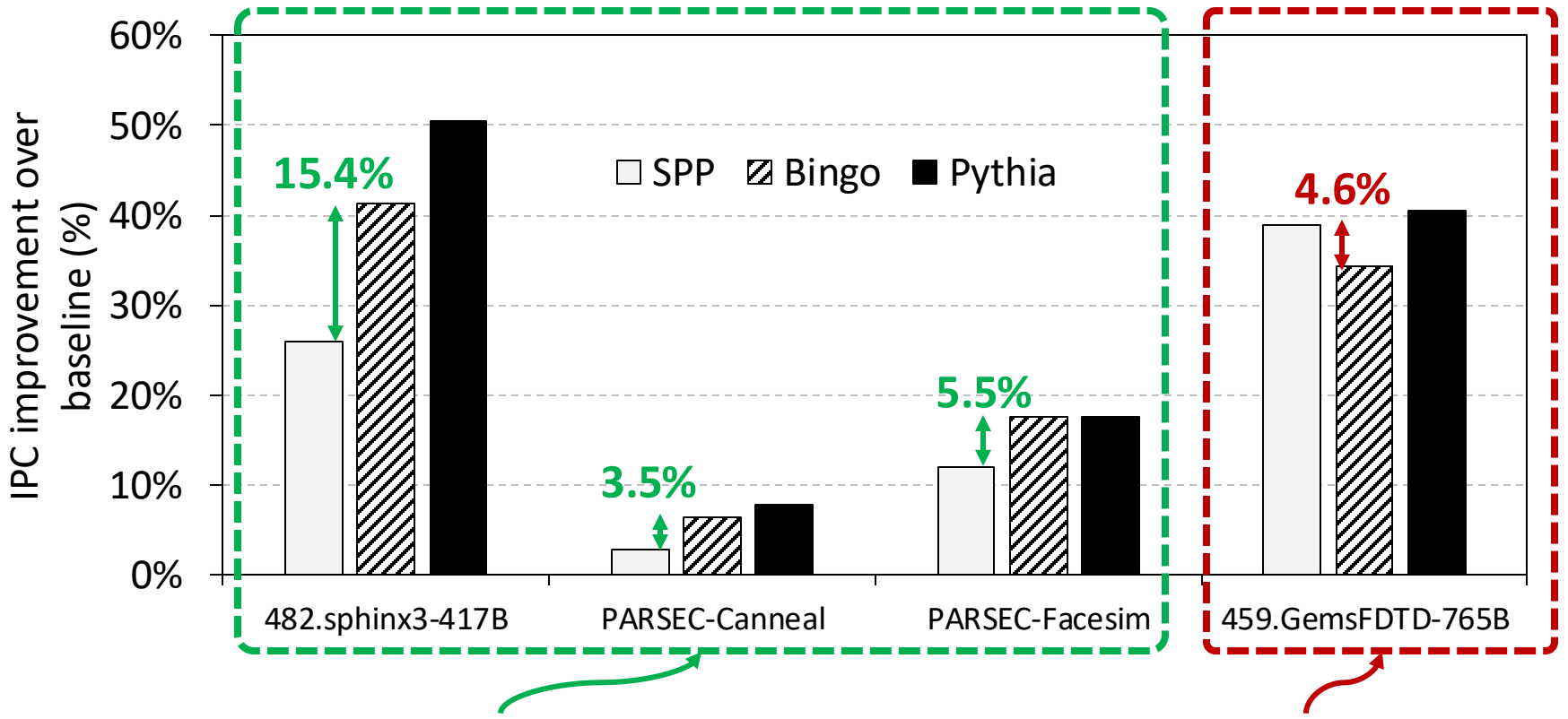
Pythia Discussion

Pythia Discussions

- **FAQs**
  - Why RL?
  - What about large page?
  - What's the prefetch degree?
  - Can customization happen during workload execution?
  - Can runtime mixing create problem?
- **Simulation and Methodology**
  - Basic Pythia configuration
  - System parameters
  - Configuration of prefetchers
  - Evaluated workloads
  - Feature selection
- **Detailed Design**
  - Reward structure
  - Design overview
  - QVStore Organization
- **More Results**
  - Comparison against other adaptive prefetchers
  - Comparison against Context prefetcher
  - Feature combination sensitivity
  - Hyperparameter sensitivity
  - Comparison with multi-level prefetchers
  - Performance in unseen workloads
  - Single-core s-curve
  - Four-core s-curve
  - Detailed performance analysis
  - Benefit of bandwidth awareness
  - Case study
  - Customizing rewards
  - Customizing features

(1) Single-Feature Prefetch Prediction

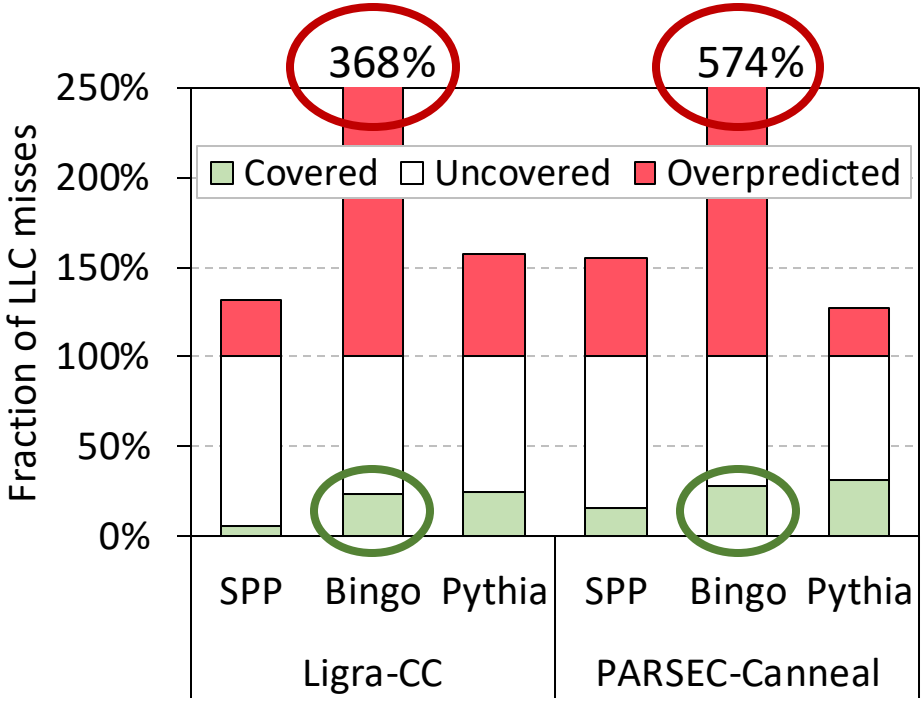
- Provides **good** performance **gains** mainly on workloads where the **feature-to-pattern correlation exists**

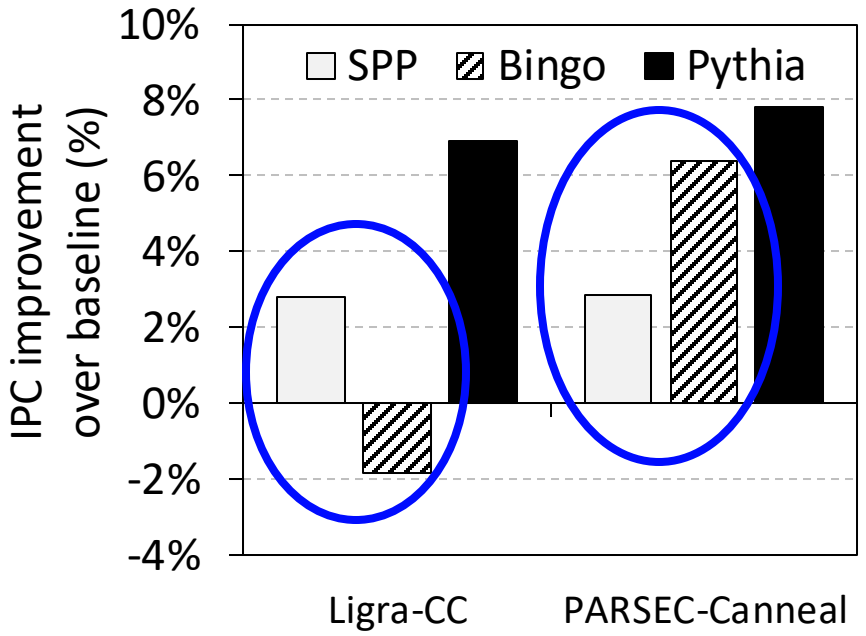


[1] Bakshalipour et al., HPCA'19 [2] Kim et al., MICRO'16

(2) Lack of Inherent System Awareness

- Little understanding of **undesirable effects** (e.g., memory bandwidth usage, cache pollution, …)
  - Performance loss in **resource-constrained** configurations





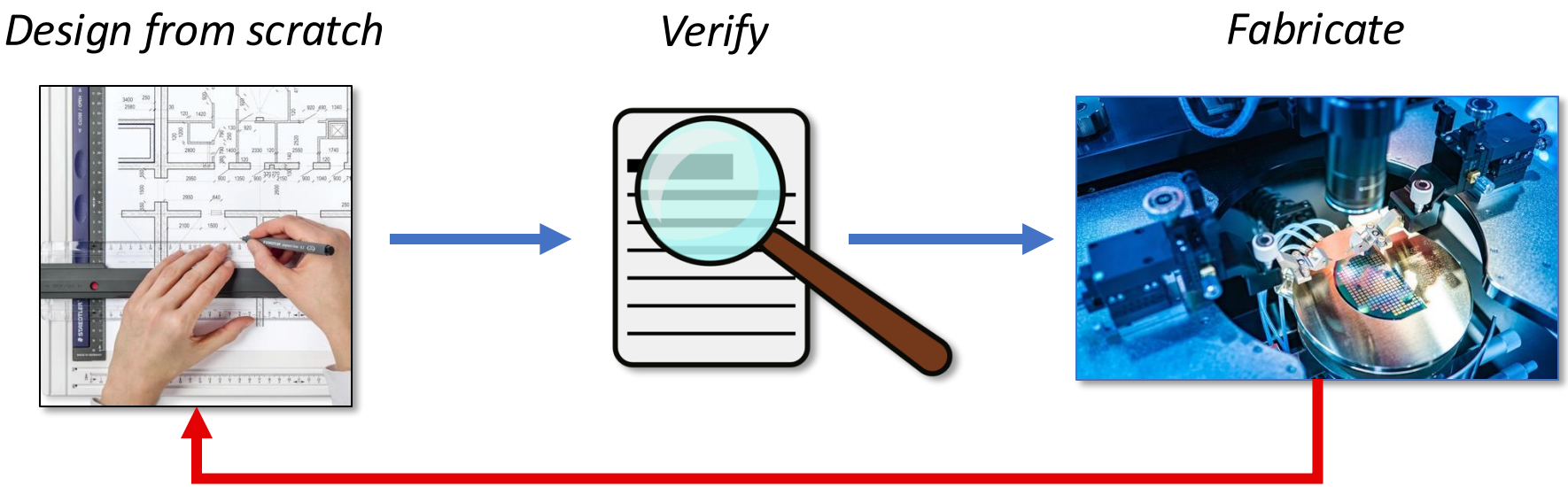
Similar coverage

Lower overpredictions

Yet, lower performance

(3) Lack of In-silicon Customizability

- Feature **statically** selected at design time
  - **Rigid hardware** designed specifically to exploit that feature
- **No way to change** program feature and/or change prefetcher's objective **in silicon**
  - **Cannot adapt** to a wide range of workload demands



Why RL? Why Not Supervised Learning?

- Determining the **benefits of prefetching** (i.e., whether a decision was good for performance or not) is **not easy**
  - **Depends on a complex set of metrics**
    - Coverage, accuracy, timeliness
    - Effects on system: b/w usage, pollution, cross-application interference, …
  - **Dynamically-changing environmental conditions** change the benefit
  - **Delayed feedback due to long latency** (might not receive feedback at all for inaccurate prefetches!)

- Differs from classification tasks (e.g., branch prediction)
  - Performance strongly correlates mainly to accuracy
  - Does not typically depend on environment
  - Bounded feedback delay

What About Large Pages?

- Pythia's framework can be **easily extended** to incorporate additional prefetch actions (i.e., possible prefetch offsets for the page size)

- To decrease the storage overhead
  - **Prune action space** via automatic design-space exploration
  - **Hash action values** to retrieve Q-values

What is the Prefetch Degree? Is It Managed by the RL Agent?

- Pythia employs **a simple degree selector**, separate from the RL agent
  - If the agent has selected the same prefetch action (O) multiple times in a row, Pythia increases the degree (A+2O, A+3O, …)
  - At most degree 4

- Future works on managing degree by the RL agent

Can the Customization Be Done While the Workload is Running?

- Certainly.
- Pythia, being an **online learning** technique, will autonomously adapt (and optimize) its policy to use the new program features or the modified reward values

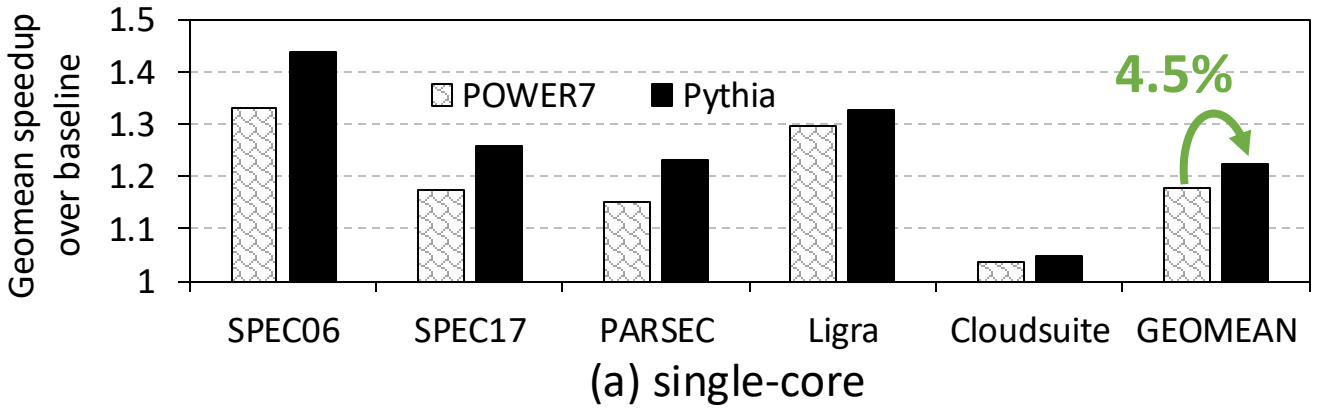
Can Runtime Workload Mix Create an Issue?

- We implement the bandwidth usage feedback using a counter in the memory controller. Thus Pythia already has a **global view of the memory bandwidth usage** that incorporates all workloads running on a multi-core system

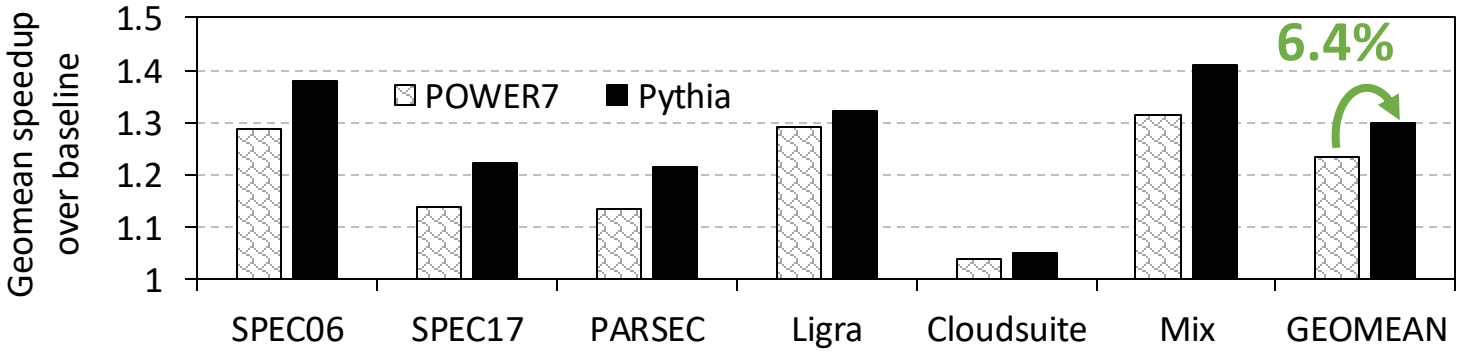
- We evaluate a diverse set (300 of each category) of four-core, eight-core, twelve-core random workload mixes
- Based on our evaluation, we observe that **Pythia dynamically adapts** itself to varying workload demands

How does Pythia Compare Against Other Adaptive Prefetching Solutions?

- We compare Pythia against **IBM POWER7**[5] prefetcher
  - Adaptively selects prefetcher degree/configuration by monitoring program IPC



(a) single-core

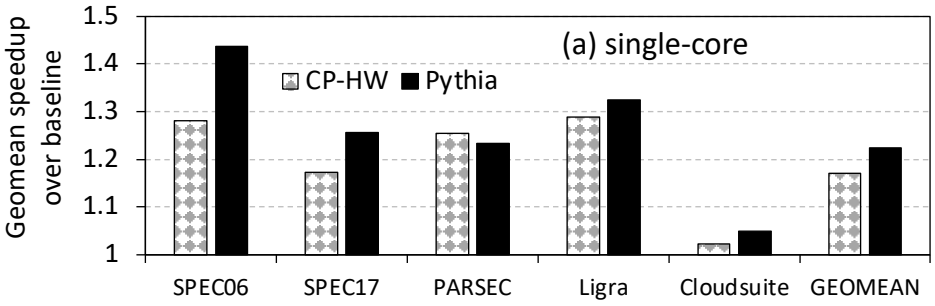


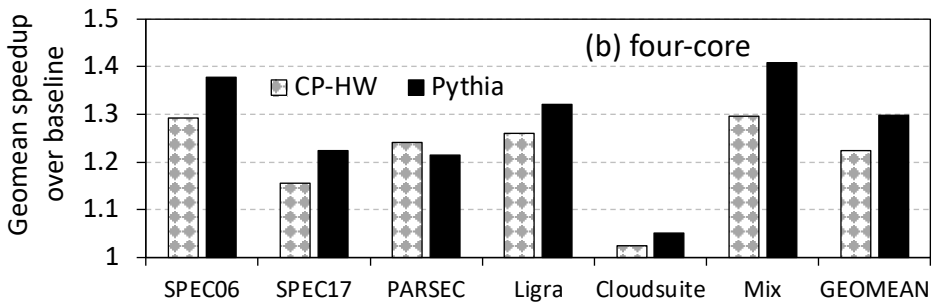
(b) four-core

[5] Jimenez et al., TOPC'14

How Does Pythia Compare Against the Context Prefetcher?

- Pythia widely differs from the Context Prefetcher (CP)[6] in all three aspects: state, action, and reward. The key differences are:
  - CP **does not consider system-level feedback**
  - CP models the agent as a contextual bandit which **takes myopic prefetch decisions** as compared to Pythia
  - CP **requires compiler support** to extract software-level features



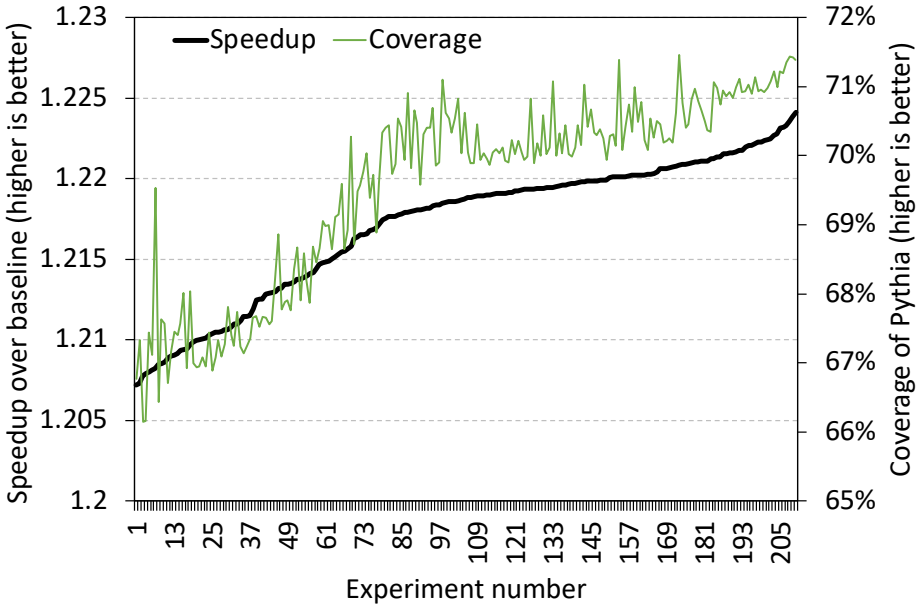


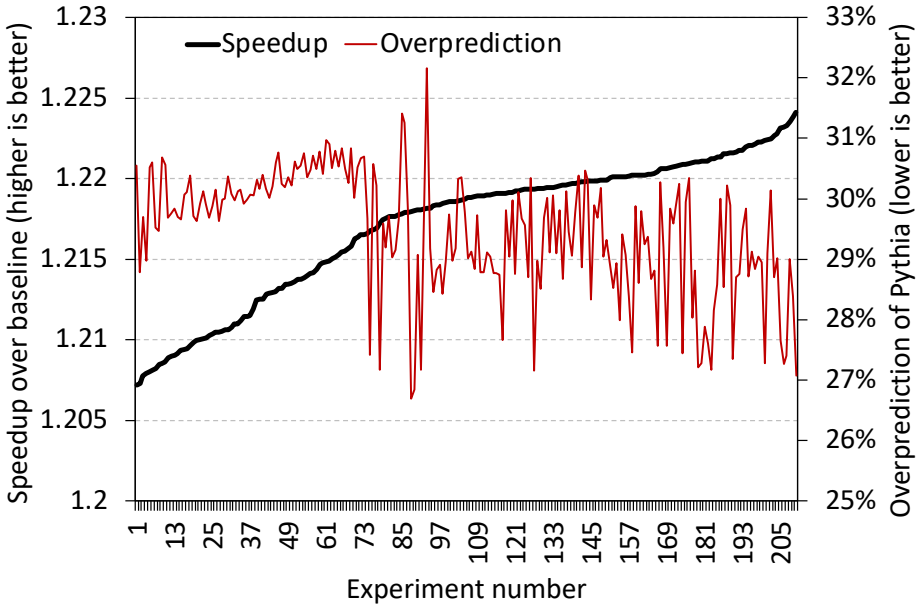
Pythia outperforms CP-HW by 5.3% in single-core and 7.6% in four-core system

[6] Leeor et al., ISCA'15

How Pythia's Performance Changes With Various State Definitions You Have Swept?

- In total we evaluate state defined as any-one, any-two, and any-three combinations of 32 features





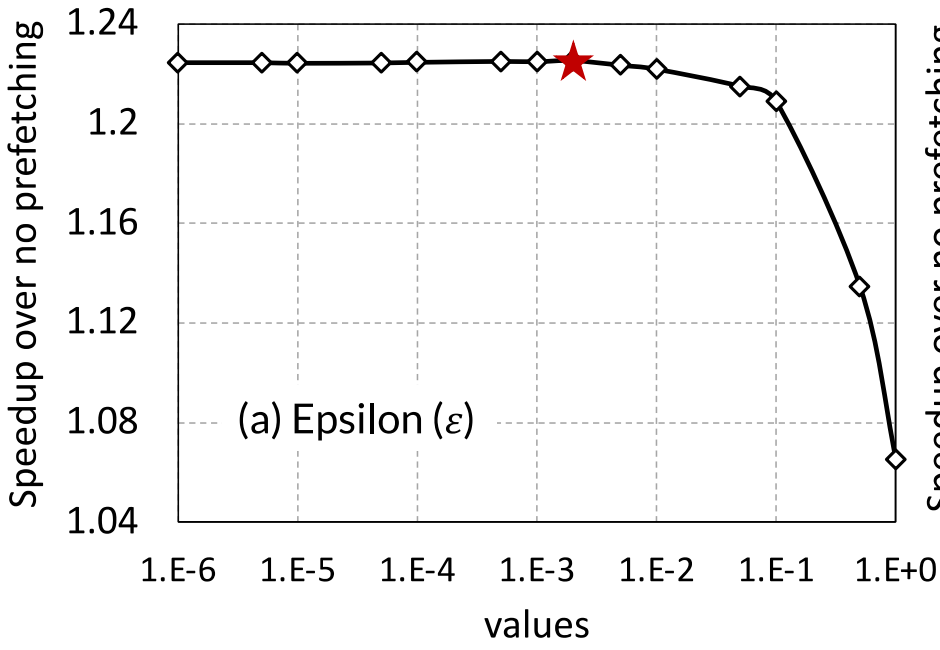
Performance gain ranges from 20.7% to 22.4%

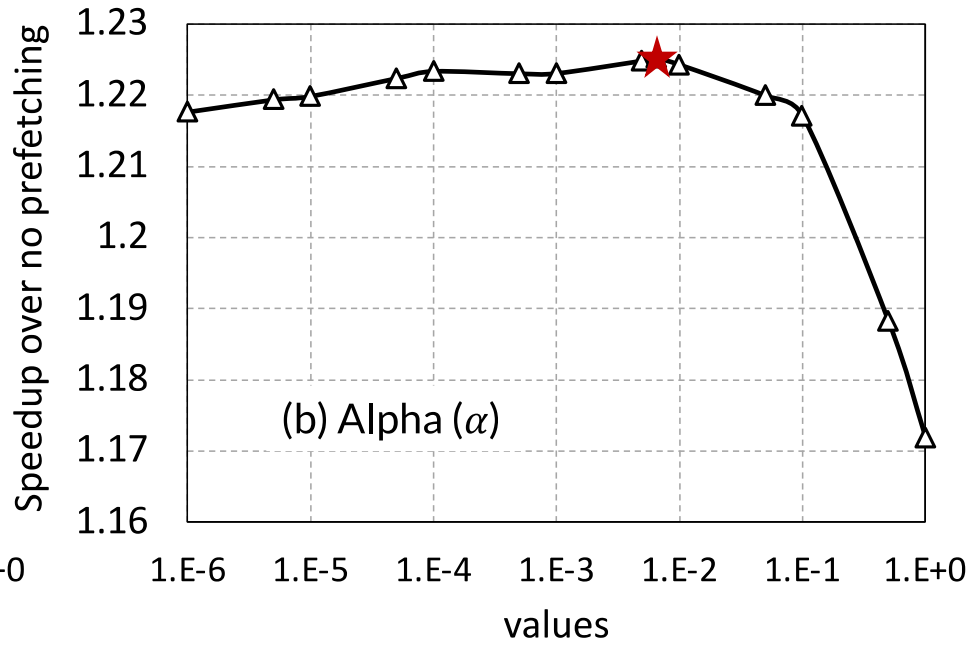
Coverage ranges from 66.2% to 71.5%

Overprediction ranges from 26.7% to 32.2%

Is Pythia Sensitive to Hyperparameter?

- Not setting hyperparameters can significantly impact the overall performance improvement

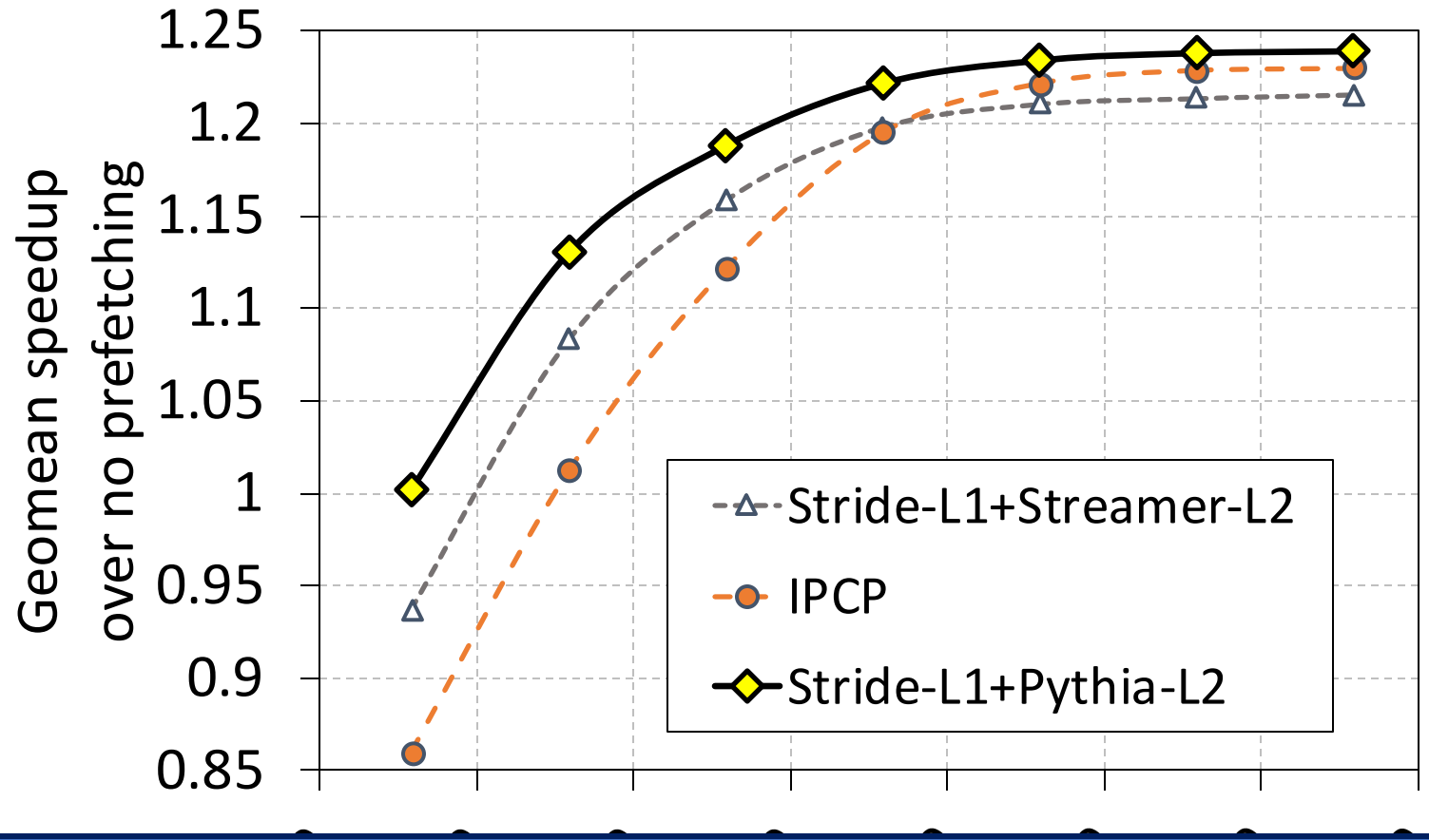




Changing ε from 0.002 to 1.0 drops perf. by 16%

Changing α from 0.0065 to 1.0 drops perf. by 5.4%

How Does Pythia Compare Against Commercial Multi-level Prefetchers?

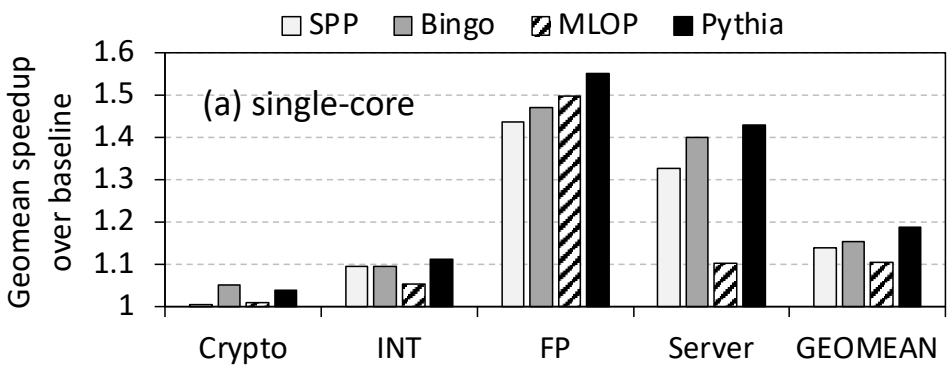


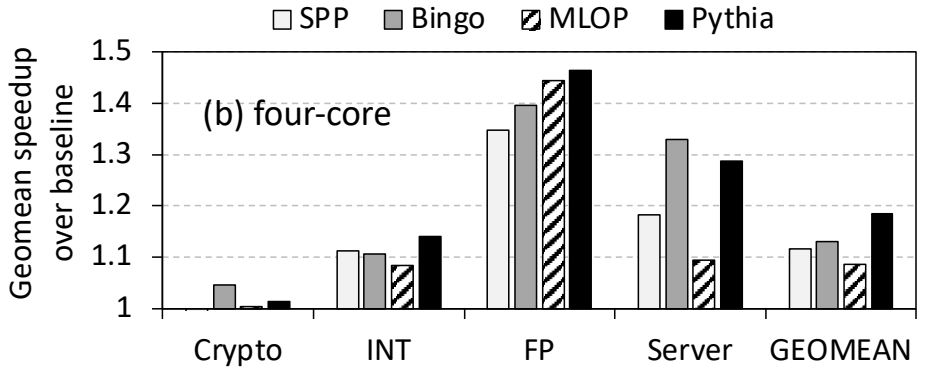
Pythia outperforms IPCP [7] by 14.2% on average in 150-MTPS

[6] Prakalapati et al., ISCA'20

Does Pythia Perform Equally Well for Unseen Workloads also?

- Evaluated with 500 traces from value prediction championship
  - **No prefetcher has been trained on these traces**





Pythia outperforms MLOP and Bingo by 8.3% and 3.5% in single-core

And 9.7% and 5.4% in four-core

Basic Pythia Configuration

Table 2: Basic Pythia configuration derived from our automated design-space exploration

Features	`PC+Delta, Sequence of last-4 deltas`
Prefetch Action List	{-6,-3,-1,0,1,3,4,5,10,11,12,16,22,23,30,32}
Reward Level Values	$\mathcal{R}_{AT}$=20, $\mathcal{R}_{AL}$=12, $\mathcal{R}_{CL}$=−12, $\mathcal{R}^{H}_{IN}$=−14, $\mathcal{R}^{L}_{IN}$=−8, $\mathcal{R}^{H}_{NP}$=−2, $\mathcal{R}^{L}_{NP}$=−4
Hyperparameters	$\alpha = 0.0065$, $\gamma = 0.556$, $\epsilon = 0.002$

System Parameters

Table 5: Simulated system parameters

Core	1-12 cores, 4-wide OoO, 256-entry ROB, 72/56-entry LQ/SQ
Branch Pred.	Perceptron-based [69], 20-cycle misprediction penalty
L1/L2 Caches	Private, 32KB/256KB, 64B line, 8 way, LRU, 16/32 MSHRs, 4-cycle/14-cycle round-trip latency
LLC	2MB/core, 64B line, 16 way, SHiP [133], 64 MSHRs per LLC Bank, 34-cycle round-trip latency
Main Memory	**1C:** Single channel, 1 rank/channel; **4C:** Dual channel, 2 ranks/channel; **8C:** Quad channel, 2 ranks/channel; 8 banks/rank, 2400 MTPS, 64b data-bus/channel, 2KB row buffer-/bank, tRCD=15ns, tRP=15ns, tCAS=12.5ns

Configuration of Prefetchers

Table 7: Configuration of evaluated prefetchers

SPP [78]	256-entry ST, 512-entry 4-way PT, 8-entry GHR	**6.2 KB**
Bingo [27]	2KB region, 64/128/4K-entry FT/AT/PHT	**46 KB**
MLOP [111]	128-entry AMT, 500-update, 16-degree	**8 KB**
DSPatch [30]	Same configuration as in [30]	**3.6 KB**
PPF [32]	Same configuration as in [32]	**39.3 KB**
Pythia	2 features, 2 vaults, 3 planes, 16 actions	**25.5 KB**

Evaluated Workloads

Table 6: Workloads used for evaluation

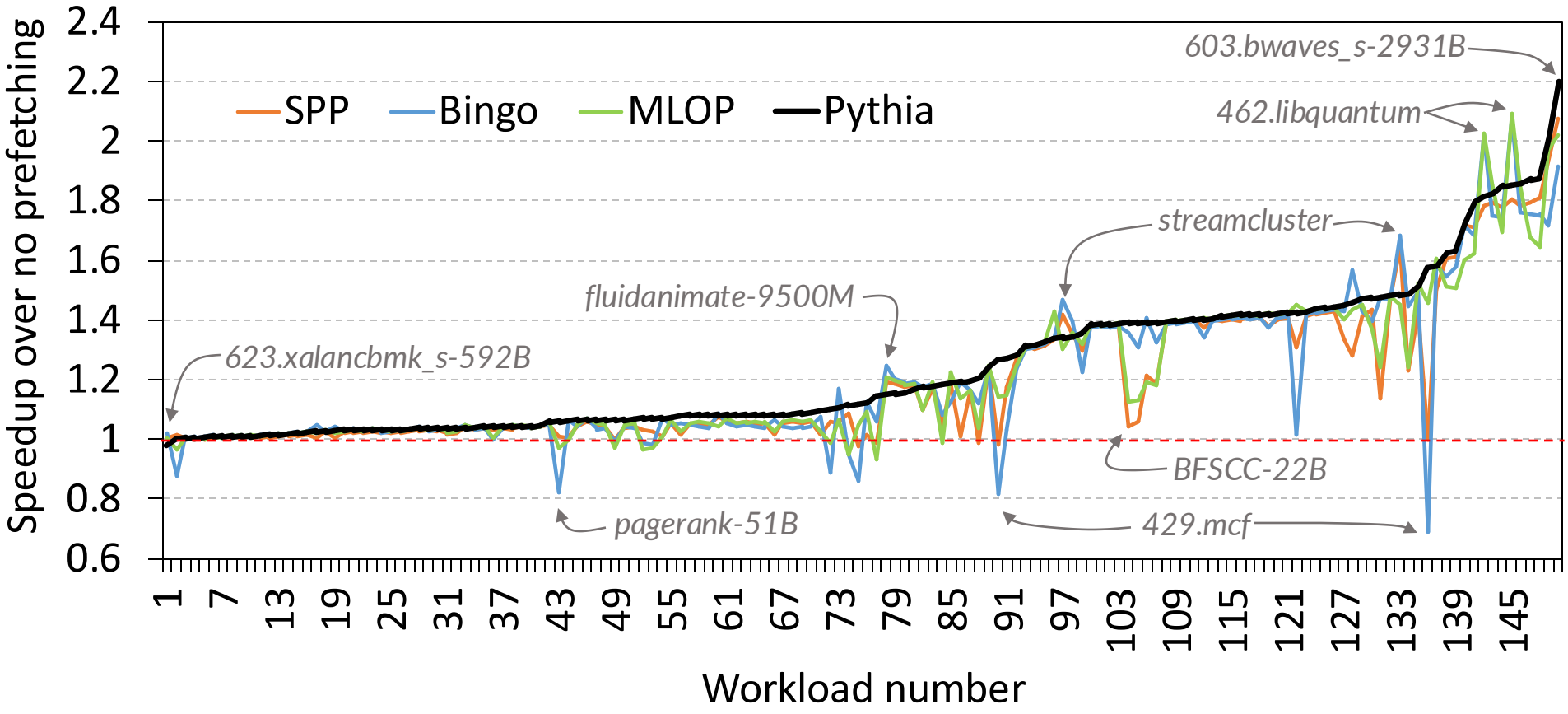
Suite	# Workloads	# Traces	Example Workloads
SPEC06	16	28	gcc, mcf, cactusADM, lbm, ...
SPEC17	12	18	gcc, mcf, pop2, fotonik3d, ...
PARSEC	5	11	canneal, facesim, raytrace, ...
Ligra	13	40	BFS, PageRank, Bellman-ford, ...
Cloudsuite	4	53	cassandra, cloud9, nutch, ...

List of Evaluated Features

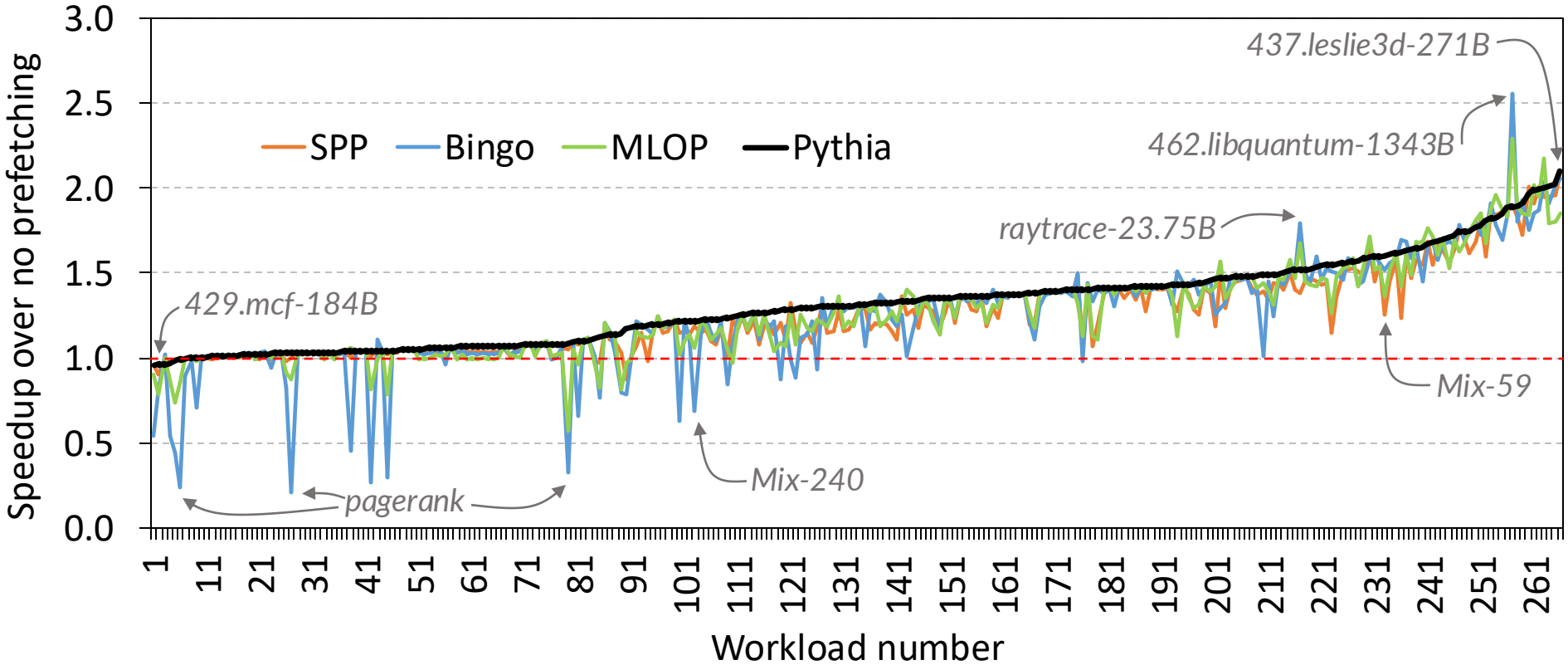
Table 3: List of program control-flow and data-flow components used to derive the list of features for exploration

Control-flow Component	Data-flow Component
(1) PC of load request (2) PC-path (XOR-ed last-3 PCs) (3) PC XOR-ed branch-PC (4) None	(1) Load cacheline address (2) Page number (3) Page offset (4) Load address delta (5) Sequence of last-4 offsets (6) Sequence of last-4 deltas (7) Offset XOR-ed with delta (8) None

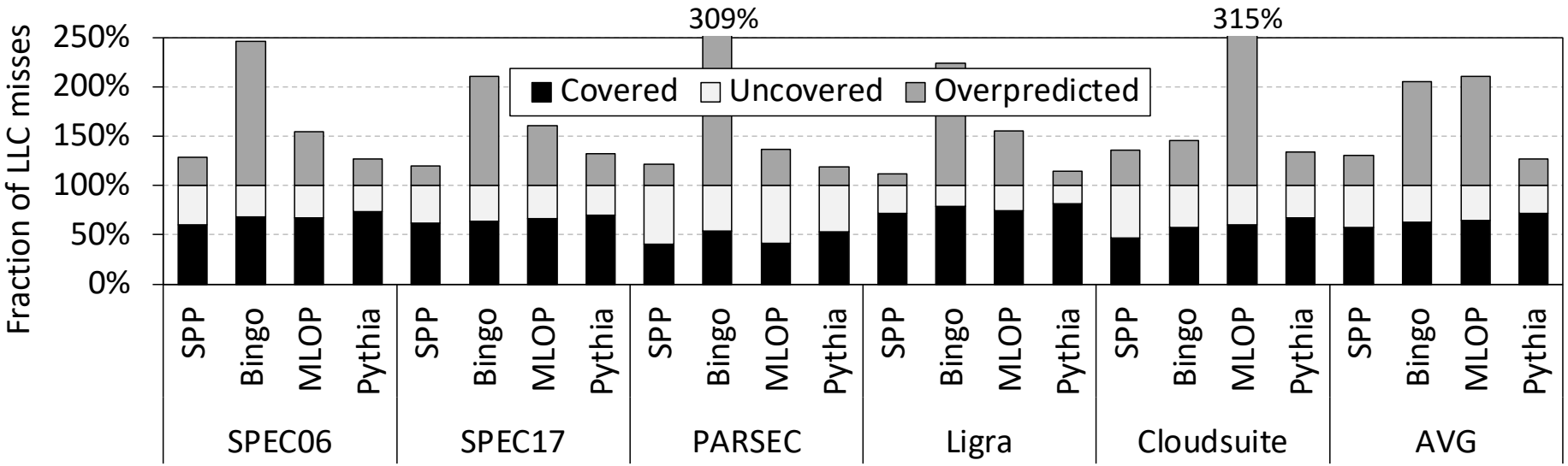
Performance S-curve: Single-core



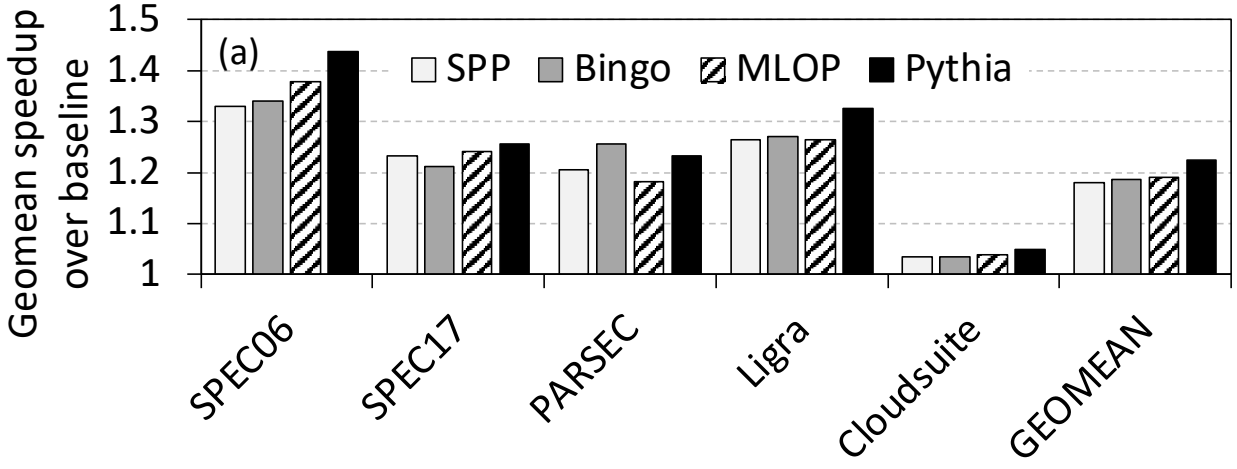
Performance S-curve: Four-core

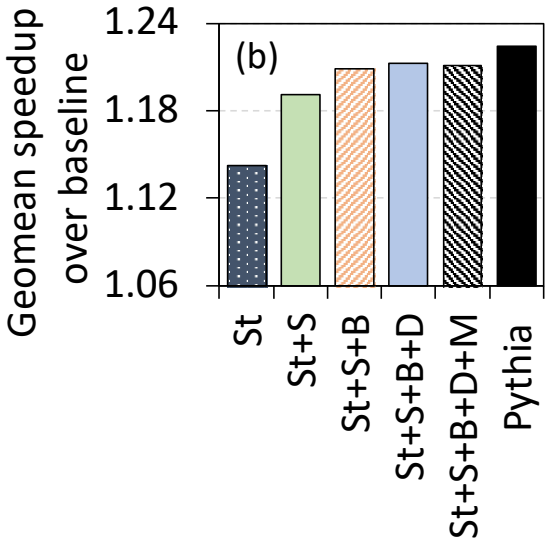


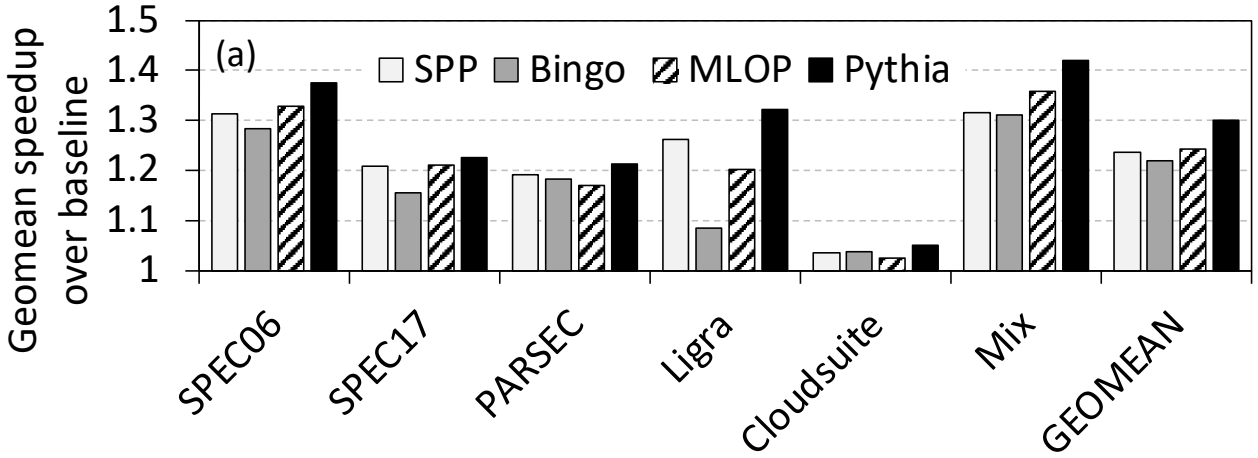
Single-core Coverage & Overprediction

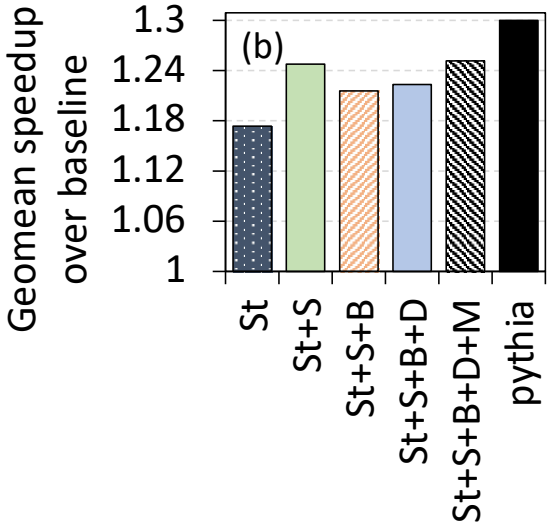


Detailed Performance

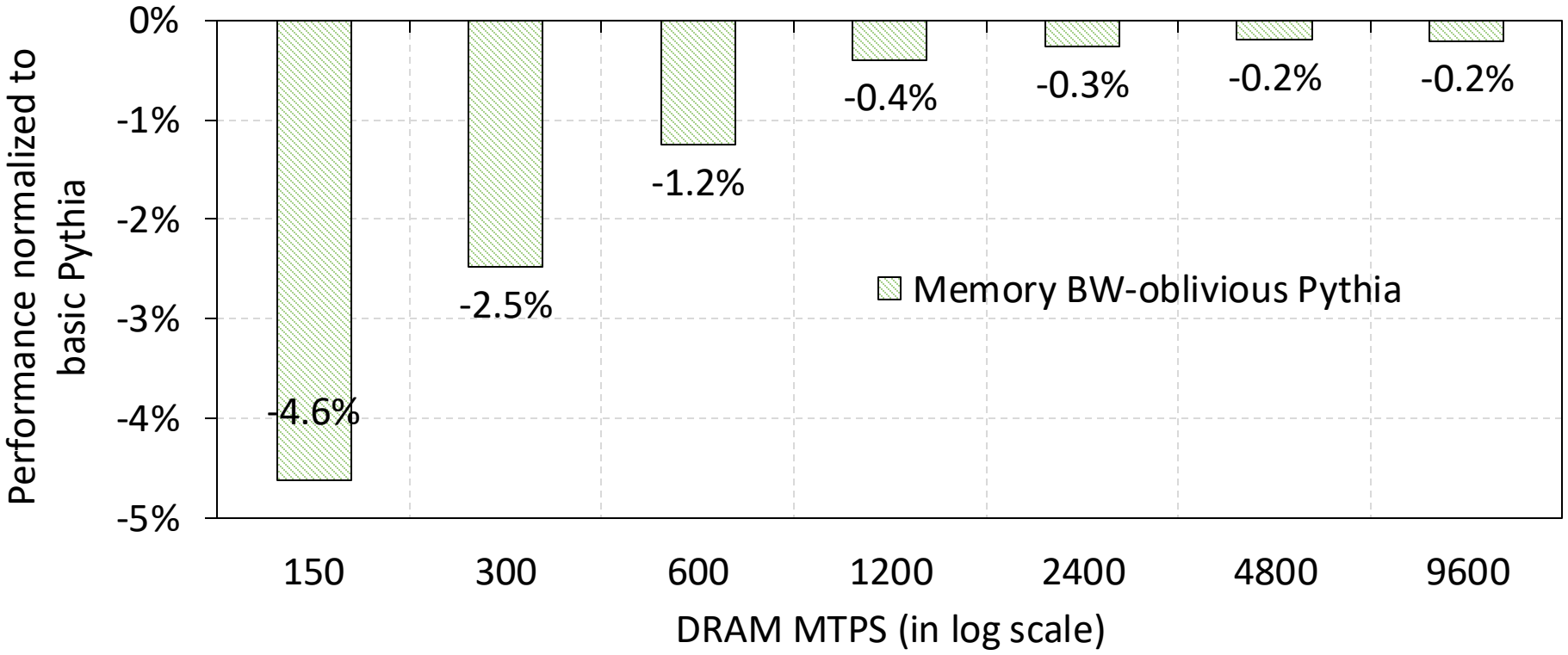








Benefit of Bandwidth Awareness



Case Study

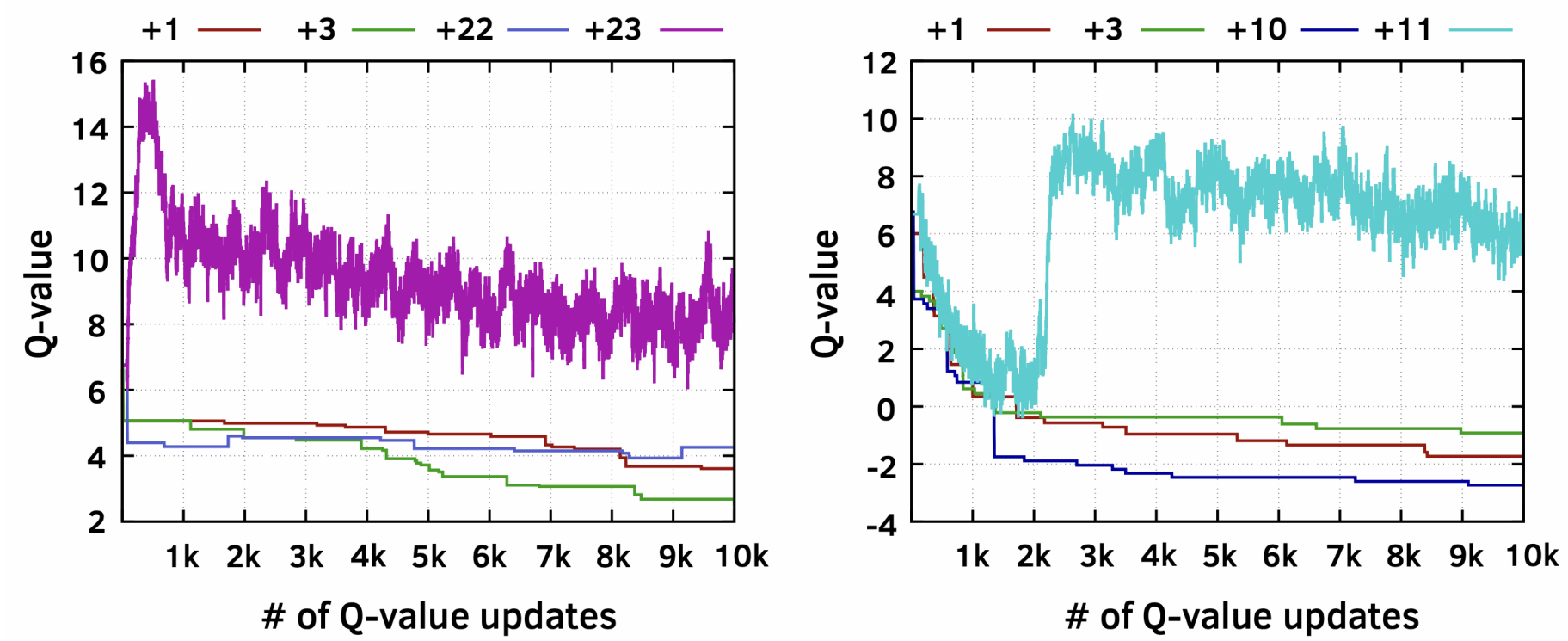


Figure 13: Q-value curves of PC+Delta feature values (a) 0x436a81+0 and (b) 0x4377c5+0 in 459.GemsFDTD-1320B.

Customizing Rewards

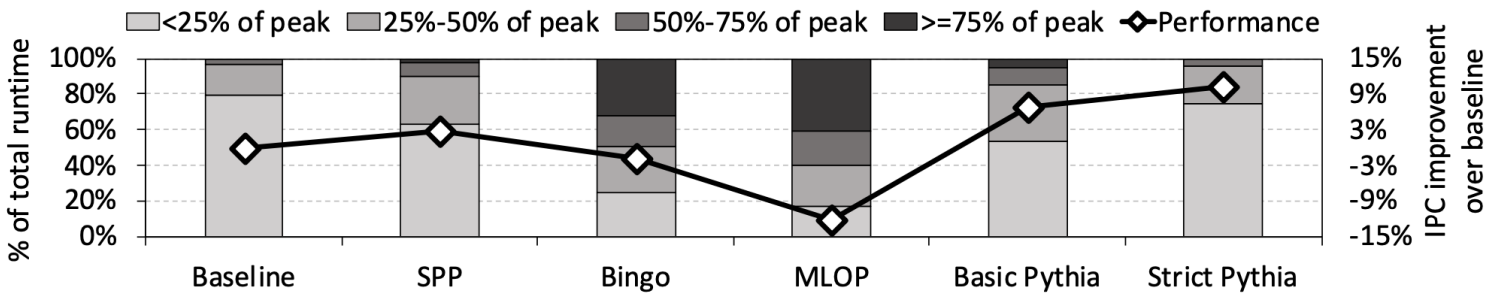


Figure 14: Performance and main memory bandwidth usage of prefetchers in Ligra-CC.

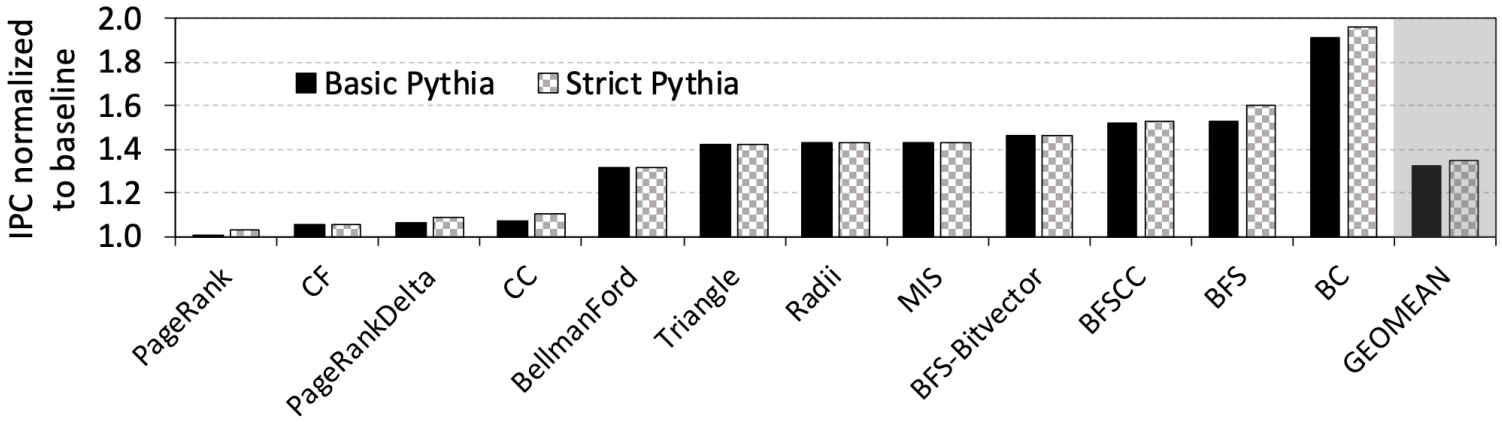


Figure 15: Performance of the basic and strict Pythia configurations on the Ligra workload suite.

Customizing Features

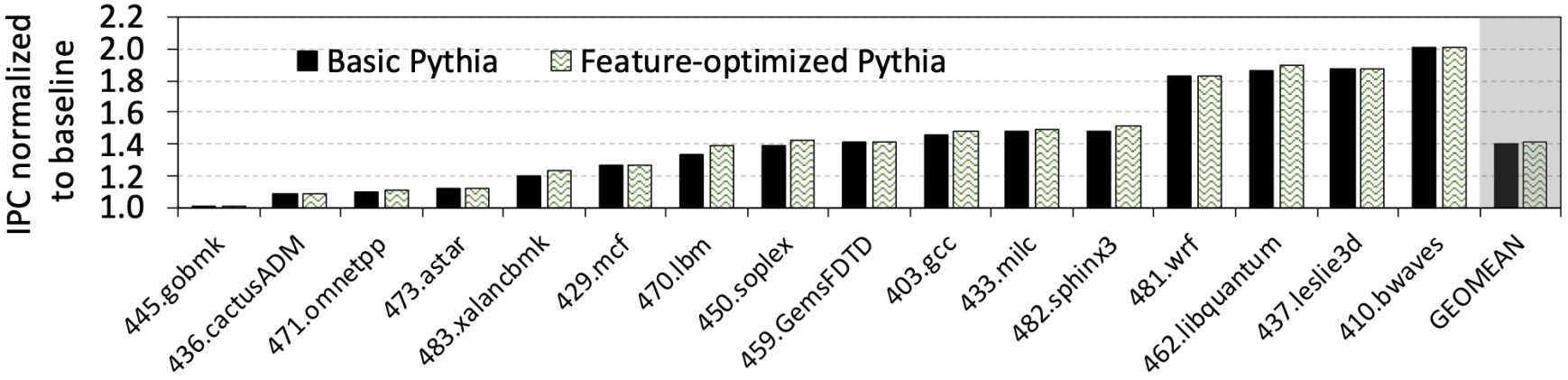


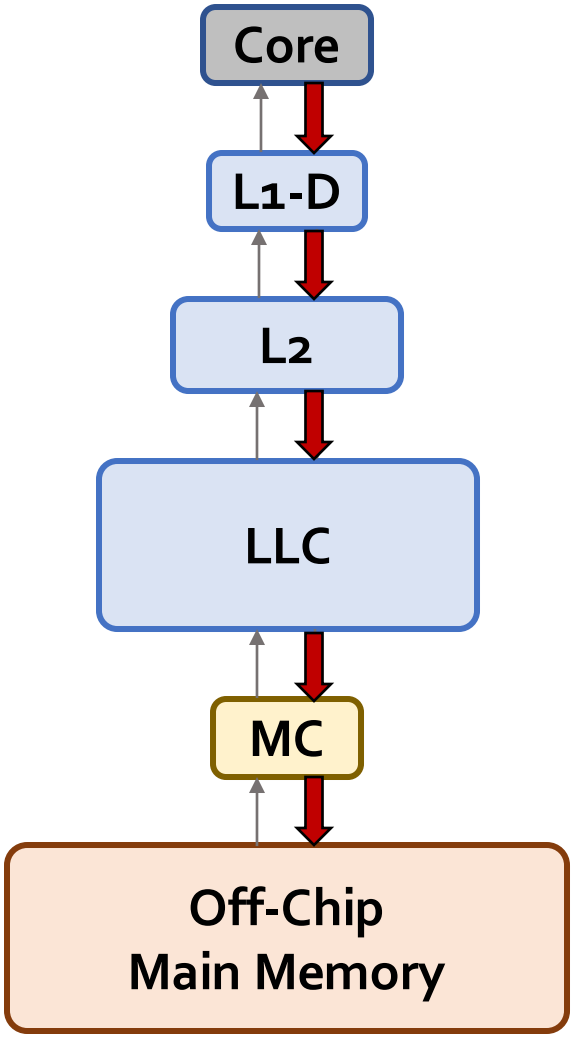
Figure 16: Performance of the basic and feature-optimized Pythia on the SPEC CPU2006 suite.

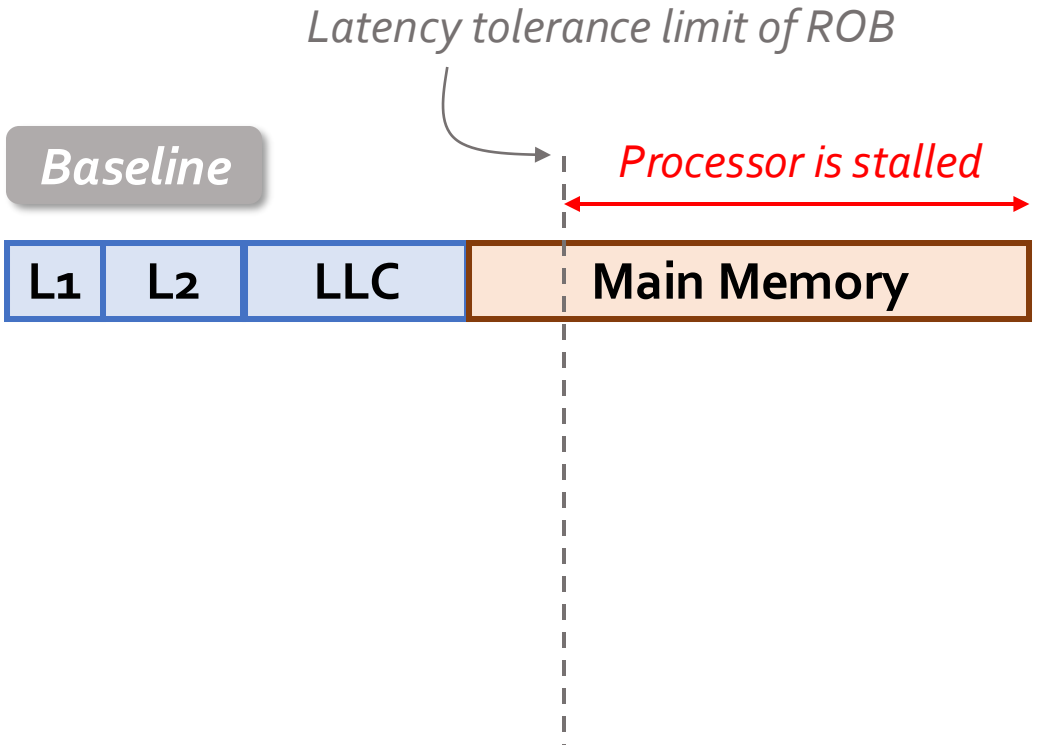
Hermes Discussions

Hermes Discussions

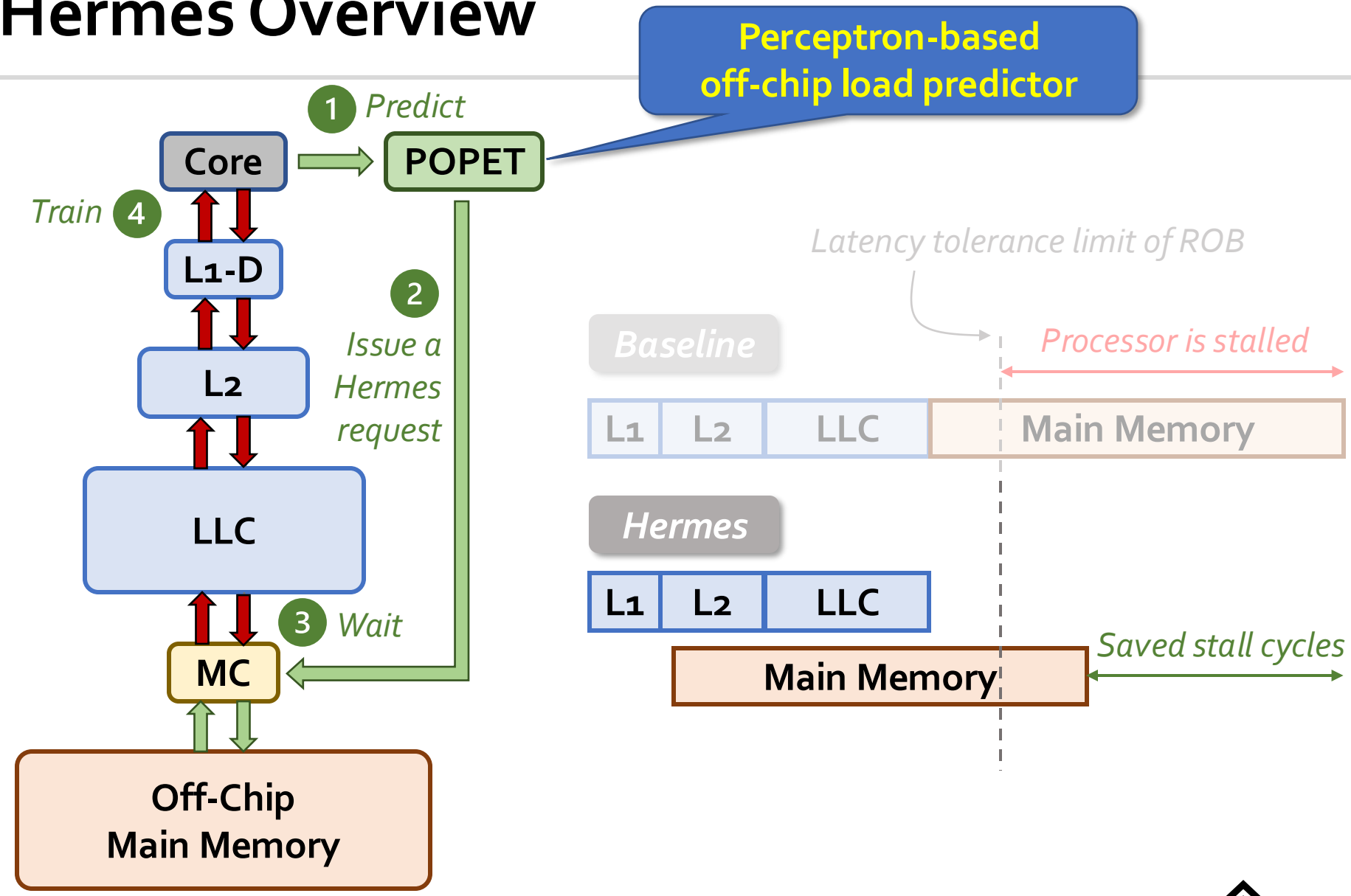
- **FAQs**
  - What are the selected set of program features?
  - Can you provide some intuition on why these features work?
  - What happens in case of a misprediction?
  - What's the performance headroom for off-chip prediction?
  - Do you see a variance of different features in final prediction accuracy?
- **Simulation Methodology**
  - System parameters
  - Evaluated workloads
- **More Results**
  - Percentage of off-chip requests
  - Reduction in stall cycles by reducing the critical path
  - Fraction of off-chip load requests
  - Accuracy and coverage of POPET
  - Effect of different features
  - Are all features required?
  - 1C performance
  - 1C performance line graph
  - 1C performance against prior predictors
  - Effect on stall cycles
  - 8C performance
  - Sensitivity:
    - Hermes request issue latency
    - Cache hierarchy access latency
    - Activation threshold
    - ROB size
    - LLC size
  - Power overhead
  - Accuracy without prefetcher
  - Main memory request overhead with different prefetchers

Hermes Overview





Hermes Overview



Designing the Off-Chip Load Predictor

History-based prediction

HMP [Yoaz+, ISCA'99] for the L1-D cache

Using branch-predictor-like hybrid predictor: Global, Gshare, and GSkew

POPET provides

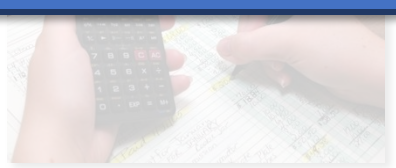
both **higher accuracy** and **higher performance**

than predictors inspired from these previous works

- Metadata size increases with cache hierarchy size

May need to track **all** cache operations

- Gets complex depending on the cache hierarchy configuration (e.g., inclusivity, bypassing,...)

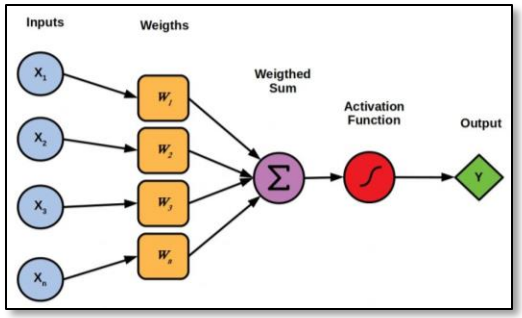


Learning from program behavior

Correlate different program features with off-chip loads

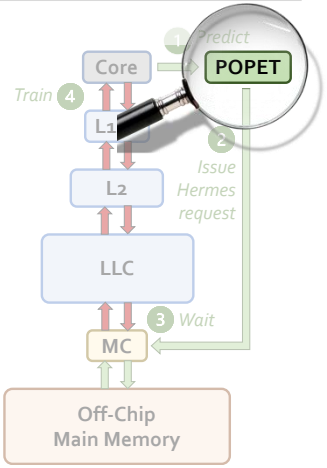
Low storage overhead

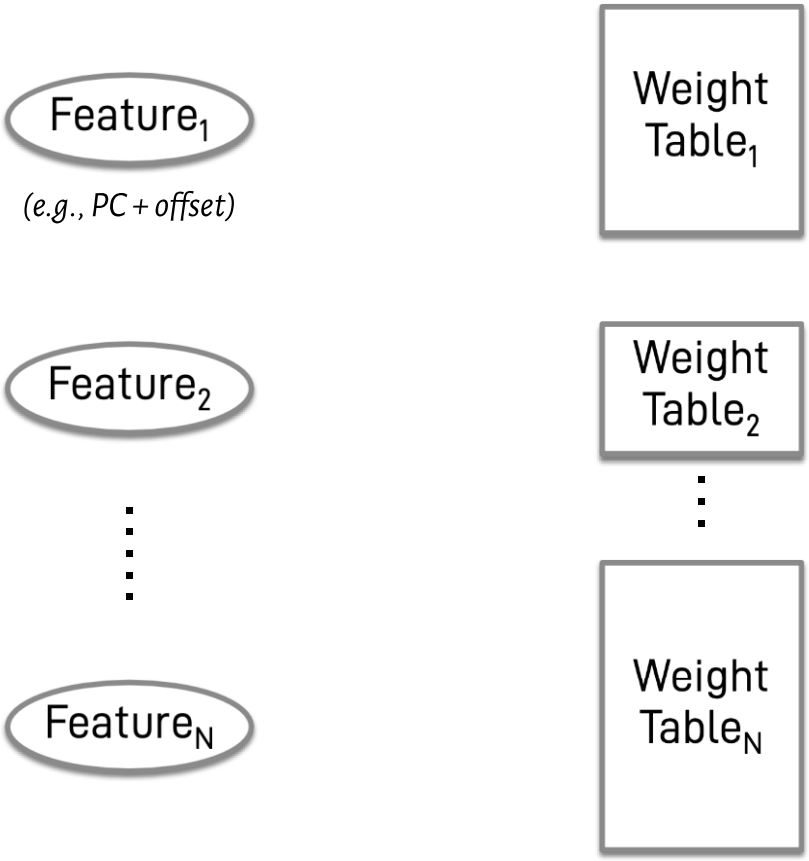
Low design complexity



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

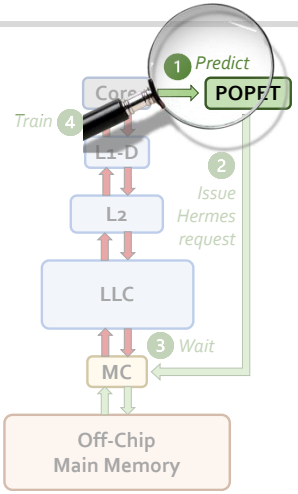


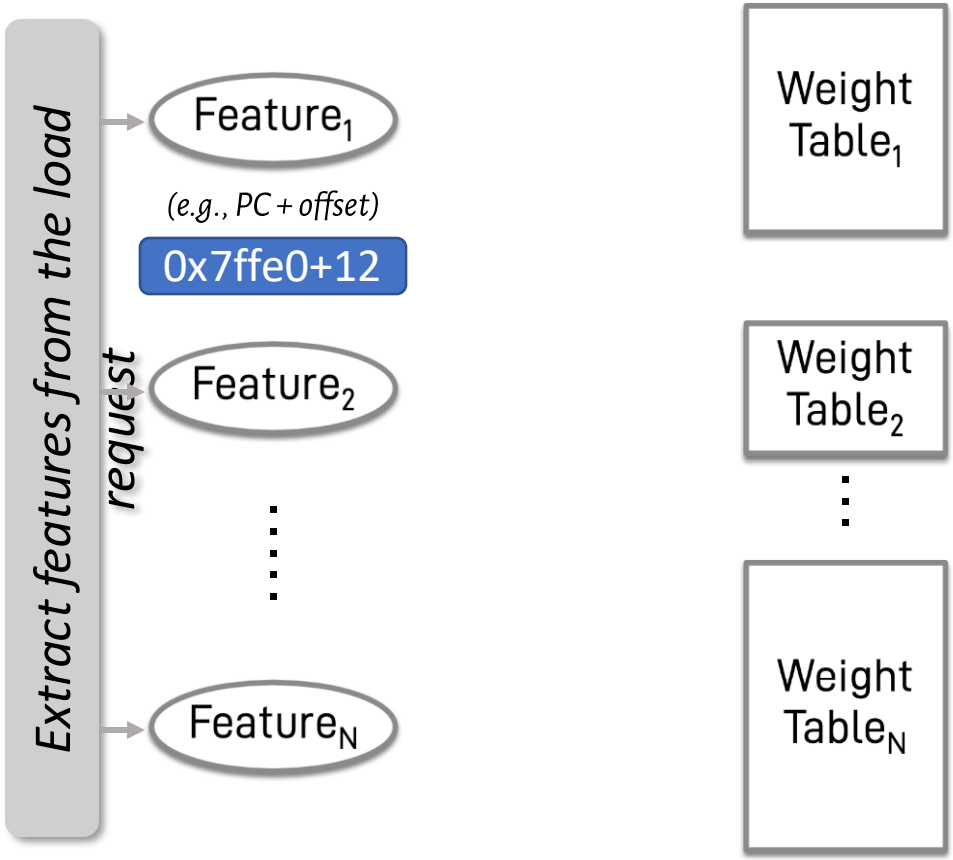


[1] D. Tarjan and K. Skadron, "Merging Path and Gshare Indexing in Perceptron Branch Prediction," TACO, 2005

Predicting using POPET

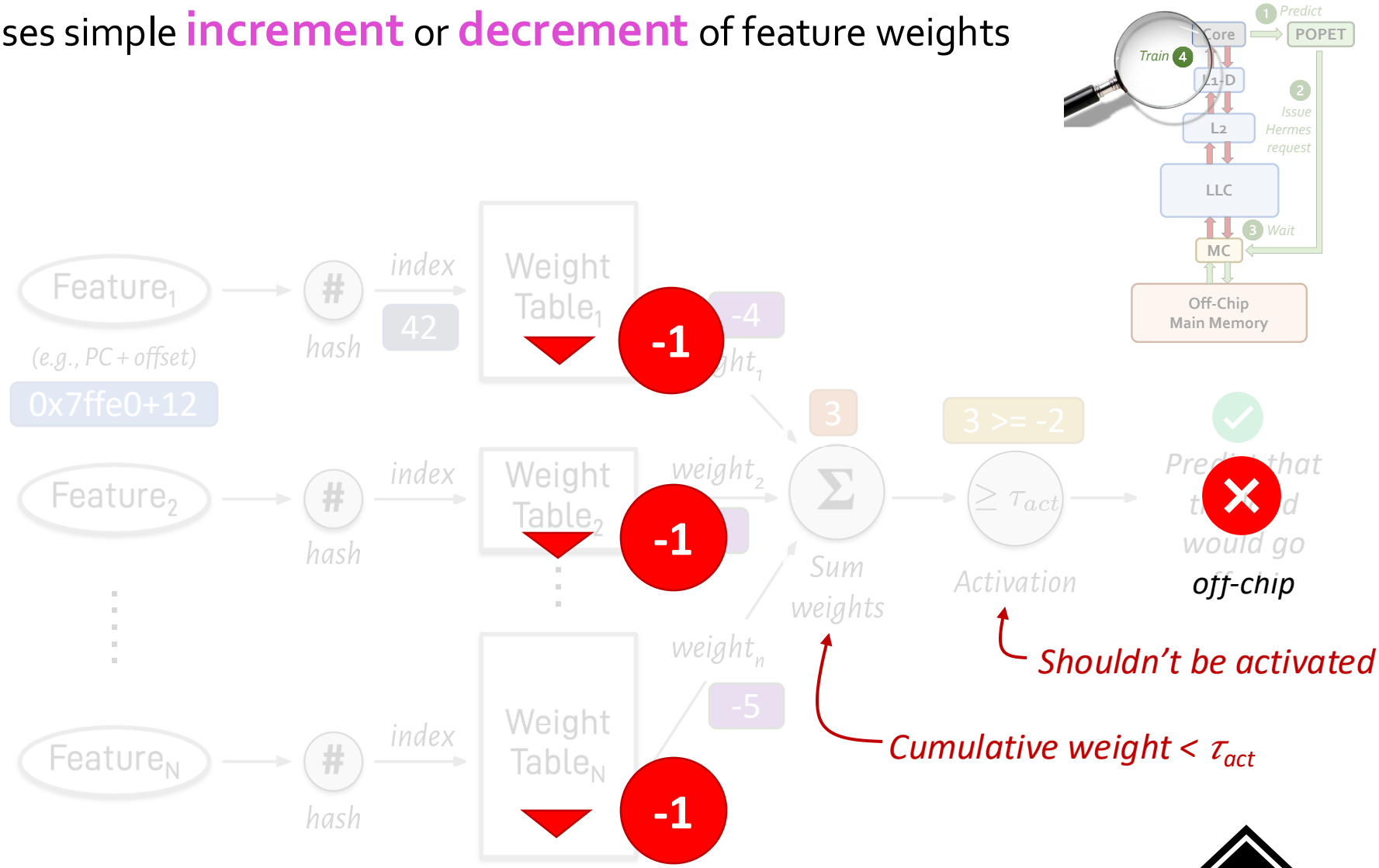
- Uses simple **table lookups**, **addition**, and **comparison**



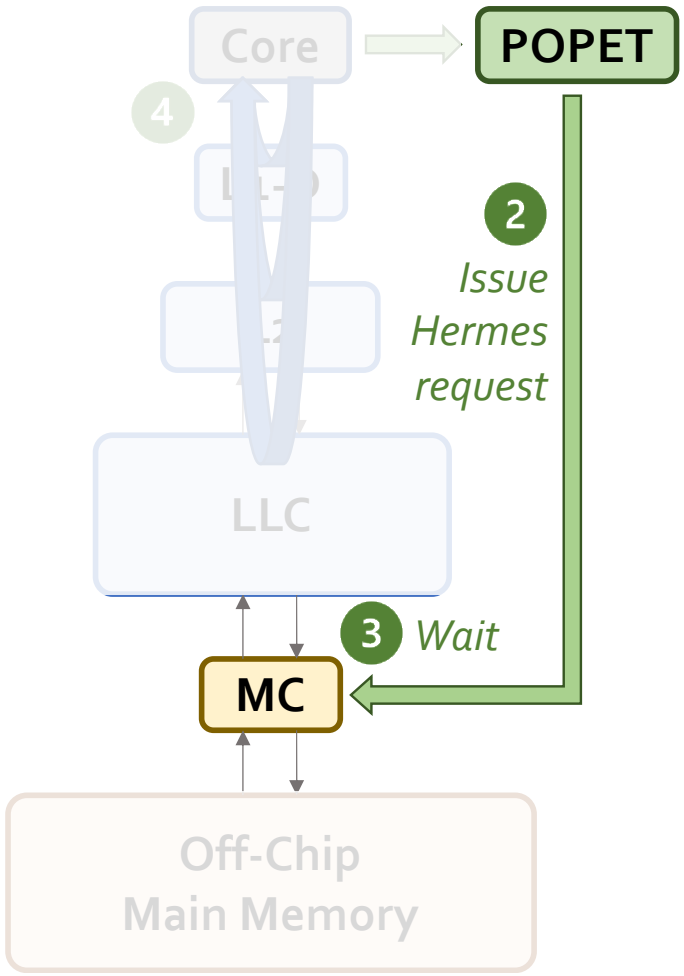


Training POPET

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



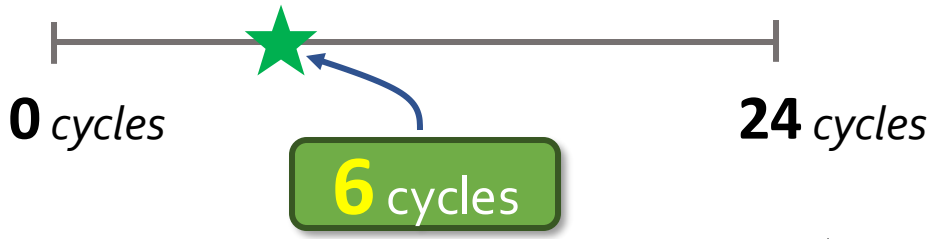
Latency Configuration



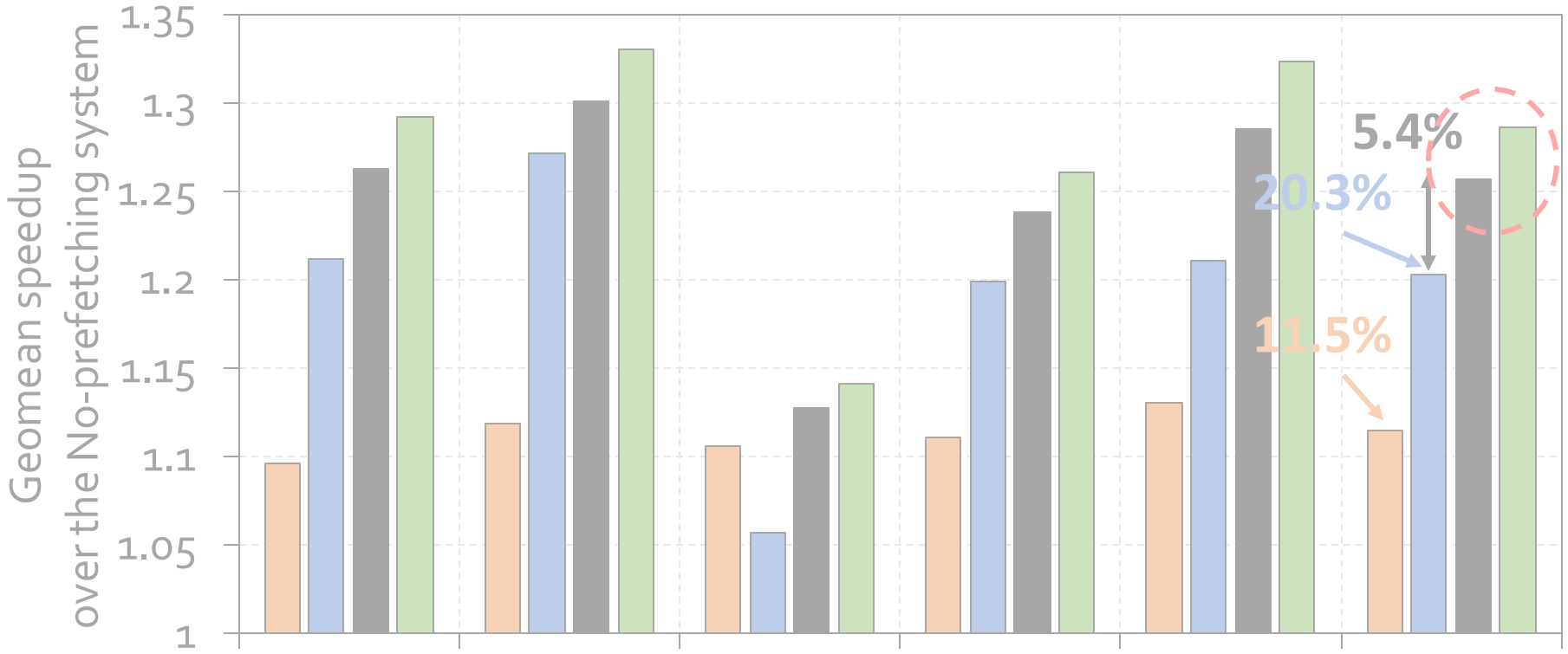
- **Cache *round-trip* latency**
  - L1-D: **5** cycles
  - L2: **15** cycles
  - LLC: **55** cycles

- **Hermes request issue latency**
  (incurred after address translation)

  Depends on
  - **Interconnect** between POPET and MC



Single-Core Performance Improvement

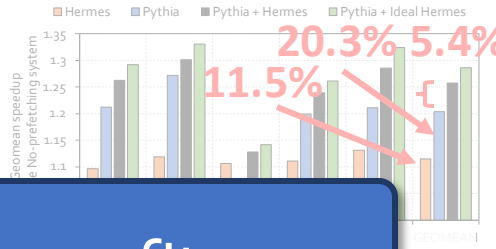


Hermes alone provides nearly

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

with only **1/3rd** storage overhead

Increase in Main Memory Requests

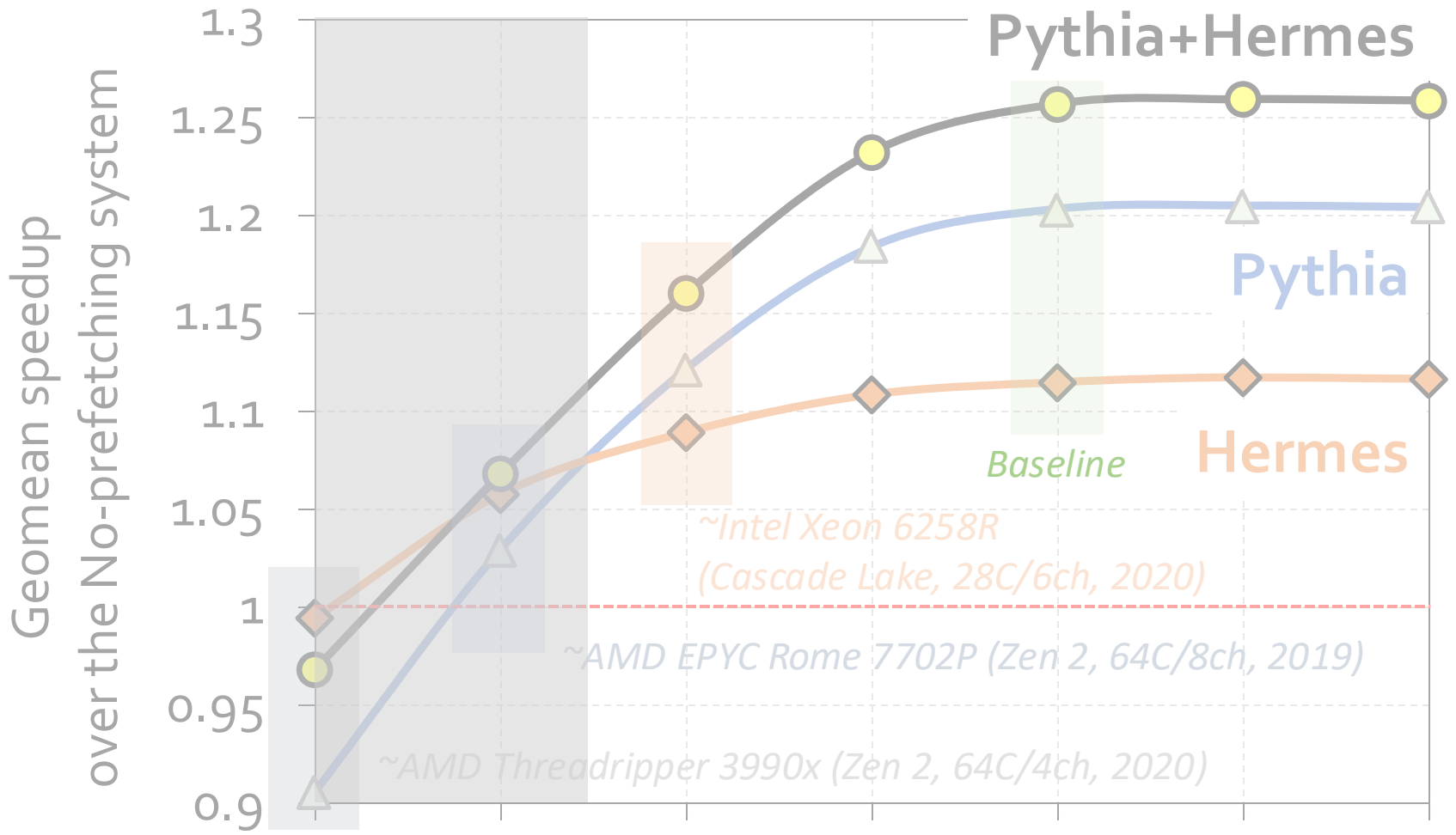


For **every 1% performance** benefit,
increase in main memory requests

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

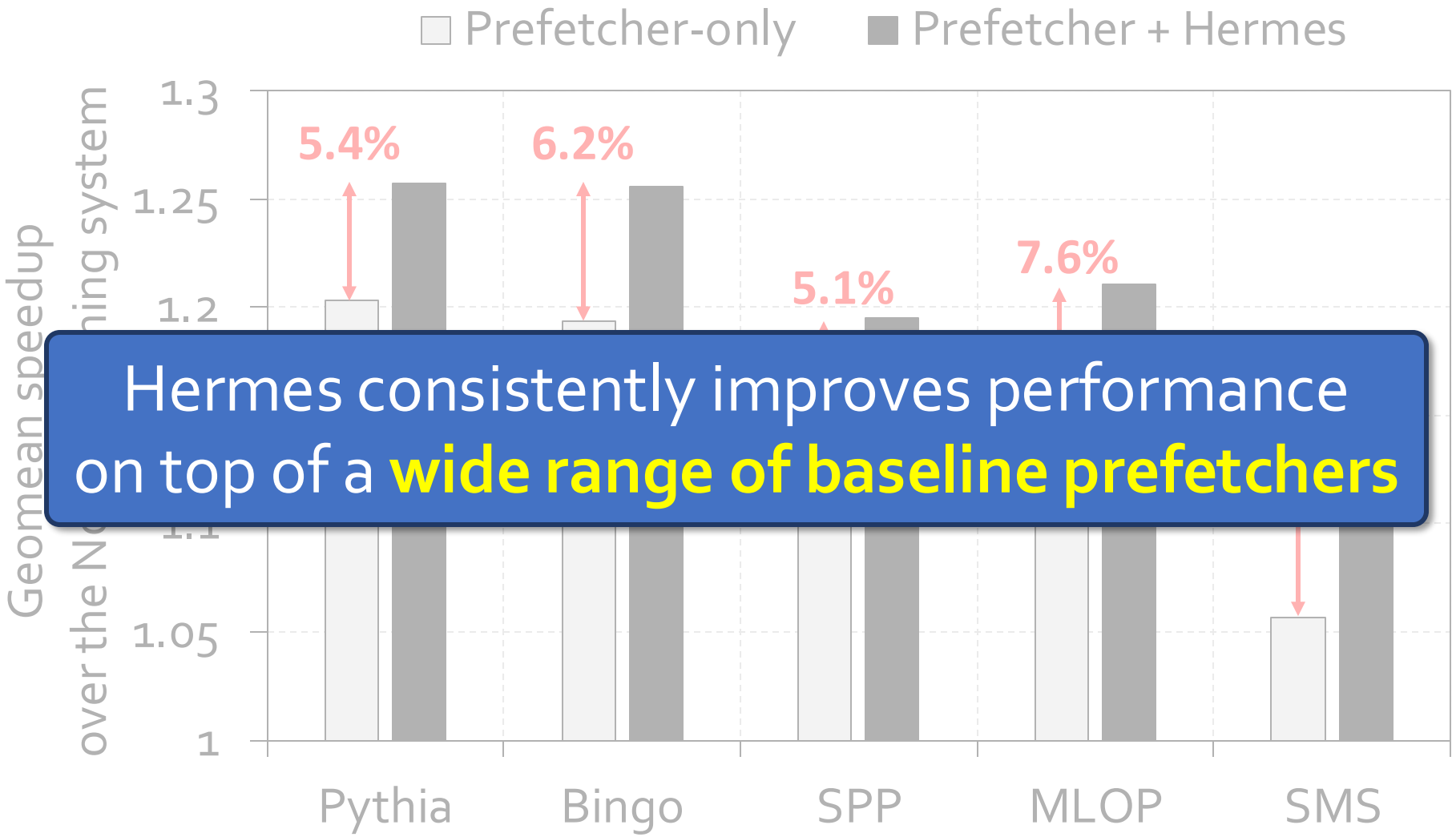
Hermes is more **bandwidth-efficient**
than even an efficient prefetcher like Pythia

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

More in the Paper

- Performance sensitivity to:
  - Cache hierarchy access latency
  - Hermes request issue latency
  - Activation threshold
  - ROB size (in extended version on arXiv)
  - LLC size (in extended version on arXiv)

- **Accuracy, coverage, and performance analysis** against **HMP** and **TTP**

- Understanding **usefulness of each program feature**

- Effect on **stall cycle reduction**

- **Performance** analysis on an **eight-core** system

Initial Set of Program Features

Features without control-flow information	Features with control-flow information
1. Load virtual address	8. Load PC
2. Virtual page number	9. PC ⊕ load virtual address
3. Cacheline offset in page	10. PC ⊕ virtual page number
4. First access	11. PC ⊕ cacheline offset
5. Cacheline offset + first access	12. PC + first access
6. Byte offset in cacheline	13. PC ⊕ byte offset
7. Word offset in cacheline	14. PC ⊕ word offset
	15. Last-4 load PCs
	16. Last-4 PCs

Selected Set of Program Features

Five features

- PC $\oplus$ cacheline offset
- PC $\oplus$ byte offset
- PC + first access
- Cacheline offset + first access
- Last-4 load PCs

A **binary hint** that represents whether or not a cacheblock has been recently touched

When A Feature Works/Does Not Work?

Trace: 462.libquantum-1343B

PC: 0x401442

... Cacheline #42 Cacheline #43 ...

Without prefetcher

- PC + first access
- Cacheline offset + first access

With a simple stride prefetcher

- Cacheline offset + first access

What Happens in case of a Misprediction?

- Two cases of mispredictions:

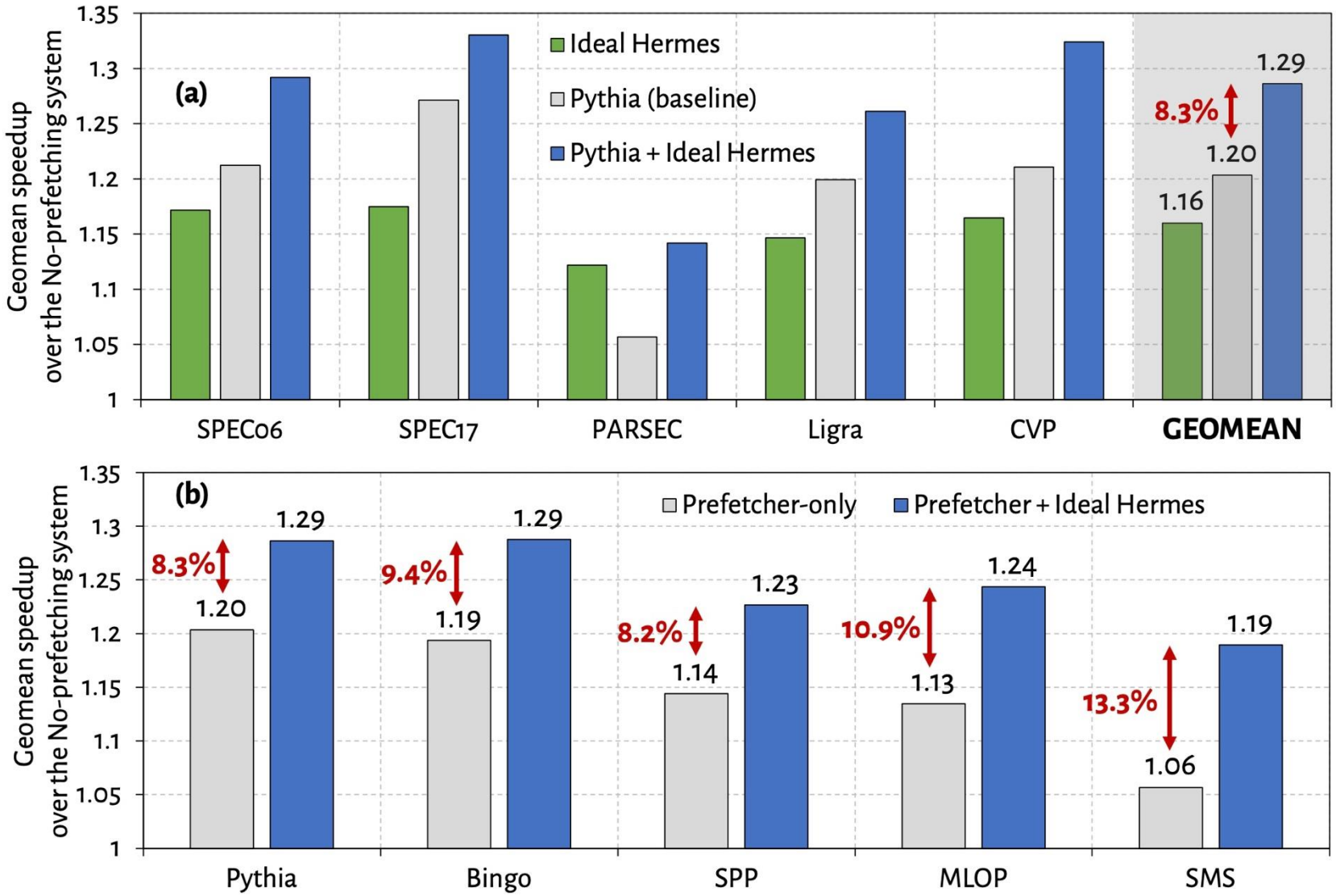
- **Predicted on-chip but actually goes off-chip**
  - Loss of performance improvement opportunity

No need for misprediction detection and recovery

- **Predicted off-chip but actually is on-chip**
  - Memory controller forwards the data to LLC **if and only if** a load to the same address have already missed LLC and arrived at the memory controller

No need for misprediction detection and recovery

Performance Headroom of Off-Chip Prediction



System Parameters

Table 4: Simulated system parameters

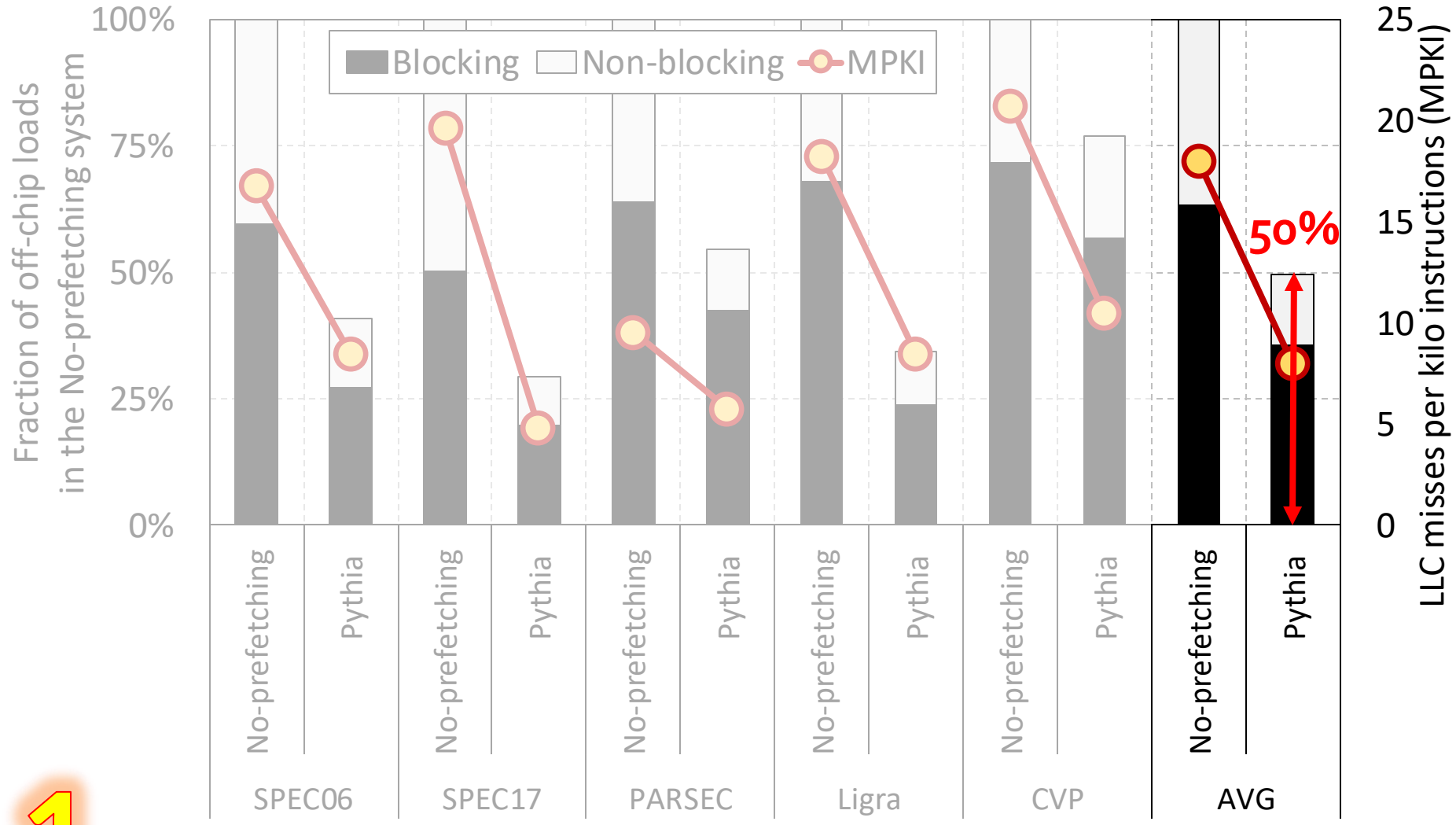
Core	1 and 8 cores, 6-wide fetch/execute/commit, 512-entry ROB, 128/72-entry LQ/SQ, Perceptron branch predictor [61] with 17-cycle misprediction penalty
L1/L2 Caches	Private, 48KB/1.25MB, 64B line, 12/20-way, 16/48 MSHRs, LRU, 5/15-cycle round-trip latency [25]
LLC	3MB/core, 64B line, 12 way, 64 MSHRs/slice, SHiP [122], 55-cycle round-trip latency [24, 25], **Pythia** prefetcher [32]
Main Memory	**1C:** 1 channel, 1 rank per channel; **8C:** 4 channels, 2 ranks per channel; 8 banks per rank, DDR4-3200 MTPS, 64b data-bus per channel, 2KB row buffer per bank, tRCD=12.5ns, tRP=12.5ns, tCAS=12.5ns
Hermes	**Hermes-O/P**: 6/18-cycle Hermes request issue latency

Evaluated Workloads

Table 5: Workloads used for evaluation

Suite	#Workloads	#Traces	Example Workloads
SPEC06	14	22	gcc, mcf, cactusADM, lbm, ...
SPEC17	11	23	gcc, mcf, pop2, fotonik3d, ...
PARSEC	4	12	canneal, facesim, raytrace, ...
Ligra	11	20	BFS, PageRank, Radii, ...
CVP	33	33	integer, floating-point, server, ...

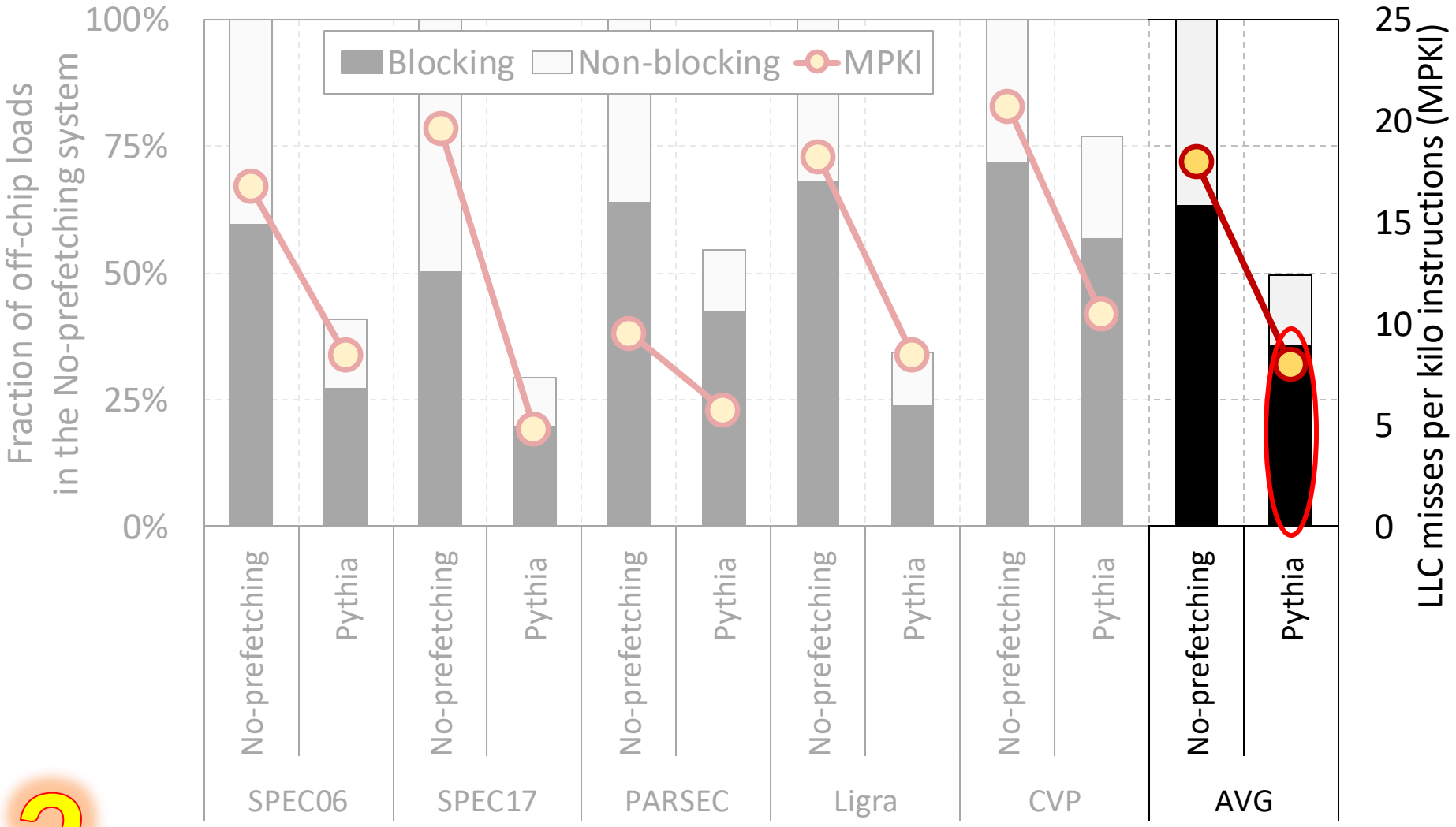
Observation: Not All Off-Chip Loads are Prefetched



1

Nearly **50%** of the loads are still **not prefetched**

Observation: Not All Off-Chip Loads are Prefetched

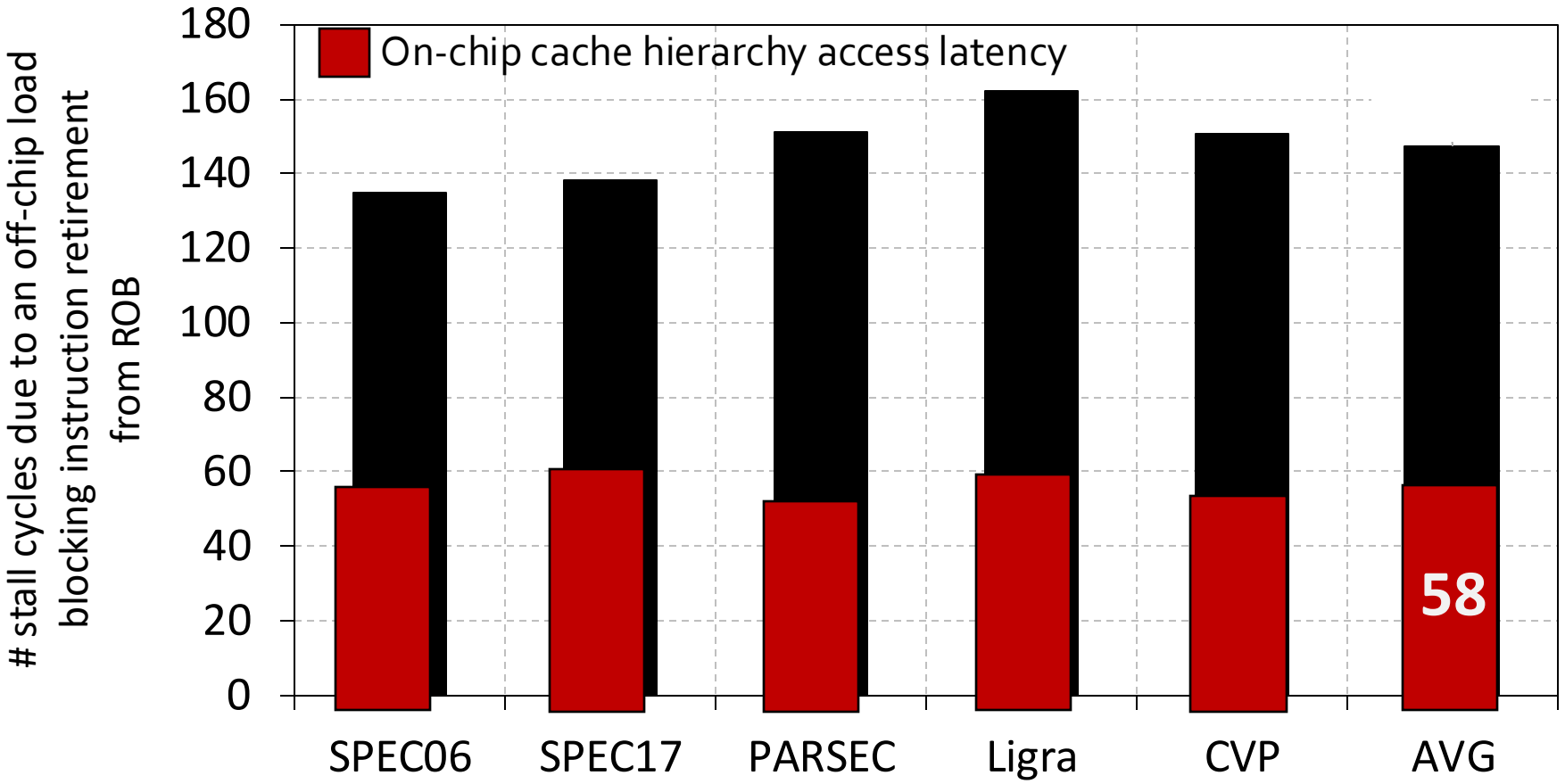


2

70% of these off-chip loads **blocks ROB**

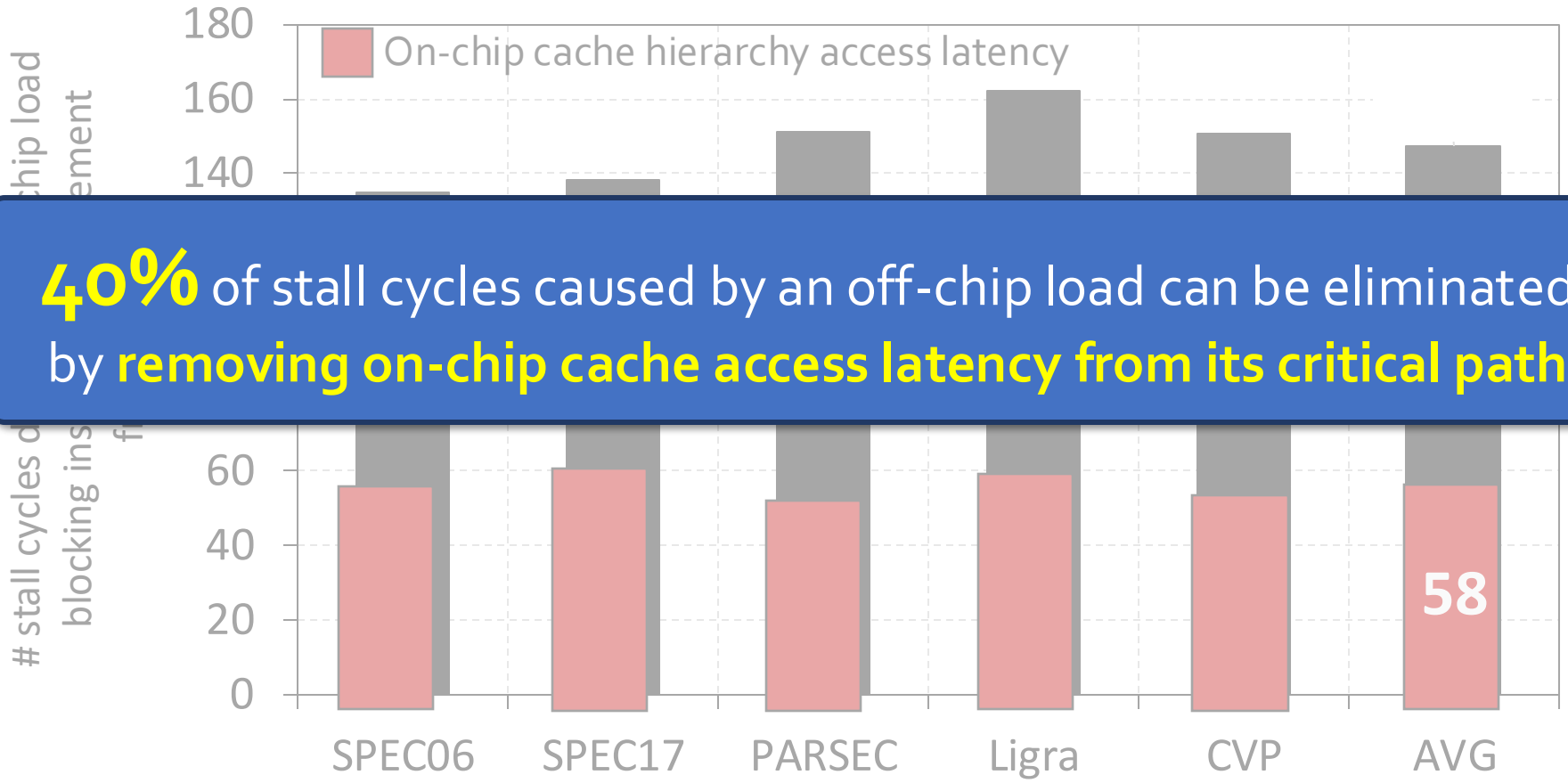
Observation: With Large Cache Comes Longer Latency

- On-chip cache access latency significantly contributes to the latency of an off-chip load



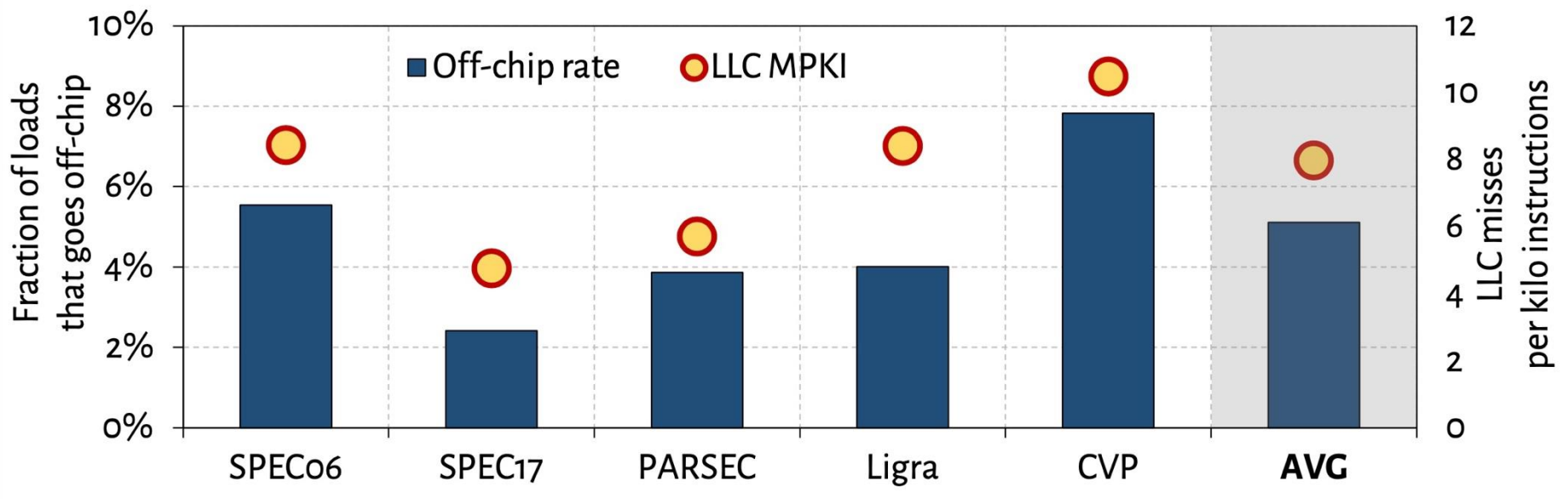
Observation: With Large Cache Comes Longer Latency

- On-chip cache access latency significantly contributes to the latency of an off-chip load

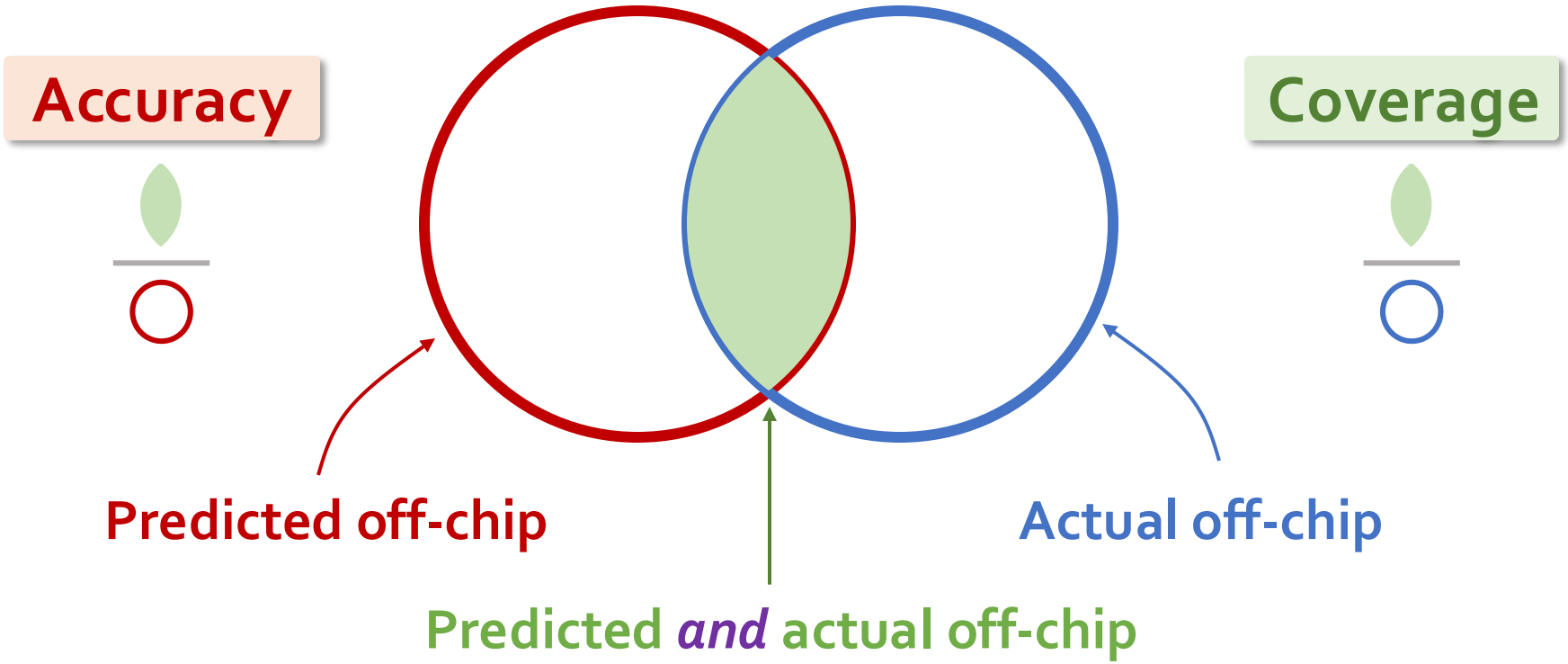


40% of stall cycles caused by an off-chip load can be eliminated by **removing on-chip cache access latency from its critical path**

What Fraction of Load Requests Goes Off-Chip?

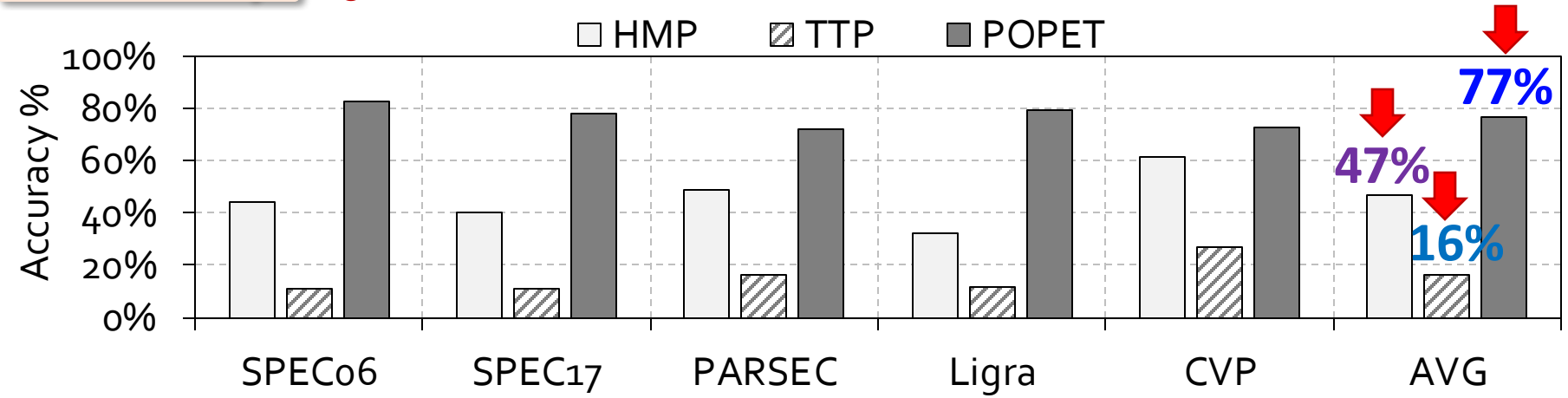


Off-Chip Prediction Quality: *Defining Metrics*

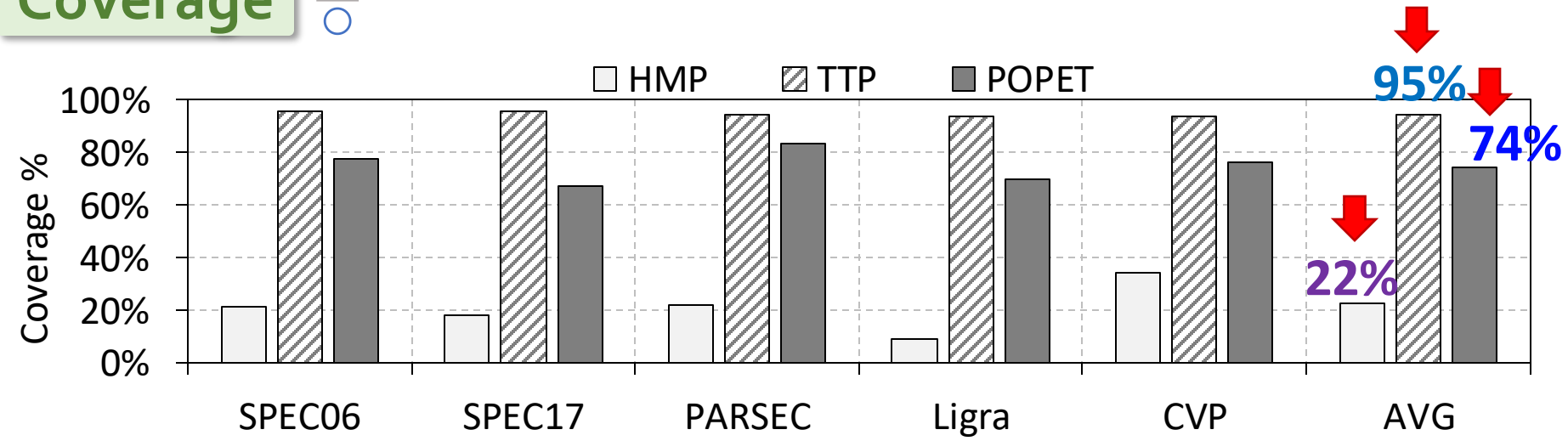


Off-Chip Prediction Quality: *Analysis*

Accuracy

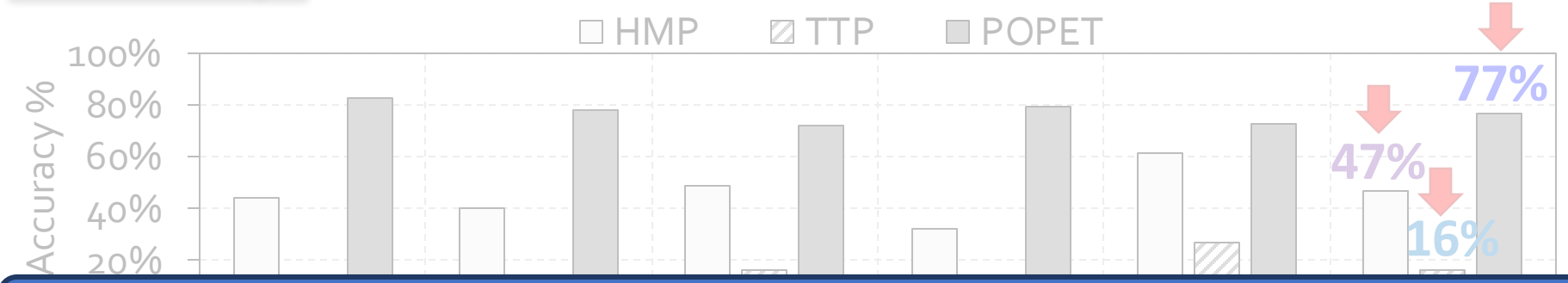


Coverage

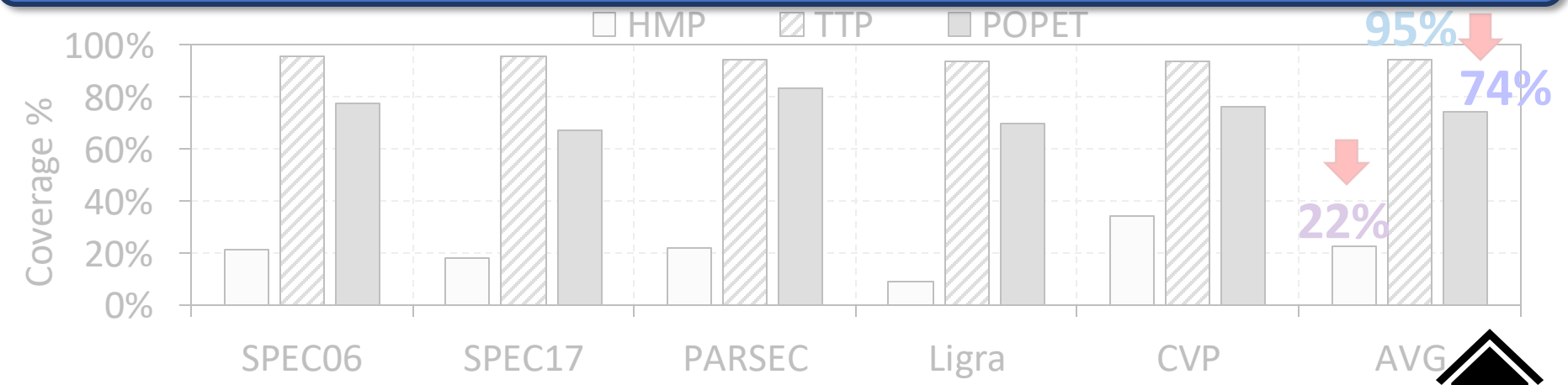


Off-Chip Prediction Quality: *Analysis*

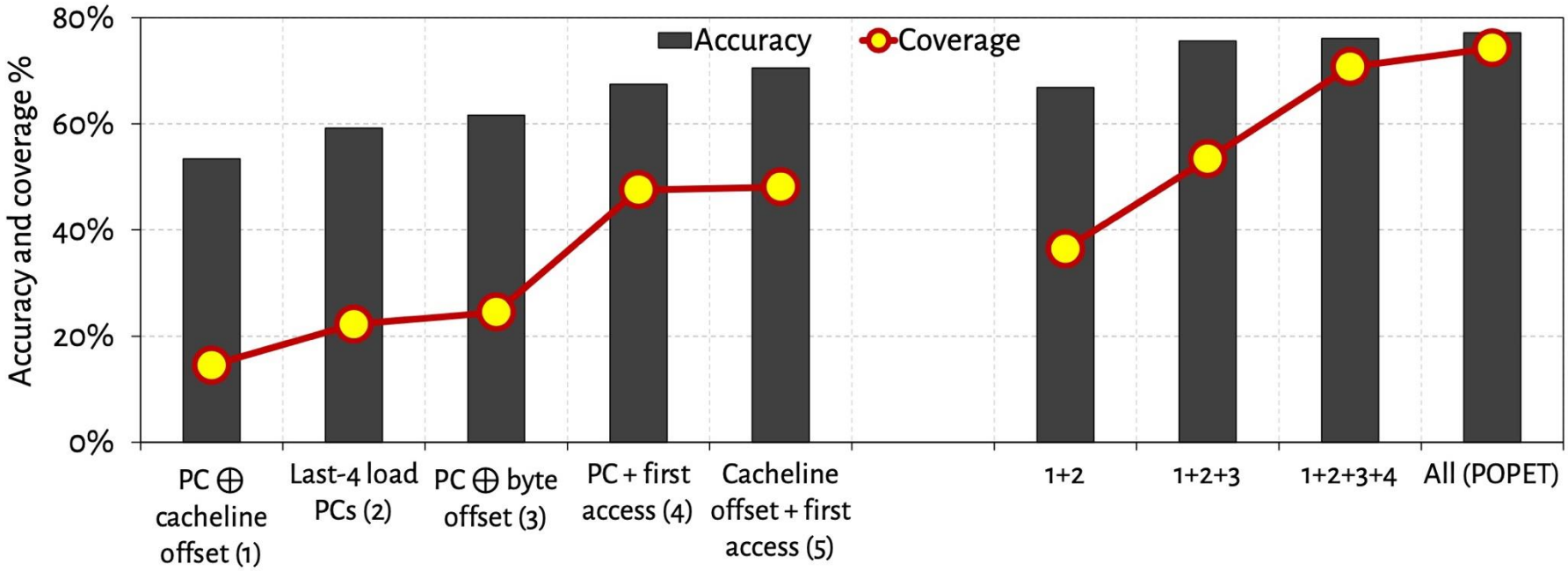
Accuracy



POPET provides off-chip predictions with **high-accuracy** and **high-coverage**

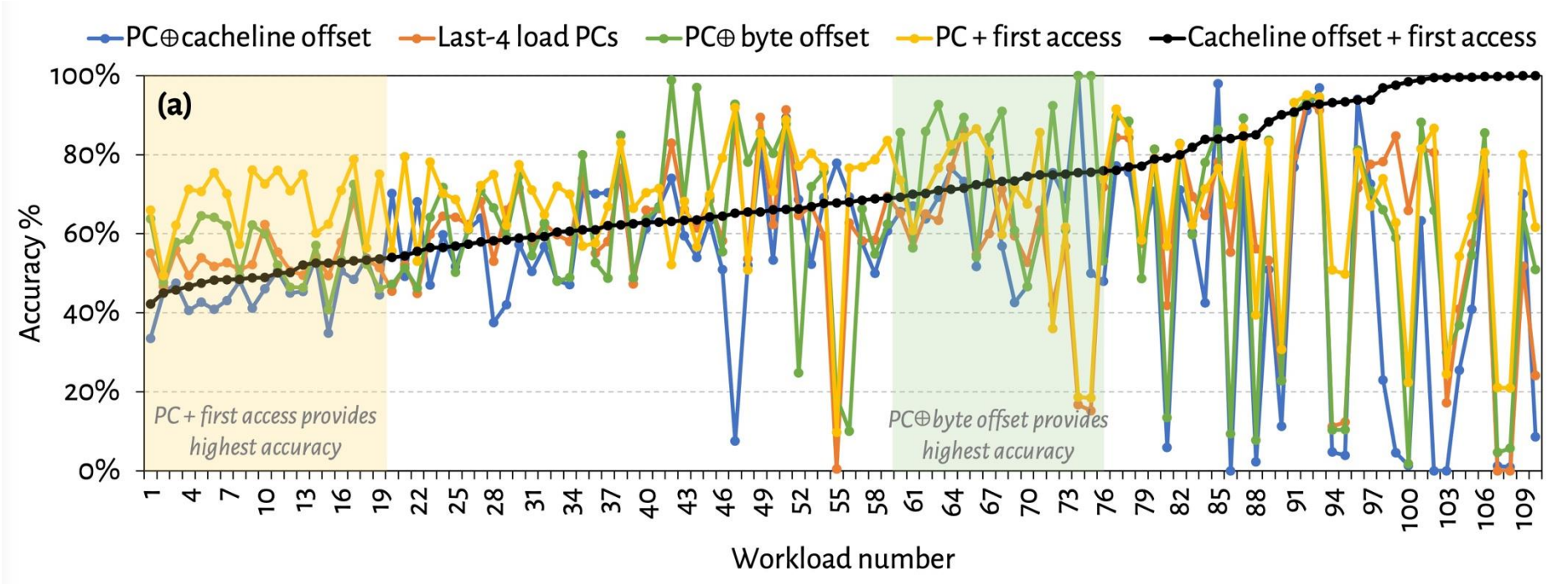


Effect of Different Features



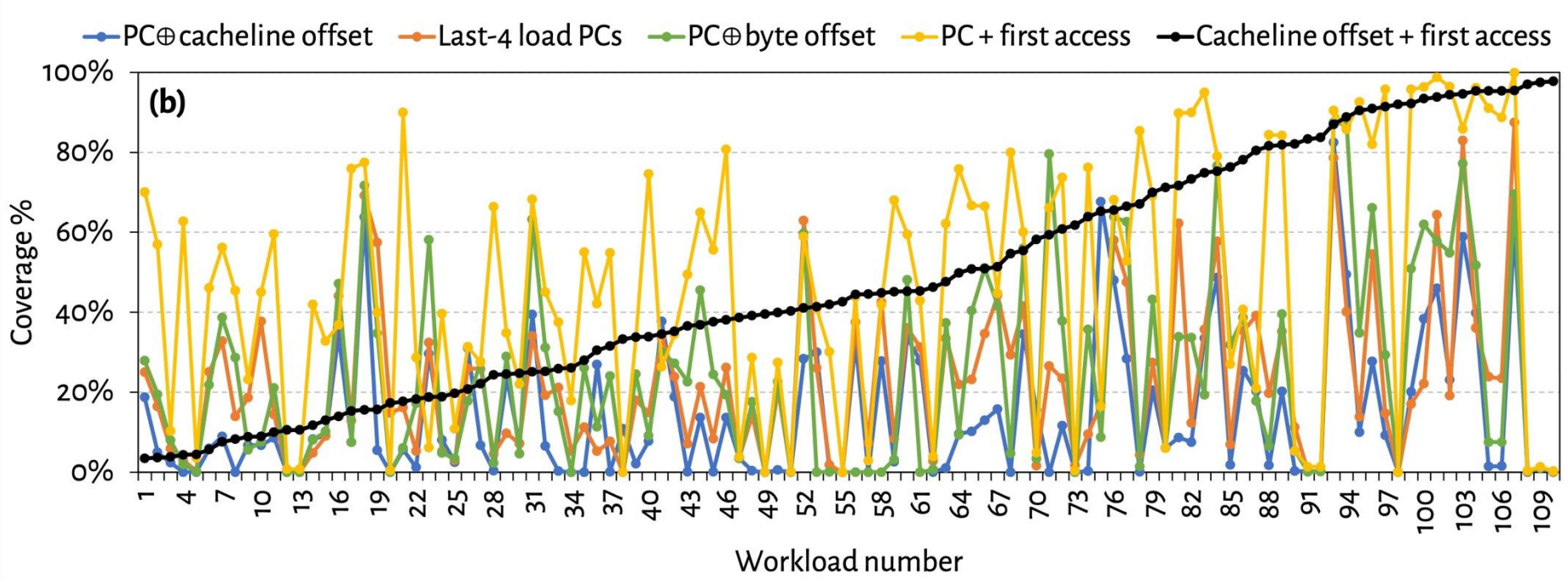
Combination of features provides both **higher accuracy and higher coverage** than any individual feature

Are All Features Required? (1)



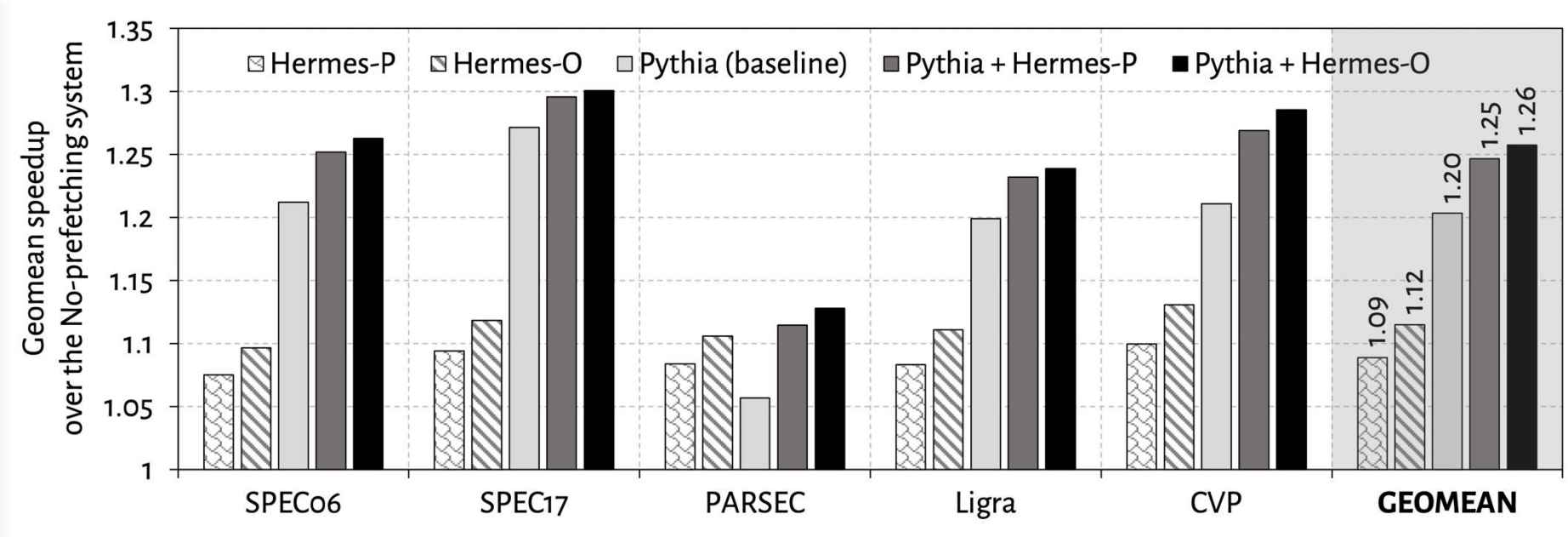
No single feature individually provides
highest prediction accuracy across ***all*** workloads

Are All Features Required? (2)



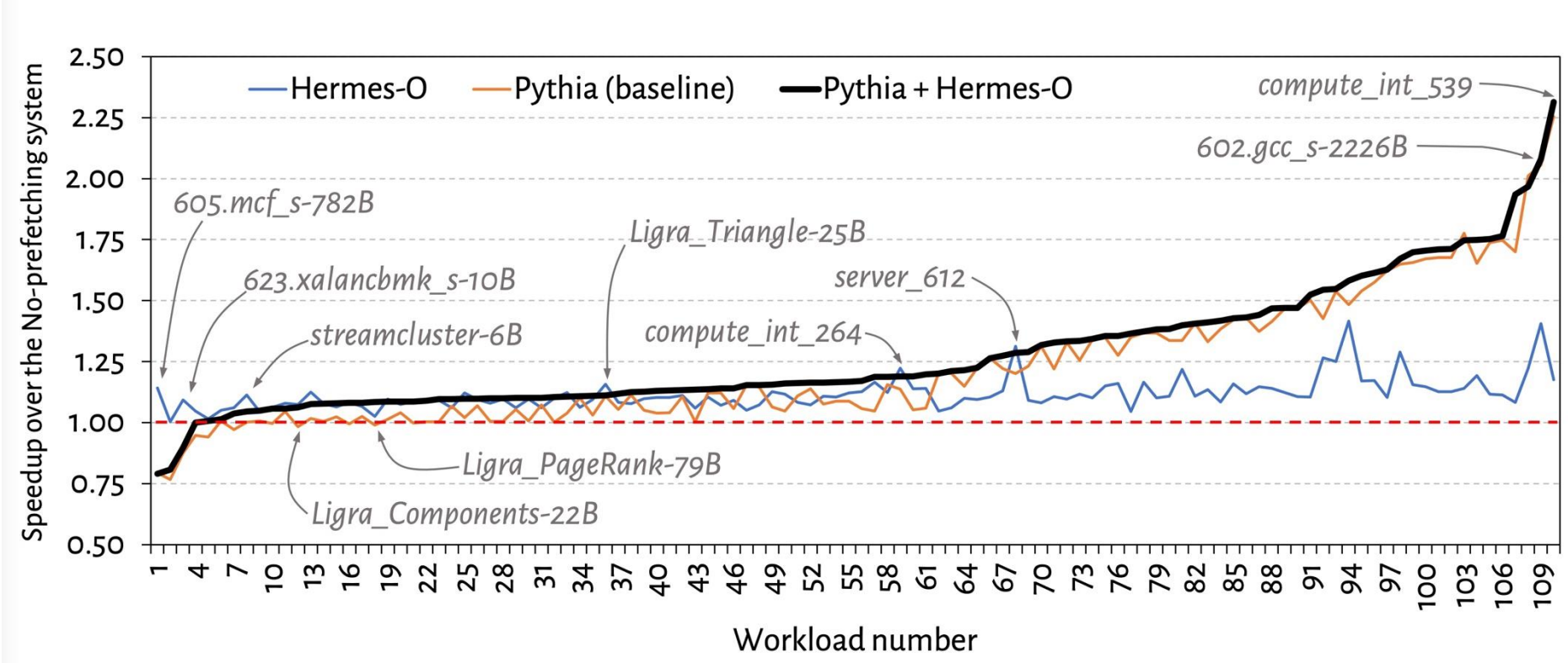
No single feature individually provides
highest prediction coverage also across ***all*** workloads

Single-Core Performance

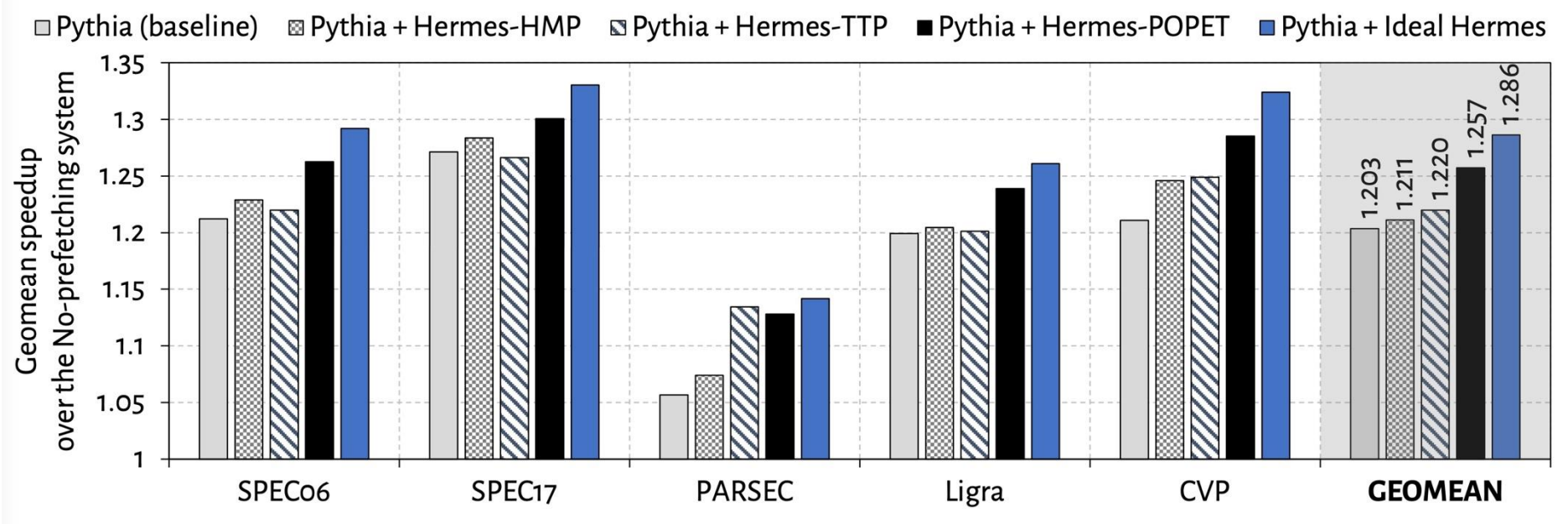


Hermes in combination with Pythia
outperforms **Pythia alone** in every workload category

Single-Core Performance Line Graph



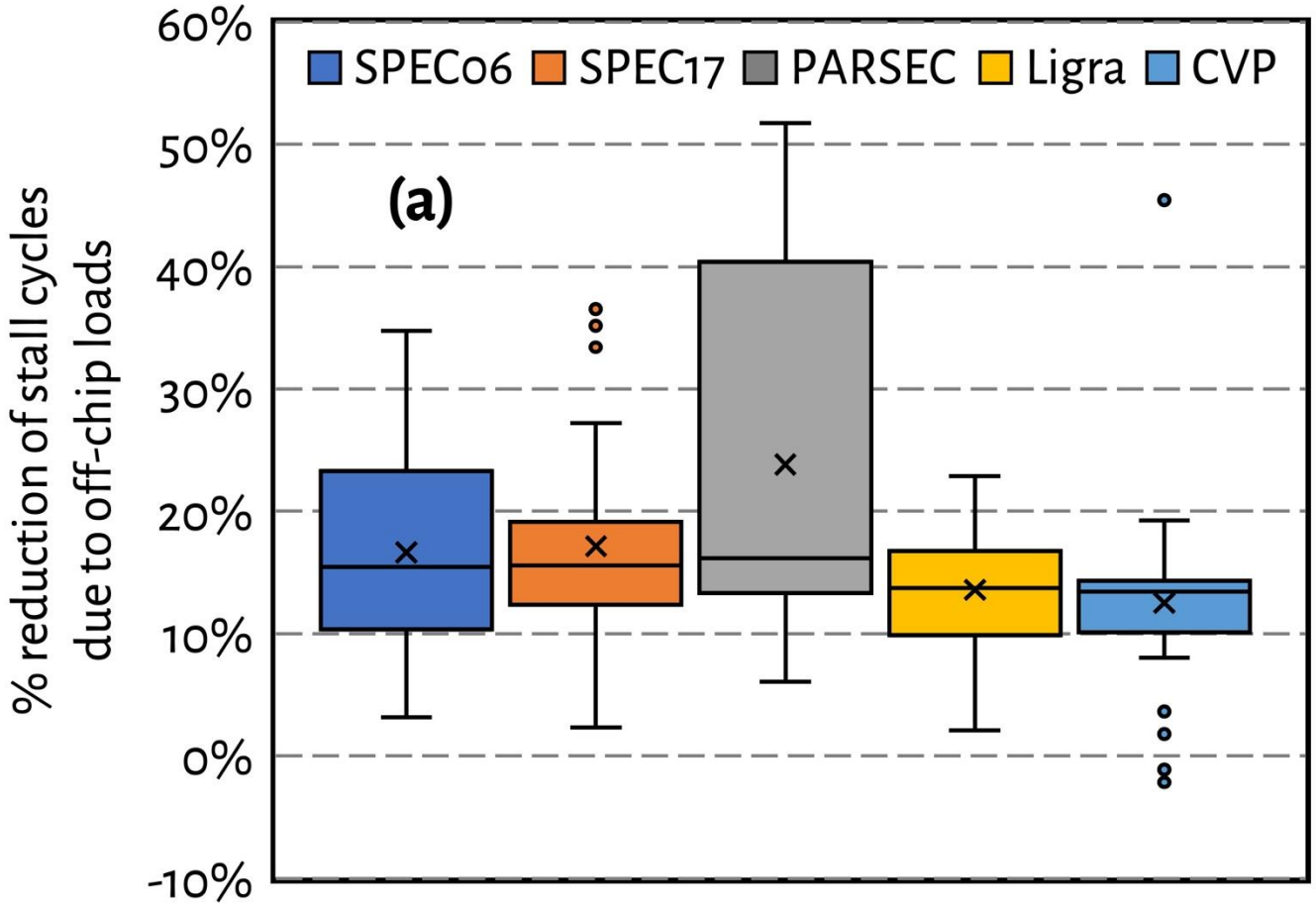
Single-Core Performance Against Prior Predictors



POPET provides higher performance benefit than prior predictors

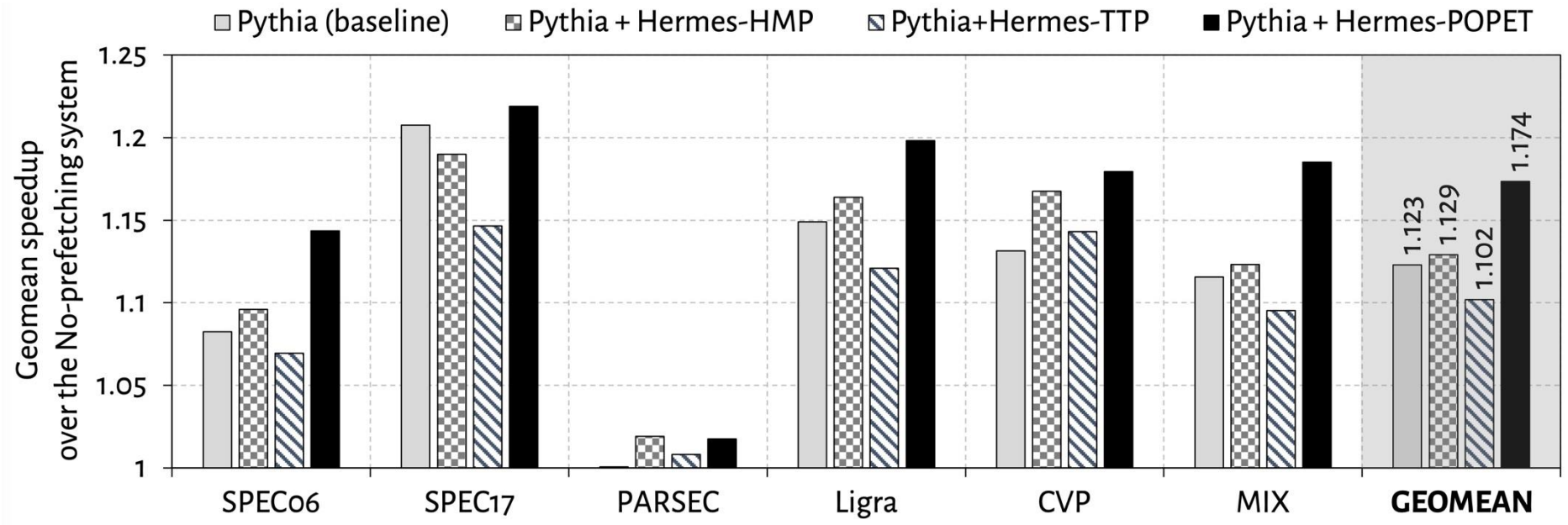
Hermes with POPET achieves nearly **90%** performance improvement of the Ideal Hermes

Effect on Stall Cycles



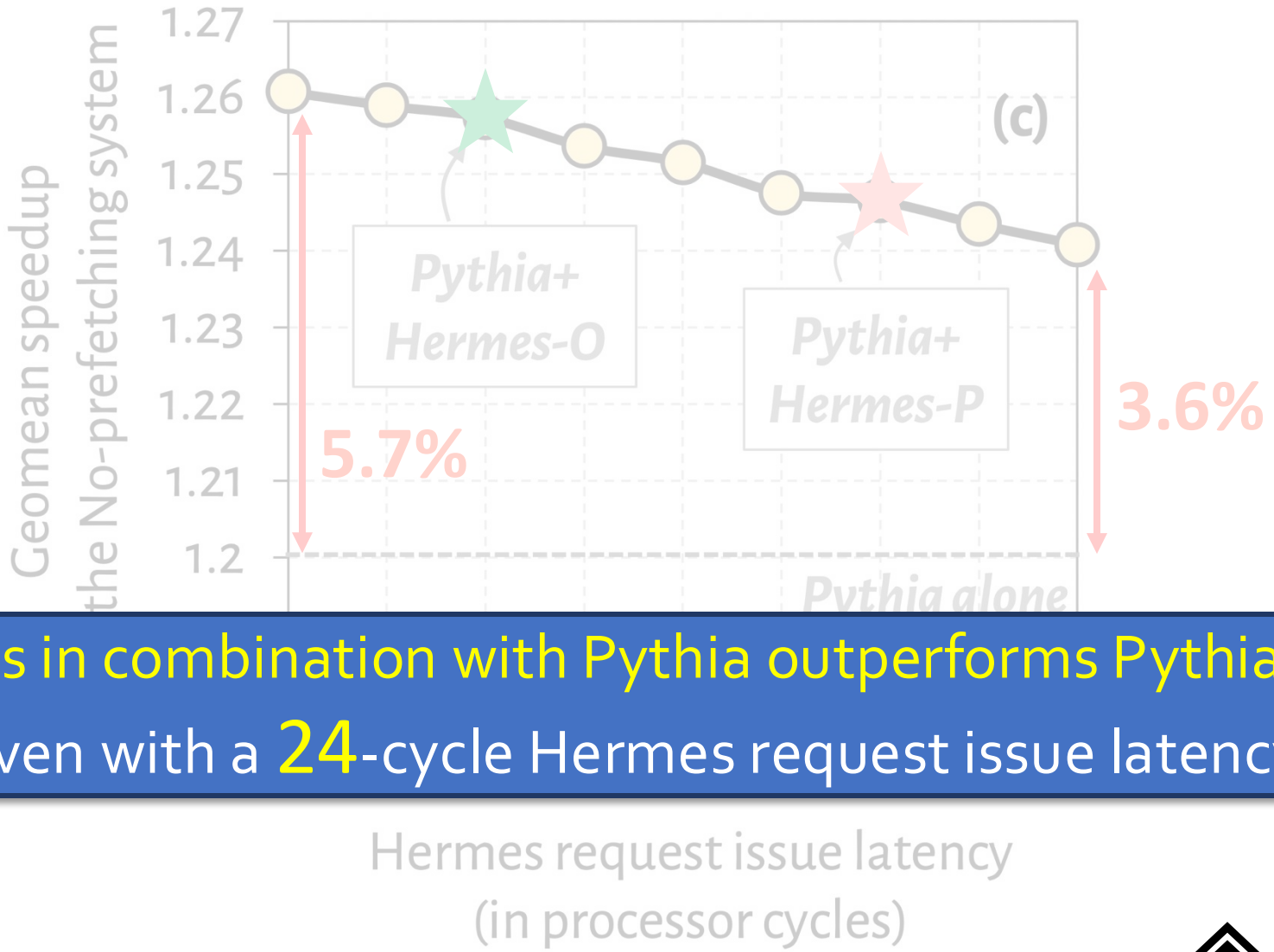
Hermes reduces off-chip load induced stall cycles on average by **16.2%** (up-to **51.8%**)

Eight-Core Performance



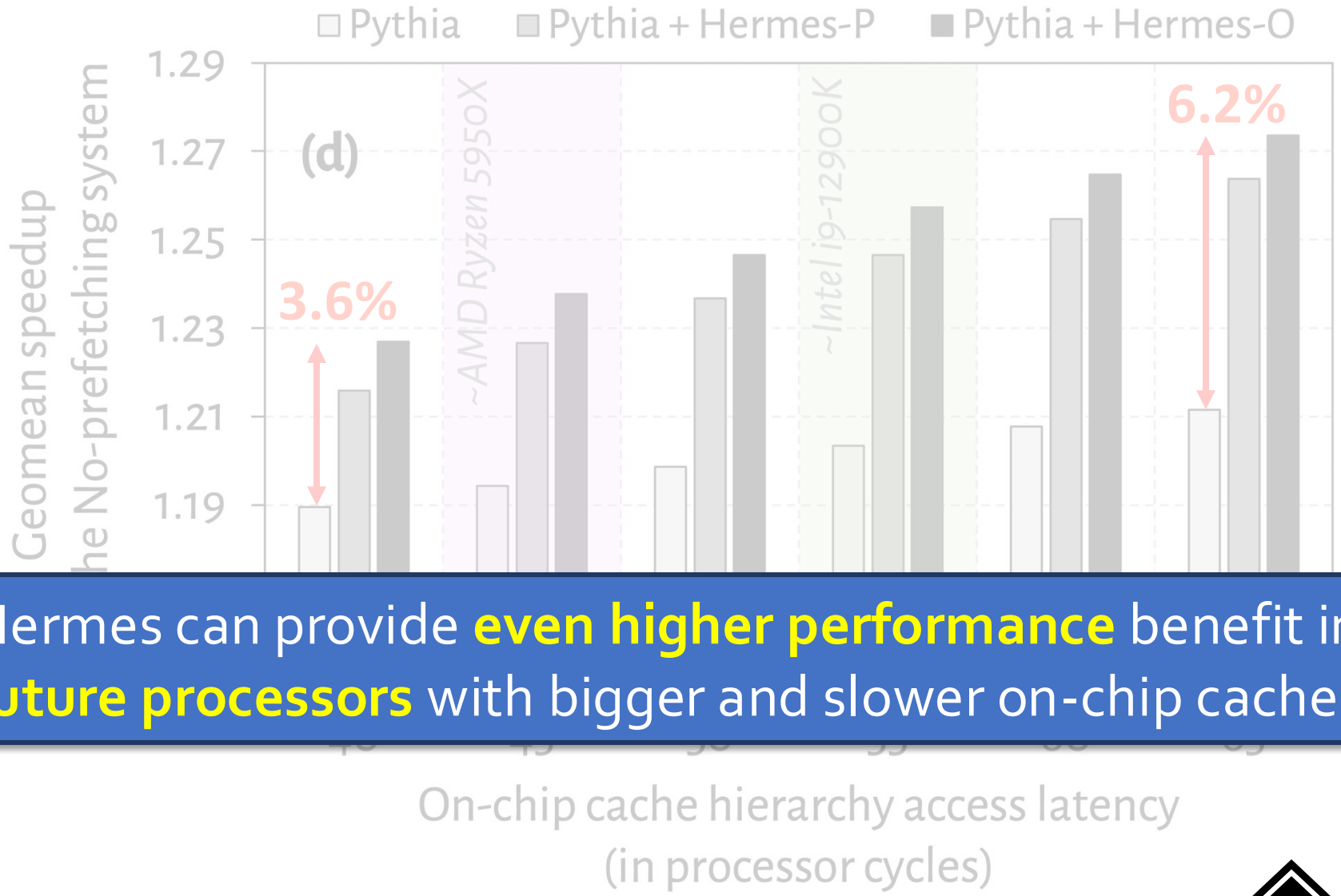
Hermes in combination with Pythia outperforms Pythia alone by **5.1%** on average

Effect of Hermes Request Issue Latency



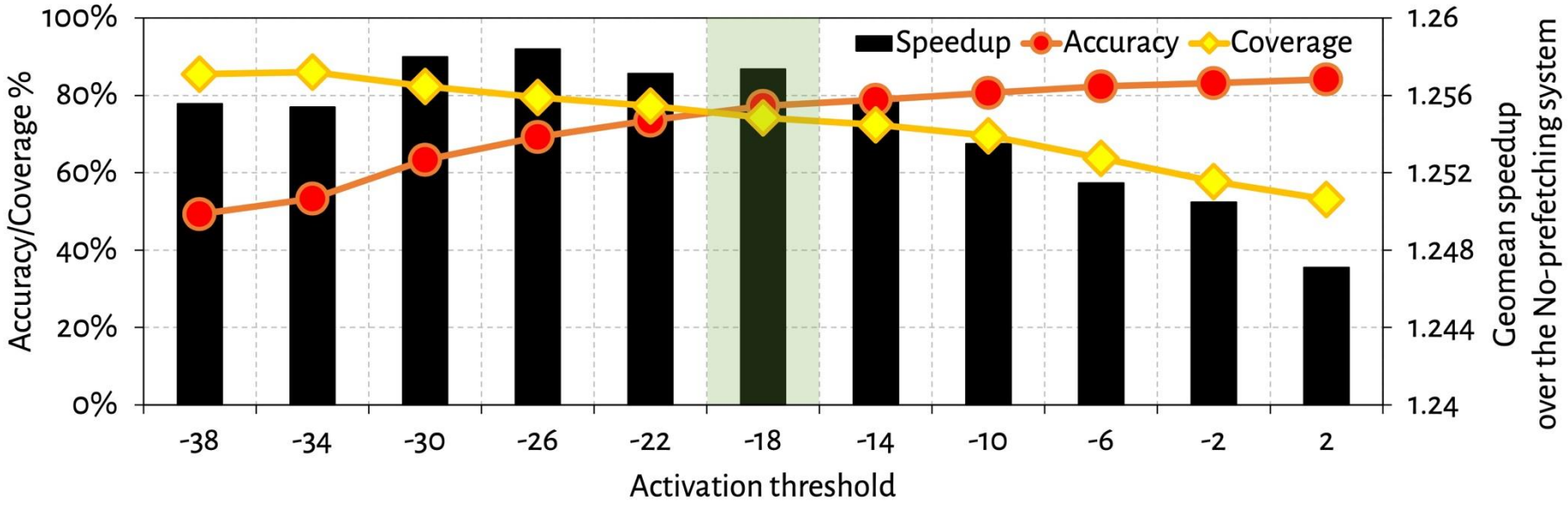
Hermes in combination with Pythia outperforms Pythia alone even with a 24-cycle Hermes request issue latency

Effect of Cache Hierarchy Access Latency



Hermes can provide **even higher performance** benefit in **future processors** with bigger and slower on-chip caches

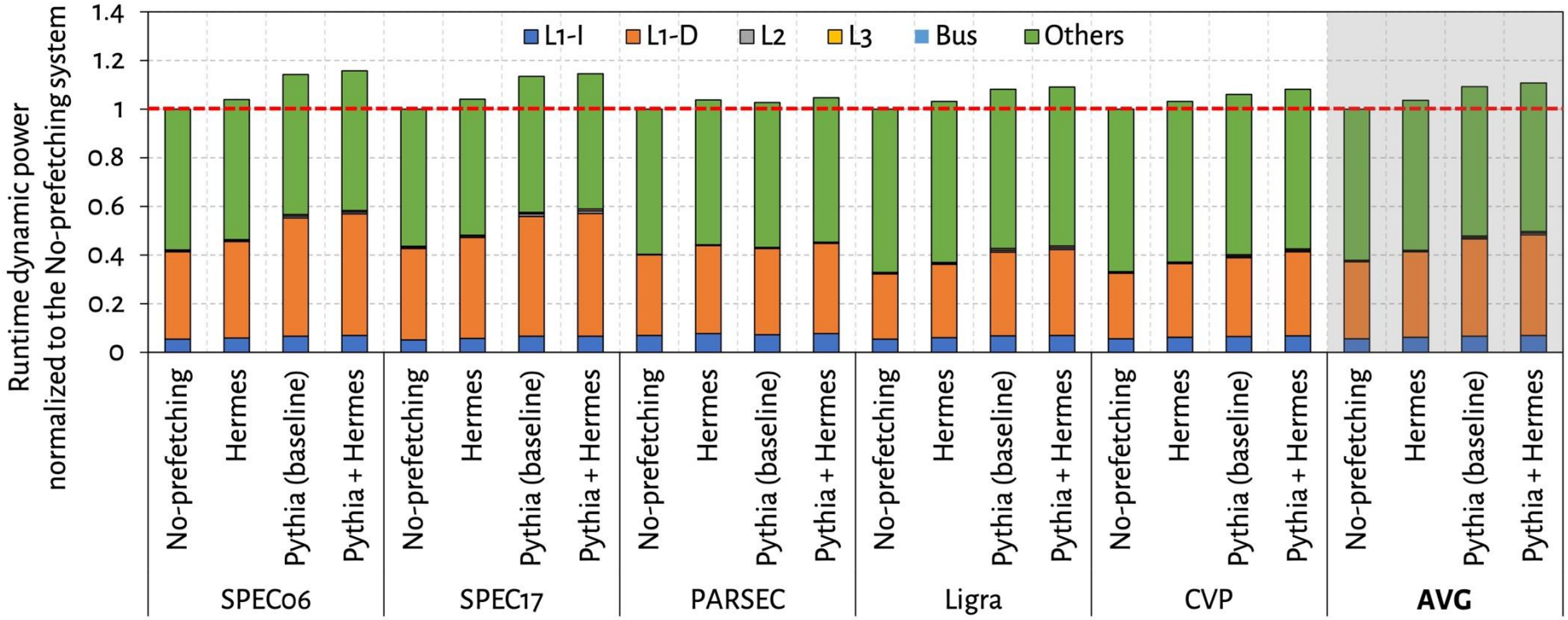
Effect of Activation Threshold



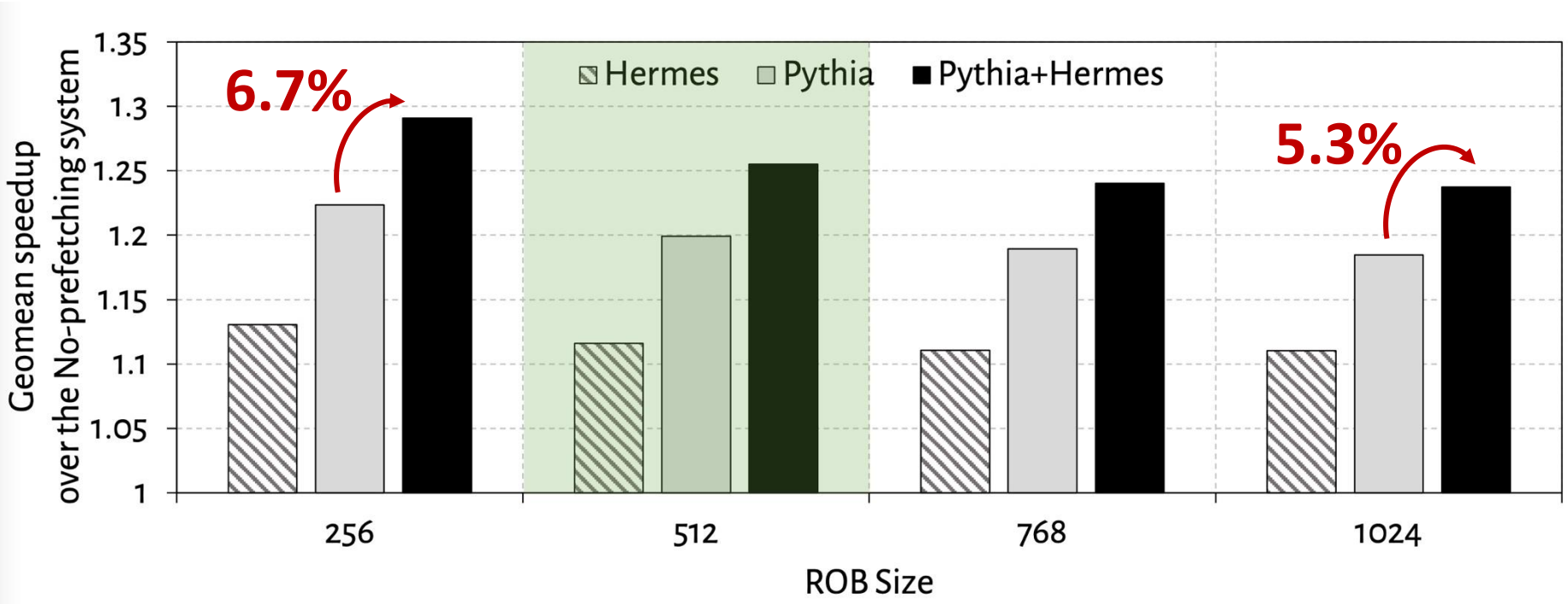
With increase in activation threshold

1. Accuracy increases
2. Coverage decreases

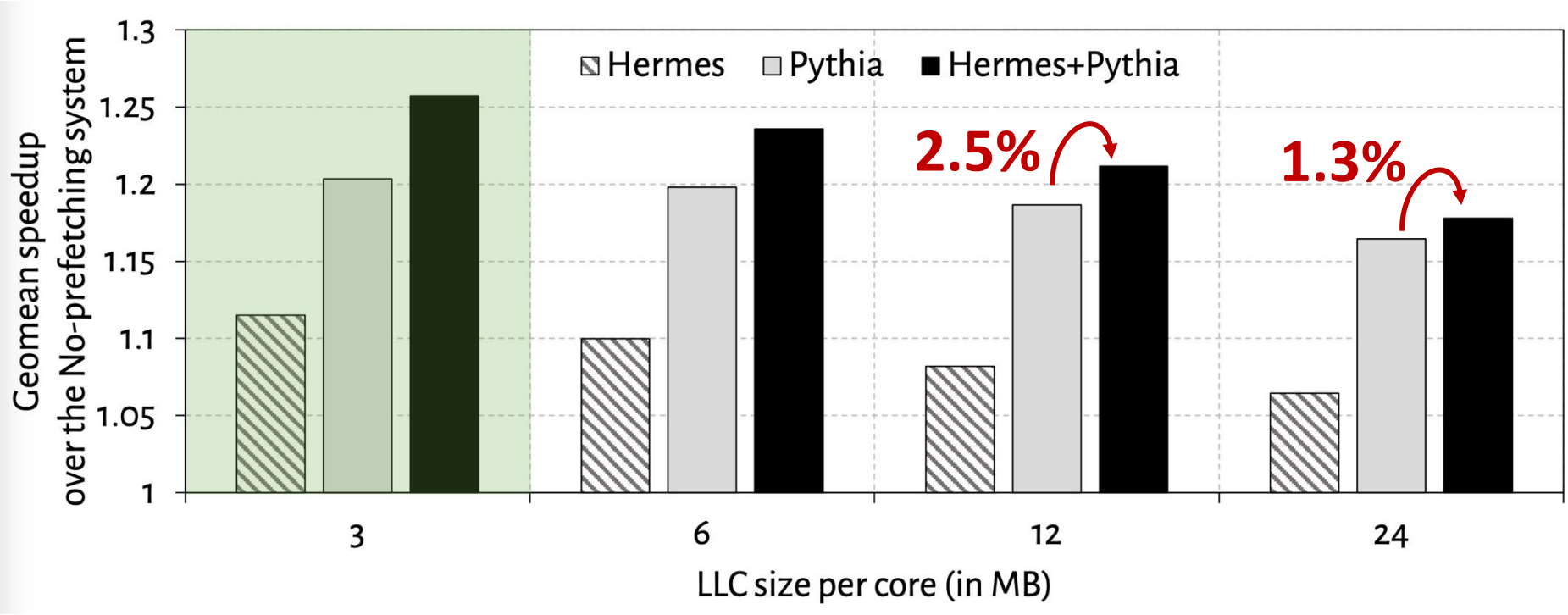
Power Overhead



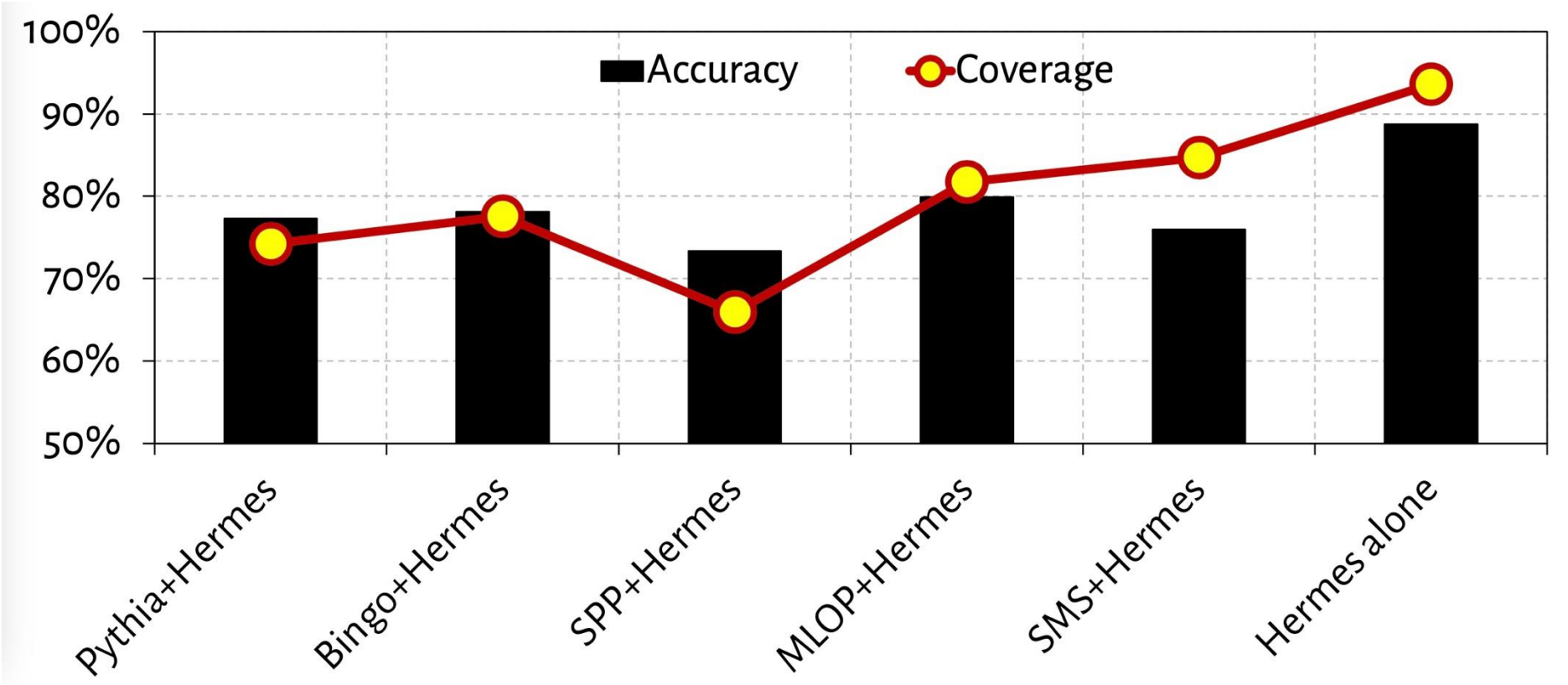
Effect of ROB Size



Effect of LLC Size

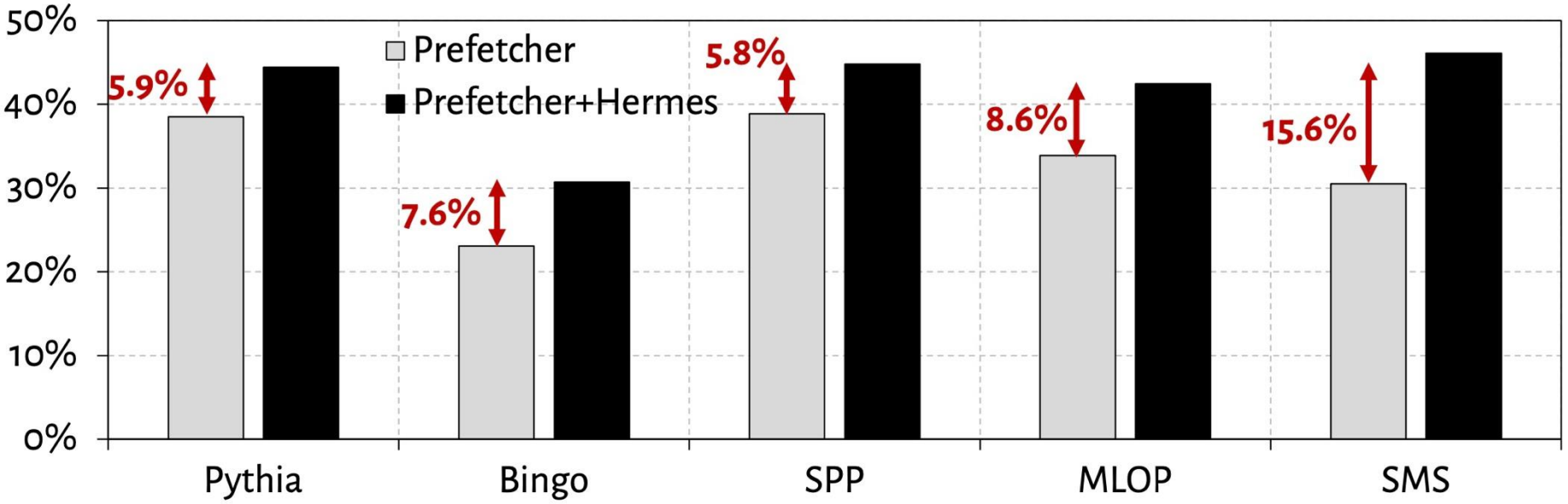


Accuracy and Coverage with Different Prefetchers

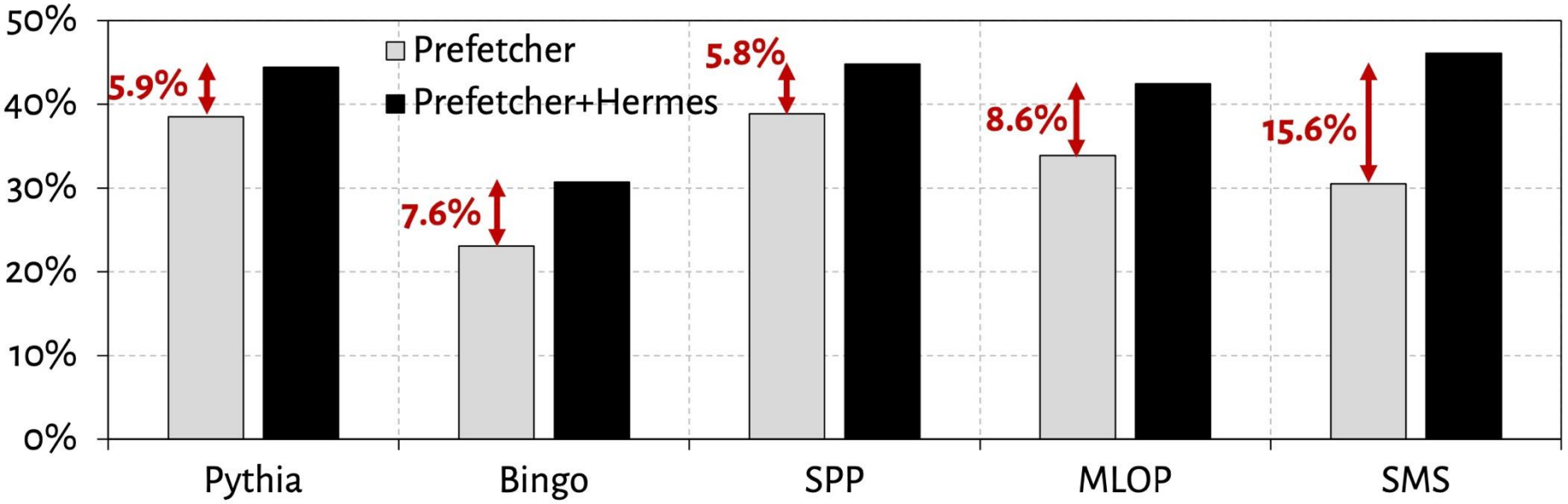


POPET's **accuracy and coverage increases significantly** in absence of a data prefetcher

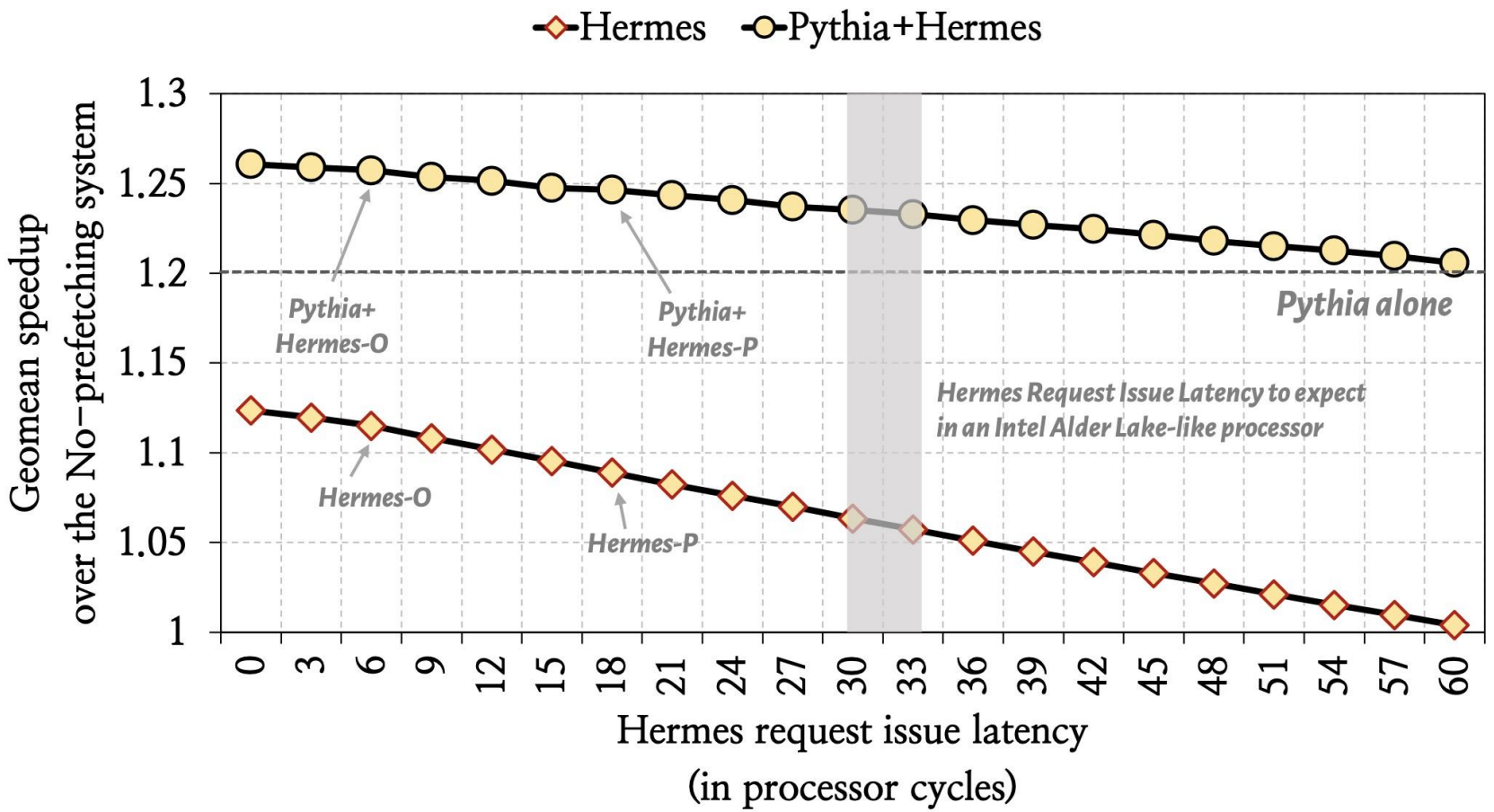
Increase in Main Memory Requests



Increase in Main Memory Requests



Hermes Limit Study

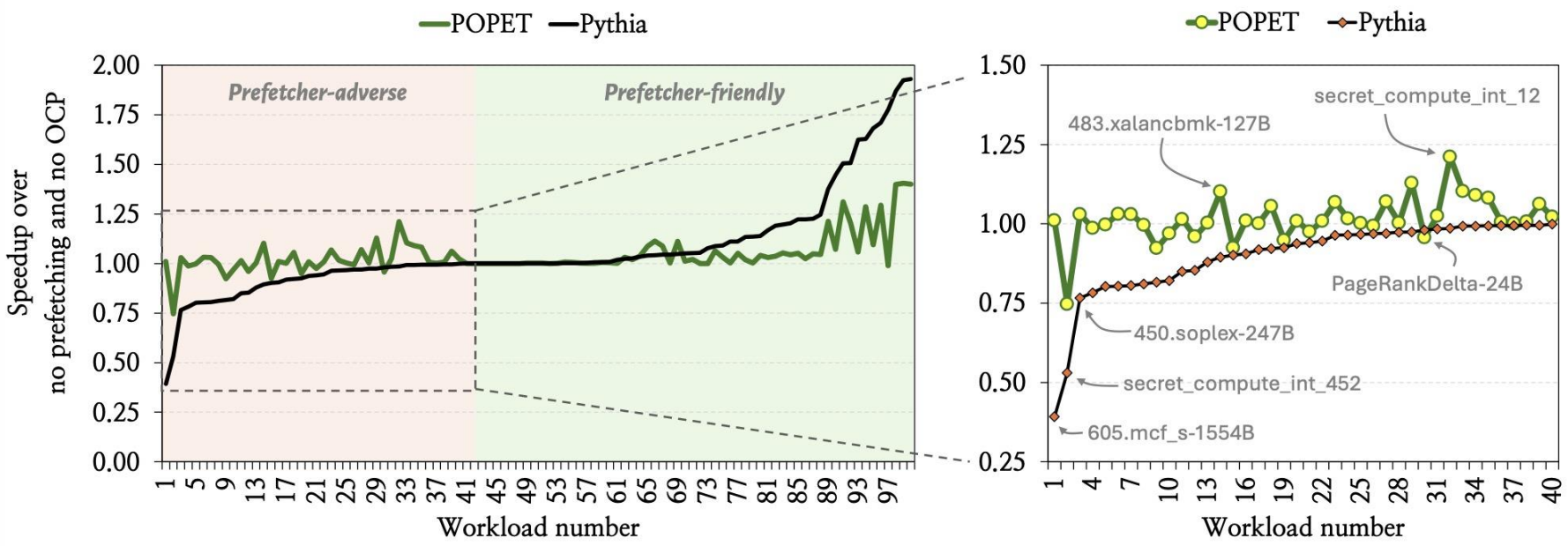


Athena Discussions

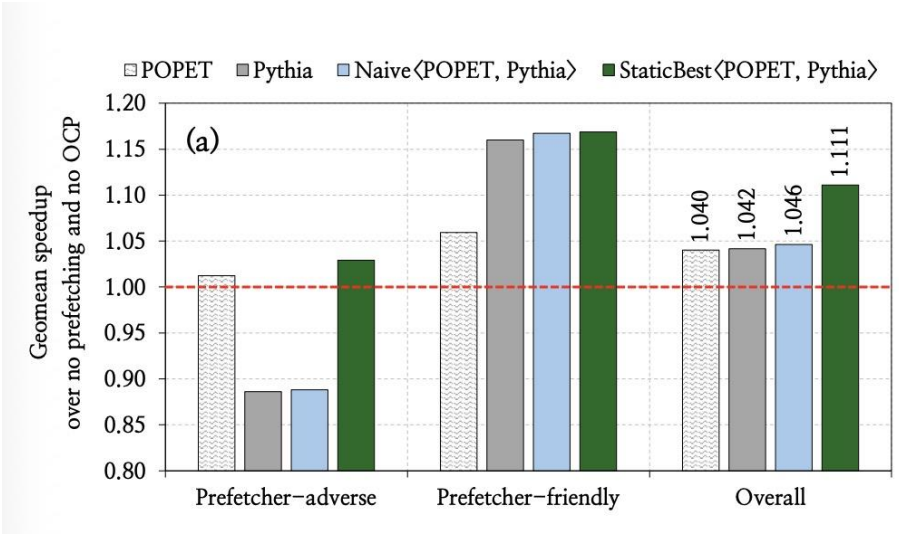
Athena Discussions

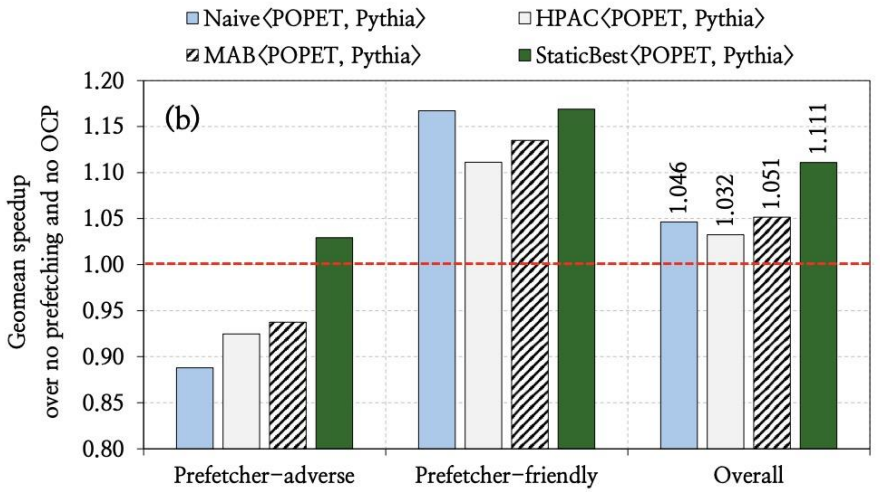
- Motivational data
- Features considered
- Q-Value table organization
- Final configuration
- Overhead analysis
- Simulation parameters
- Evaluated workloads
- Evaluated mechanisms
- Perf in CD1
- Perf in CD2
- Perf in CD3
- Perf in CD4
- Prefetcher sensitivity
- OCP sensitivity
- Bandwidth sensitivity
- Four-core perf
- Eight-core perf
- Impact of uncorrelated reward
- Understanding Athena

Motivational Data



Motivational Data





Features Considered

Feature	Measurement	Rationale
Prefetcher accuracy	$\frac{\text{\# demand hits}}{\text{\# prefetches issued}}$	Effectiveness of prefetching
OCP accuracy	$\frac{\text{\# off-chip demand hits}}{\text{\# off-chip predictions}}$	Effectiveness of OCP
Bandwidth usage	$\frac{\text{current DRAM bandwidth}}{\text{max DRAM bandwidth}}$	Memory bus pressure
Cache pollution	$\frac{\text{\# prefetch-evicted demand misses}}{\text{\# total demand misses}}$	Interference caused by prefetches
Prefetch bandwidth	$\frac{\text{\# prefetch requests to DRAM}}{\text{\# total DRAM requests}}$	Prefetcher's share of memory traffic
OCP bandwidth	$\frac{\text{\# OCP requests to DRAM}}{\text{\# total DRAM requests}}$	OCP's share of memory traffic
Demand bandwidth	$\frac{\text{\# demand requests to DRAM}}{\text{\# total DRAM requests}}$	Demand's share of memory traffic

Q-Value Table Organization

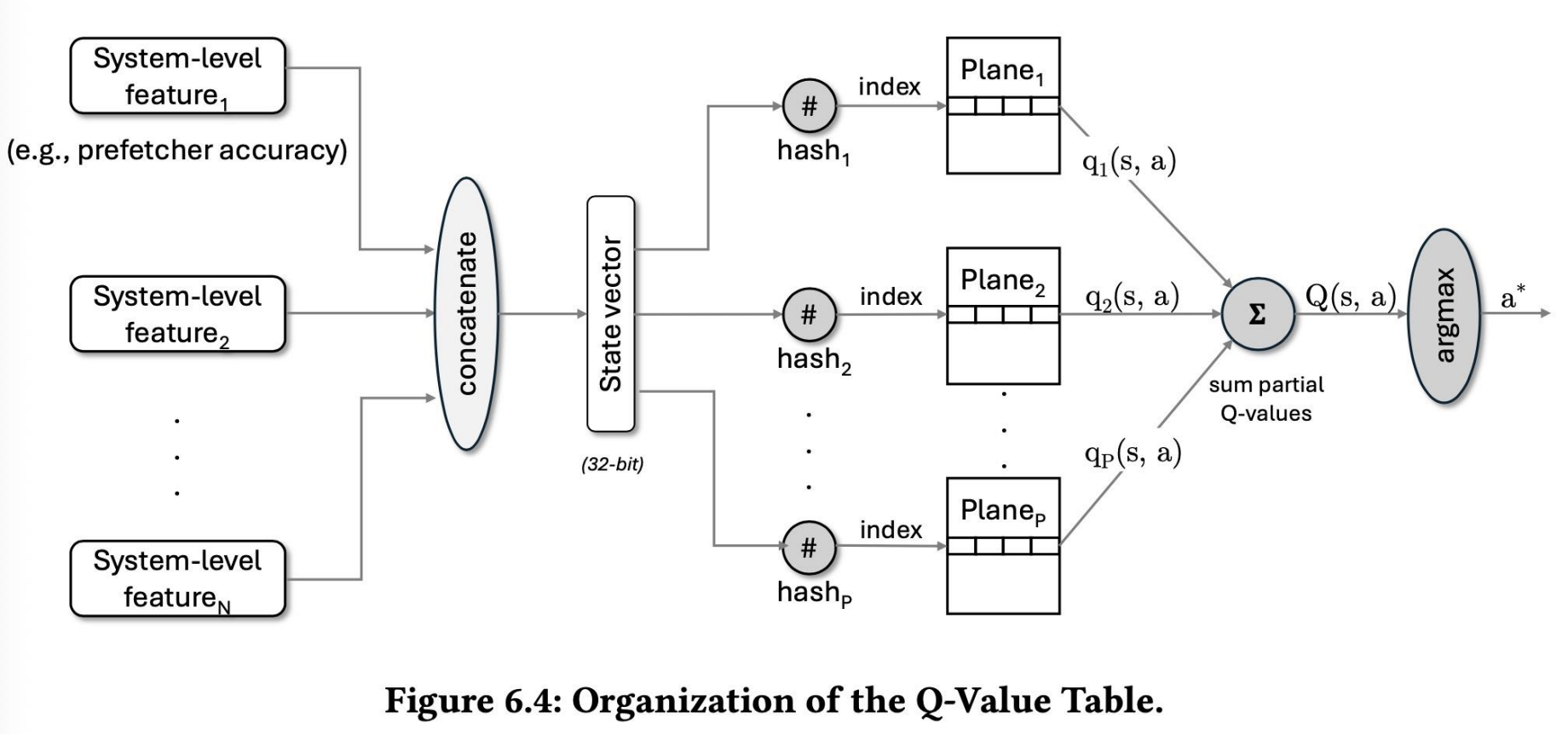


Figure 6.4: Organization of the Q-Value Table.

Final Configuration

Category	Final Values
Selected features	Prefetcher accuracy, OCP accuracy, bandwidth usage, prefetch-induced cache pollution
Reward Weights	$\lambda_{\text{cycle}} = 1.6$, $\lambda_{\text{LLC}_\text{m}} = 0.0$, $\lambda_{\text{LLC}_\text{t}} = 0.0$, $\lambda_{\text{load}} = 0.7$, $\lambda_{\text{MBr}} = 0.6$
Hyperparamteres	$\alpha = 0.60$, $\gamma = 0.60$, $\epsilon = 0.01$, $\Delta Q = 0.12$, Epoch Length = 2.4K Instructions

Overhead Analysis

Structure	Description	Size
Q-Value Table	# planes = 8, # rows = 32 # columns = 4, Entry size = 16 bits	**2 KB**
Accuracy Tracker	4096-bit Bloom filter, 2 hash functions	**0.5 KB**
Pollution Tracker	4096-bit Bloom filter, 2 hash functions	**0.5 KB**
Total		**3 KB**

Simulation Parameters

Core	6-wide fetch/issue/commit, 512-entry ROB, 128-entry LQ, 72-entry SQ, perceptron branch predictor [255], 17-cycle misprediction penalty
L1I/D	Private, 48KB, 64B line, 12-way, 16 MSHRs, LRU, 4/5-cycle round-trip latency
L2C	Private, 1.25MB, 64B line, 20-way, 48 MSHRs, LRU, 15-cycle round-trip latency [52]
LLC	Shared, 3MB/core, 64B line, 12-way, 64 MSHRs/slice, SHiP [553], 55-cycle round-trip latency [48, 52]
Main Memory	1 rank per channel, 8 banks per rank, 64-bit data bus, 2KB row buffer, $t_{RCD} = t_{RP} = t_{CAS}$ = 12.5ns; DDR4 with 3.2 GB/s per core

Evaluated Workloads

Suite	# Workloads	Example Workloads
SPEC CPU06	29	astar, leslie3d, libquantum, milc, omnetpp, sphinx3, soplex ...
SPEC CPU17	20	bwaves, cactuBSSN, fotonik3d, gcc, lbm, mcf, xalancbmk ...
Ligra	13	BC, BFS, BFSCC, PageRankDelta, Radii, ...
PARSEC	13	canneal, facesim, fluidanimate, raytrace, streamcluster ...
CVP	25	integer, floating point, ...

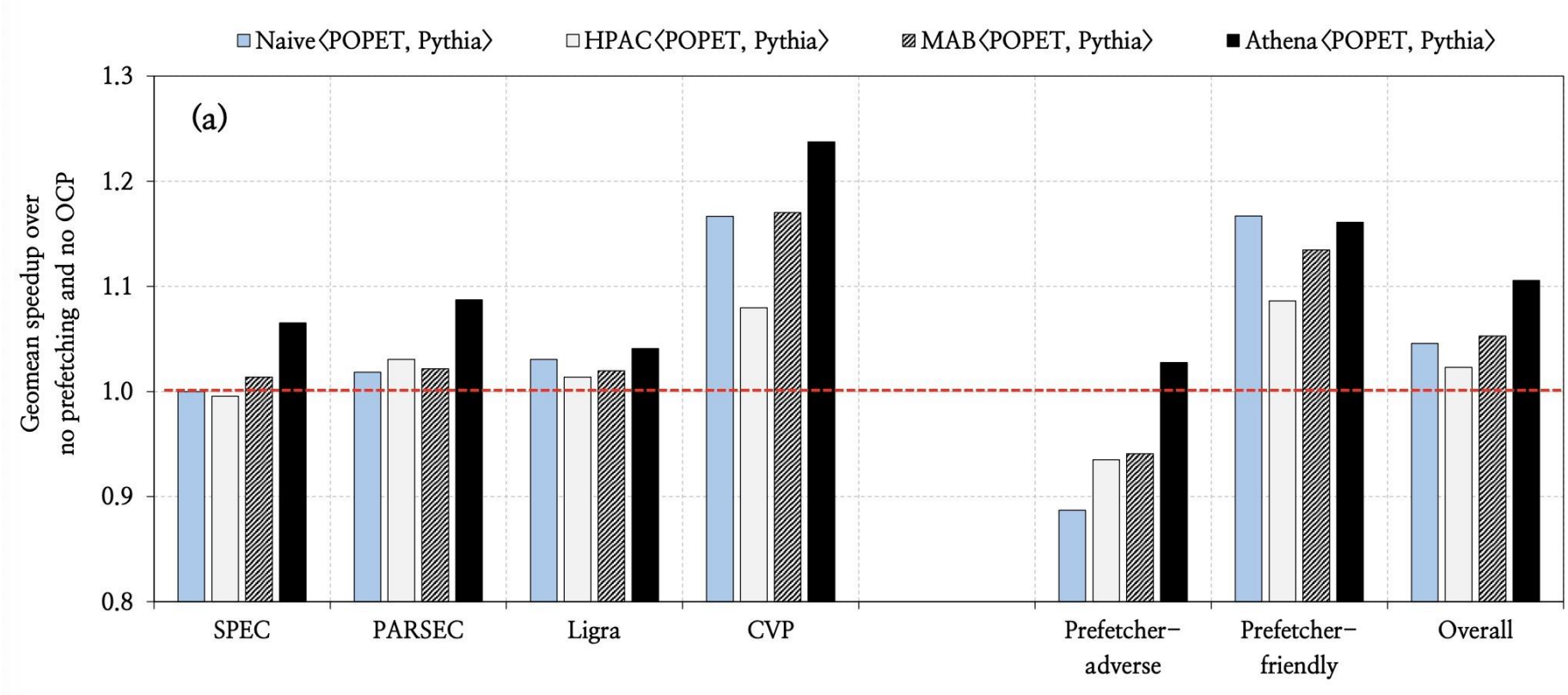
Evaluated Cache Designs

Cache Design	Description	Comparison Points
CD1	OCP + 1 L2C prefetcher	HPAC, MAB
CD2	OCP + 1 L1D prefetcher	HPAC, MAB, TLP
CD3	OCP + 2 L2C prefetchers	HPAC, MAB
CD4	OCP + 1 L1D + 1 L2C prefetcher	HPAC, MAB, TLP

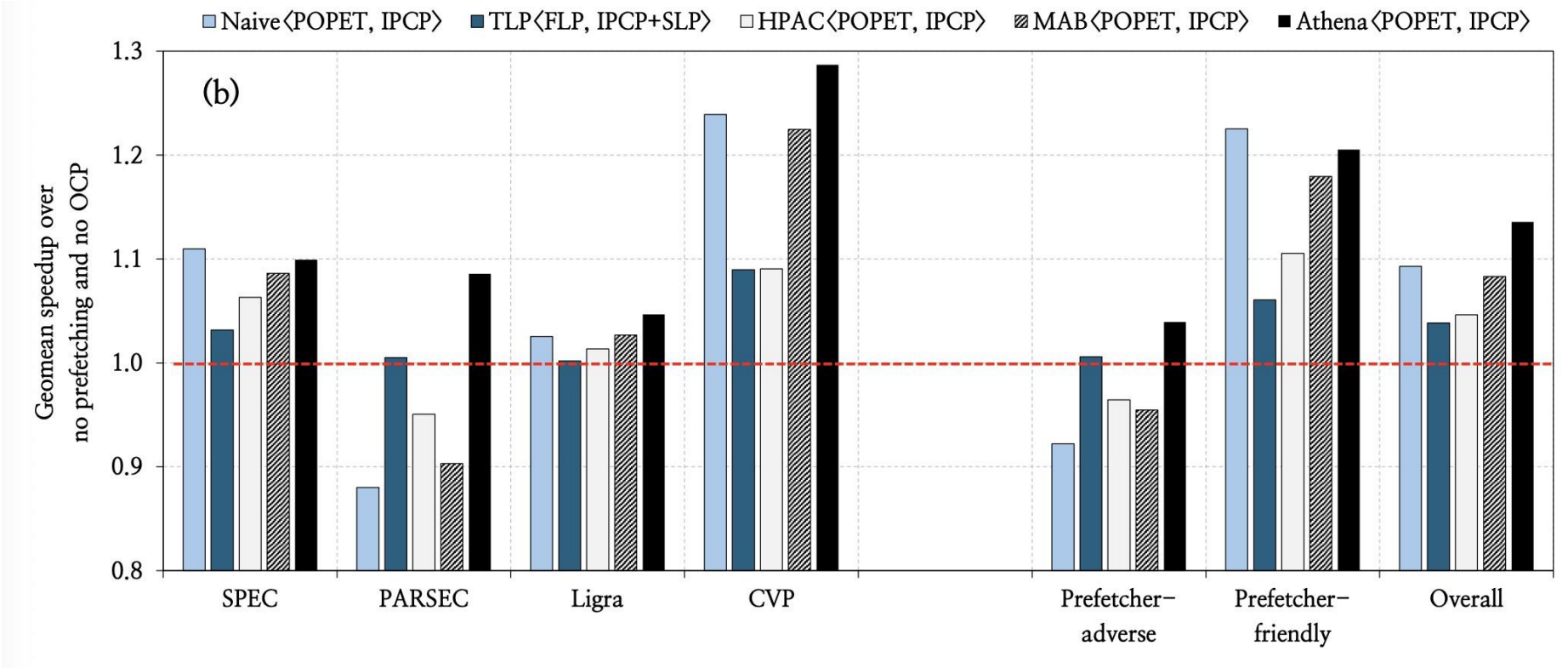
Evaluated Mechanisms

Prefetchers	**IPCP** [406], as an L1-only prefetcher	**0.7 KB**
	IP-based stride [84, 197], with degree 2	**2.25 KB**
	Pythia, as configured in [100]	**25.5 KB**
	SPP+PPF, as configured in [106, 300]	**39.3 KB**
	MLOP, as configured in [477]	**8 KB**
	SMS, as configured in [502]	**20 KB**
OCPs	**POPET** [99], with 5 features	**4 KB**
	HMP [576], with local, gshare, and gskew predictors	**11 KB**
	TTP [248], with metadata budget similar to L2 cache	**1536 KB**
Policies	**TLP**, as in [250]	**7 KB**
	HPAC [172], adapted for OCP	**0.5 KB**
	MAB [205], adapted for OCP	**0.1 KB**
	Athena (this work)	**3 KB**

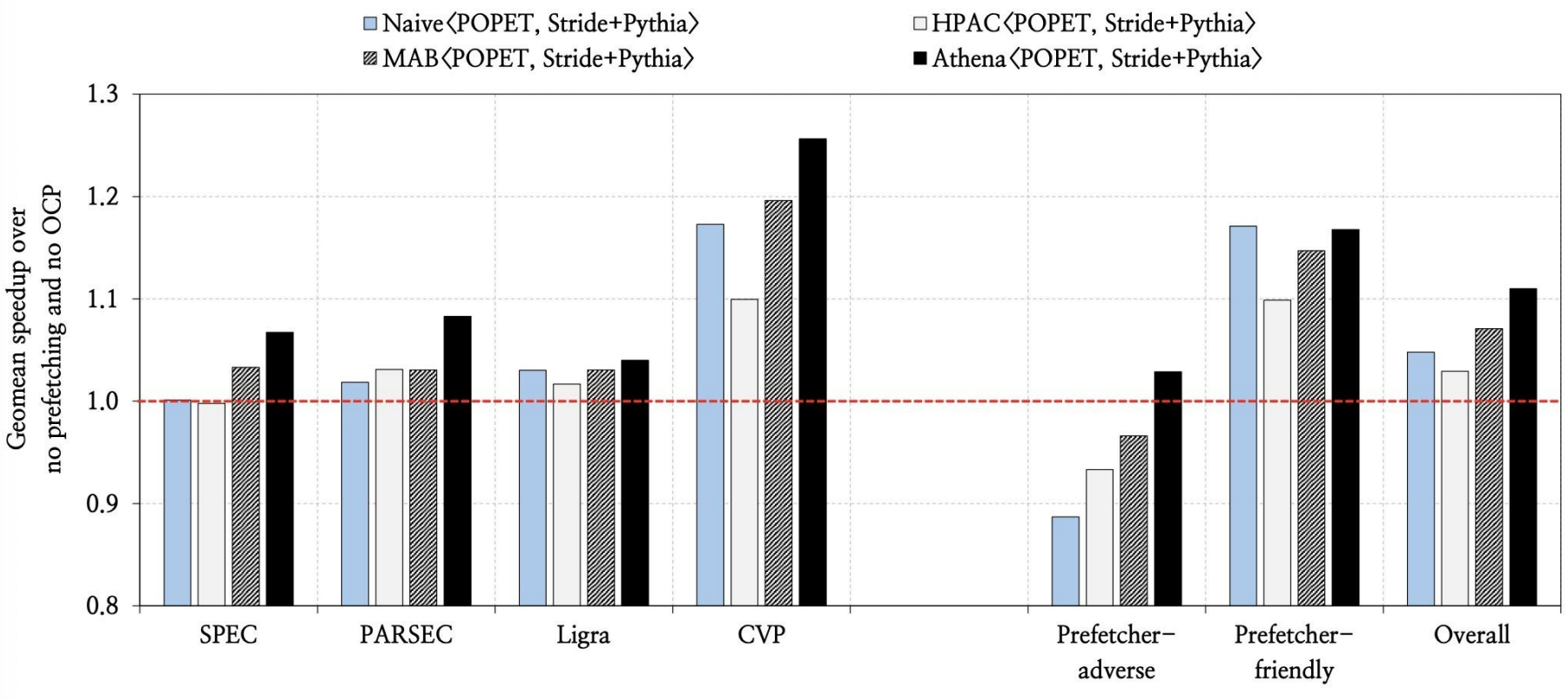
Performance Gain in CD1



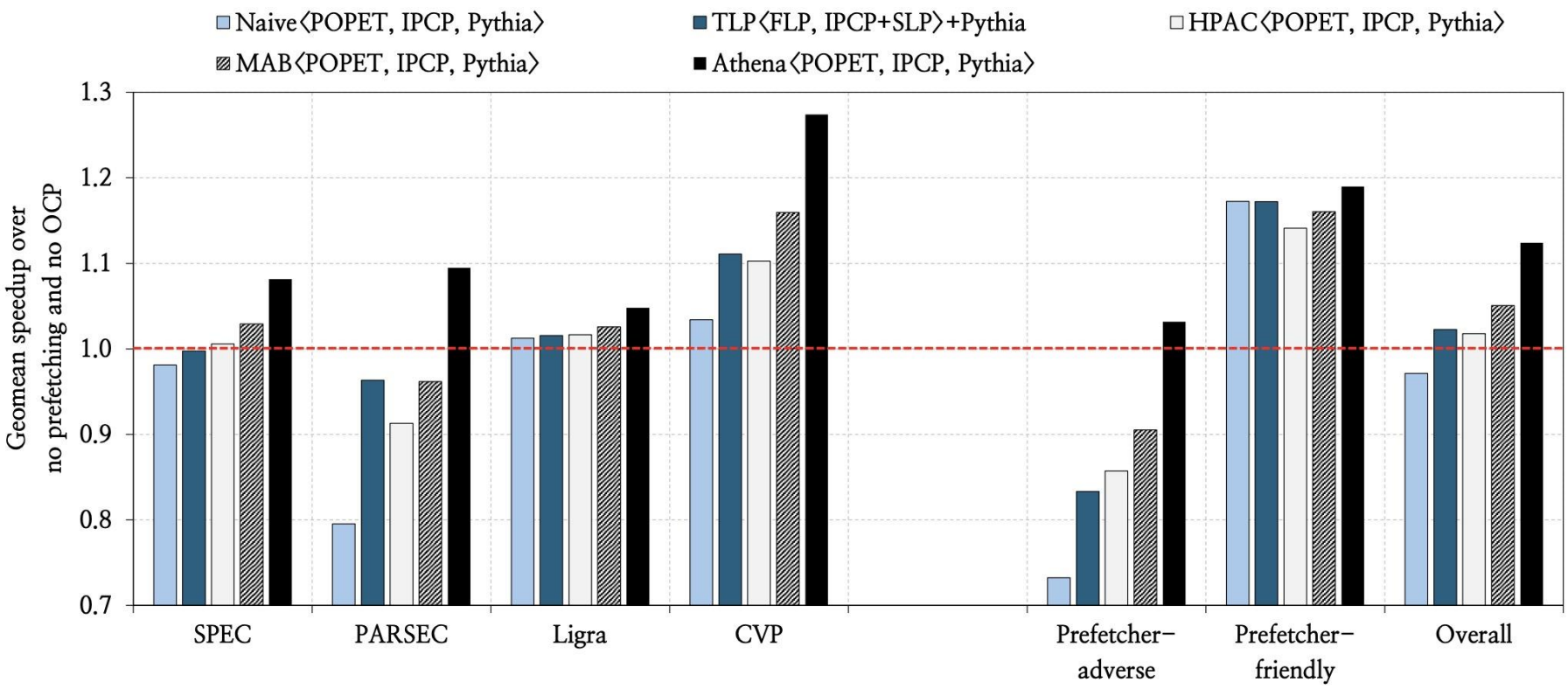
Performance Gain in CD2



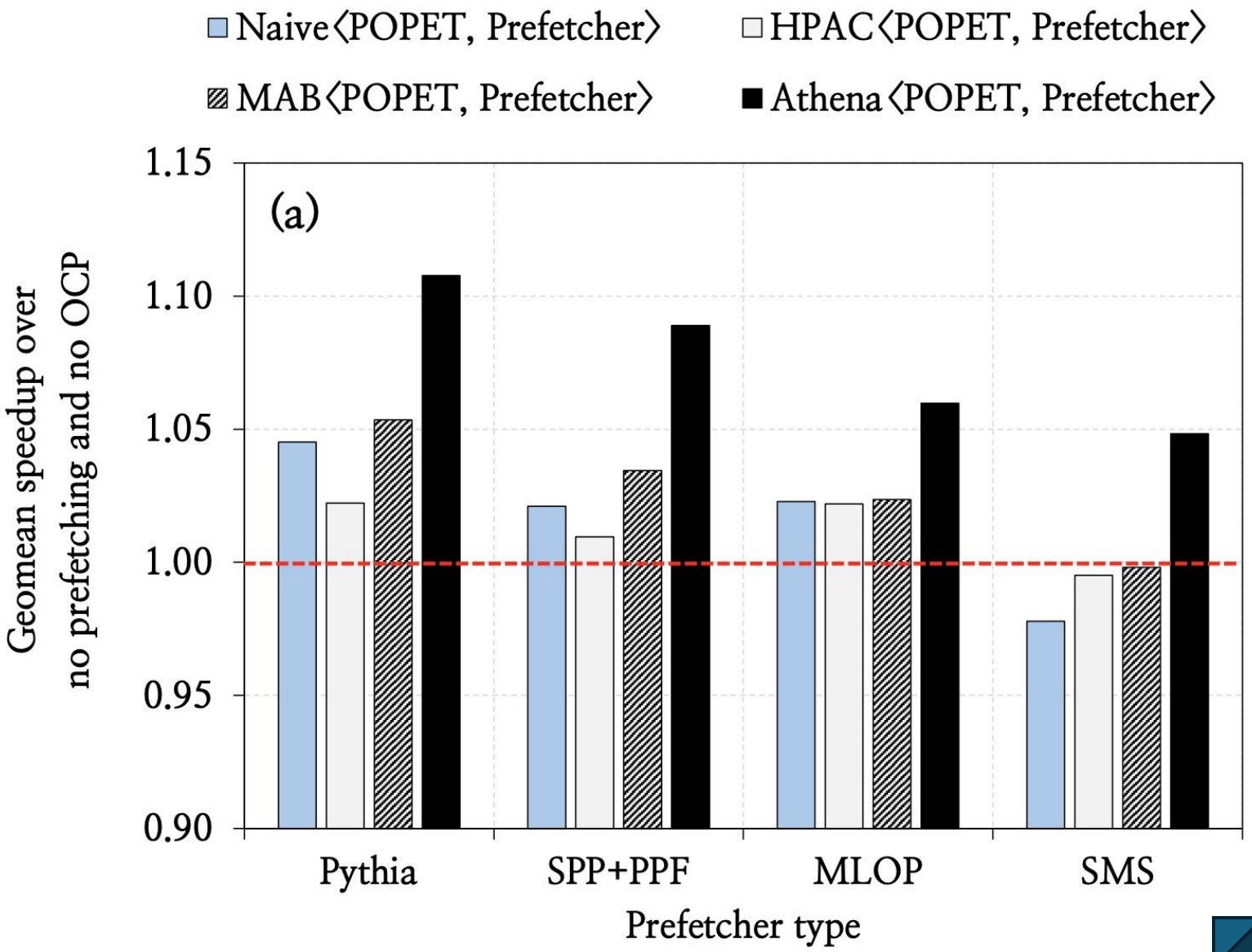
Performance Gain in CD3



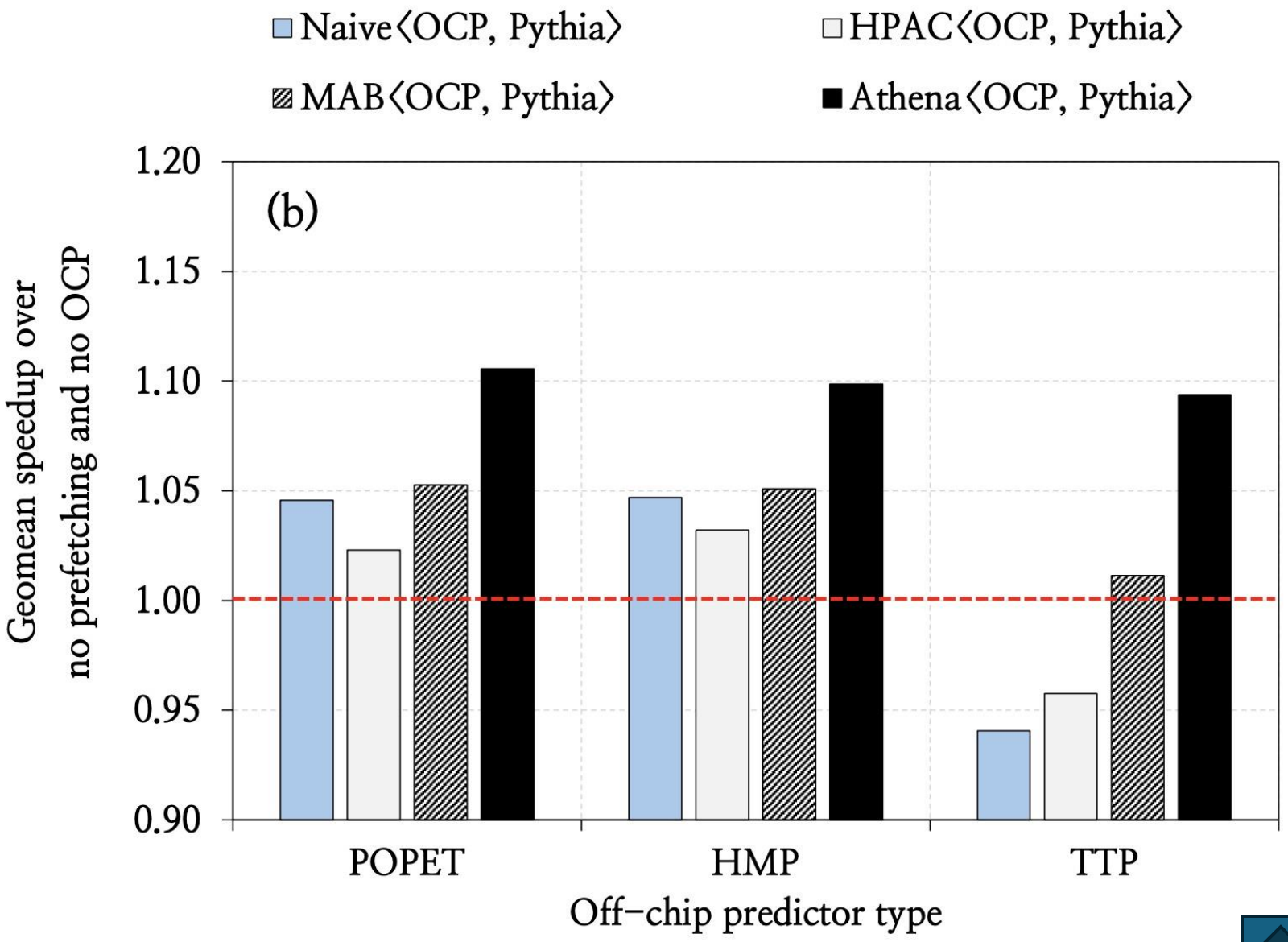
Performance Gain in CD4



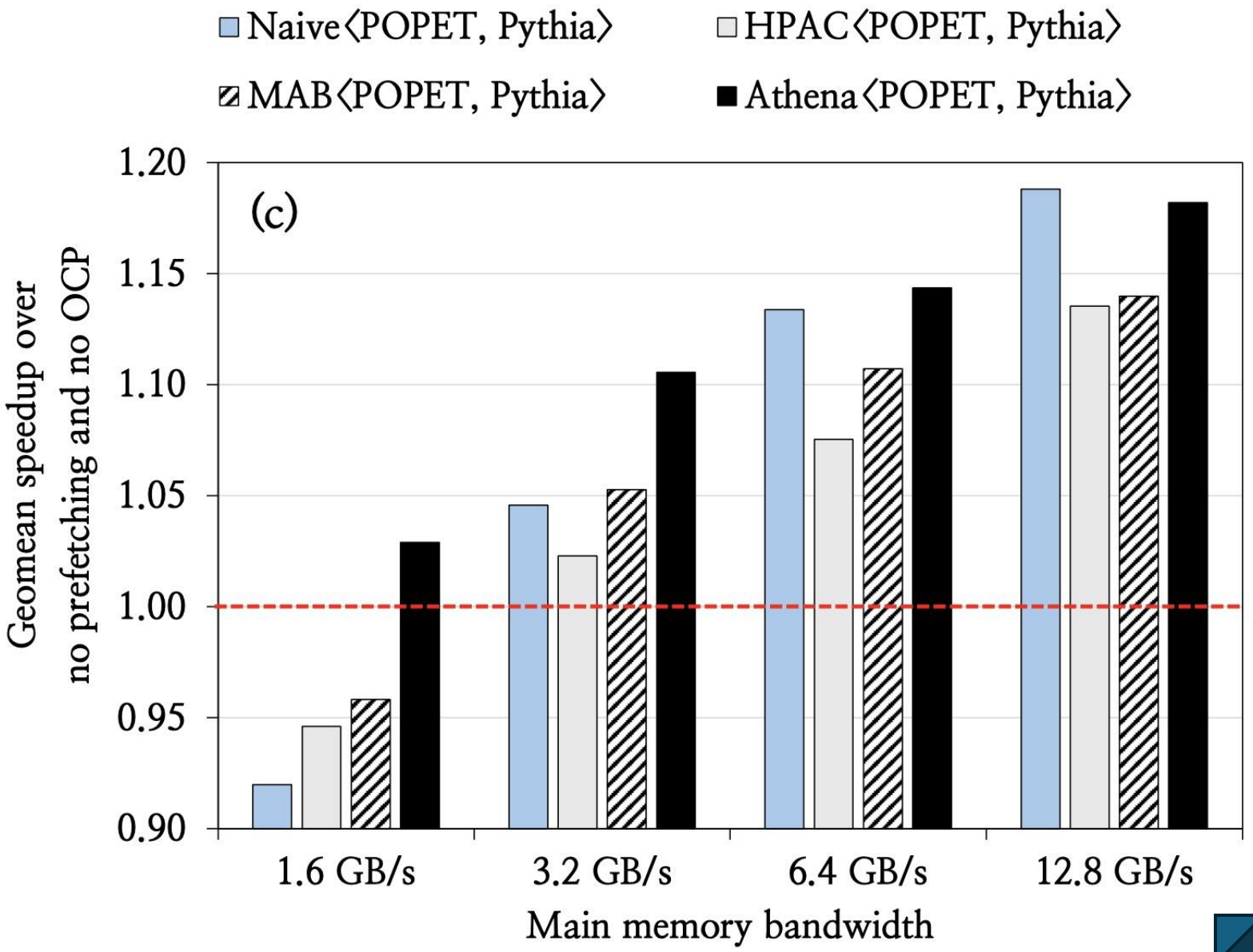
Sensitivity to Prefetcher Type



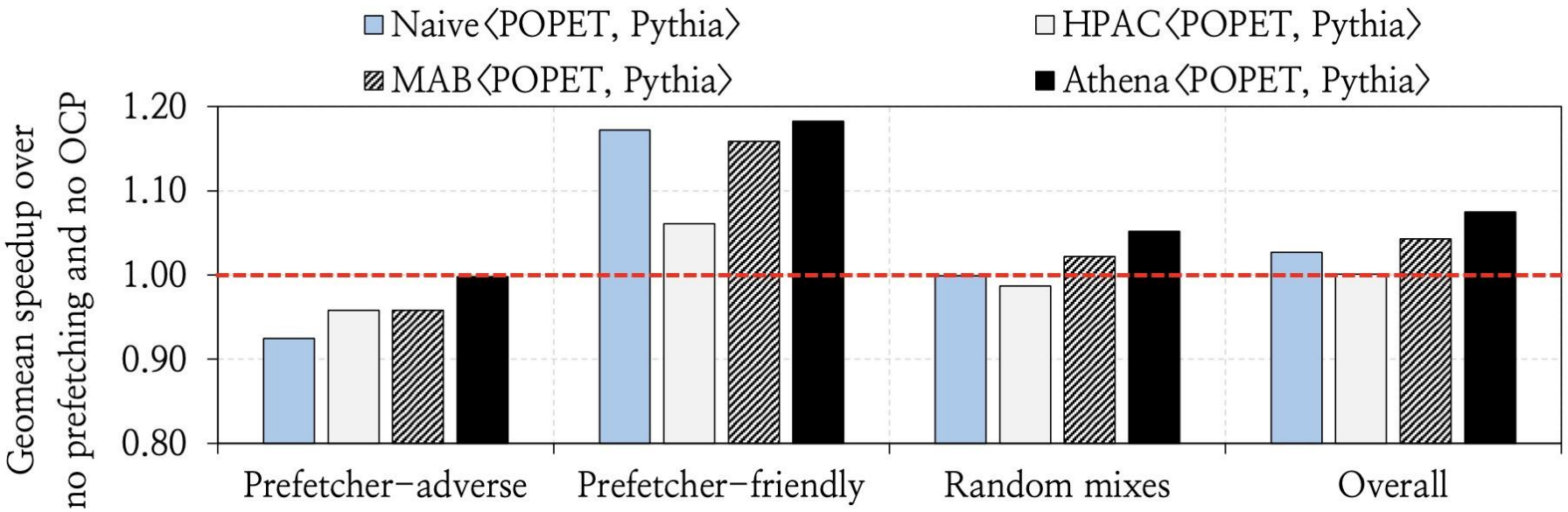
Sensitivity to OCP Type



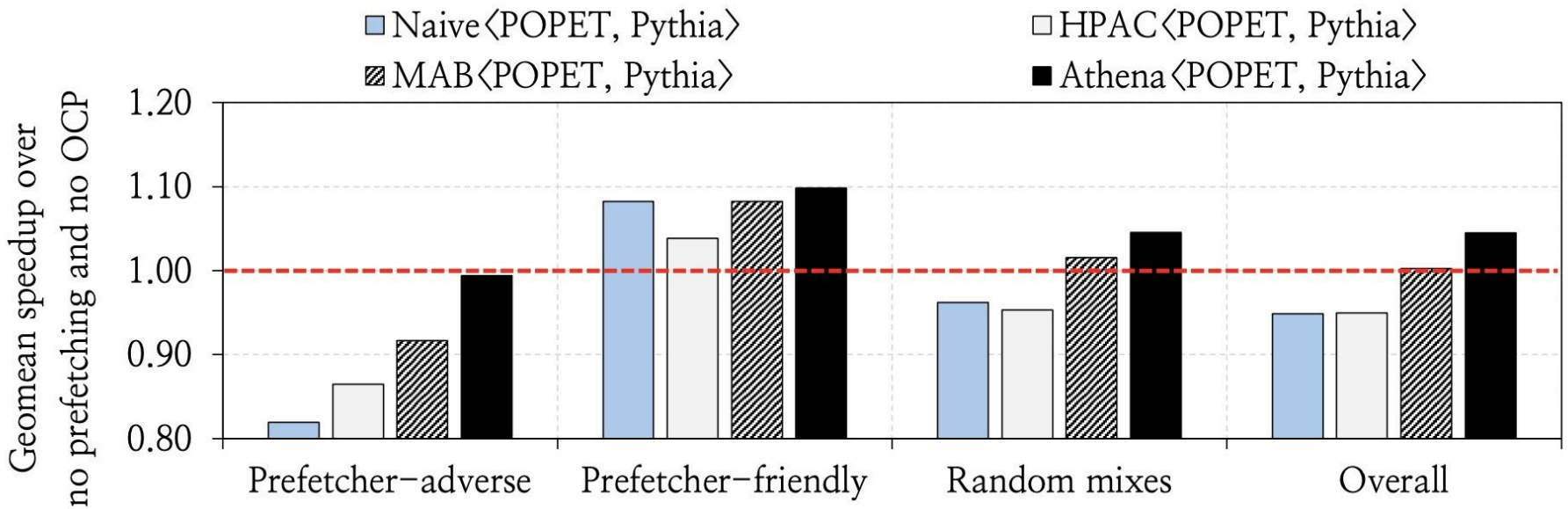
Sensitivity to Main Memory Bandwidth



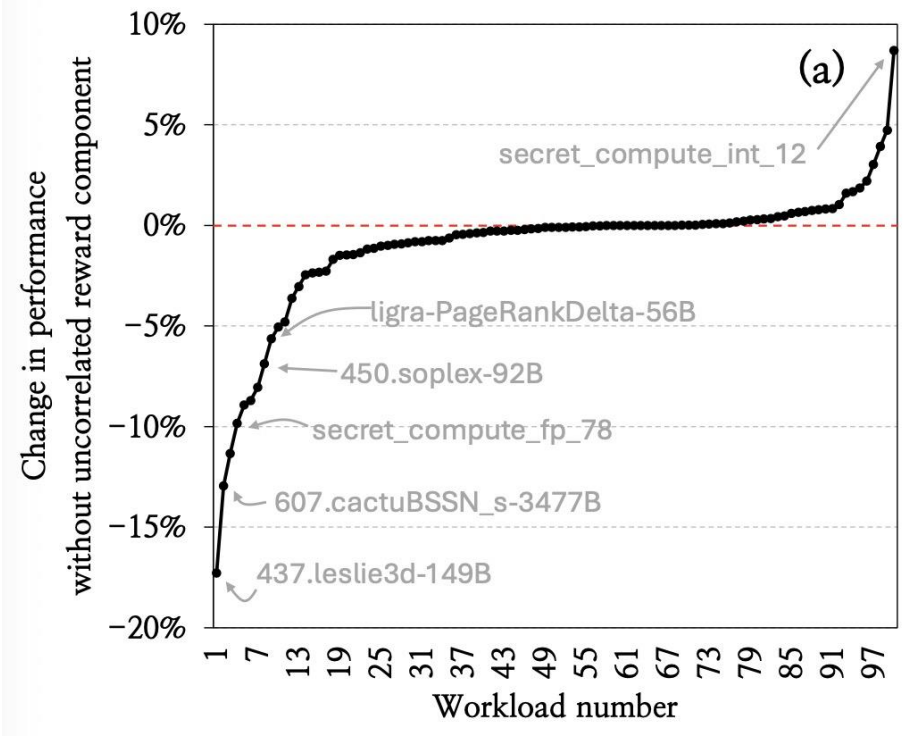
Performance Improvement in Four-Core

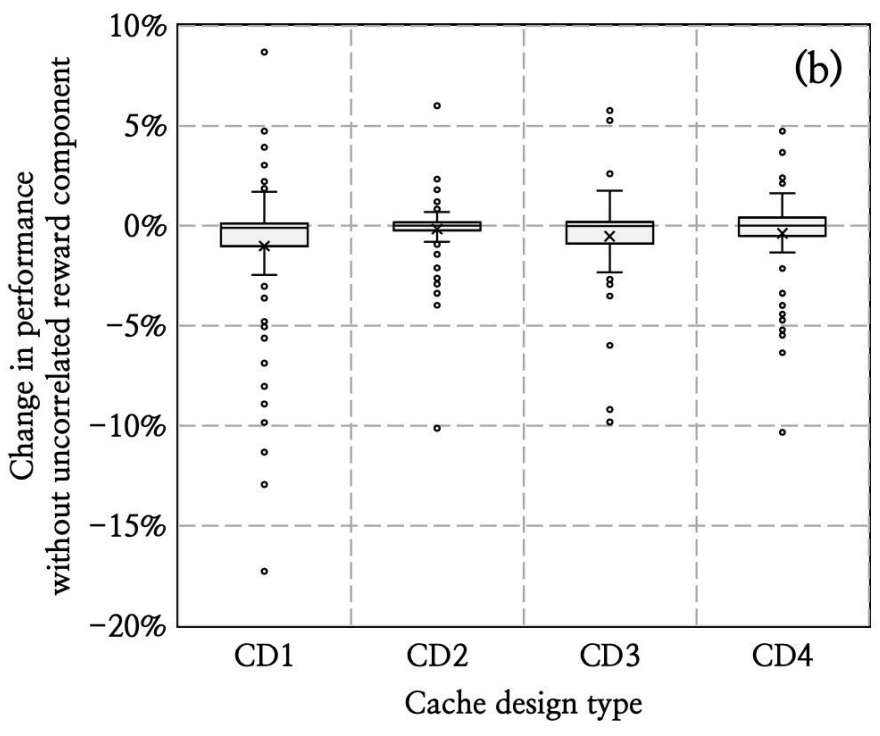


Performance Improvement in Eight-Core



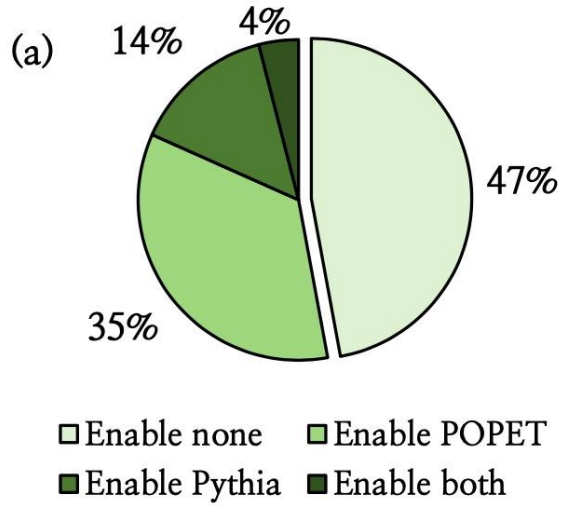
Impact of Uncorrelated Reward

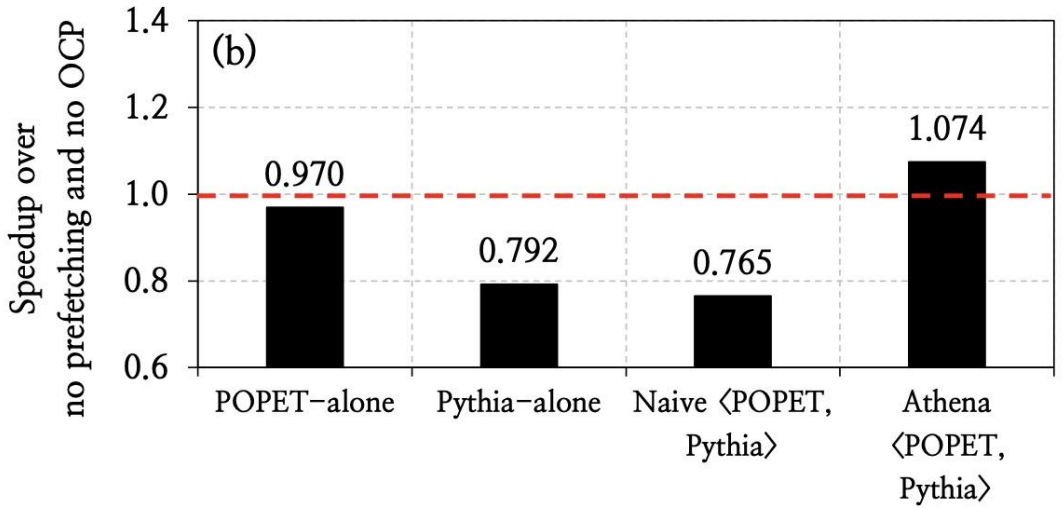




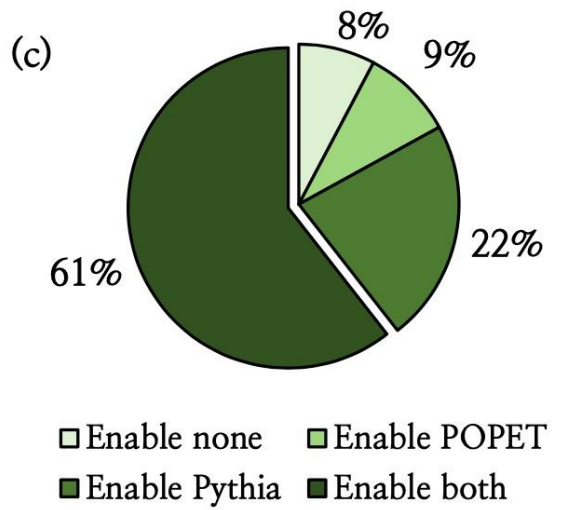
Understanding Athena

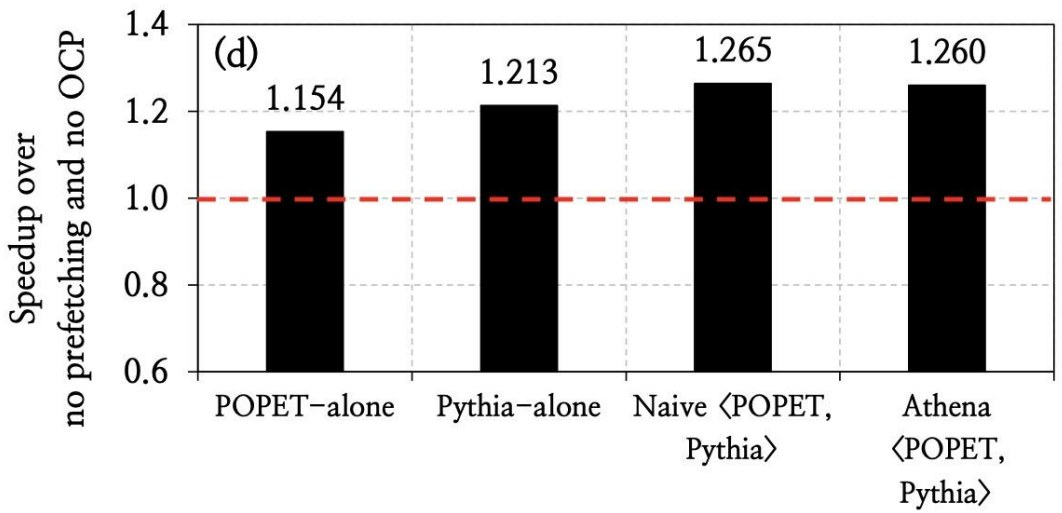
With 3.2 GB/s main memory bandwidth





With 25.6 GB/s main memory bandwidth





Constable Discussions

Constable Discussions

Motivation & Design

- Why do global-stable loads exist?
- Effect of increasing registers
- Characterization of global-stable loads
- Resource dependence by global-stable loads
- Performance headroom
- Constable overview
- Architecting elimination table
- Effect of wrong-path update
- Example of Constable's operation
- Ensuring coherence
- Storage overhead
- Area/energy overhead

Methodology & Evaluation

- Simulation parameters
- Evaluated workloads
- Workload-wise performance
- Load-category-wise performance
- Comparison with prior works
- Elimination coverage
- Coverage of global-stable loads
- Reduction in power
- Scaling width
- Scaling depth
- Violating memory ordering

Why Do Global-Stable Loads Exist?

```
Random* Random::s_rng = 0;
```

Global scope variable

```
Random* Random::get_Rng(void)
{
    if (s_rng == 0)
        s_rng = new Random();
    return s_rng;
}
```

Function to access the variable

Example code from **541.leela_r**

Why Do Global-Stable Loads Exist?

```
Random* Random::s_rng = 0;

Random* Random::get_Rng(void)
{
    if (s_rng == 0)
        s_rng = new Random();
    return s_rng;
}
```

```
624: mov rax, [rip+0x1f4ac5]
62b: test   rax,rax
62e: je     0x638
630: ret
631: nop
638: sub    rsp,0x8
63c: mov    edi,0xc
641: call   0x460
```

Example code from **541.leela_r**

Disassembly (compiled by GCC* with –O3)

Why Do Global-Stable Loads Exist?

Gets initialized only once and never changes

```
Random* Random::s_rng = 0;

Random* Random::get_Rng(void)
{
    if (s_rng == 0)
        s_rng = new Random();
    return s_rng;
}
```

Global-stable load

```
624: mov rax, [rip+0x1f4ac5]
62b: test   rax,rax
62e: je     0x638
630: ret
631: nop
638: sub    rsp,0x8
63c: mov    edi,0xc
641: call   0x460
```

Example code from **541.leela_r**

Disassembly (compiled by GCC* with –O3)

Why Do Global-Stable Loads Exist?

```
1  static inline bool
2    rc_shift_low       ( lzma_range_encoder        * rc , uint8_t *        out ,
3                        size_t      * out_pos   ,   size_t      out_size    )
4  {
5    ...
6    do
7    {
8      if (*              out_pos     ==   out_size      )
9        return true;
10             out  [*  out_pos   ] =     rc - >cache + (uint8_t)(          rc - >low >> 32);
11     ++*            out_pos   ;
12             rc - >cache = 0xFF;
13   } while (          --  rc - >cache_size       != 0);
14   ...
15 }
```

```
4134c8: mov r10d,edi
4134cb: mov      rdi,QWORD    PTR [r15]                    ;   rdi     = *  out_pos
4134ce:    jmp   4134f0    <lzma_lzma_encode+0x3f0>
4134d0:    movzx   r8d,BYTE PTR [rbx+0x14]                 ; r8d =        rc - >cache
4134d5: mov r14,QWORD PTR [        rsp  ]                  ; r14 = out
4134d9: add r8d,r10d
4134dc: mov BYTE PTR [r14+rdi*1],r8b
4134e0:    inc    rdi                                     ; ++*    out_pos
4134e3: mov QWORD PTR [r15],          rdi
4134e6: dec QWORD PTR [rbx+0x8]                           ;   --  rc - >cache_size
4134ea: mov BYTE PTR [rbx+0x14],0xff
4134ee: je 413500          <lzma_lzma_encode+0x400>       ; Breaking the while loop
4134f0:      cmp  QWORD PTR [rsp+0x8],      rdi           ; if (*        out_pos      ==   out_size      )
4134f5:    jne    4134d0   <lzma_lzma_encode+0x3d0>       ; do   - while loop
4134f7:    jmp    4133e9   <lzma_lzma_encode+0x2e9>       ; return true
```

Global-stable loads *accessing function arguments. Fur*
repeatedly called by the same caller with the same

Why Do Global-Stable Loads Exist?

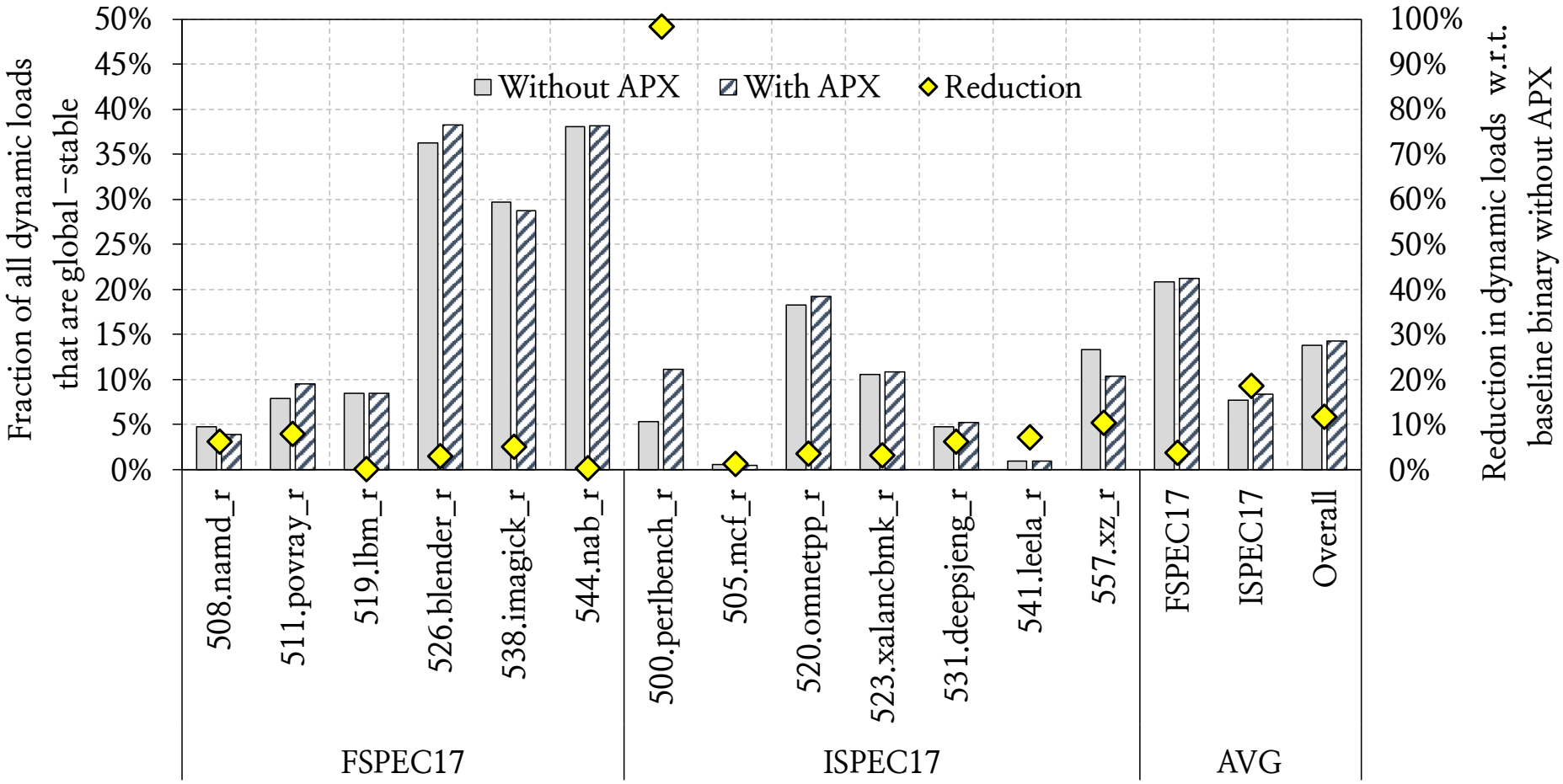
```
1 static inline bool
2   rc_shift_low      ( lzma_range_encoder      * rc , uint8_t *      out ,
3                       size_t     * out_pos   ,   size_t     out_size   )
4 {
5
```

Global-stable loads exist for many reasons:

- Accessing global variables
- Accessing local variables of an inline function
- Limited architectural registers
- …

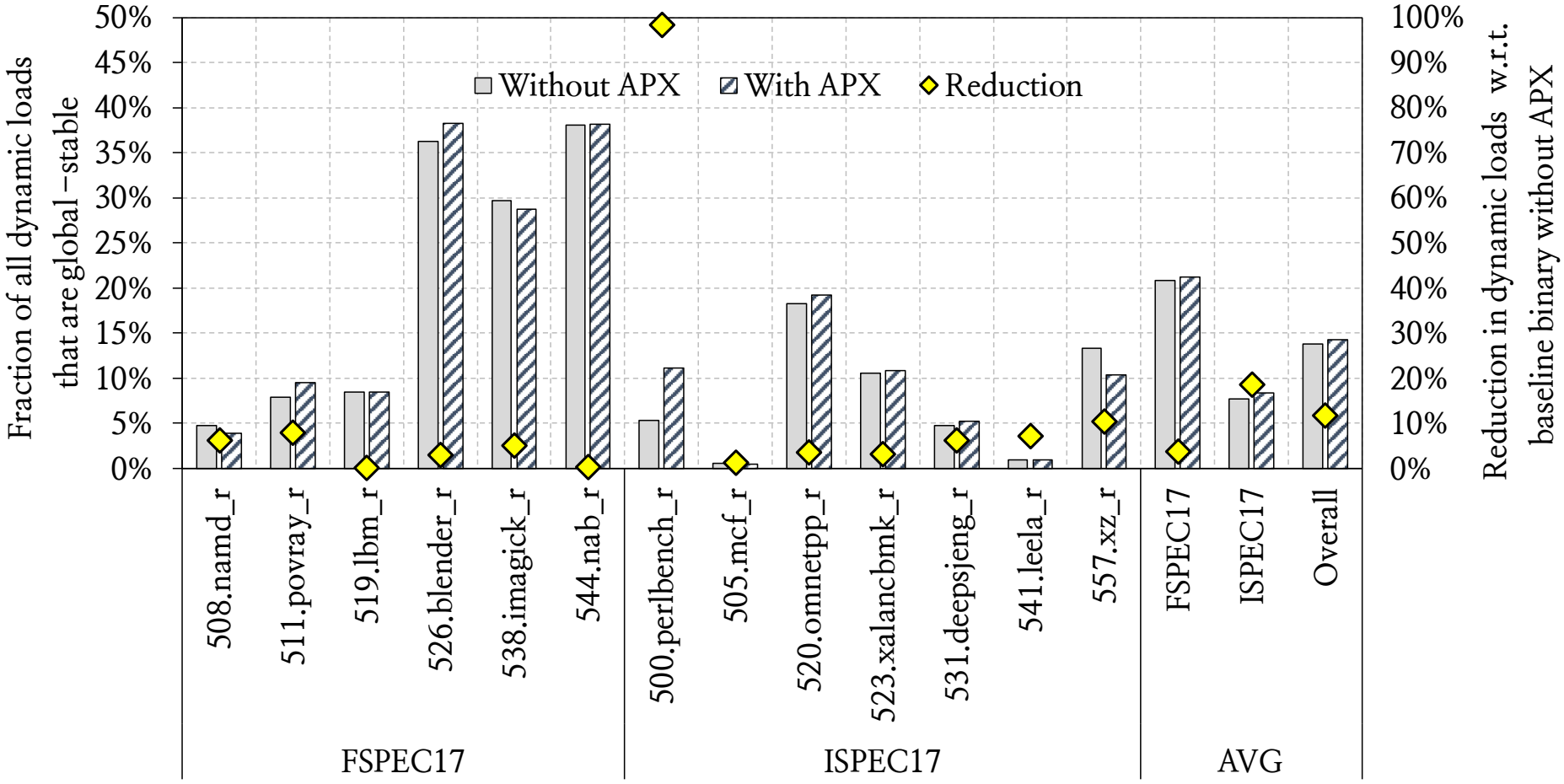
```
4134e6: dec QWORD PTR [rbx+0x8]                    ;   --  rc - >cache_size
4134ea: mov BYTE PTR [rbx+0x14],0xff
4134ee: je 413500          <lzma_lzma_encode+0x400>  ; Breaking the while loop
4134f0:     cmp  QWORD PTR [rsp+0x8],     rdi        ; if (*       out_pos     ==  out_size     )
4134f5:     jne    4134d0  <lzma_lzma_encode+0x3d0>  ; do   - while loop
4134f7:     jmp    4133e9  <lzma_lzma_encode+0x2e9>  ; return true
```

Effects of Increasing Architectural Registers



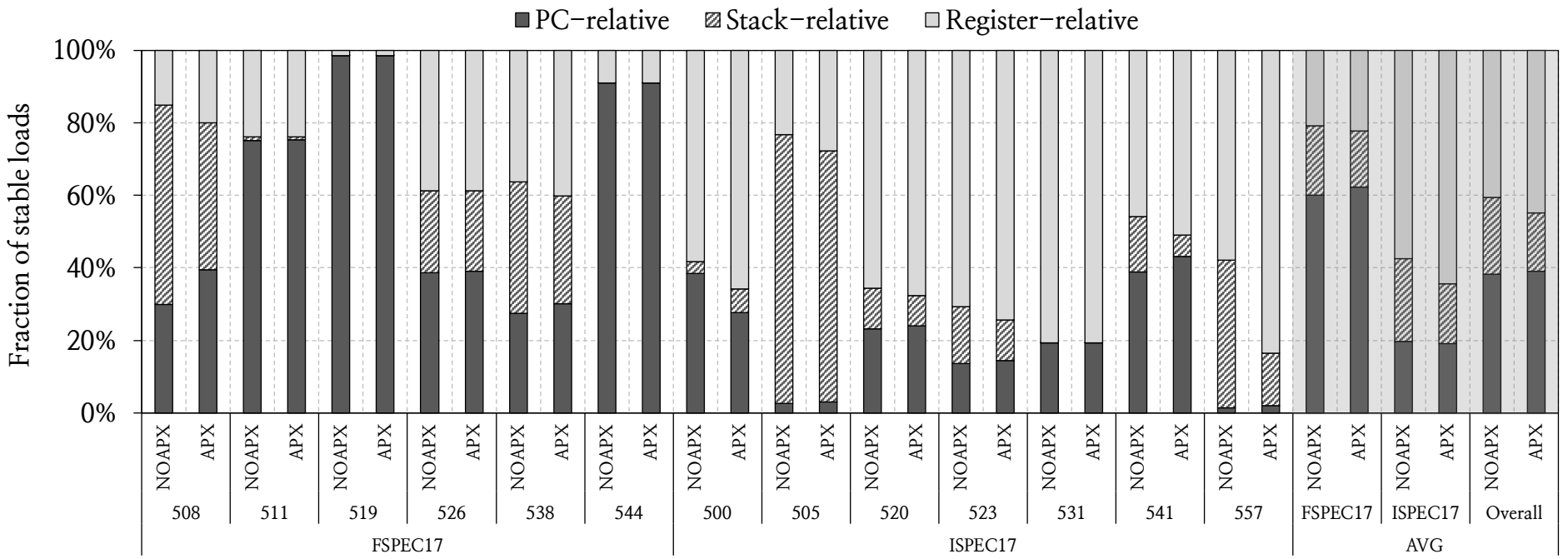
Fraction of global-stable loads are nearly same without or with APX

Effects of Increasing Architectural Registers



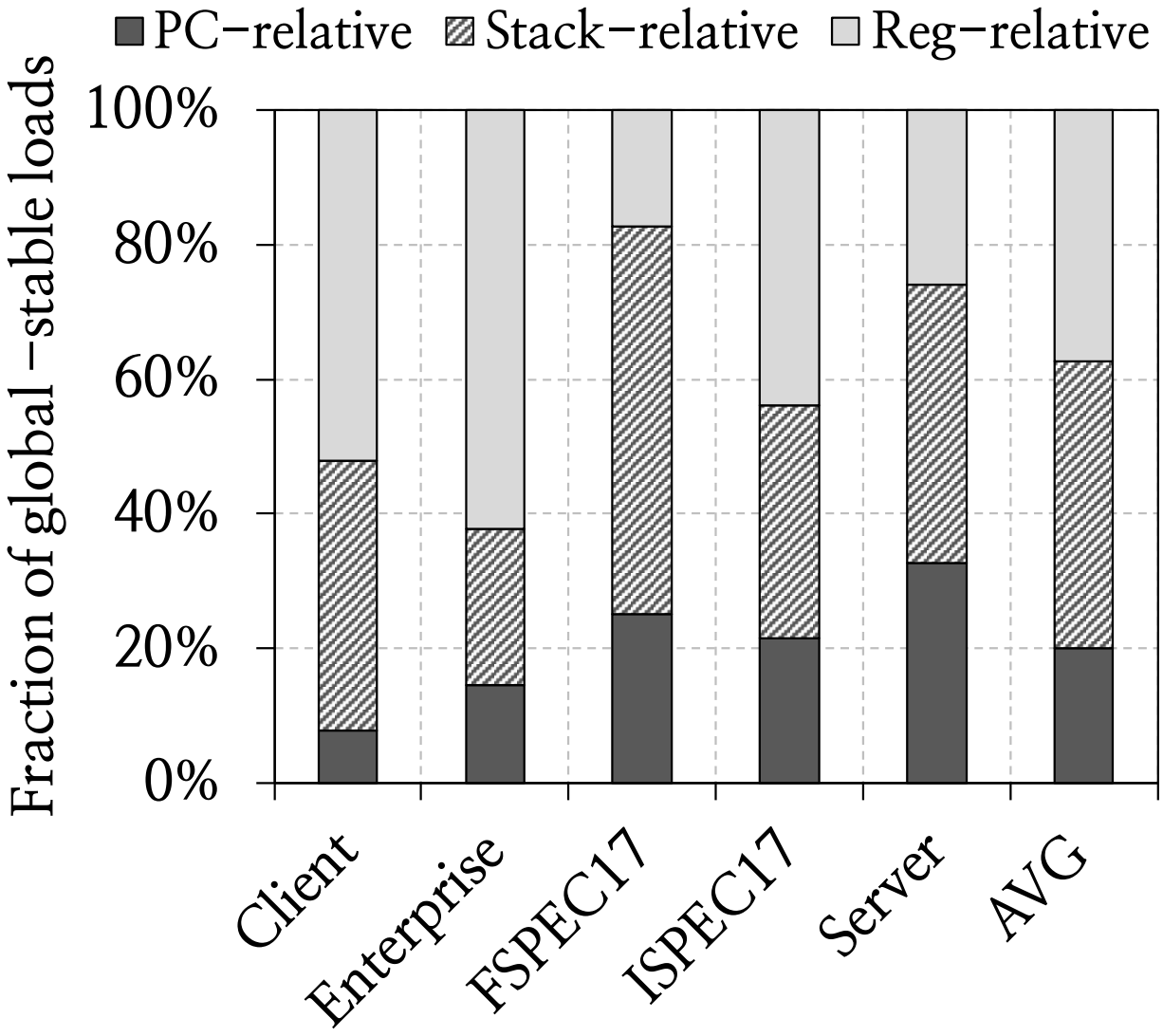
Fraction of global-stable loads (i.e., Constable's opportunities) are much higher than reduction in loads by APX

Effects of Increasing Architectural Registers

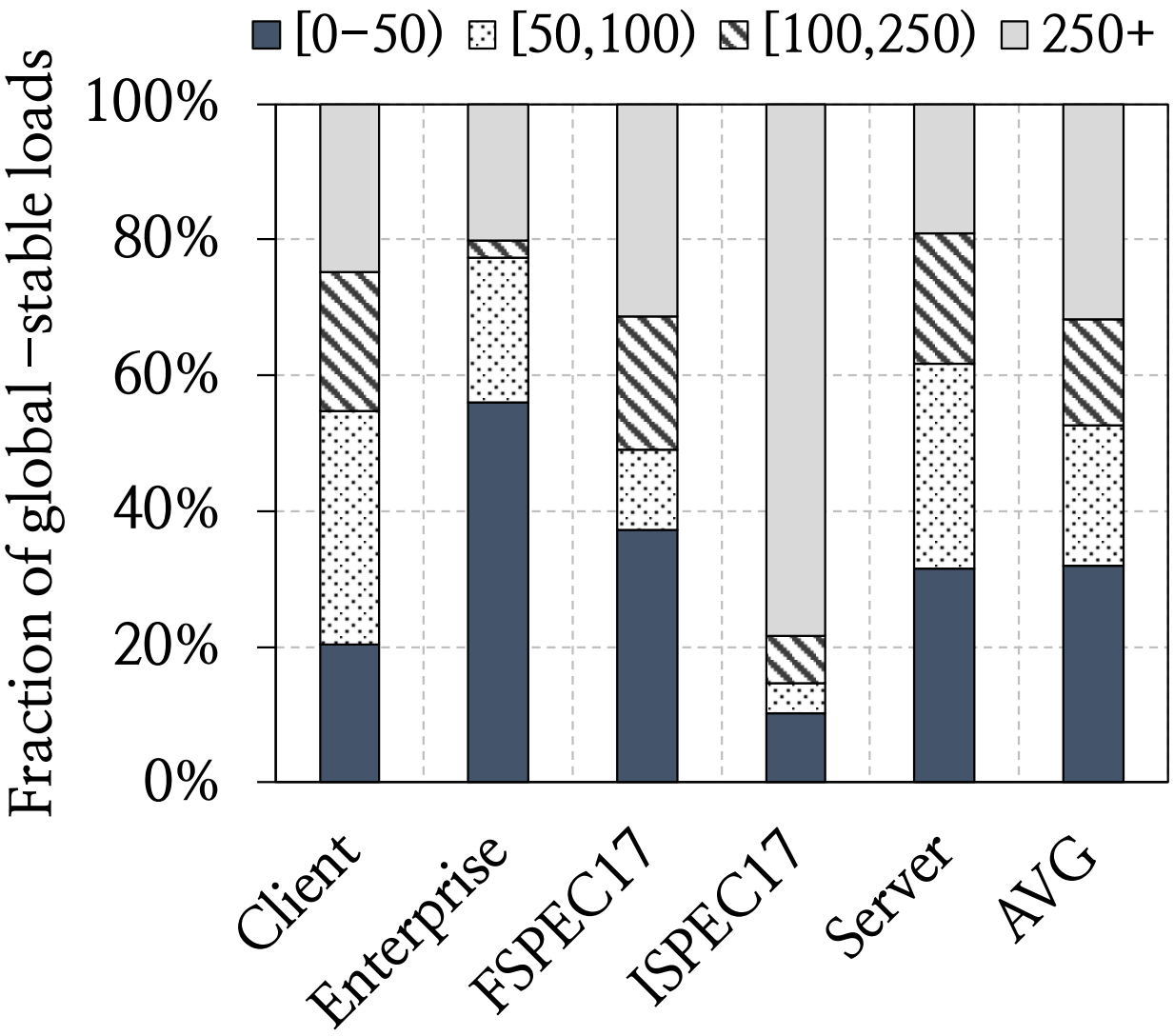


The profile of global-stable loads stays largely unchanged after doubling registers using APX

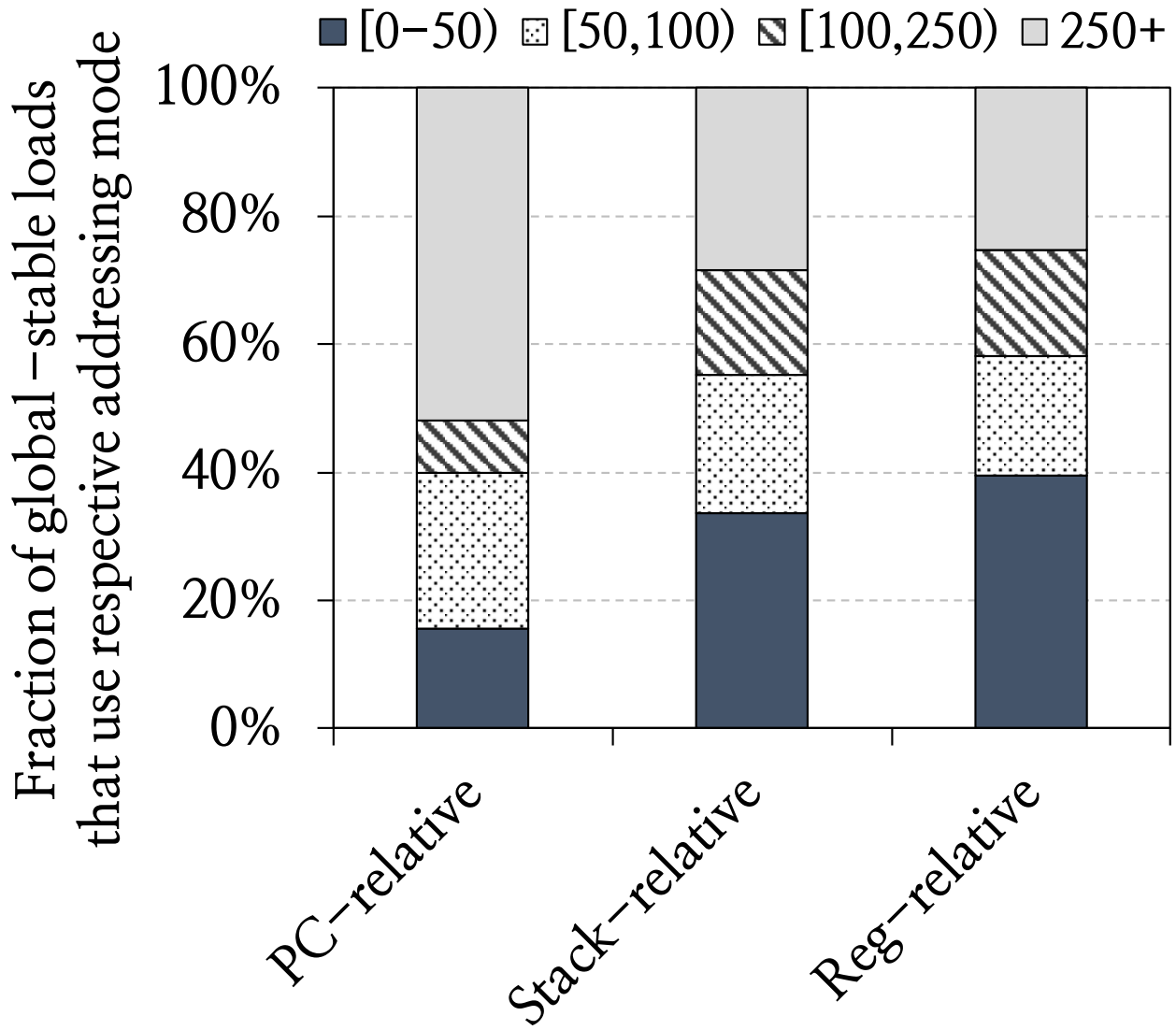
Characterization of Global-Stable Loads (I)



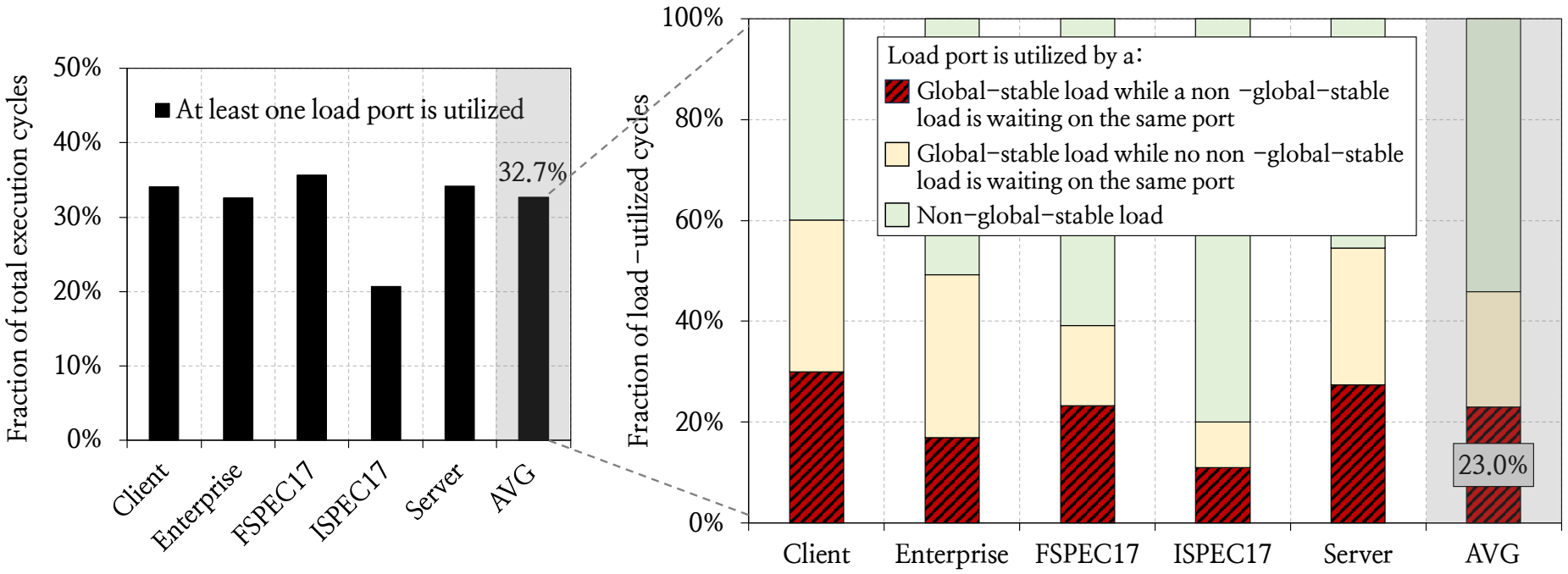
Characterization of Global-Stable Loads (II)



Characterization of Global-Stable Loads (III)

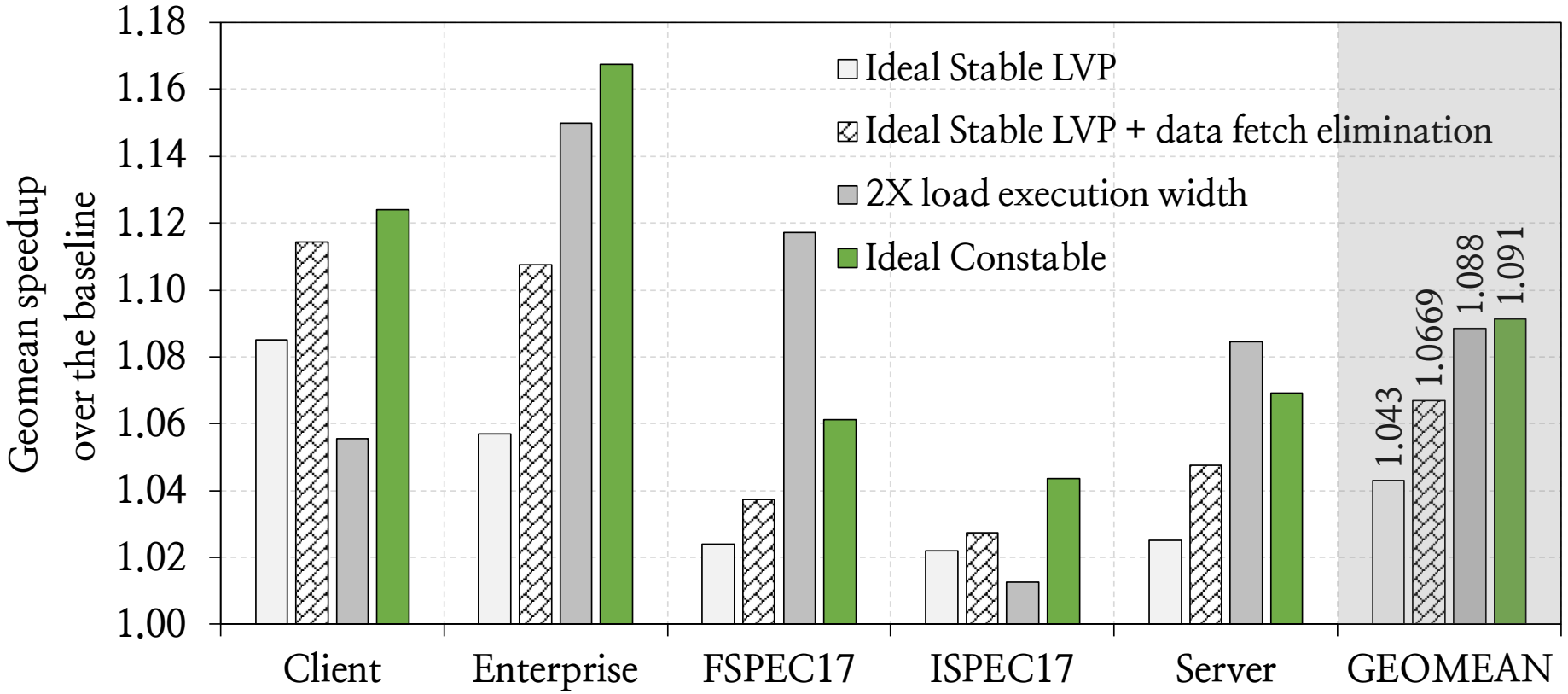


Resource Dependence by Global-Stable Loads

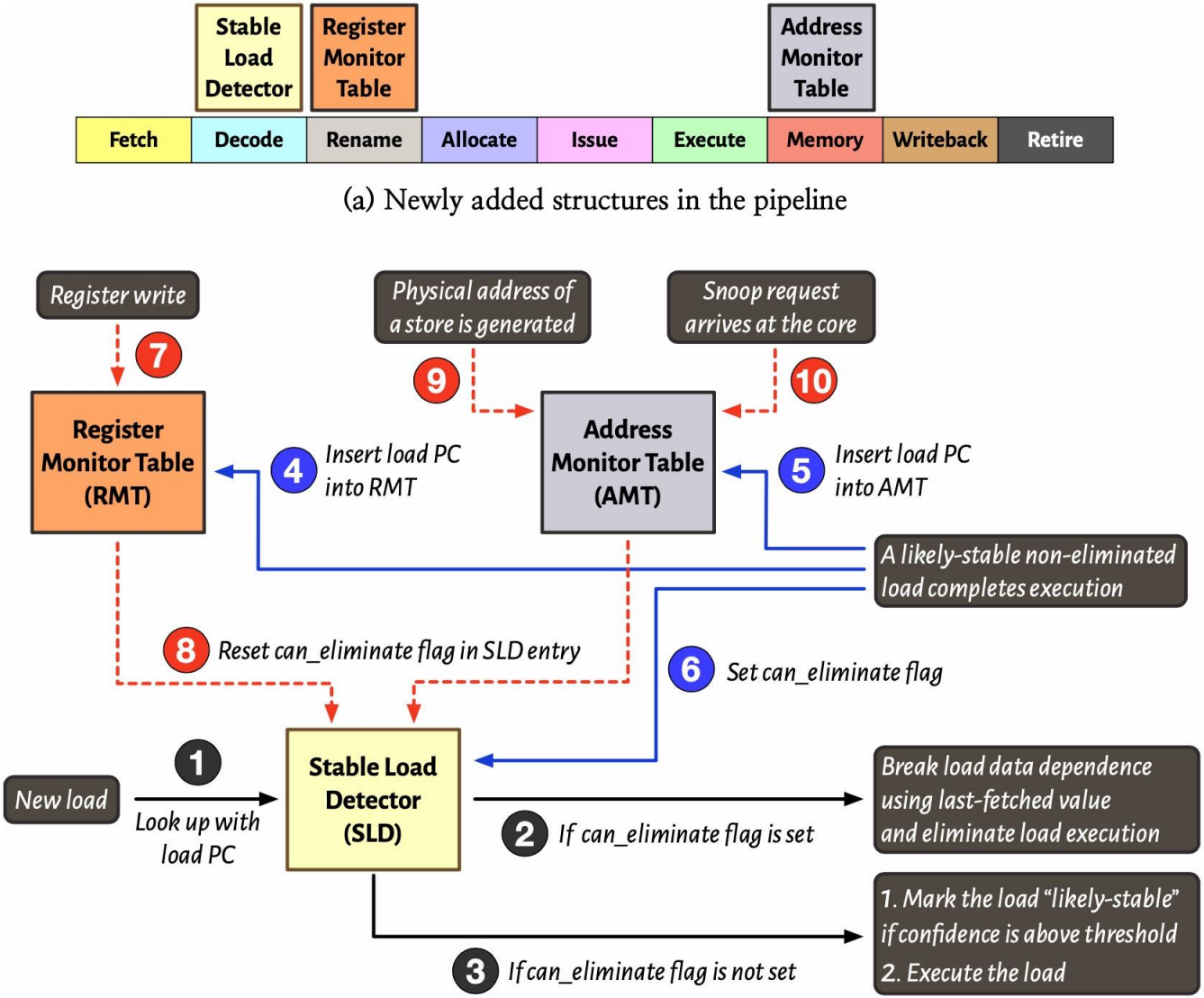


Global-stable loads cause significant resource dependence

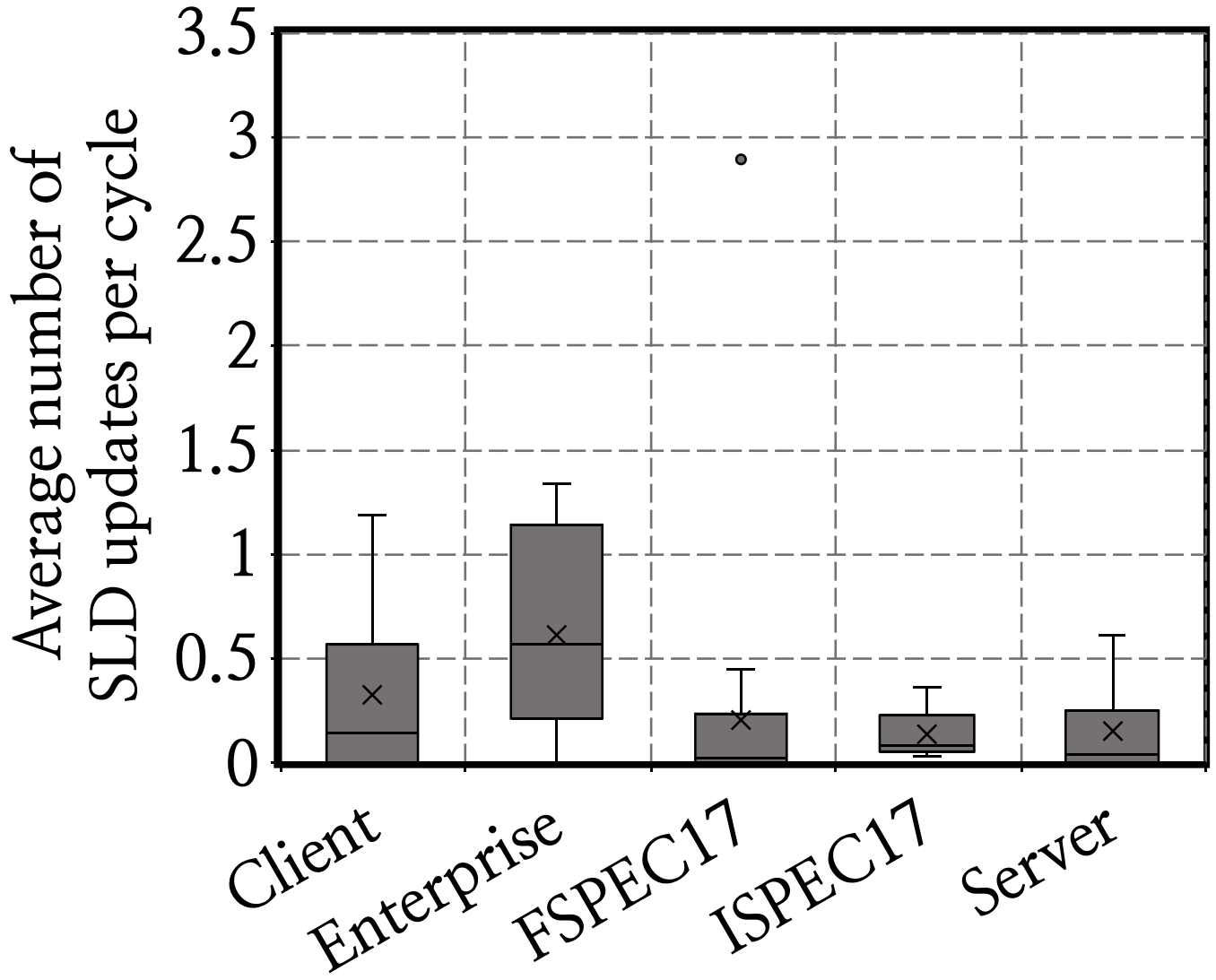
Performance Headroom Analysis



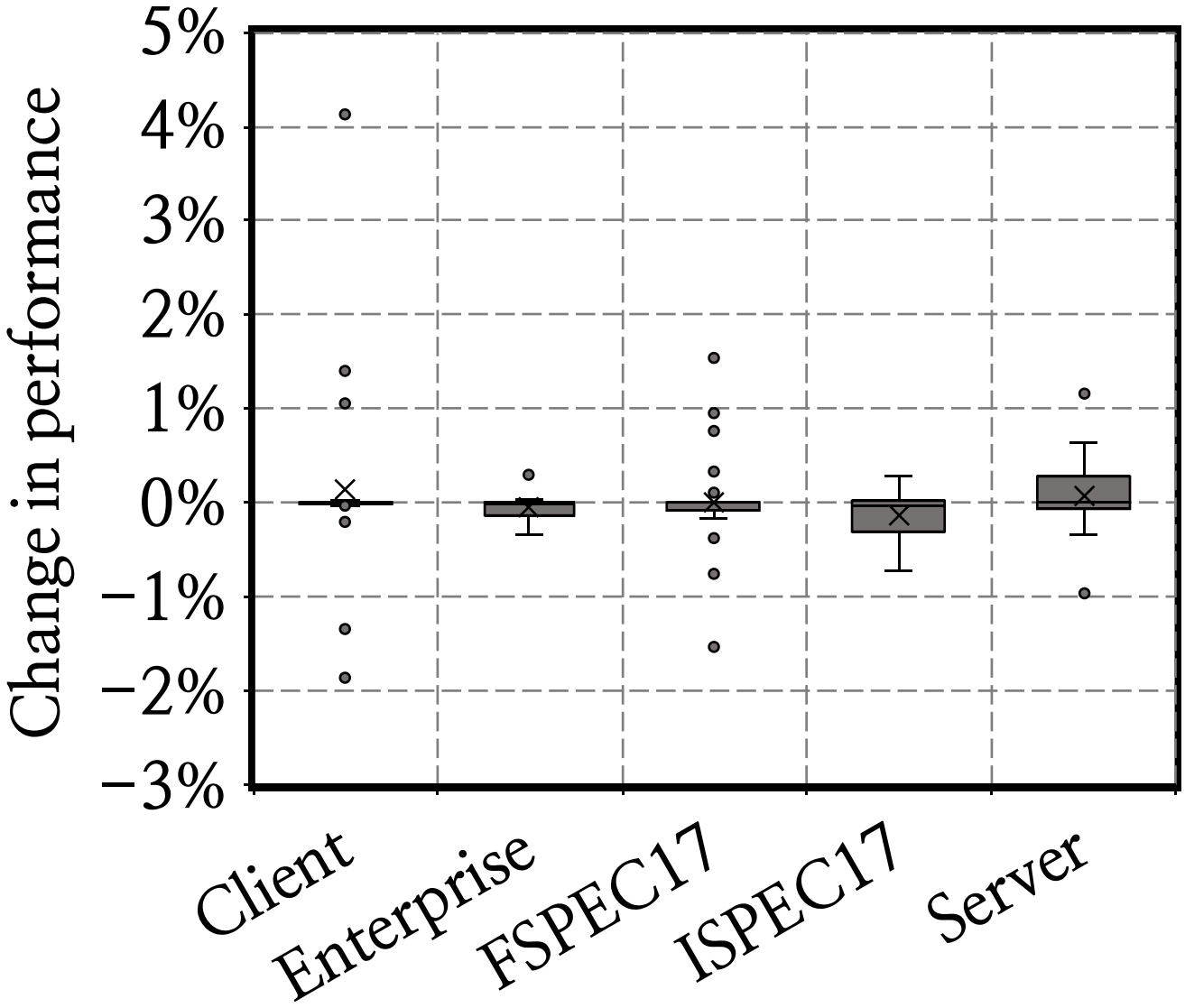
Constable Overview



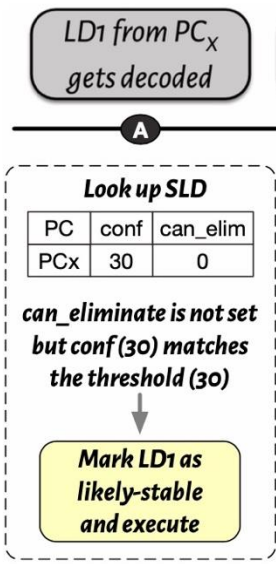
Architecting SLD



Effect of Wrong-Path Update

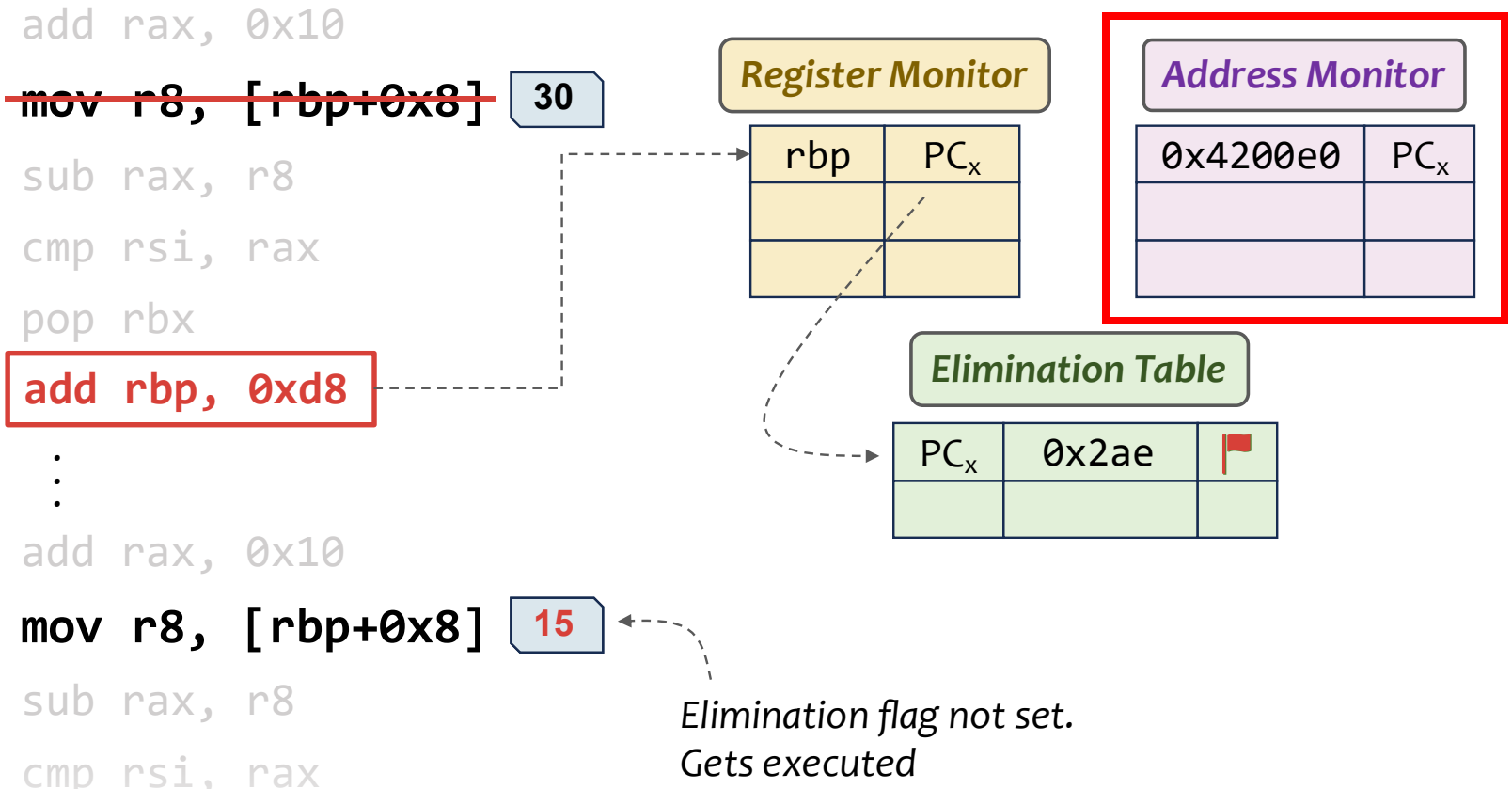


An Example of Constable's Operation



Ensuring Coherence

- Constable relies on **snoop requests** to observe **modifications to a memory address by other cores**



Ensuring Coherence

- Constable relies on **snoop requests** to observe **modifications to a memory address by other cores**

- Evicting a cacheline from core-private cache **resets the core-valid bit** in directory
  - What if a **clean eviction**?
    - Unnecessary elimination opportunity loss

- Constable **pins the CV-bit of an eliminated load's cacheline**
  - Even if the cacheline gets evicted from core-private cache, snoop request gets delivered

Constable's Storage Overhead

Structure	Description	Size
SLD	• # entries: 512 (32 sets × 16 ways) • Entry size: tag (24b) + addr (32b) + val (64b) + confidence level (5b) + can_eliminate flag (1b)	**7.9 KB**
RMT	• 16 load PCs for each stack registers (RSP and RBP) • 8 load PCs for each remaining 14 architectural registers in x86-64	**0.4 KB**
AMT	• # entries: 256 (32 sets × 8 ways) • Entry size: physical address tag (32b) + # hashed load PCs (4 × 24b)	**4.0 KB**
Total		**12.4 KB**

Area and Power Overhead of Constable's Structures

Component	Read access energy (pJ)	Write access energy (pJ)	Leakage power (mW)	Area (mm2)
SLD (7.9KB, 3R/2W ports)	10.76	16.70	1.02	0.211
RMT (0.4KB, 2R/6W ports)	0.15	0.20	0.31	0.004
AMT (4.0KB, 1R/1W ports)	1.58	4.22	0.74	0.017

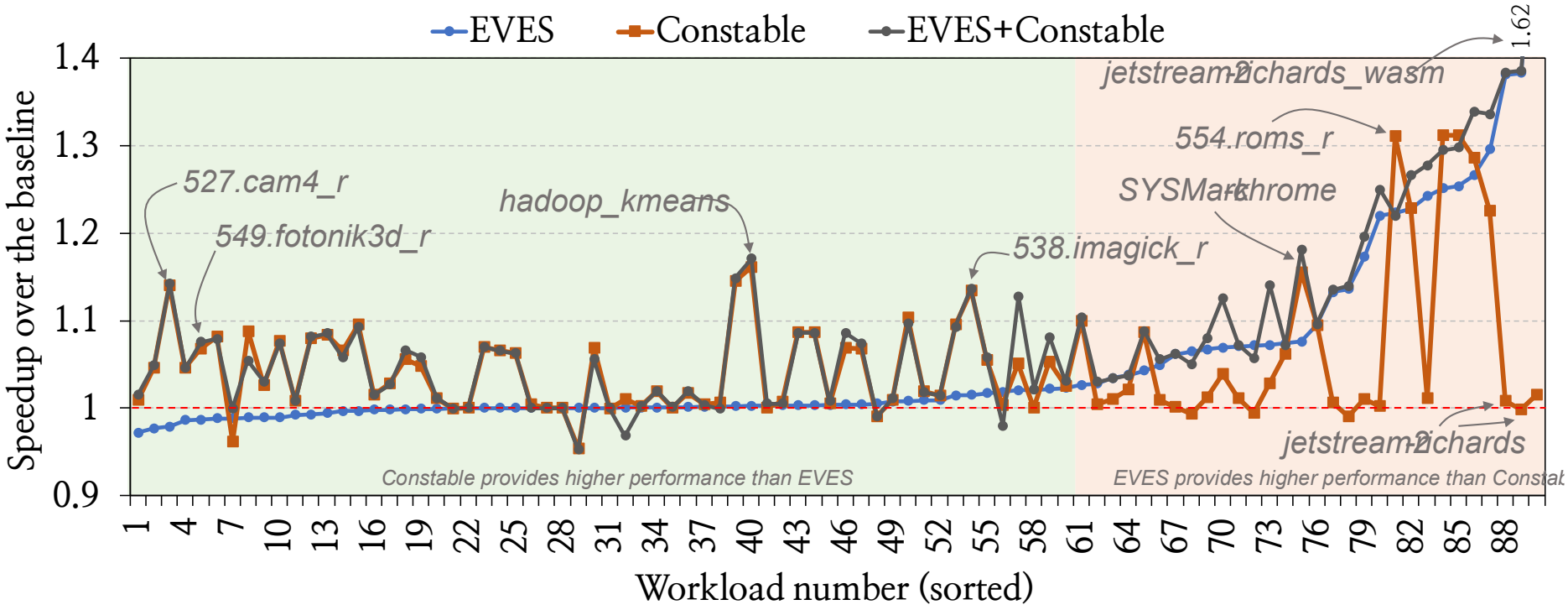
Simulated System Parameters

Basic	x86-64 core clocked at 3.2 GHz with 2-way SMT support
Fetch & Decode	8-wide fetch, TAGE/ITTAGE branch predictors [156], 20-cycle misprediction penalty, 32KB 8-way L1-I cache, 4K-entry 8-way micro-op cache, 6-wide decode, 144-entry IDQ, loop-stream detector [41]
Rename	6-wide, 288 integer, 220 512-bit and 320 256-bit physical registers, **Memory Renaming [177], zero elimination [68], move elimination [64, 68], constant folding [64, 68], branch folding [60]**
Allocate	512-entry ROB, 240-entry LB, 112-entry SB, 248-entry RS
Issue & Retire	6-wide issue to 12 execution ports; 5, 3, 2, and 2 ports for ALU, load, store-address, and store-data execution. Port 0, 1, and 5 are used for vector instructions. Aggressive out-of-order load scheduling with memory dependence prediction [51, 120], 6-wide retire
Caches	**L1-D**: 48KB, 12-way, 5-cycle latency, LRU, PC-based stride prefetcher [69]; **L2**: 2MB, 16-way, 12-cycle round-trip latency, LRU, stride + streamer [47] + SPP [101]; **LLC**: 3MB, 12-way, 50-cycle data round trip latency [15, 36], dead-block-aware replacement policy [100], streamer, MESIF [118] protocol
Memory	4 channels, 2 ranks/channel, 8 banks/rank, 2KB row-buffer/rank, 64b bus/channel, DDR4, tCAS=22ns, tRCD=22ns, tRP=22ns, tRAS=56ns
Optional	• **EVES** [155]: exact design of the CVP-1 **32KB** storage track winner • **ELAR** [34]: with additional adder and RSP register in decode stage • **RFP** [164]: 2K-entry prefetch table, 64-entry page address table, 128-entry RFP-inflight table. Total size: **12.5KB** • **Constable** (this work). Total size: **12.4KB**

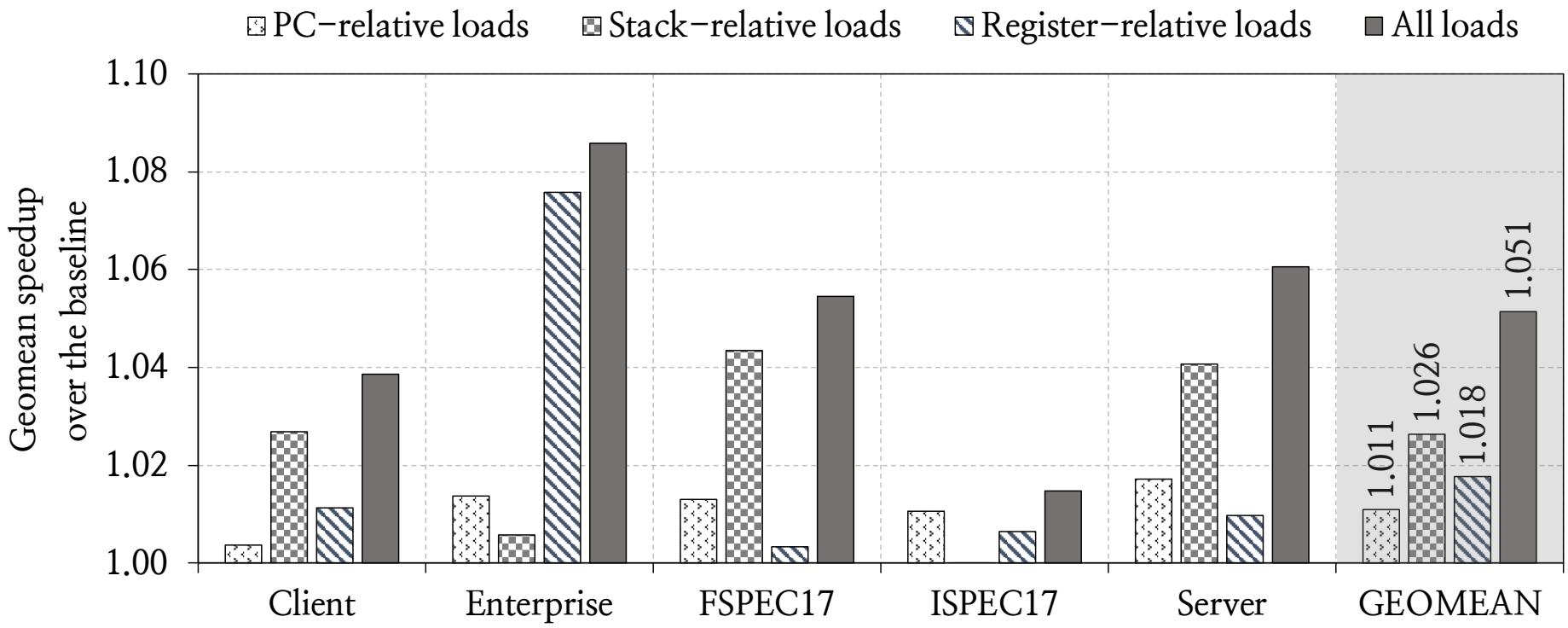
Evaluated Workloads

Suite	#Workloads	#Traces	Example Workloads
Client	16	22	DaCapo [3], SYSmark [20], TabletMark [21], JetStream2 [12]
Enterprise	9	14	SPECjEnterprise [19], SPECjbb [18], LAMMPS [13]
FSPEC17	13	29	All from SPECrate FP 2017 [17]
ISPEC17	10	11	All from SPECrate Integer 2017 [17]
Server	10	14	Hadoop [1], Linpack [9], Snort [16], BigBench [2]

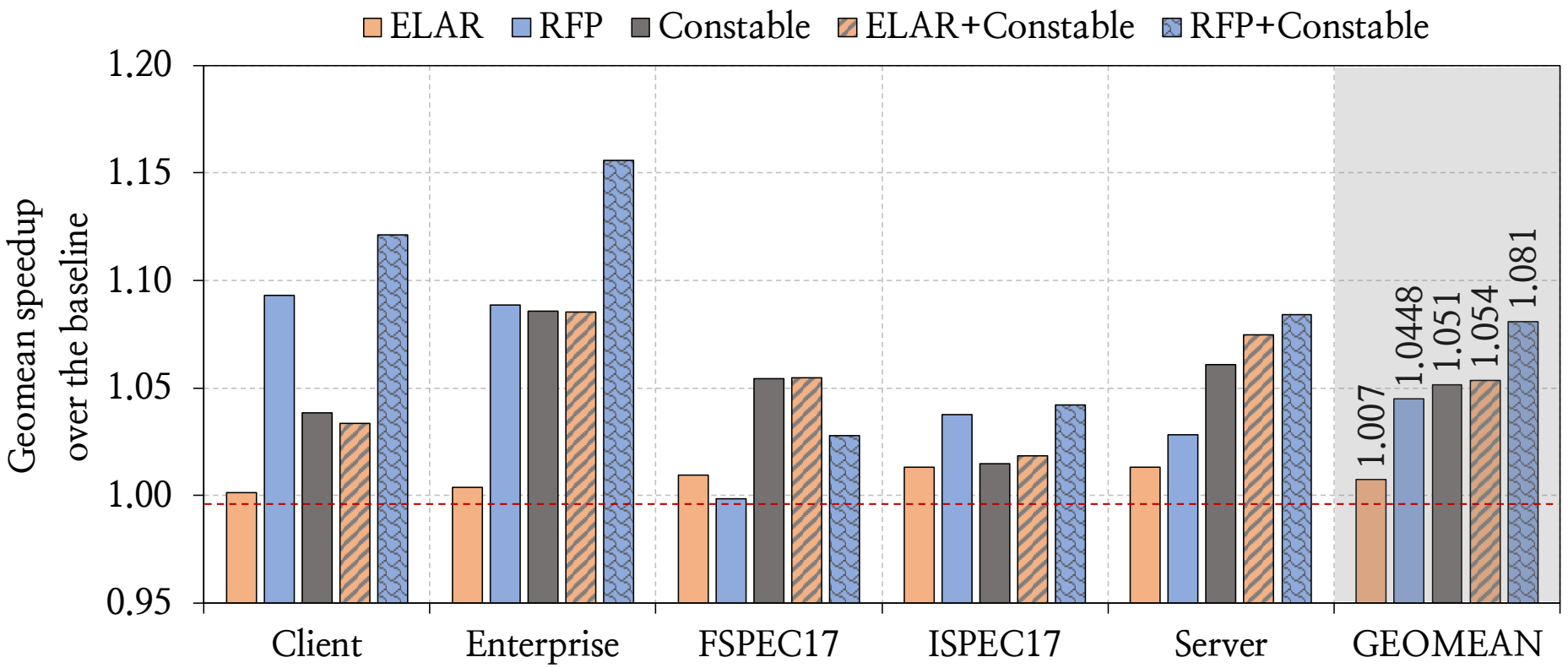
Workload-Wise Performance



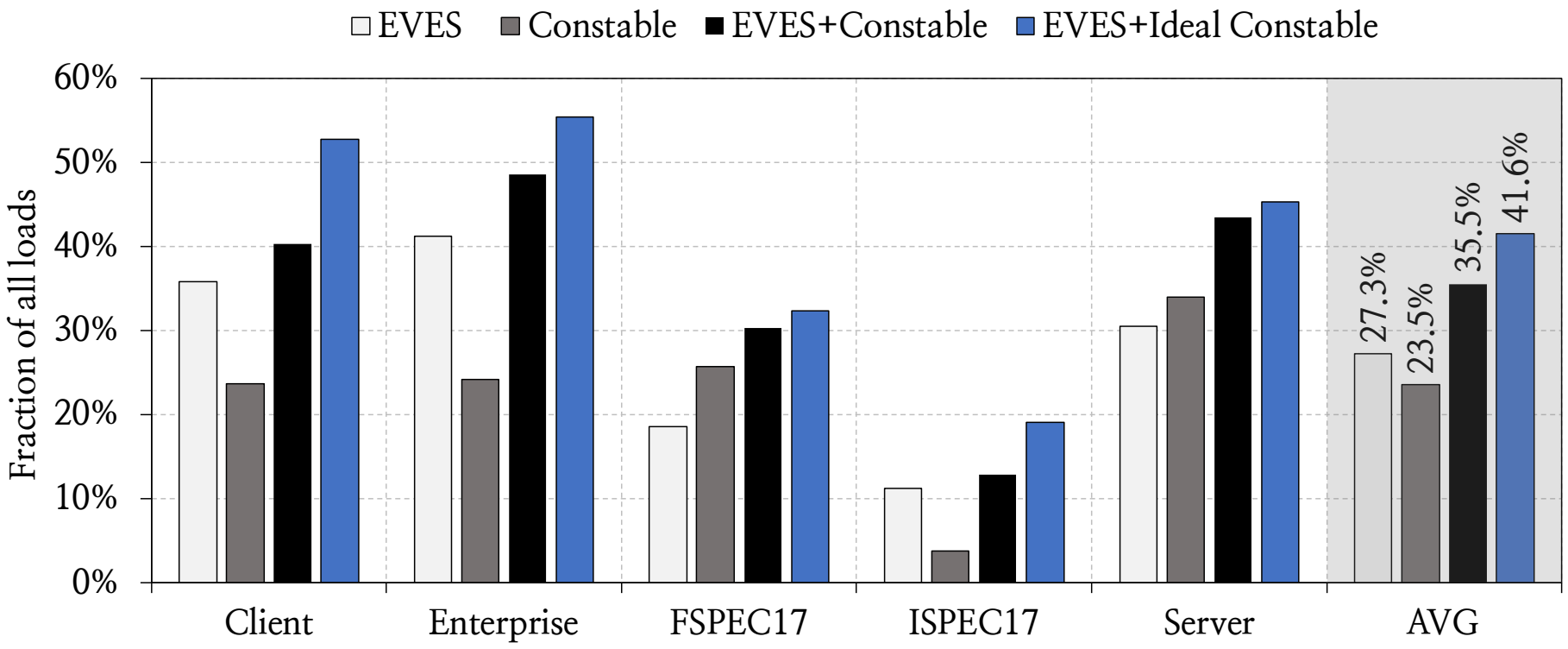
Load Category-Wise Performance



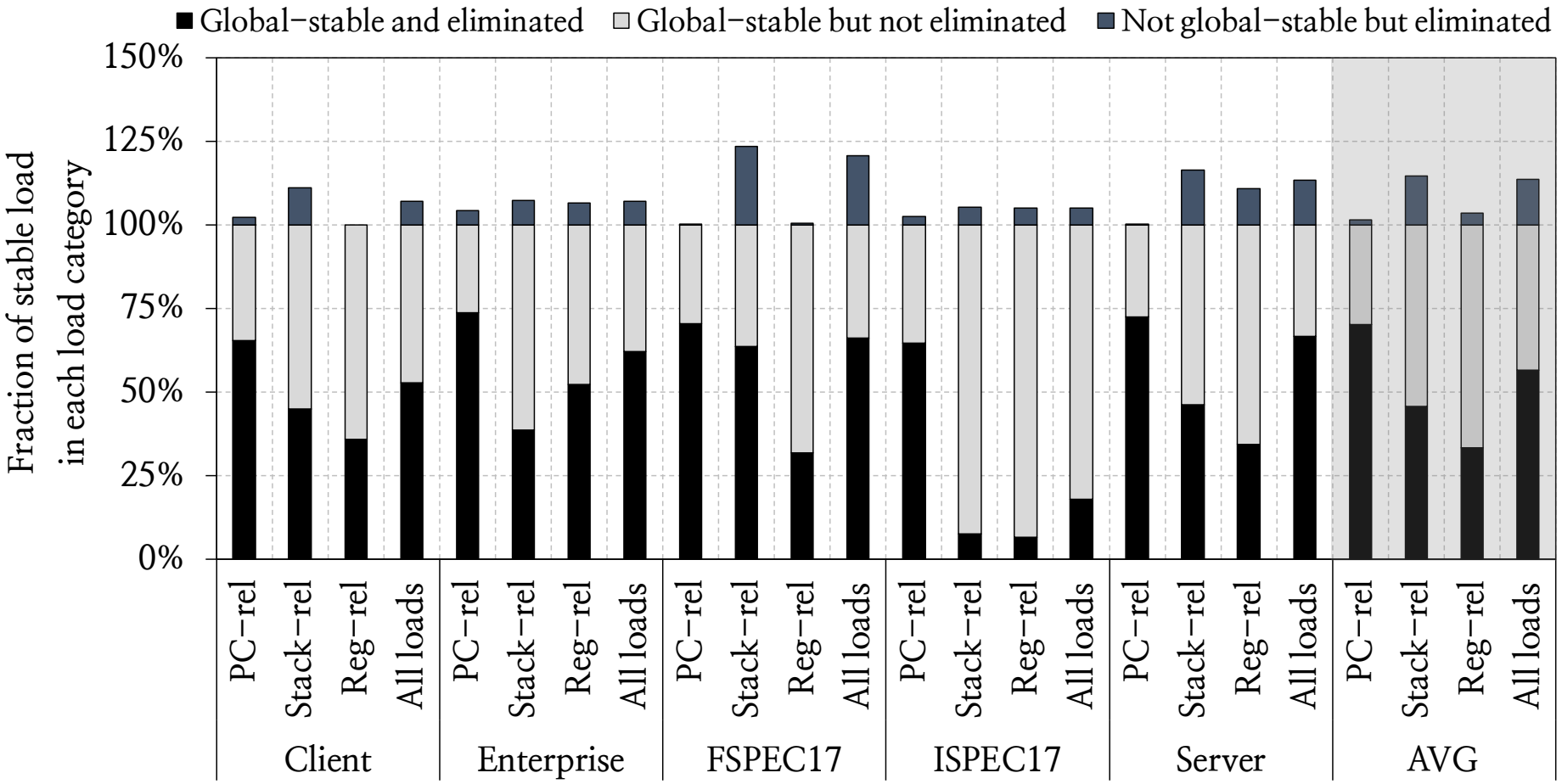
Performance Comparison with Prior Works



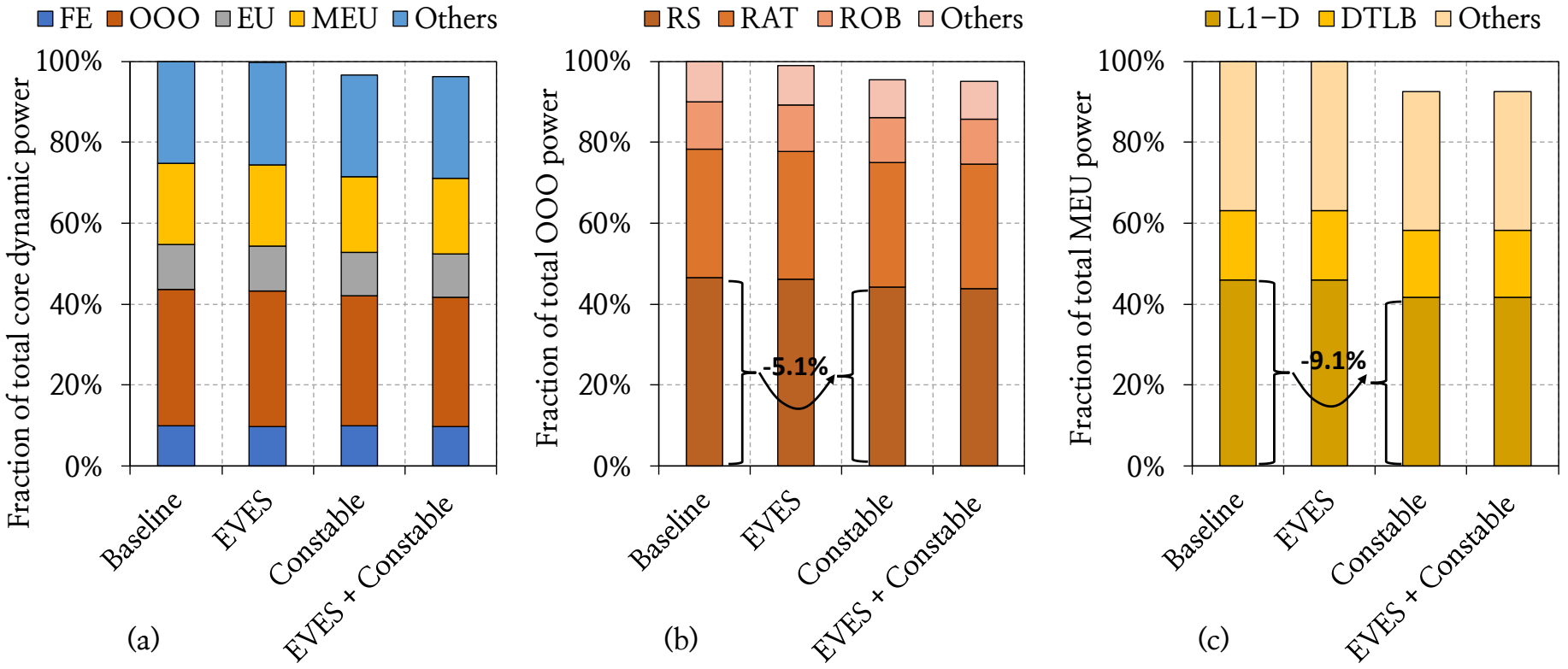
Elimination Coverage



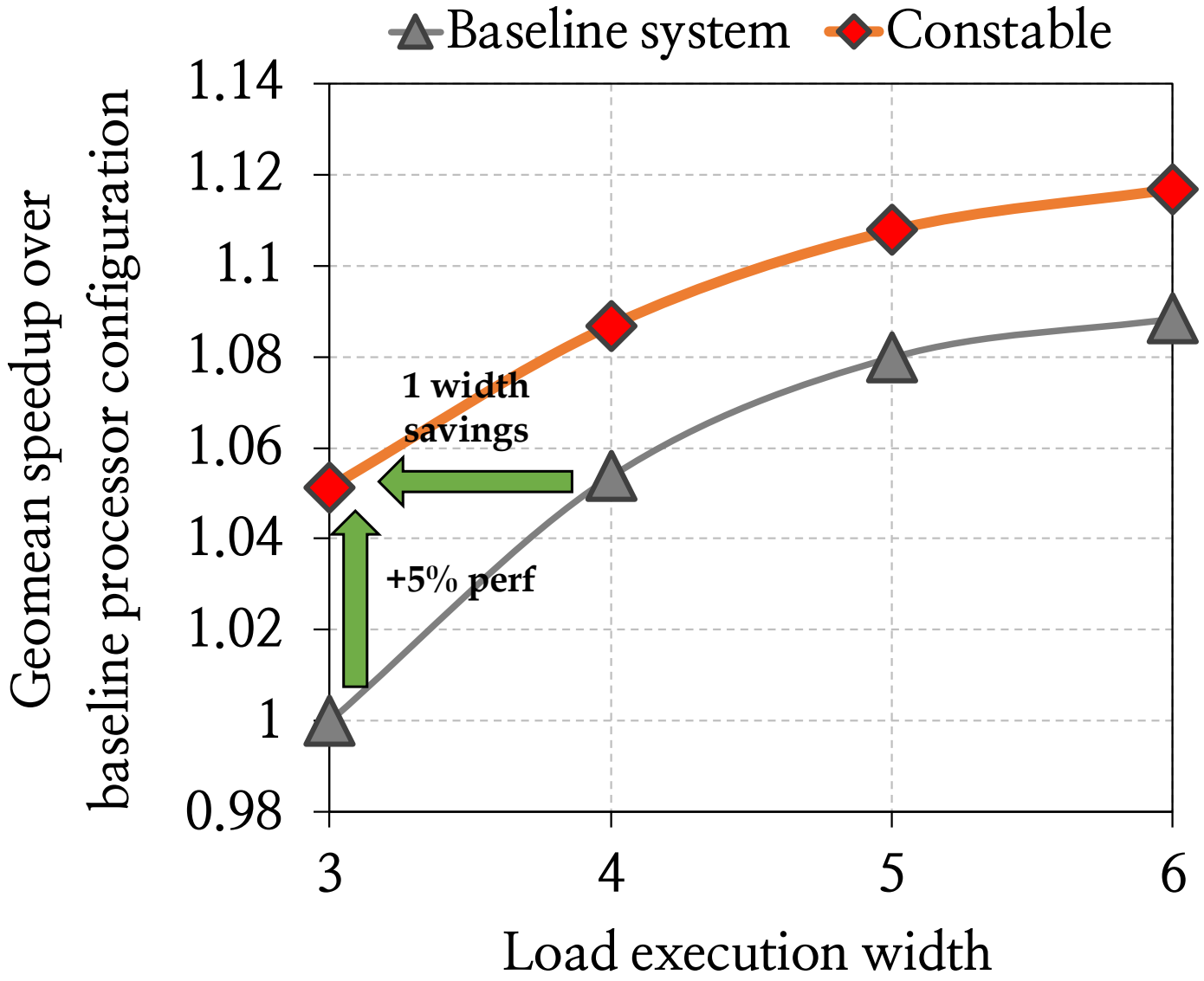
Coverage of Global-Stable Loads



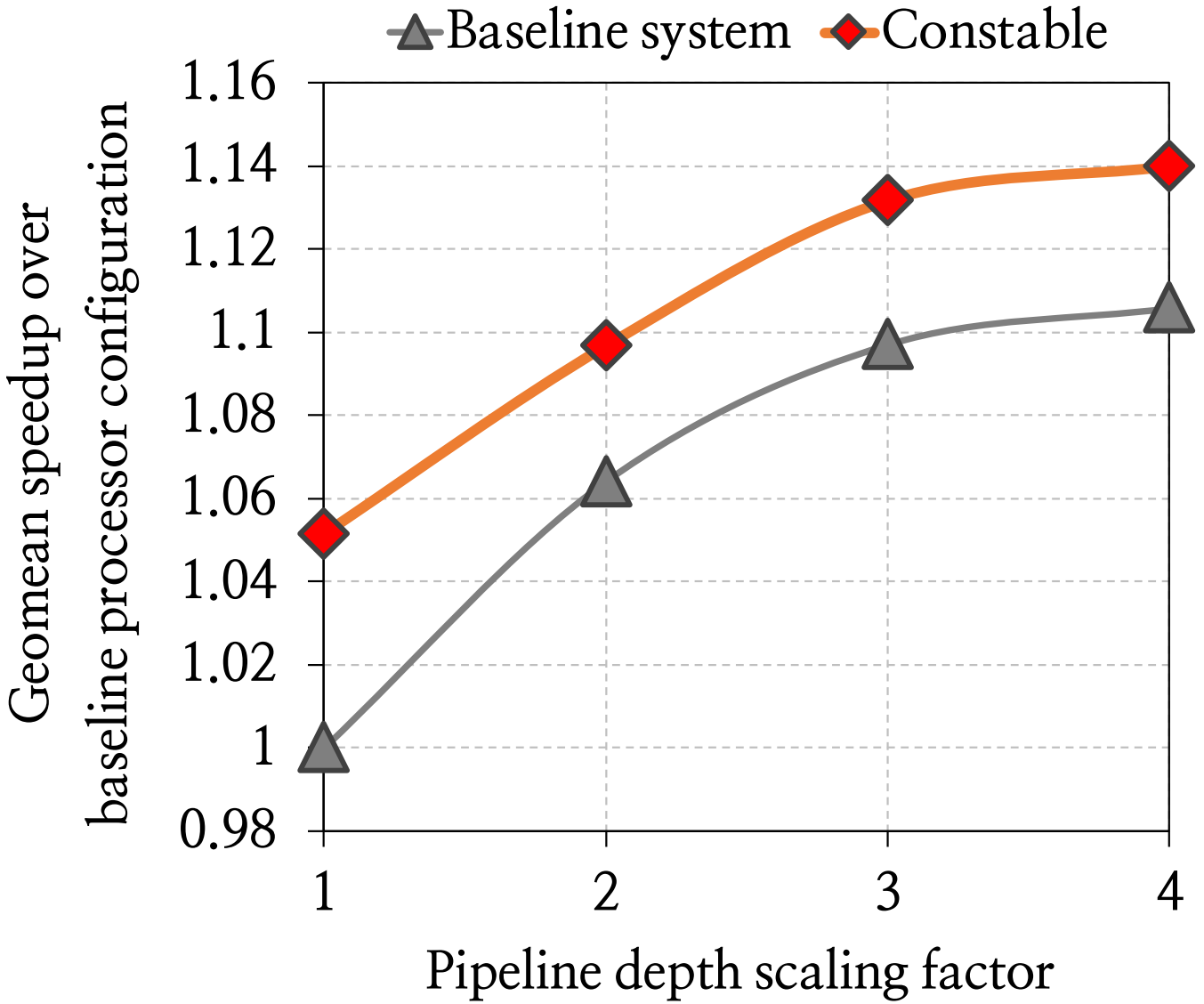
Reduction in Dynamic Power



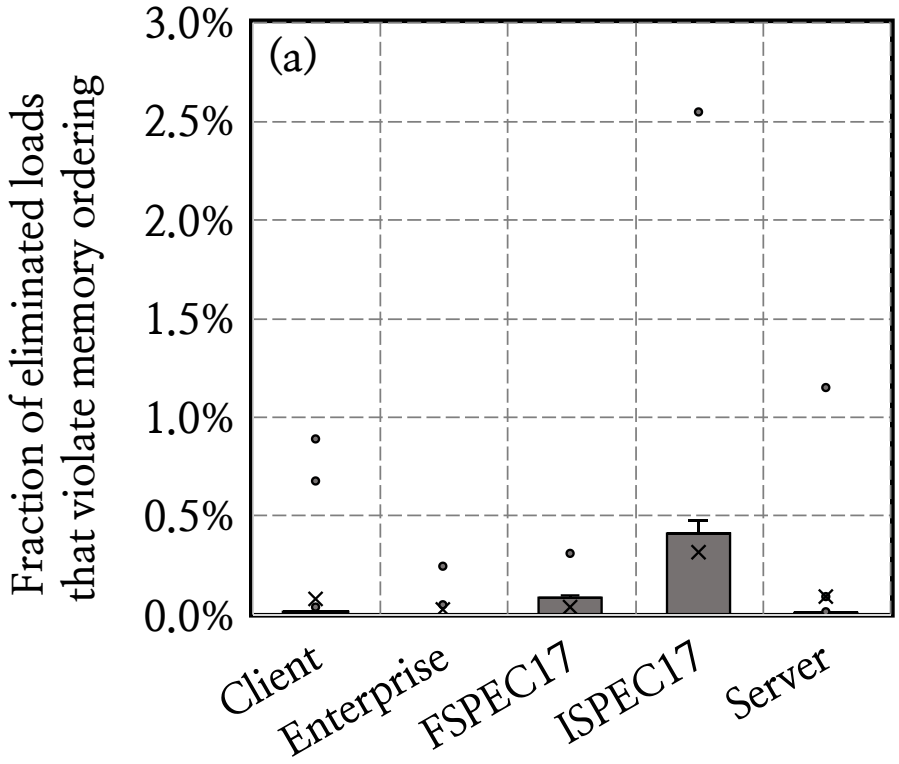
Performance Sensitivity to Load Execution Width Scaling



Performance Sensitivity to Pipeline Depth Scaling

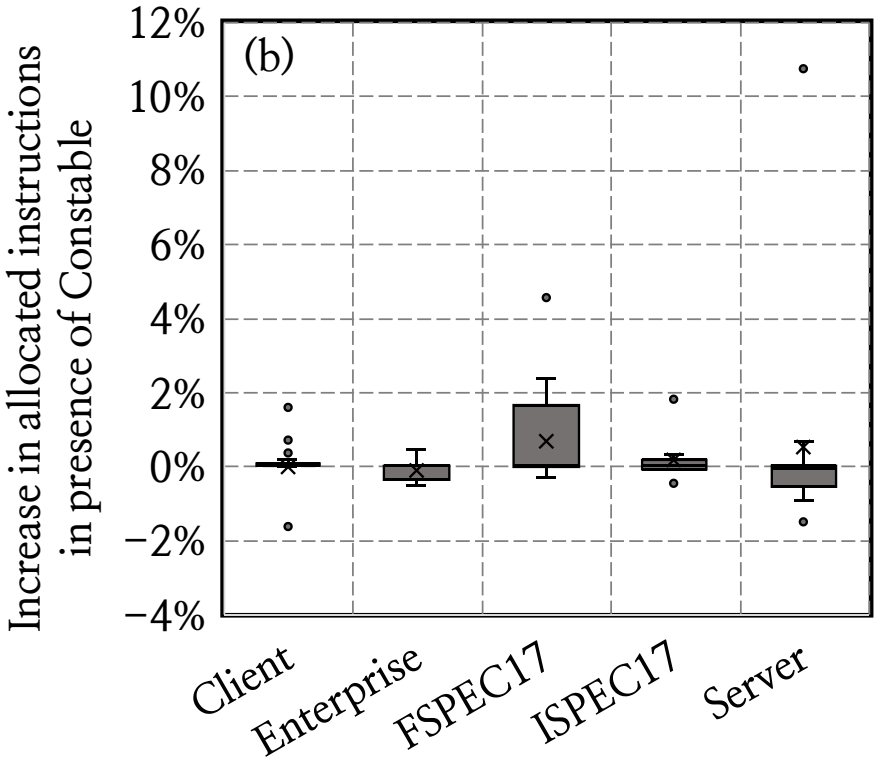


Eliminated Loads that Violates Memory Ordering



Only 0.09% of all eliminated loads violate memory ordering

Eliminated Loads that Violates Memory Ordering



Eliminated loads that violate memory ordering increase number of allocated instructions only by 0.3%