CONSTABLE

Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

Rahul Bera* Adithya Ranganathan* Joydeep Rakshit Sujit Mahto

Anant V. Nori Jayesh Gaur Ataberk Olgun Konstantinos Kanellopoulos

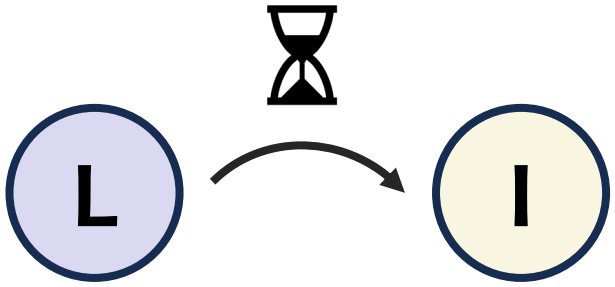
Mohammad Sadrosadati Sreenivas Subramoney Onur Mutlu

Key Problem

Load instructions are a key limiter of instruction-level parallelism (ILP)

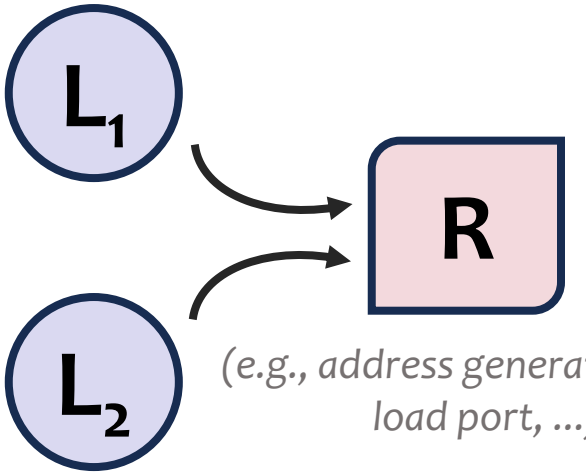
Data Dependence

Stall load-dependent instructions due to **long load execution latency**



Resource Dependence

Stall other loads due to contention **in load execution resources**

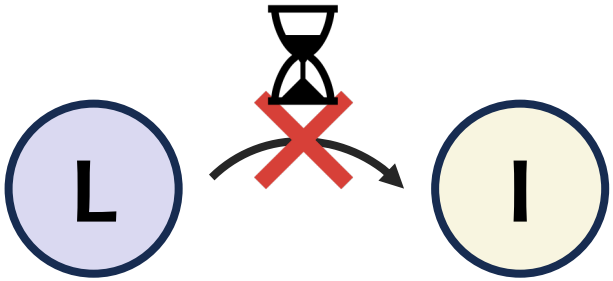


Prior Works on Tolerating Load Latency

- Load value prediction (LVP) [Lipasti+, ASPLOS'96; Sazeides+, MICRO'96; ...]
- Memory renaming (MRN) [Moshovos+, ISCA'97; Tyson+, MICRO'97; ...]

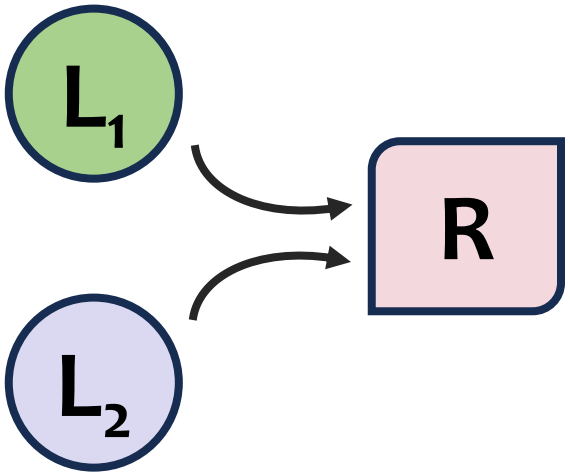
Mitigate Data Dependence

By **speculatively executing** load-dependent instructions using a predicted load value



Do Not Mitigate Resource Dependence

Predicted load still gets executed to verify speculation, consuming execution resources



Motivation

Safely breaking load data dependency
without executing a load instruction
may provide additional performance benefits

How do we start?

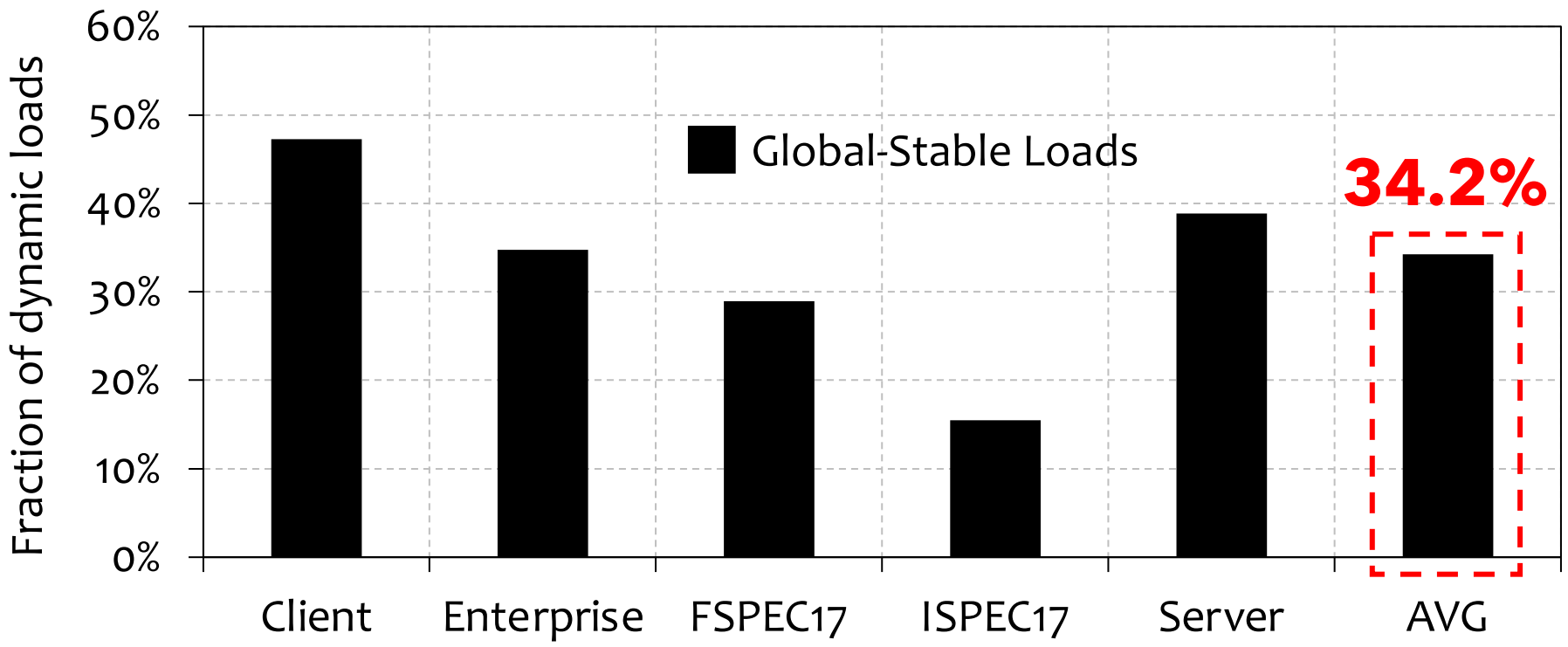
By finding load instructions that repeatedly produce identical results across dynamic instances

Key Finding I: Global-Stable Loads

- Some loads repeatedly fetch **the same data value** from **same load address** across **entire workload**
  - Both operations, **address generation** & **data fetch**, produce identical results across all dynamic instances
  - Prime targets for **breaking data dependency without execution**

Global-Stable Load

Key Finding I: Global-Stable Loads



Nearly **1 in every 3** dynamic loads is a **global-stable load**

Across a wide range of 90 workloads

In the Paper: Analysis of Global-Stable Loads

- **Why do these loads even exist** in well-optimized real-world workloads?
  - Accessing **global-scope variables**
  - Accessing **local variables** of **inline functions**
  - **Limited** set of architectural **registers**

- **Can increasing architectural registers help**?
  - **Very small change** even after doubling x64 registers

- **Deeper characterization** of global-stable loads
  - Which **addressing mode** do they use?
  - How **far away do they appear** in a workload?

In the Paper: Analysis of Global-Stable Loads

- Why do these loads even exist in well-optimized
-
-
- How

Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

*Rahul Bera[1] *Adithya Ranganathan[2] Joydeep Rakshit[2] Sujit Mahto[2]
Anant V. Nori[2] Jayesh Gaur[2] Ataberk Olgun[1] Konstantinos Kanellopoulos[1]
Mohammad Sadrosadati[1] Sreenivas Subramoney[2] Onur Mutlu[1]

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

Load instructions often limit instruction-level parallelism (ILP) in modern processors due to data and resource dependences they cause. Prior techniques like Load Value Prediction (LVP) and Memory Renaming (MRN) mitigate load data dependence by predicting the data value of a load instruction. However, they fail to mitigate load resource dependence as the predicted load instruction gets executed nonetheless (even on a correct prediction), which consumes hard-to-scale pipeline resources that otherwise could have been used to execute other load instructions.

Our goal in this work is to improve ILP by mitigating both *load data dependence and resource dependence. To this end, we propose a purely-microarchitectural technique called* Constable*, that safely eliminates the execution of load instructions. Constable dynamically identifies load instructions that have re-*

stall for multiple cycles, which can limit ILP. On the other hand, a load instruction consumes multiple hard-to-scale hardware resources (e.g., reservation station entry, port to address generation unit, L1-data cache read port) which often causes resource dependence in the pipeline, also limiting ILP.

Prior works propose many latency tolerance techniques to improve ILP by mitigating load data dependence. Load Value Prediction (LVP) [32,42,43,71,98,107,114,139–143,151,153–155, 159,160] and Memory Renaming (MRN) [120,121,147,177,178] are two such widely-studied techniques that mitigate load data dependence via data value speculation. LVP and MRN speculatively execute load-data-dependent instructions using the predicted value of the load instruction, thus improving ILP.

https://arxiv.org/pdf/2406.18786

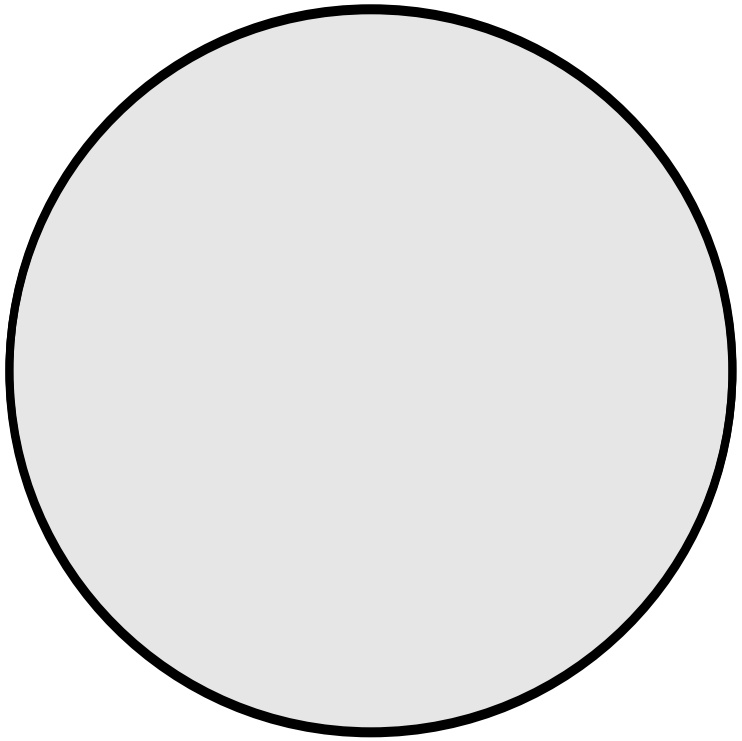
A significant fraction of loads are **global-stable**

But do they limit ILP even when using load value prediction and memory renaming?

Key Finding II: Global-Stable Loads Cause Resource Dependence

All execution cycles where

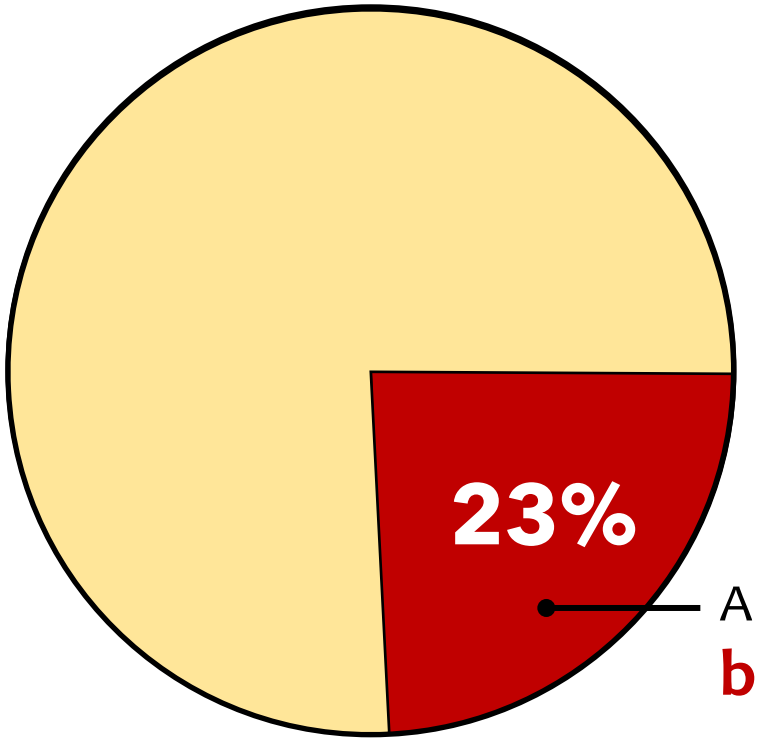
at least one load port is utilized



*In an aggressive OoO processor with 6-wide issue, 3 load ports, a **load value predictor** (EVES [Seznec, CVP'18]), and **memory renaming enabled***

Key Finding II: Global-Stable Loads Cause Resource Dependence

All execution cycles where
at least one load port is utilized



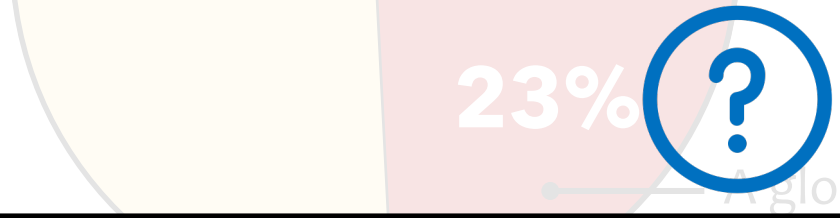
A global-stable load utilizes a load port
blocking a non-global-stable load

*In an aggressive OoO processor with 6-wide issue, 3 load ports, a **load value predictor** (EVES [Seznec, CVP'18]), and **memory renaming enabled***

Key Finding II: Global-Stable Loads Cause Resource Dependence

All execution cycles where
at least one load port is utilized

Even when using load value prediction and memory renaming, global-stable loads limit ILP due to resource dependence



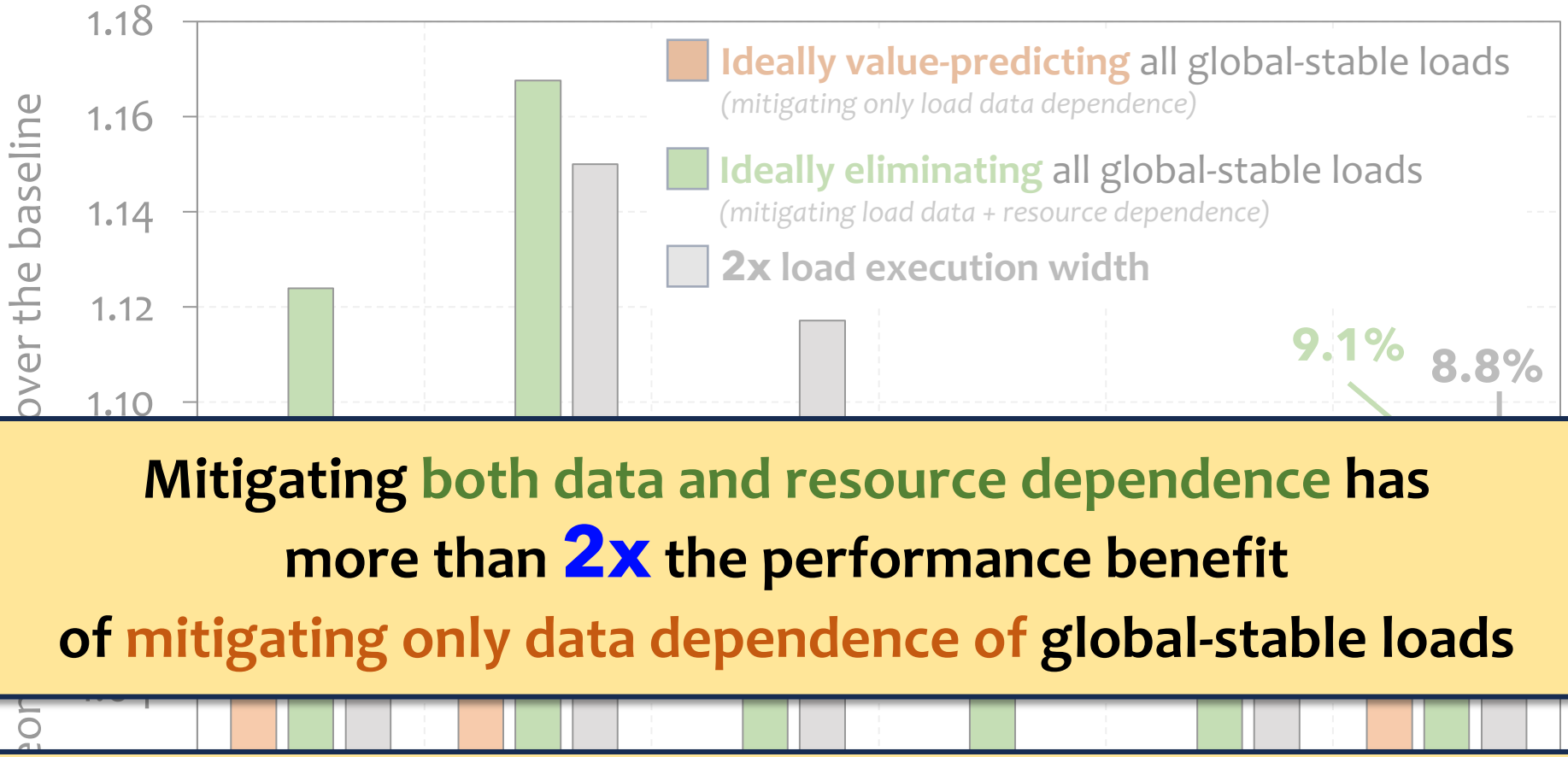
A global-stable load utilizes a load port

load

What's the performance headroom of mitigating the resource dependence?

a load value predictor (EVES [Seznec, CVP'18]), and memory renaming enabled

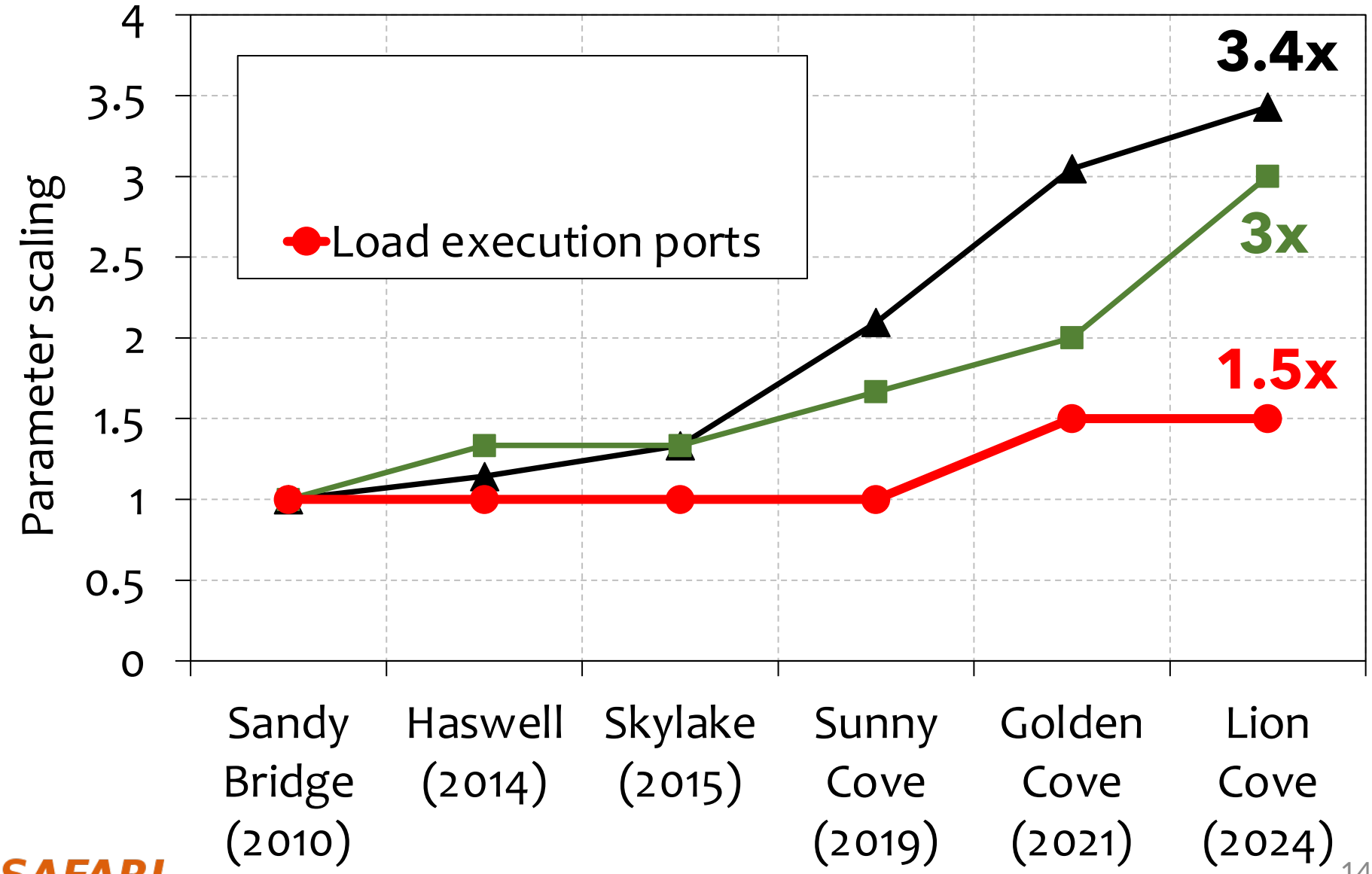
Key Finding III: High Performance Headroom



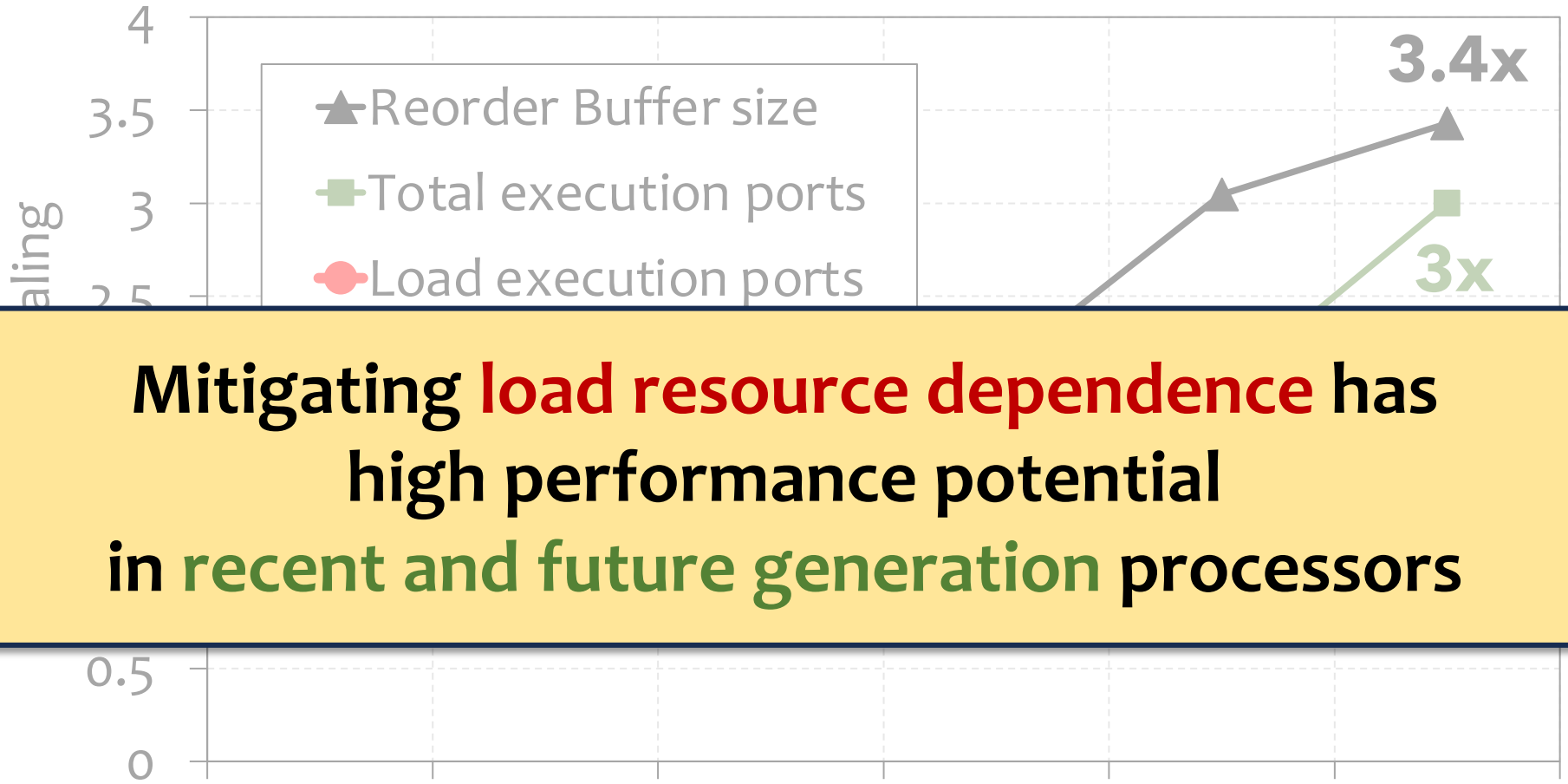
Mitigating both data and resource dependence has more than 2x the performance benefit of mitigating only data dependence of global-stable loads

Ideal elimination of global-stable loads exceeds performance of a processor with 2x wider load execution

Load Execution Resources Lag Behind



Load Execution Resources Lag Behind



Mitigating load resource dependence has high performance potential in recent and future generation processors

Our Goal

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

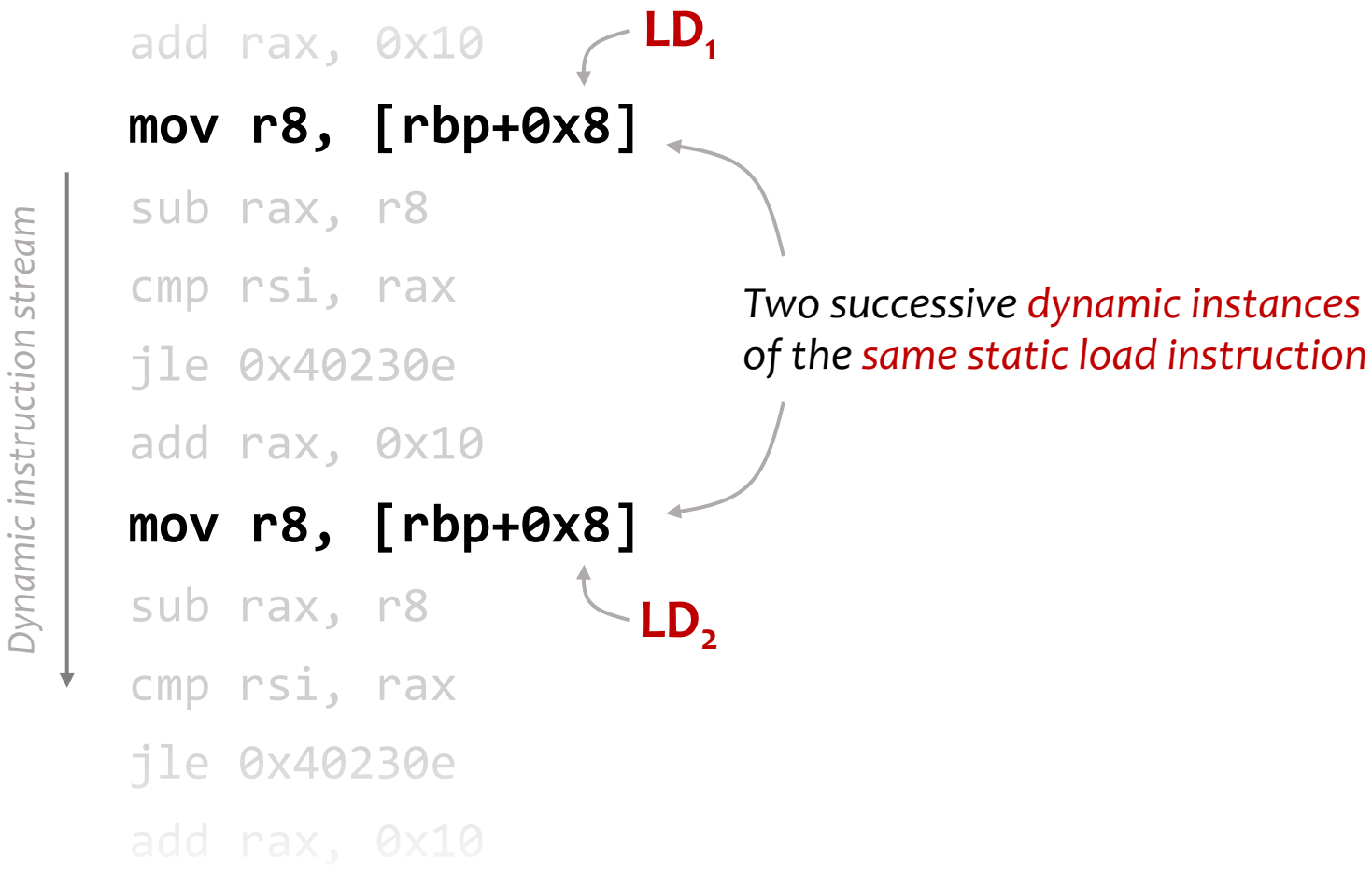
CONSTABLE

A purely-microarchitectural technique

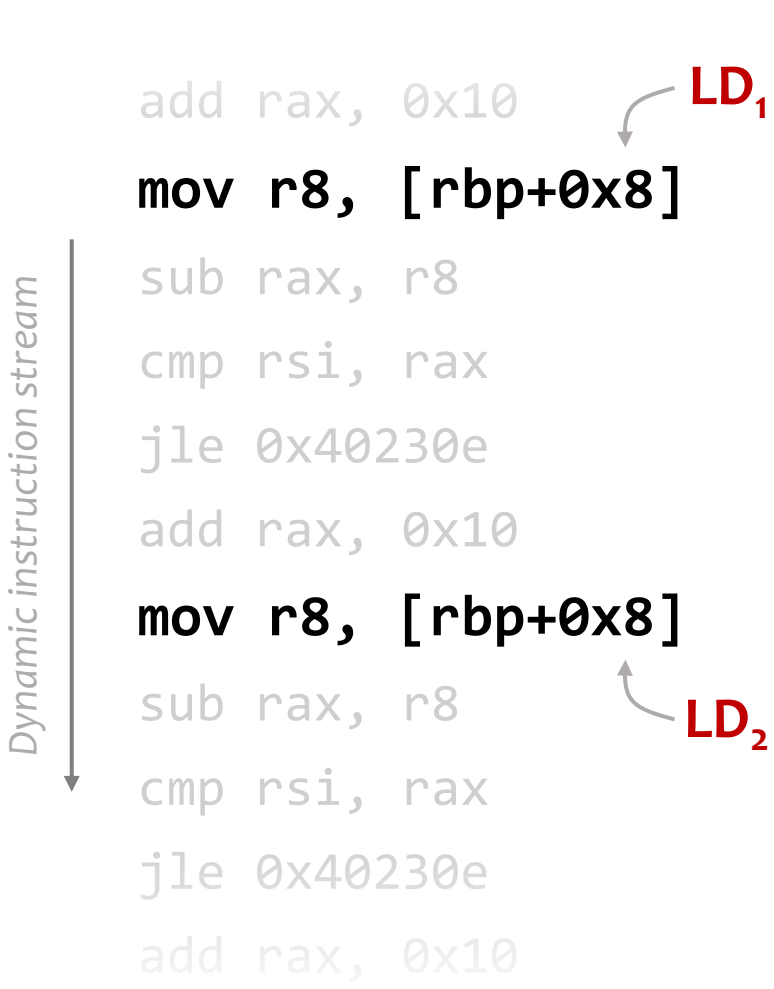
Mitigates both **load data dependence** and **load resource dependence**

By **safely** eliminating the **entire execution** of a load instruction

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

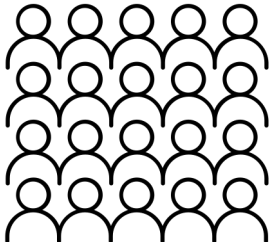
Dynamically identify load instructions
that have historically fetched
the same data from the same load address
(i.e., *likely-stable*)

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

Prior Related Literature

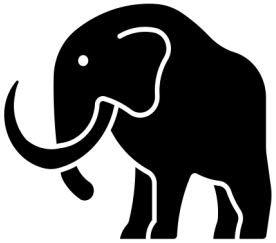
Rich literature on skipping redundant computations
by **memoizing previously-computed results**

[Michie, Nature'68; Harbison+, ASPLOS'82; Richardson, SCA'93; Sodani+, ISCA'97; González+, ICPP'99; ...]



Aim to memoize every instruction

including multiple dynamic instances of each instruction



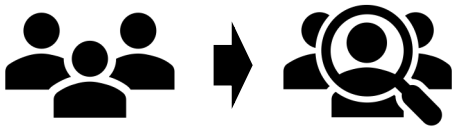
Require large memoization buffer

Often bigger than the size of L1 data cache

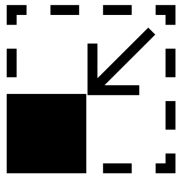
Key Improvements over Literature

Rich literature on skipping redundant computations
by **memoizing previously-computed results**

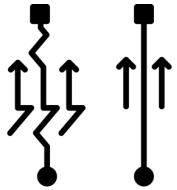
[Michie, Nature'68; Harbison+, ASPLOS'82; Richardson, SCA'93; Sodani+, ISCA'97; González+, ICPP'99; ...]



1 **Focus** only on loads that are likely stable



Lower storage overhead
with high load elimination coverage



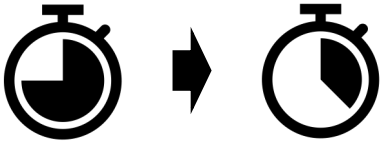
Lower design complexity
Fewer port requirements, lower power

Key Improvements over Literature

Rich literature on skipping redundant computations
by **memoizing previously-computed results**

[Michie, Nature'68; Harbison+, ASPLOS'82; Richardson, SCA'93; Sodani+, ISCA'97; González+, ICPP'99; ...]

1 **Focus** only on loads that are likely stable



2 Eliminate loads **early** in the pipeline

Elimination at rename stage
by explicitly monitoring changes to the source registers
and load address of a likely-stable load

Key Improvements over Literature

Rich literature on skipping redundant computations
by **memoizing previously-computed results**

[Michie, Nature'68; Harbison+, ASPLOS'82; Richardson, SCA'93; Sodani+, ISCA'97; González+, ICPP'99; ...]

1. **Focus** only on loads that are likely stable
2. Eliminate loads **early** in the pipeline
3. Ensure **correctness** in today's processors

- Maintain correctness in presence of **out-of-order load issue**
- Maintain coherence in **multi-threaded & multi-core execution**

Design Overview

Constable: Key Steps

Identify

likely-stable loads

Eliminate

by tracking modifications

Identify a Likely-Stable Load

```
add rax, 0x10
mov r8, [rbp+0x8]      5
sub rax, r8
cmp rsi, rax
add rax, 0x10
mov r8, [rbp+0x8]      6
sub rax, r8
cmp rsi, rax
ret
mov r8, [rbp+0x8]      3
sub rax, r8
cmp rsi, rax
jle 0x40230e
```

- Using a **stability confidence** counter per load instruction

Same data & address as last dynamic instance

+1

Stability Confidence

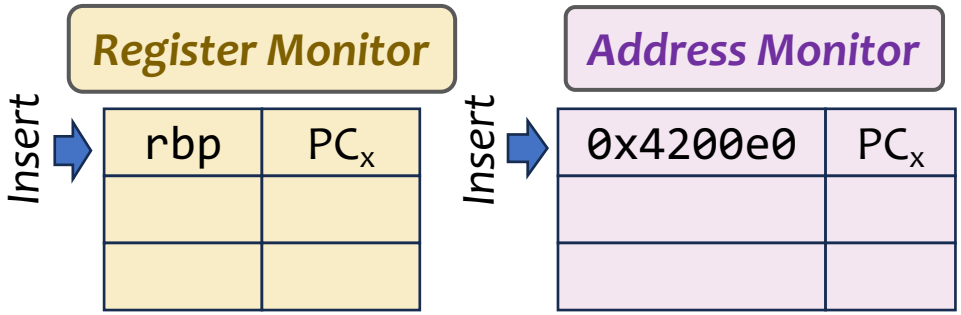
/2

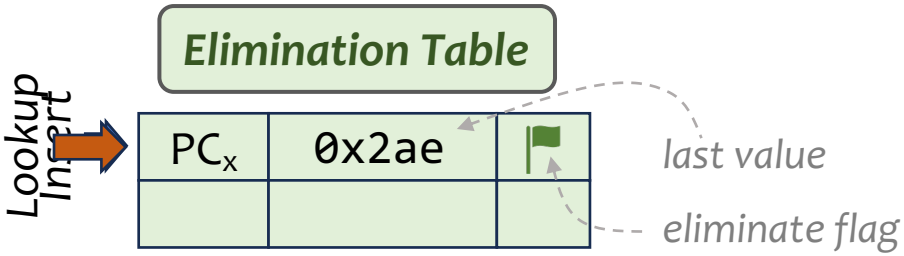
Different data or different address

Eliminate a Likely-Stable Load

```
jle 0x40230e
add rax, 0x10
mov r8, [rbp+0x8]      30
sub rax, r8
cmp rsi, rax
add rax, 0x10
mov r8, [rbp+0x8]      30
sub rax, r8
cmp rsi, rax
add rax, 0x10
mov r8, [rbp+0x8]      30
sub rax, r8
cmp rsi, rax
```

Stability confidence crosses threshold

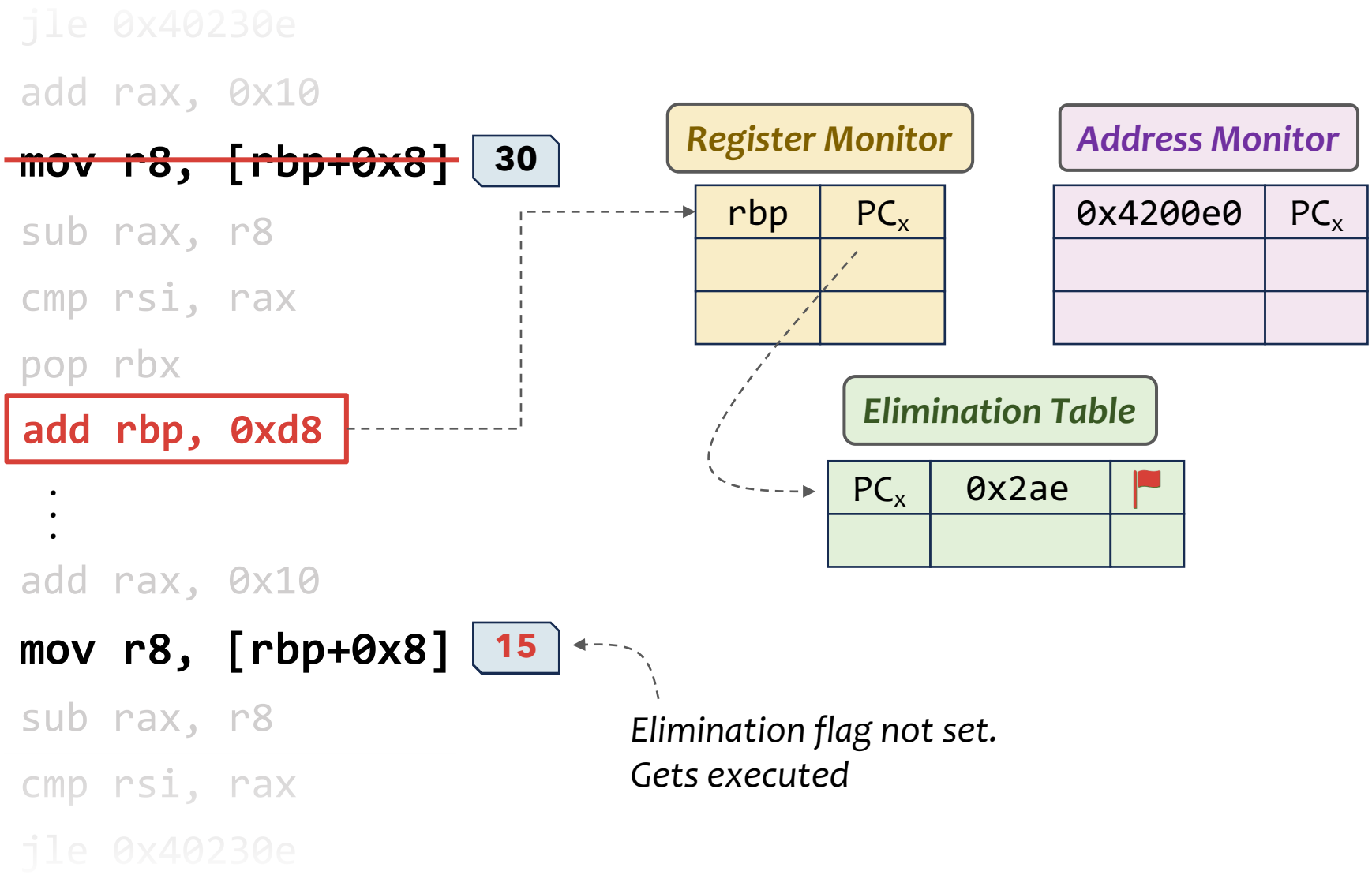




- *No reservation station*
- *No address generation unit*
- *No load port*
- *Still takes ROB and load buffer*

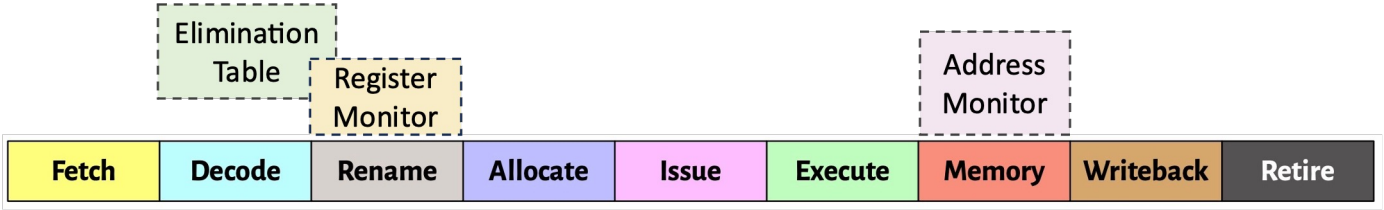
to handle correct elimination of in-flight loads

Stop Elimination of a Likely-Stable Load



More in the Paper

- **Ensuring safe and correct elimination** in presence of
  - Out-of-order load issue
  - Multi-threaded & multi-core execution
  - Wrong-path execution
- **Integration of Constable** into the processor pipeline



- **Microarchitecture for breaking data** dependence on the eliminated loads
- **Microarchitecture of Constable's own structures**
  - Read and write port requirements

More in the Paper

- Ensuring safe and correct elimination in presence of
  - Out-of-order load issue
- In
- M
- M
  - Rea

Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

*Rahul Bera[1] *Adithya Ranganathan[2] Joydeep Rakshit[2] Sujit Mahto[2]
Anant V. Nori[2] Jayesh Gaur[2] Ataberk Olgun[1] Konstantinos Kanellopoulos[1]
Mohammad Sadrosadati[1] Sreenivas Subramoney[2] Onur Mutlu[1]

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

Load instructions often limit instruction-level parallelism (ILP) in modern processors due to data and resource dependences they cause. Prior techniques like Load Value Prediction (LVP) and Memory Renaming (MRN) mitigate load data dependence by predicting the data value of a load instruction. However, they fail to mitigate load resource dependence as the predicted load instruction gets executed nonetheless (even on a correct prediction), which consumes hard-to-scale pipeline resources that otherwise could have been used to execute other load instructions.

Our goal in this work is to improve ILP by mitigating both *load data dependence and resource dependence. To this end, we propose a purely-microarchitectural technique called* Constable, *that safely eliminates the execution of load instructions. Constable dynamically identifies load instructions that have re-*

stall for multiple cycles, which can limit ILP. On the other hand, a load instruction consumes multiple hard-to-scale hardware resources (e.g., reservation station entry, port to address generation unit, L1-data cache read port) which often causes resource dependence in the pipeline, also limiting ILP.

Prior works propose many latency tolerance techniques to improve ILP by mitigating load data dependence. Load Value Prediction (LVP) [32,42,43,71,98,107,114,139–143,151,153–155,159,160] and Memory Renaming (MRN) [120,121,147,177,178] are two such widely-studied techniques that mitigate load data dependence via data value speculation. LVP and MRN speculatively execute load-data-dependent instructions using the predicted value of the load instruction, thus improving ILP.

https://arxiv.org/pdf/2406.18786

Evaluation

Methodology

- Industry-grade x86-64 simulator modeling aggressive OoO processor
  - 8-wide fetch, 6-wide issue to 3 load ports, 512-entry ROB
  - With memory renaming, zero/constant/move elimination, branch folding
  - Five prefetchers throughout cache hierarchy

- **90 workloads** of wide variety
  - All from **SPEC CPU** 2017
  - **Client** (SYSMark, DaCapo, ...)
  - **Enterprise** (SPECjbb, SPECjEnterprise, ...)
  - **Server** (BigBench, Hadoop, ...)

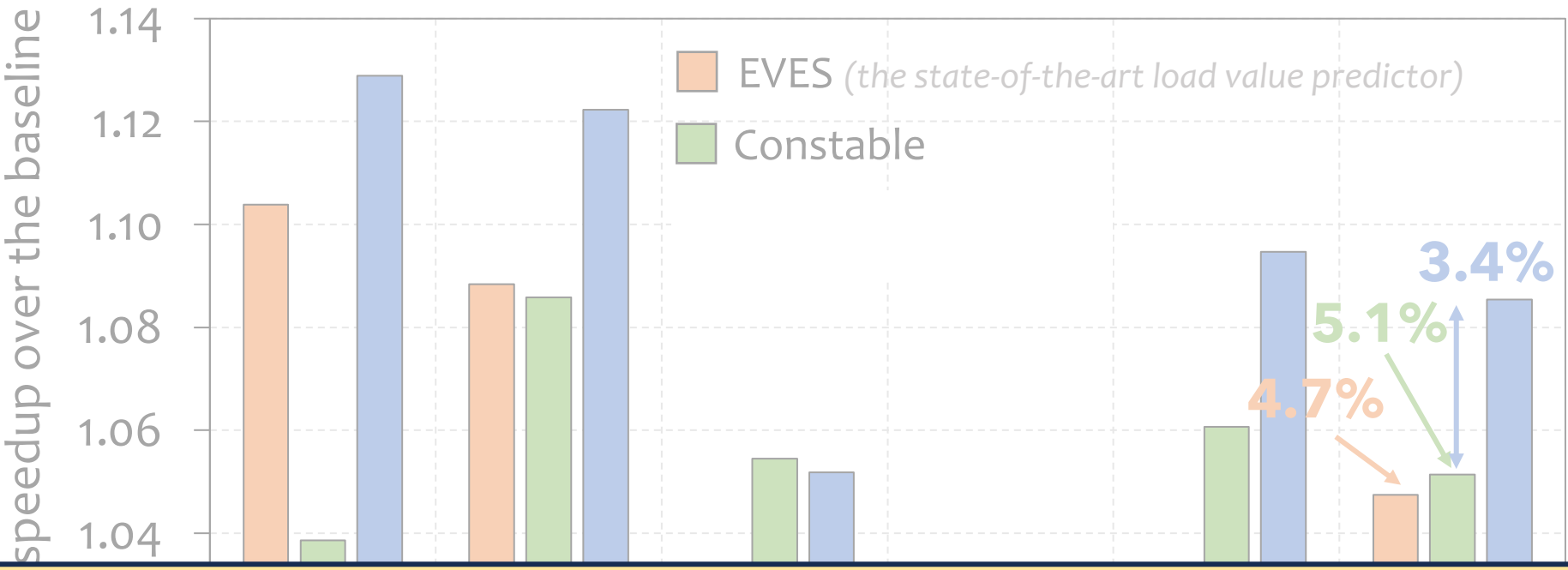
Mechanisms compared against

- EVES, the state-of-the-art load value predictor [Seznec, CVP'18]
- Early Load Address Resolution [Bekerman+, ISCA'00]
- Register File Prefetching [Shukla+, ISCA'22]

Configurations

- No simultaneous multi-threading (SMT)
- 2-way SMT

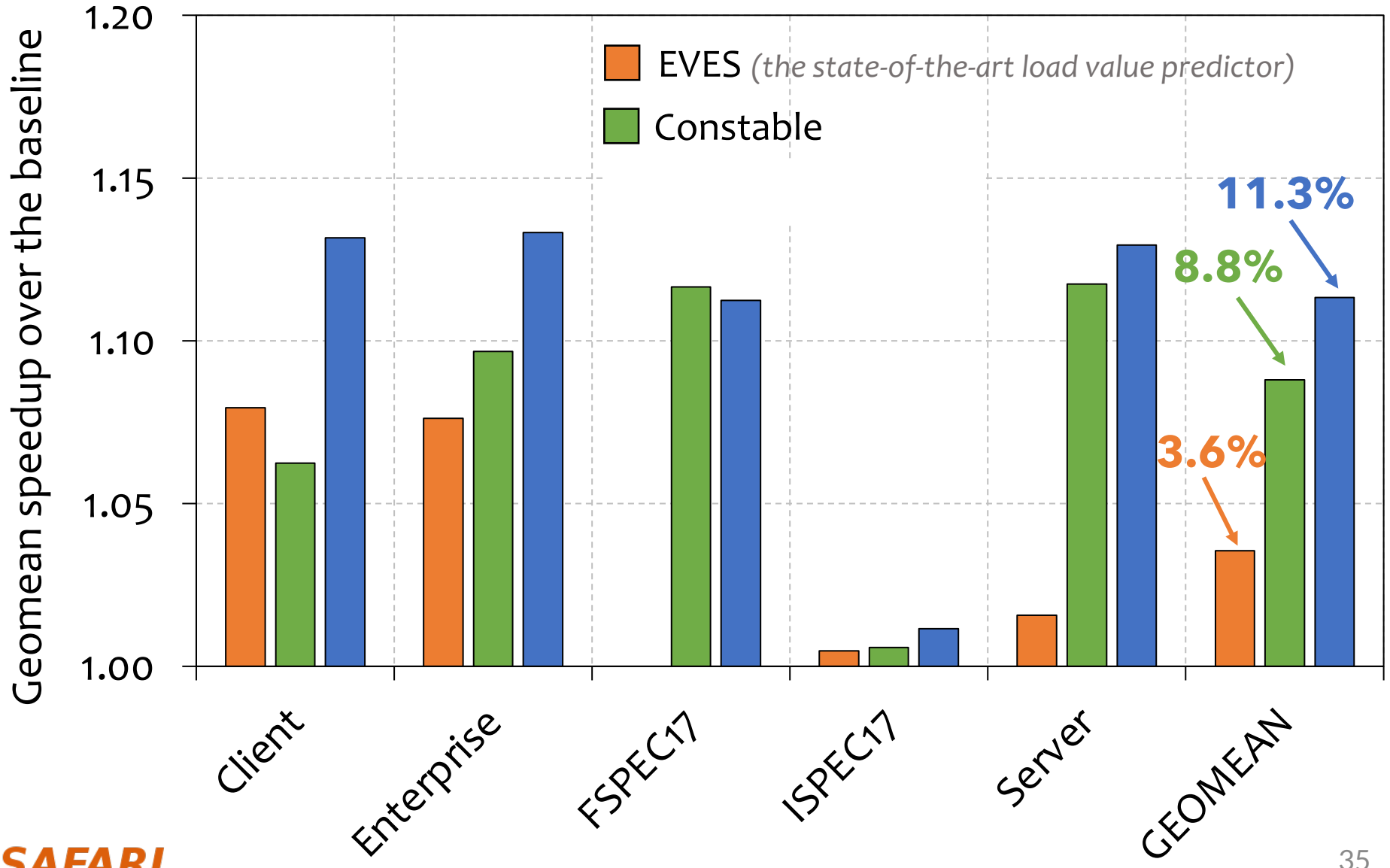
Performance Improvement in noSMT



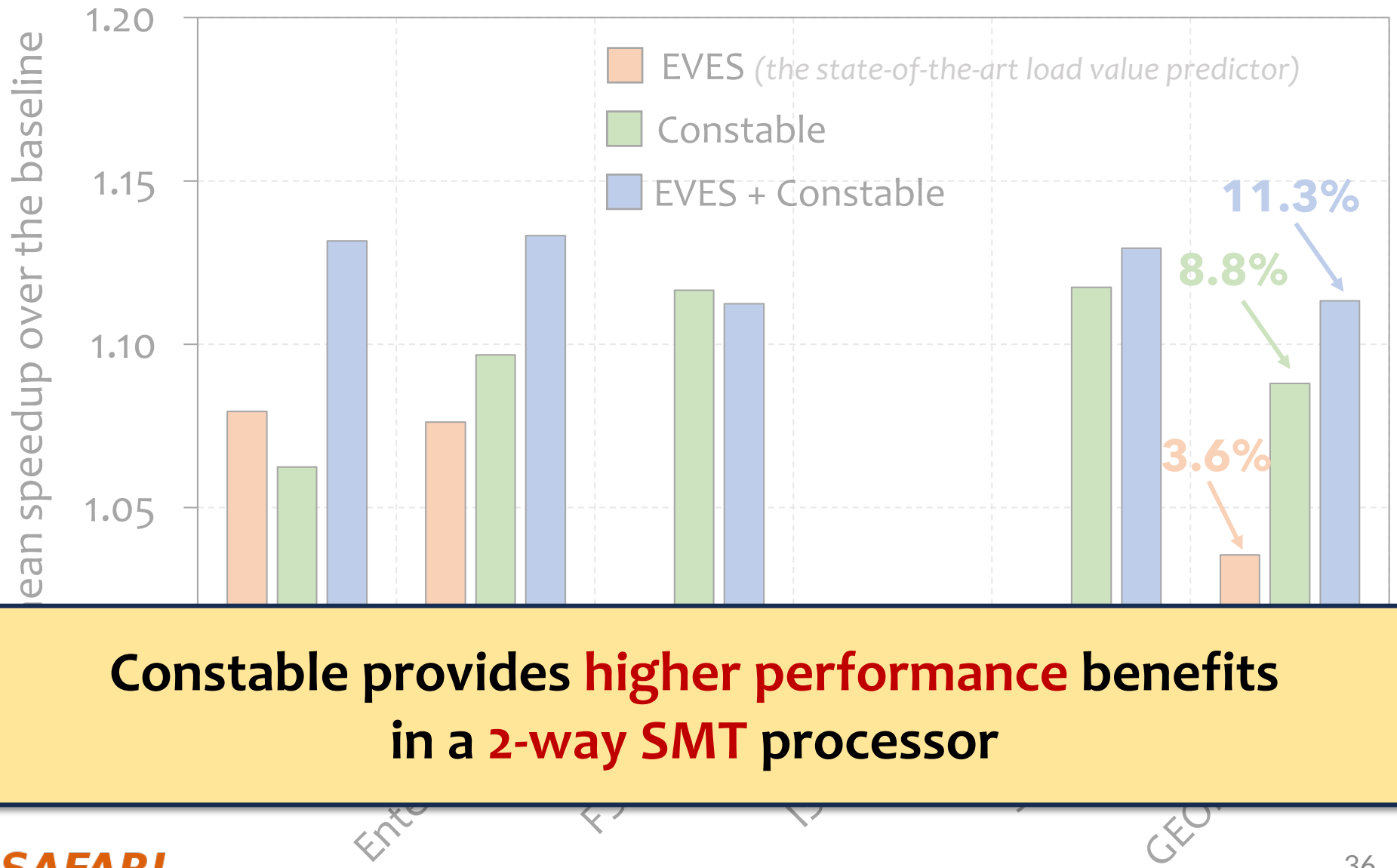
Constable alone provides similar performance as EVES with only ½ of EVES' storage overhead

Constable on top of EVES outperforms EVES alone

Performance Improvement in 2-way SMT

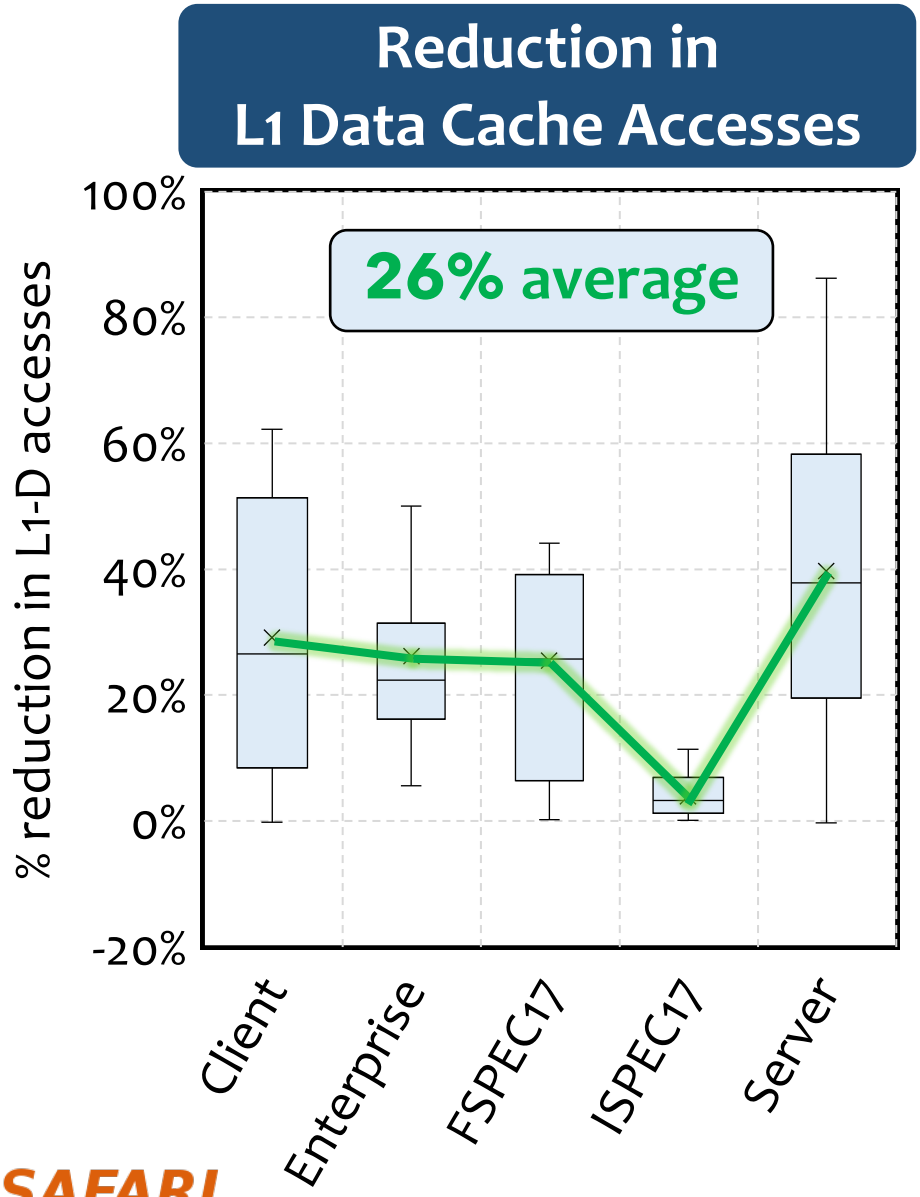


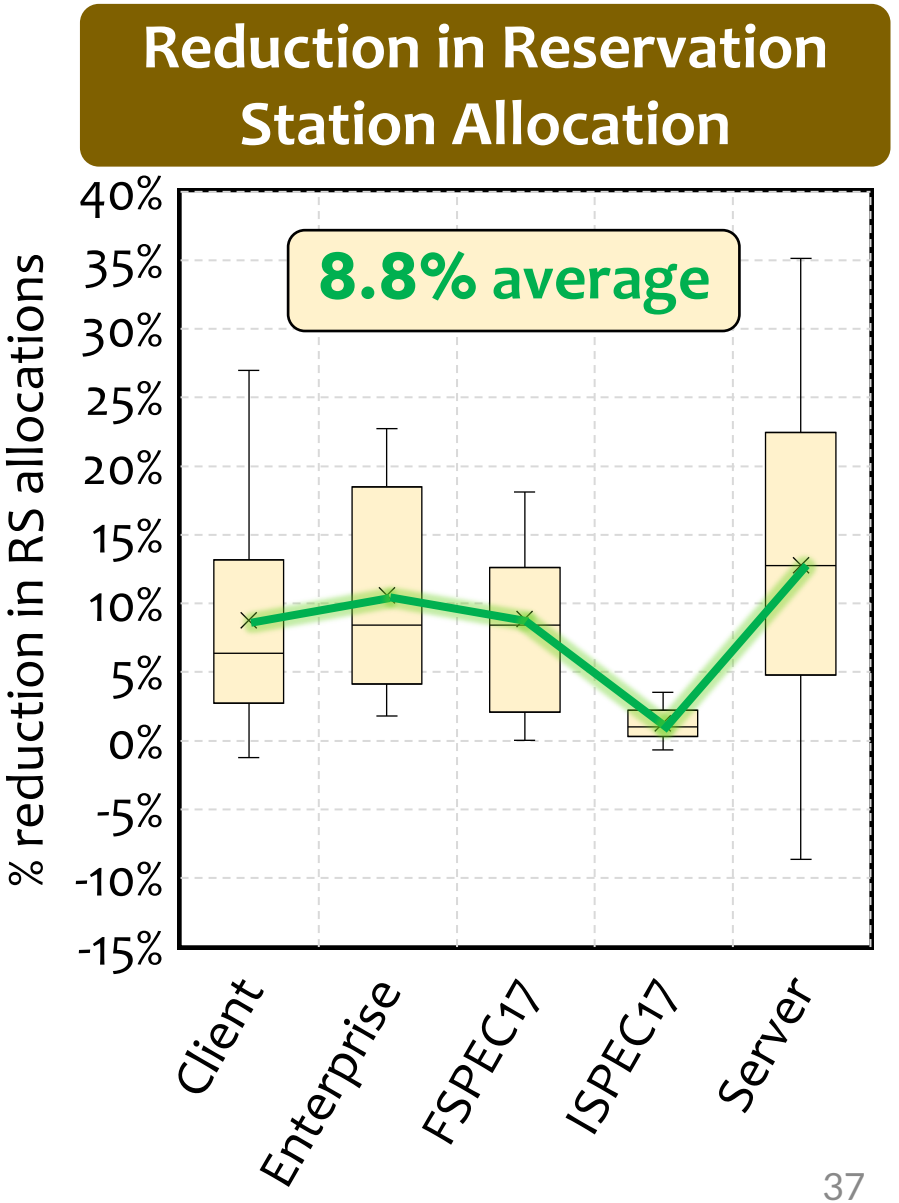
Performance Improvement in 2-way SMT



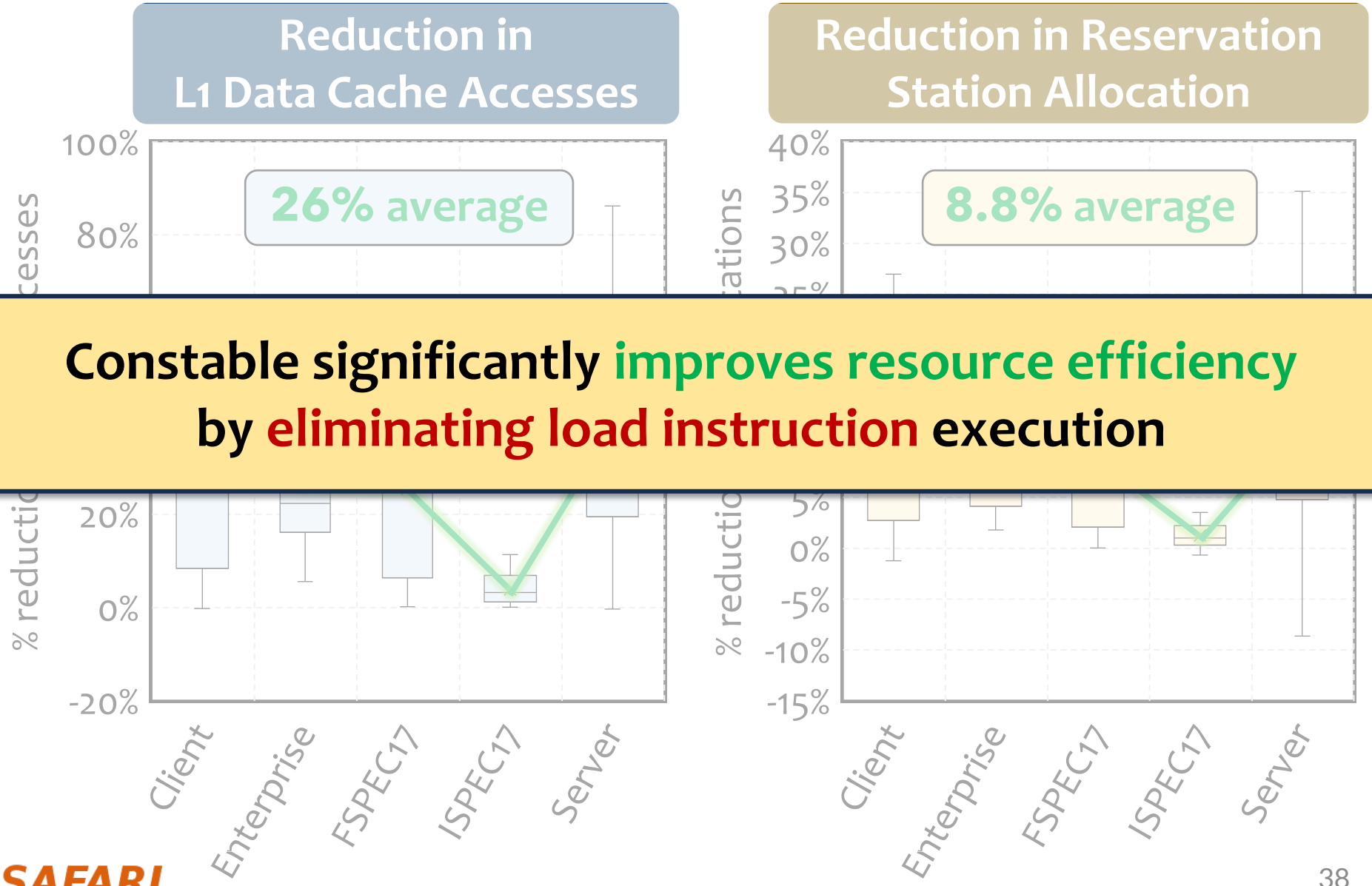
Constable provides higher performance benefits in a 2-way SMT processor

Improvement in Resource Efficiency





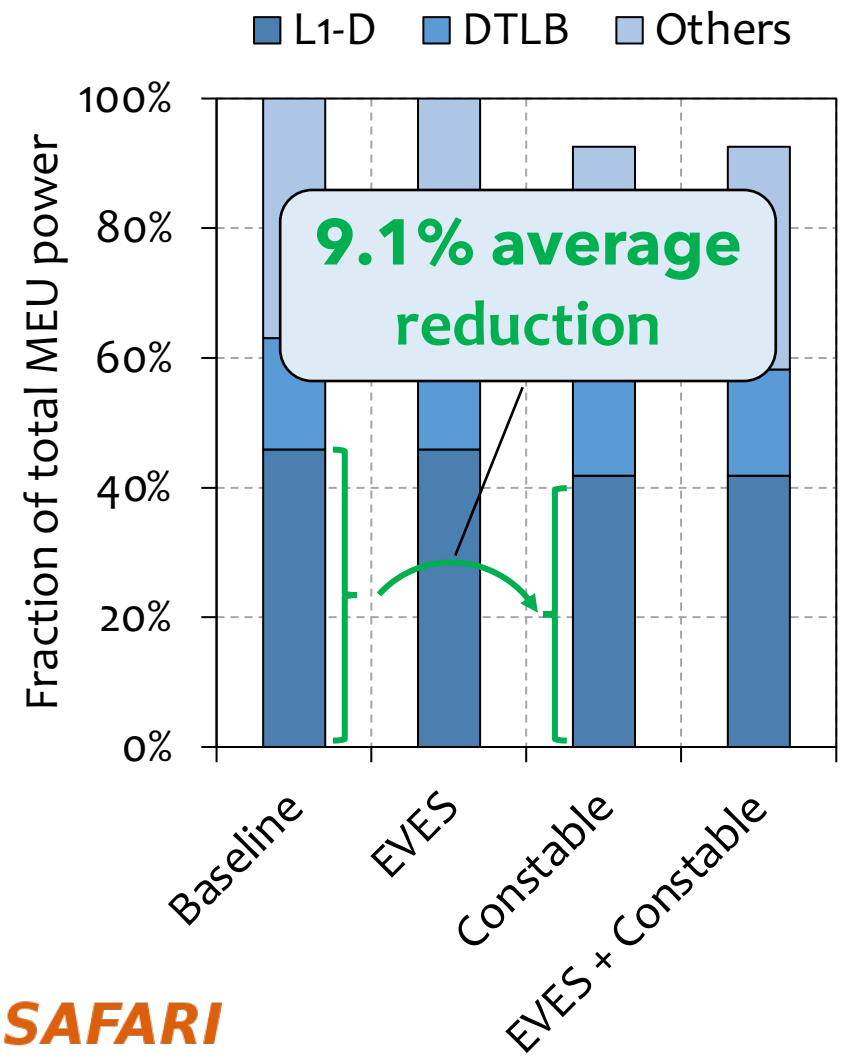
Improvement in Resource Efficiency



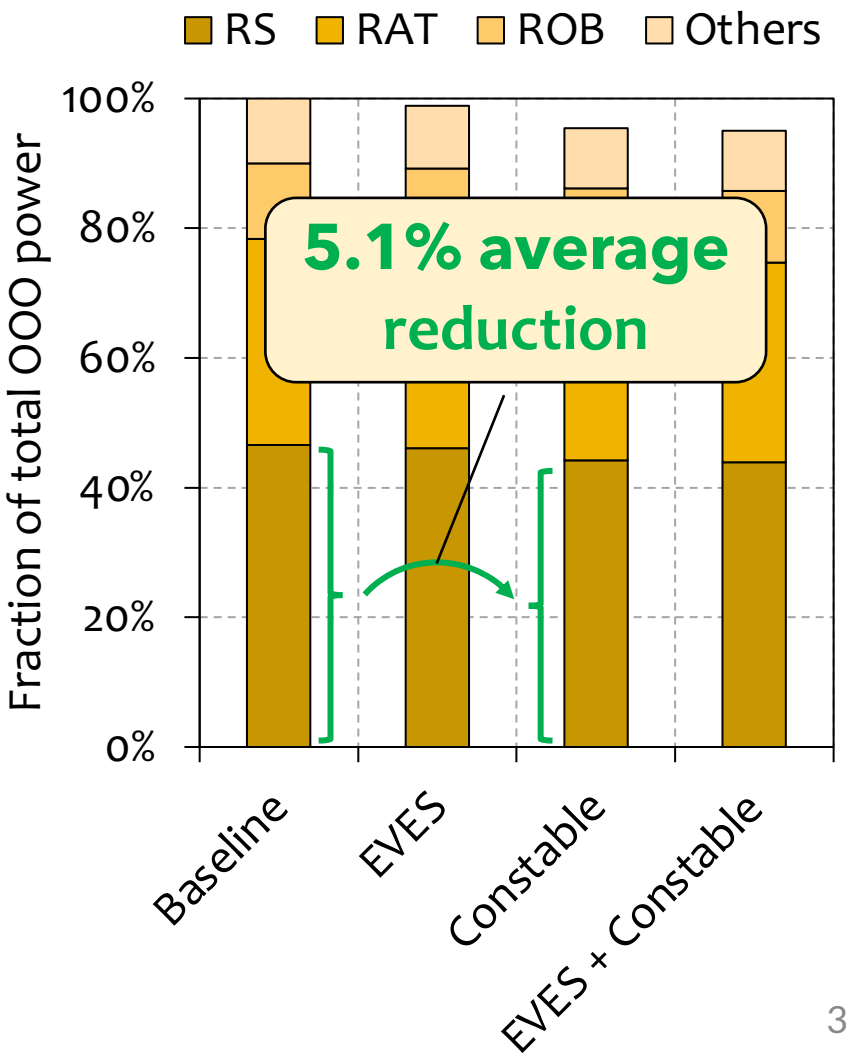
Constable significantly improves resource efficiency by eliminating load instruction execution

Reduction in Dynamic Power

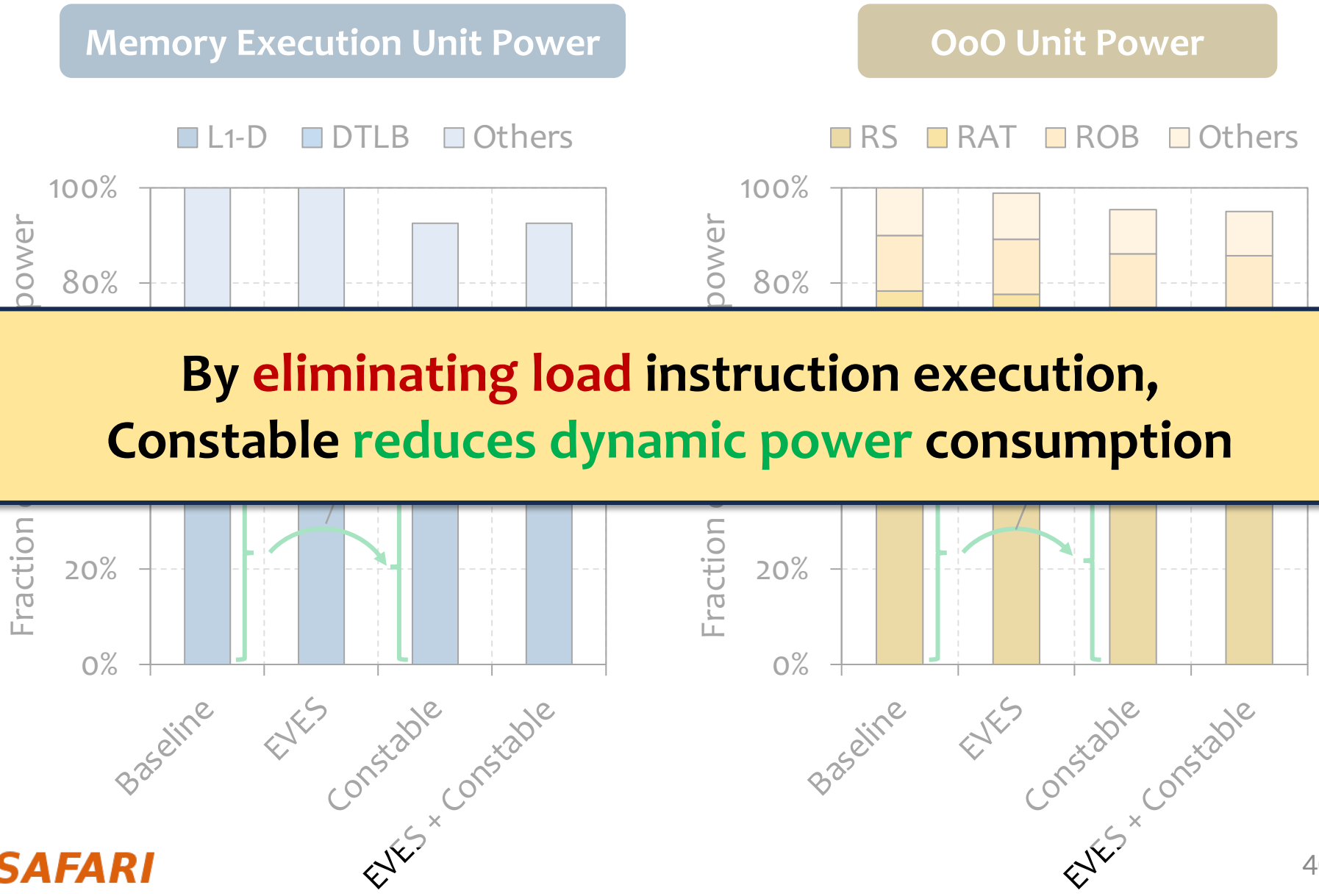
Memory Execution Unit Power



OoO Unit Power

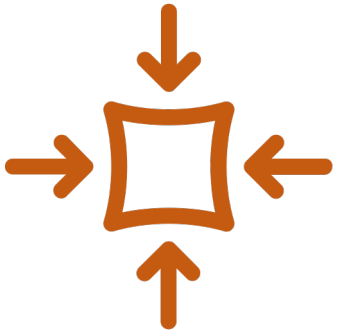


Reduction in Dynamic Power



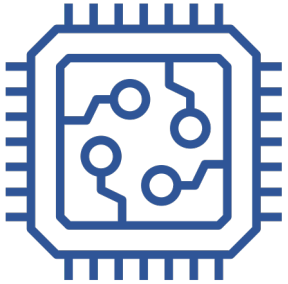
By eliminating load instruction execution, Constable reduces dynamic power consumption

Area and Power Overhead of Constable's Own Structures



12.4 KB

Storage overhead per core



0.232 mm²

0.0061% area of Intel Alderlake-S processor

Low Energy

Up to 10.8 pJ/read and 16.7 pJ/write

More in the Paper

- **Load elimination coverage** of Constable
  - **23.5%** of all dynamic loads are eliminated

- **Per-workload performance** analysis
  - Up to **31.2%** over baseline
  - 60/90 workloads outperforms EVES by more than 5%

- **Performance contribution per load category**
  - Stack loads contribute the highest

- Performance improvement over **prior works**
  - **4.7%** over **Early load address resolution**
  - **3.6%** over **Register file prefetching**

- Performance **sensitivity**:
  - Higher performance in every configuration up to **2X load execution width**
  - Higher performance in every configuration up to **2X pipeline depth**

More in the Paper

- Load elimination coverage of Constable
  - 23.5% of all dynamic loads are eliminated
- Pe
- Pe
- Pe
- Pe
  - Higher performance in every configuration up to 2X load execution width
  - High

Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

*Rahul Bera[1] *Adithya Ranganathan[2] Joydeep Rakshit[2] Sujit Mahto[2]
Anant V. Nori[2] Jayesh Gaur[2] Ataberk Olgun[1] Konstantinos Kanellopoulos[1]
Mohammad Sadrosadati[1] Sreenivas Subramoney[2] Onur Mutlu[1]

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

Load instructions often limit instruction-level parallelism (ILP) in modern processors due to data and resource dependences they cause. Prior techniques like Load Value Prediction (LVP) and Memory Renaming (MRN) mitigate load data dependence by predicting the data value of a load instruction. However, they fail to mitigate load resource dependence as the predicted load instruction gets executed nonetheless (even on a correct prediction), which consumes hard-to-scale pipeline resources that otherwise could have been used to execute other load instructions.

Our goal in this work is to improve ILP by mitigating both *load data dependence and resource dependence. To this end, we propose a purely-microarchitectural technique called* Constable, *that safely eliminates the execution of load instructions. Constable dynamically identifies load instructions that have re-*

stall for multiple cycles, which can limit ILP. On the other hand, a load instruction consumes multiple hard-to-scale hardware resources (e.g., reservation station entry, port to address generation unit, L1-data cache read port) which often causes resource dependence in the pipeline, also limiting ILP.

Prior works propose many latency tolerance techniques to improve ILP by mitigating load data dependence. Load Value Prediction (LVP) [32,42,43,71,98,107,114,139–143,151,153–155, 159,160] and Memory Renaming (MRN) [120,121,147,177,178] are two such widely-studied techniques that mitigate load data dependence via data value speculation. LVP and MRN speculatively execute load-data-dependent instructions using the predicted value of the load instruction, thus improving ILP.

https://arxiv.org/pdf/2406.18786

To Summarize…

Our Key Findings

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

2. These global-stable loads cause significant ILP loss due to **resource dependence**

3. **Eliminating** global-stable load execution provides more than **2x the performance** benefit of just **breaking** their **load data dependency**

Our Proposal

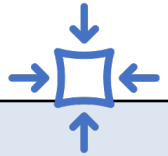
Constable

Identifies and eliminates loads
that repeatedly fetch same data from same address

High performance benefit
over a strong baseline system
without (**5.1%**) and with SMT (**8.8%**)

Improves resource efficiency
L1-D access reduction by **26%**
RS allocation reduction by **8.8%**

Reduces dynamic power
L1-D power by **9.1%**
RS power by **5.1%**



Low storage overhead
Only **12.4 KB**/core,
0.232 mm^2 in 14-nm technology

There's Still Headroom...

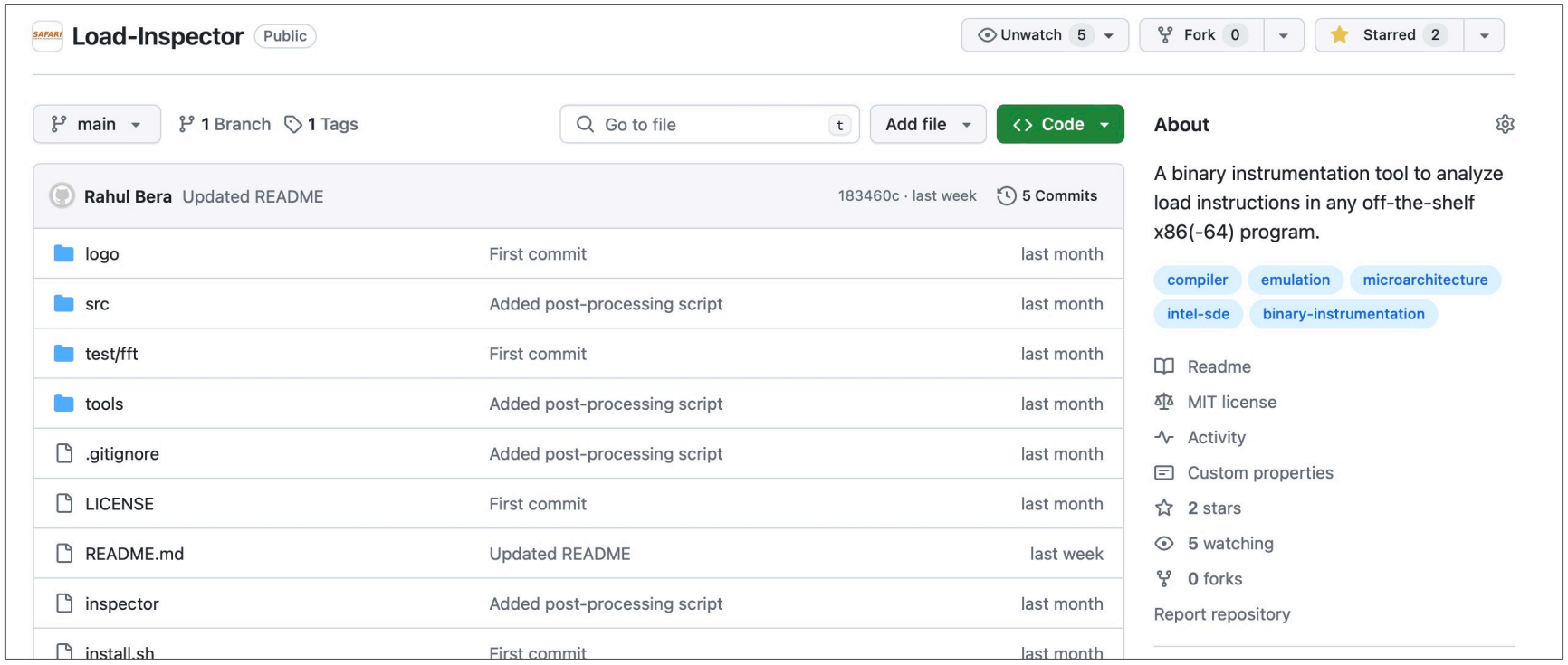
Constable successfully eliminates
57% of all global-stable loads at runtime

43% of global-stable loads
do not get eliminated

We need to understand more software primitives that generate global-stable loads

Open-Source Tool

A tool to analyze load instructions in any off-the-shelf x86(-64) program



https://github.com/CMU-SAFARI/Load-Inspector

Open-Source Tool

Load Inspector

A tool to analyze load instructions in any off-the-shelf x86(-64) program

Load-Inspector Public

Unwatch 5 Fork 0 Starred 2

Study global-stable loads

Study the effects of increasing architectural registers using APX extension to x64 ISA

logo First commit last month

x86(-64) program.

README.md Updated README last week

inspector Added post-processing script last month

install.sh First commit last month

5 watching

0 forks

Report repository

https://github.com/CMU-SAFARI/Load-Inspector

CONSTABLE

Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution

arXiv

Load Inspector

SAFARI
SAFARI Research Group
safari.ethz.ch

ETH zürich

intel

BACKUP

Index

Motivation & Design

Methodology & Evaluation

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;
}
```

Example code from **541.leela_r**

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
```

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. Function is repeatedly called by the same caller with the same arguments*

Why Do Global-Stable Loads Exist?

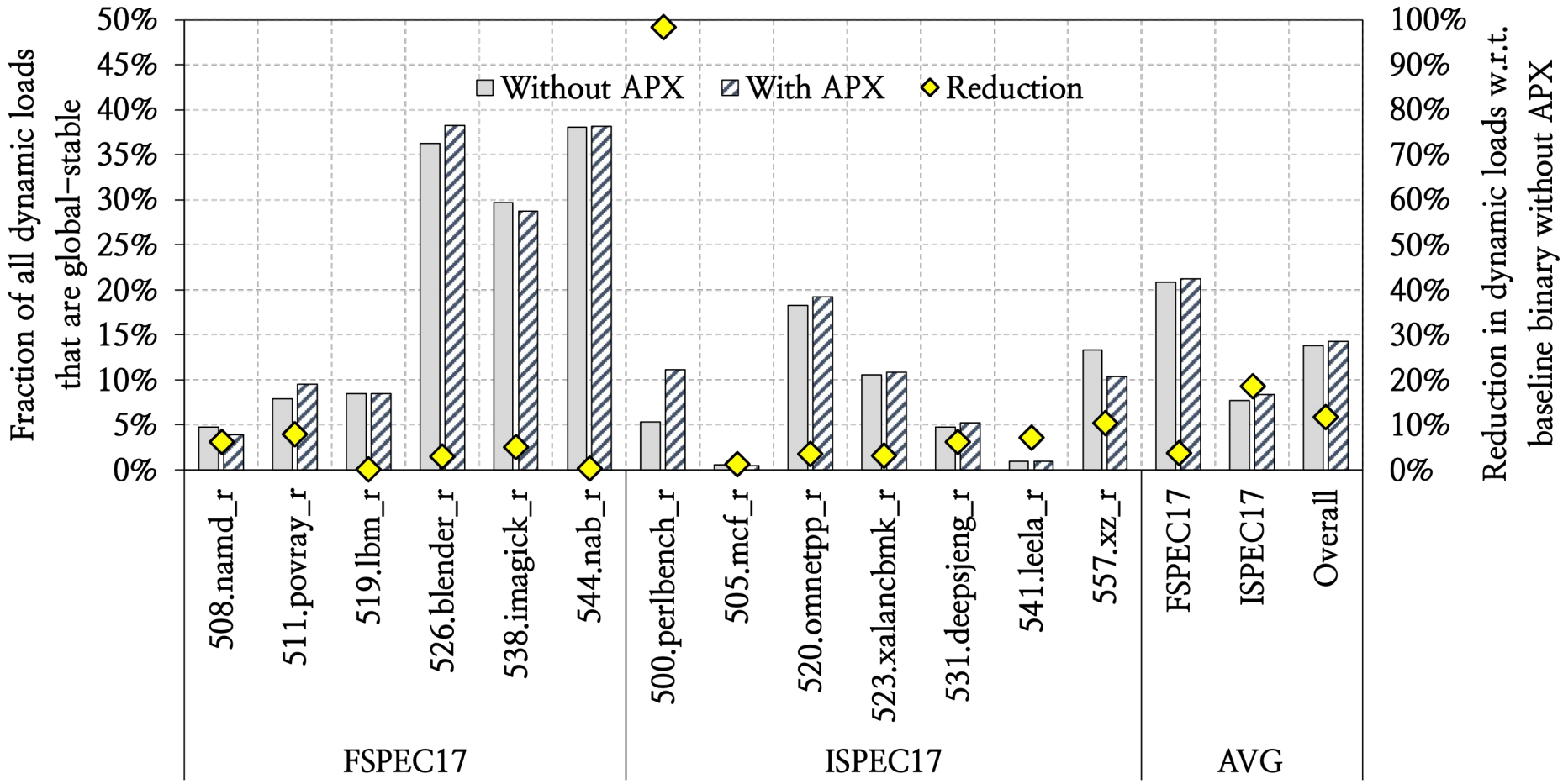
```
1  static inline bool
2  rc_shift_low(lzma_range_encoder *rc, uint8_t *out,
3               size_t *out_pos, size_t out_size)
4  {
5
```

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
```

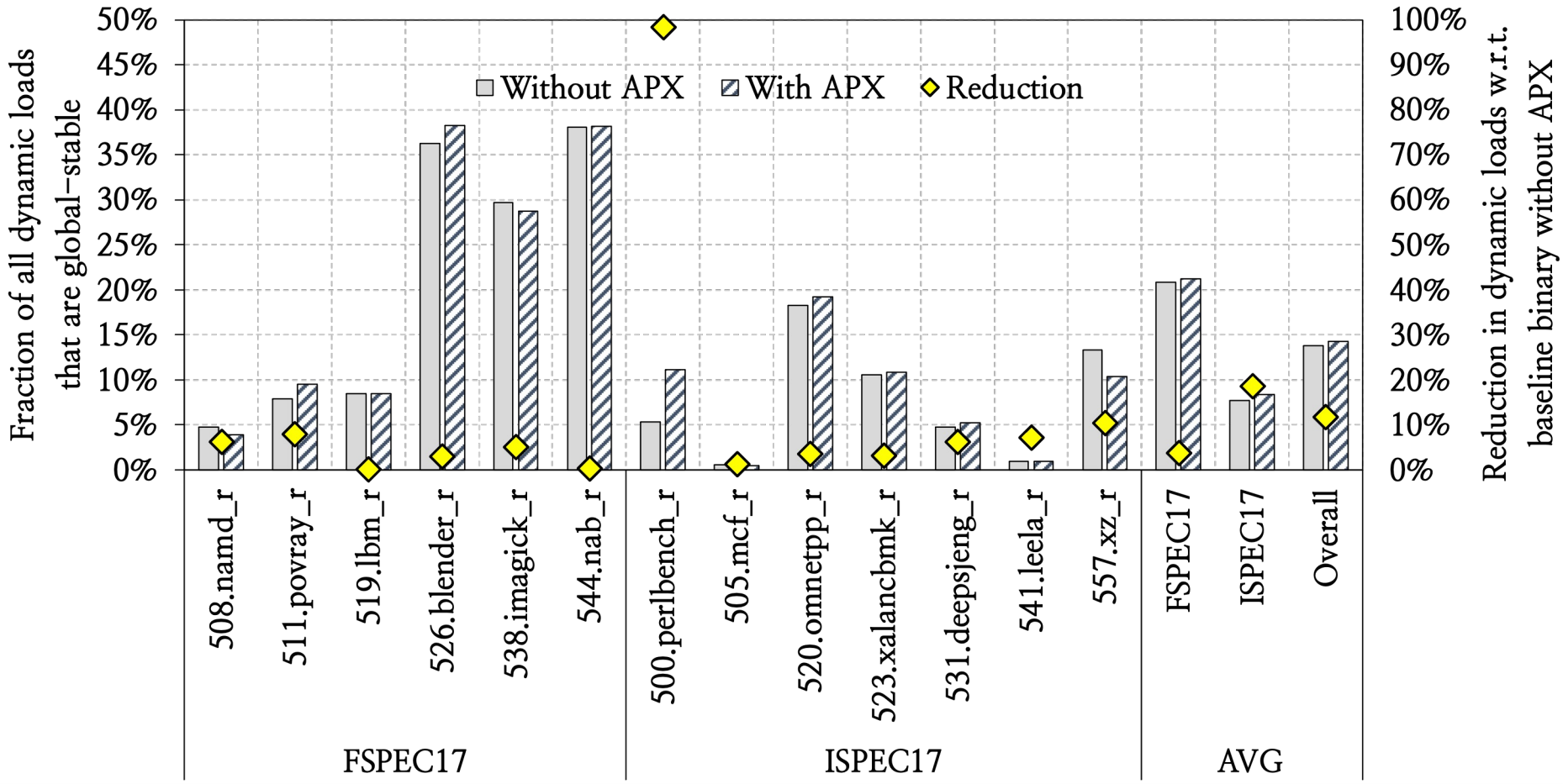
Effects of Increasing Architectural Registers



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

Compiled with Clang 18.1.3 with and without `-mapxf`

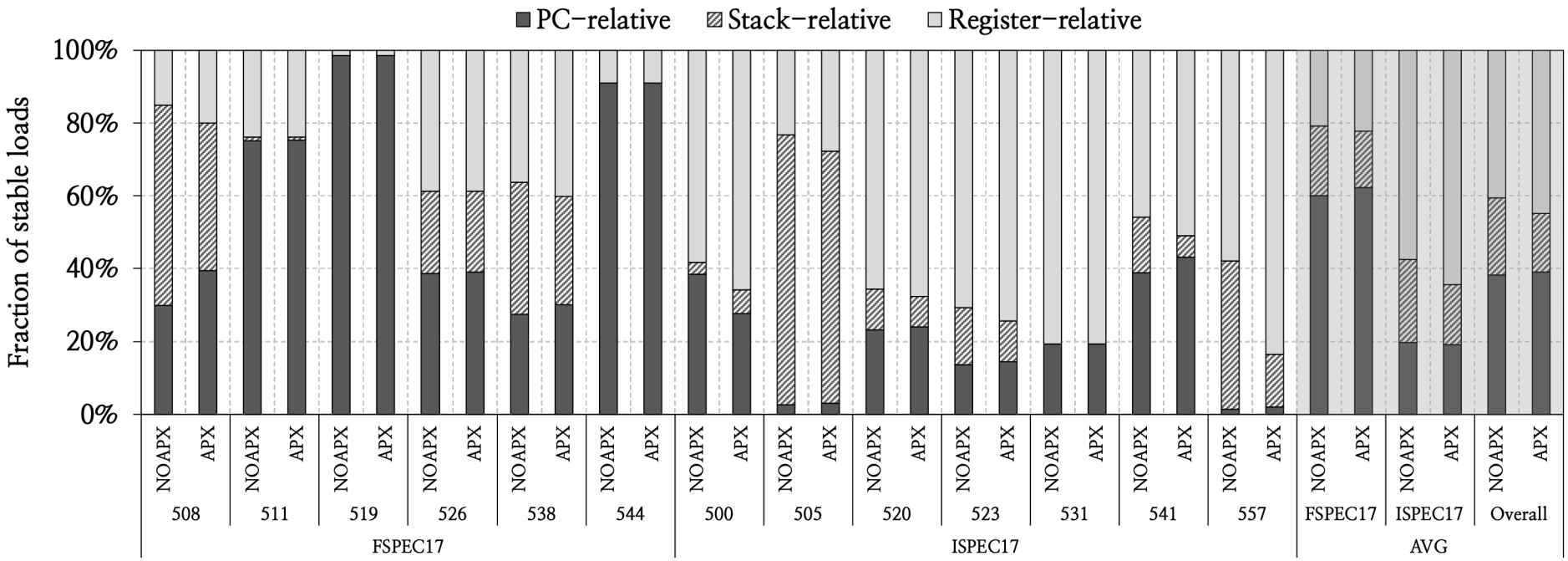
Effects of Increasing Architectural Registers



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

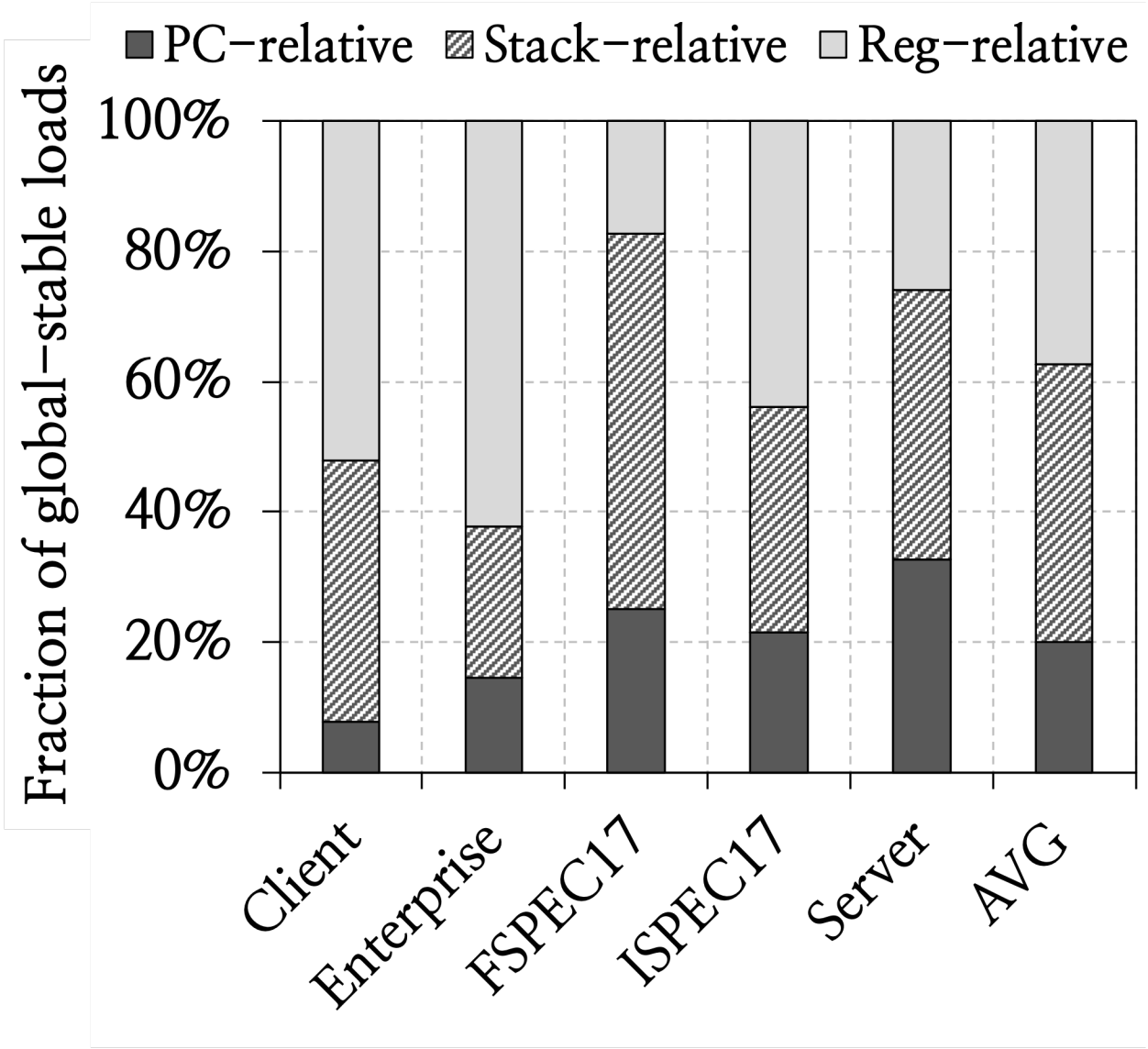
Compiled with Clang 18.1.3 with and without `-mapxf`

Effects of Increasing Architectural Registers

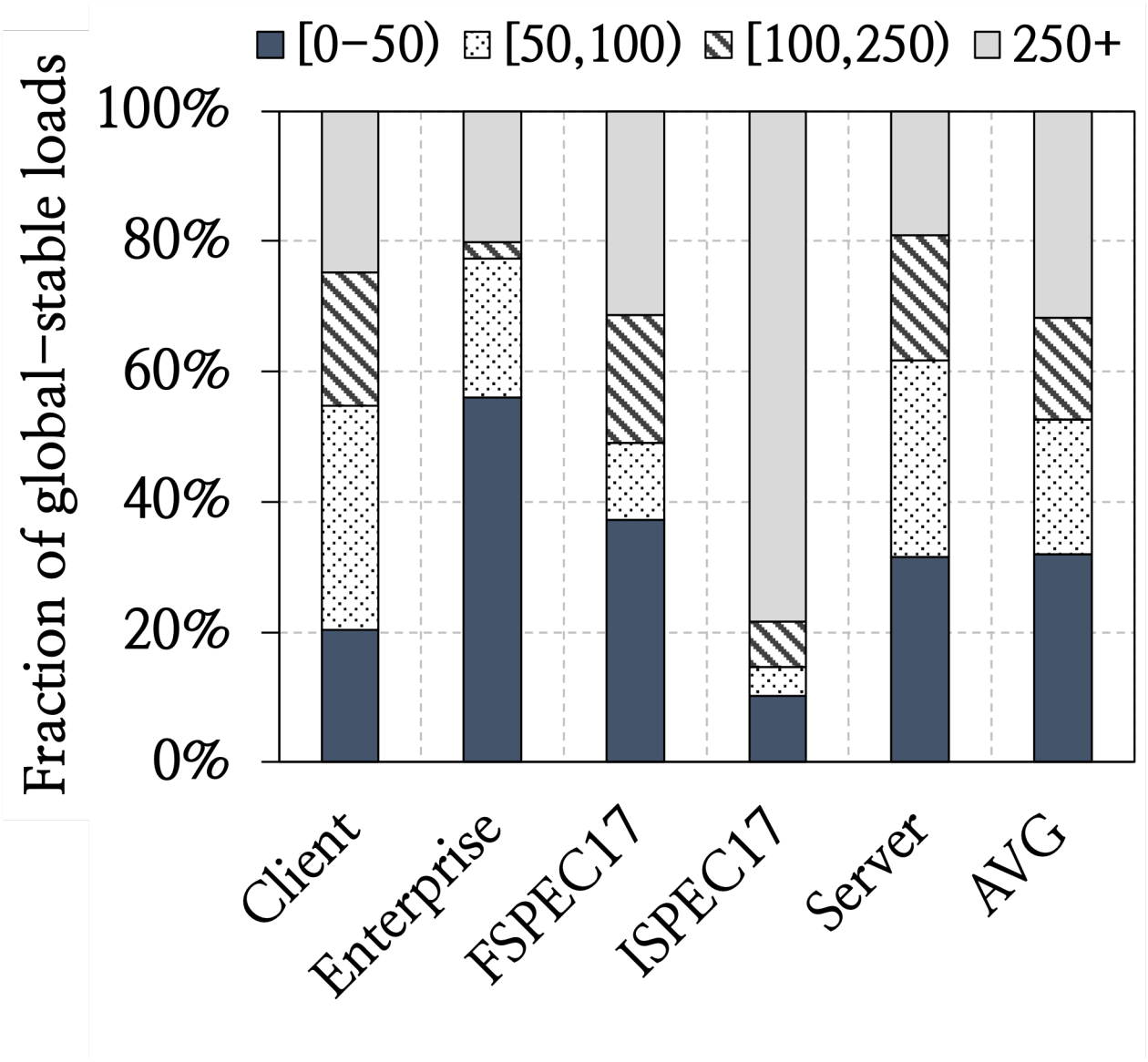


The profile of global-stable loads stays largely unchanged after doubling registers using APX

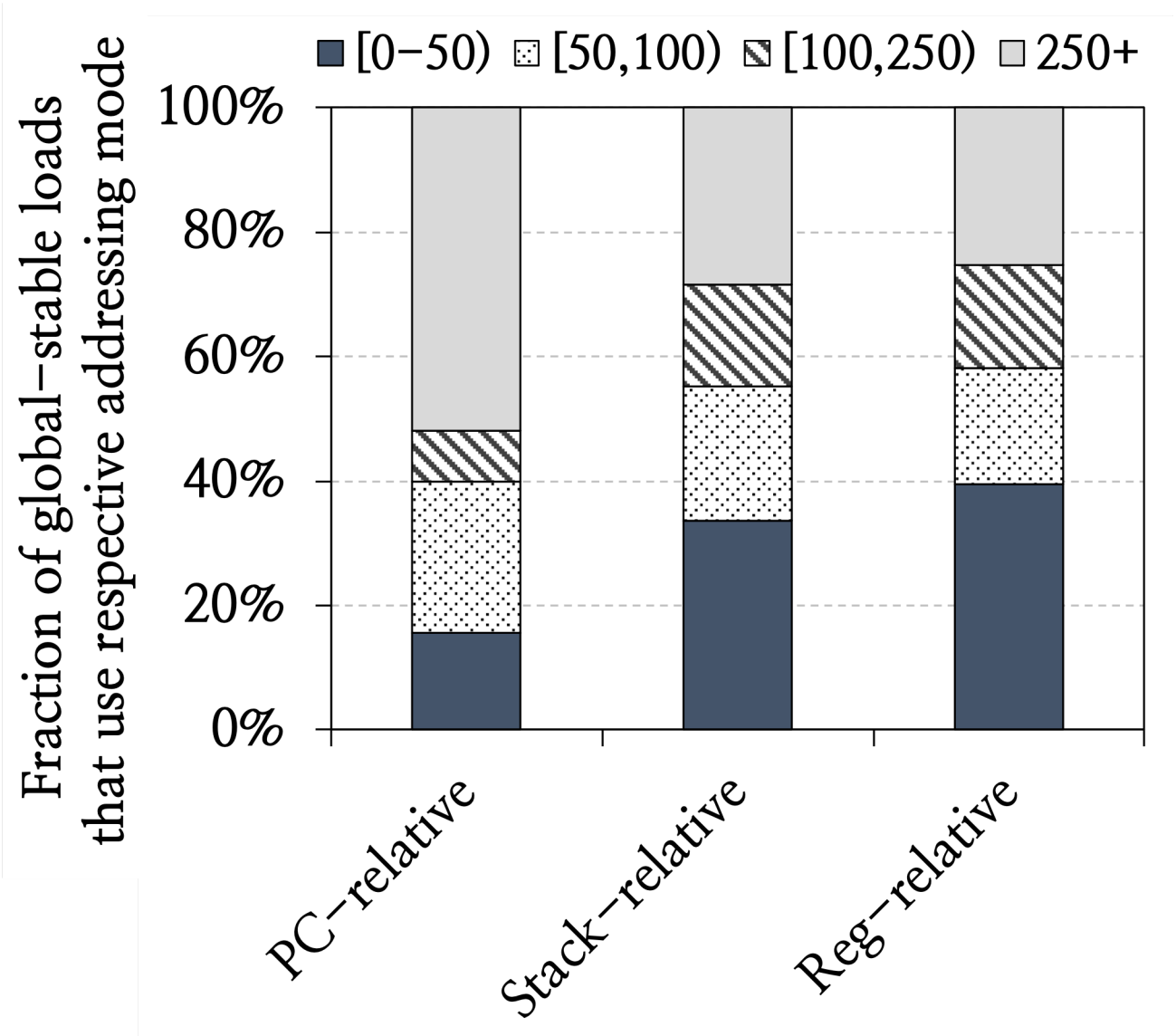
Characterization of Global-Stable Loads (I)



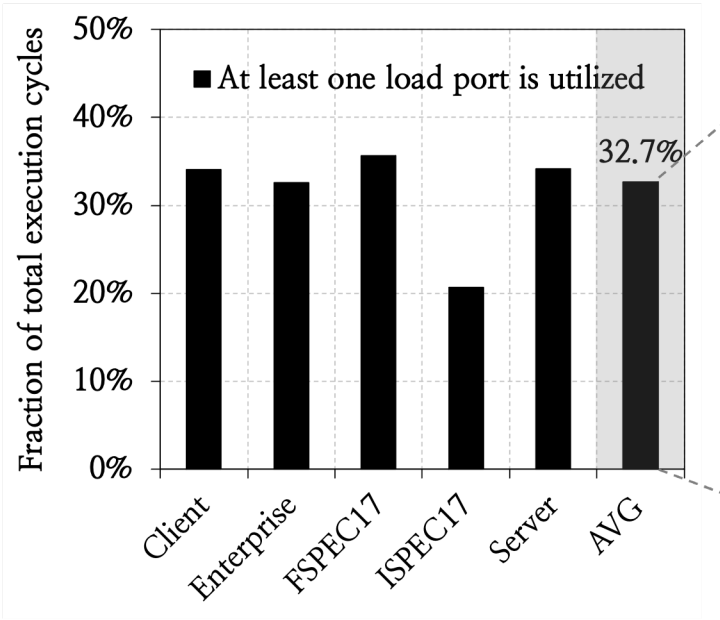
Characterization of Global-Stable Loads (II)

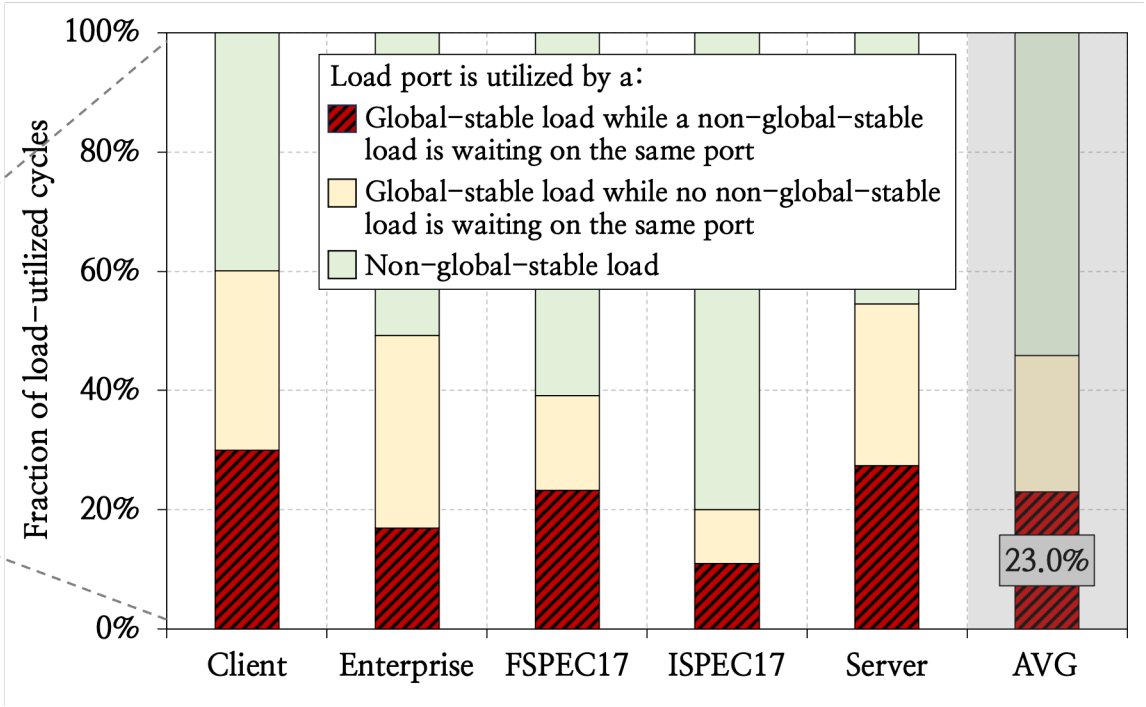


Characterization of Global-Stable Loads (III)



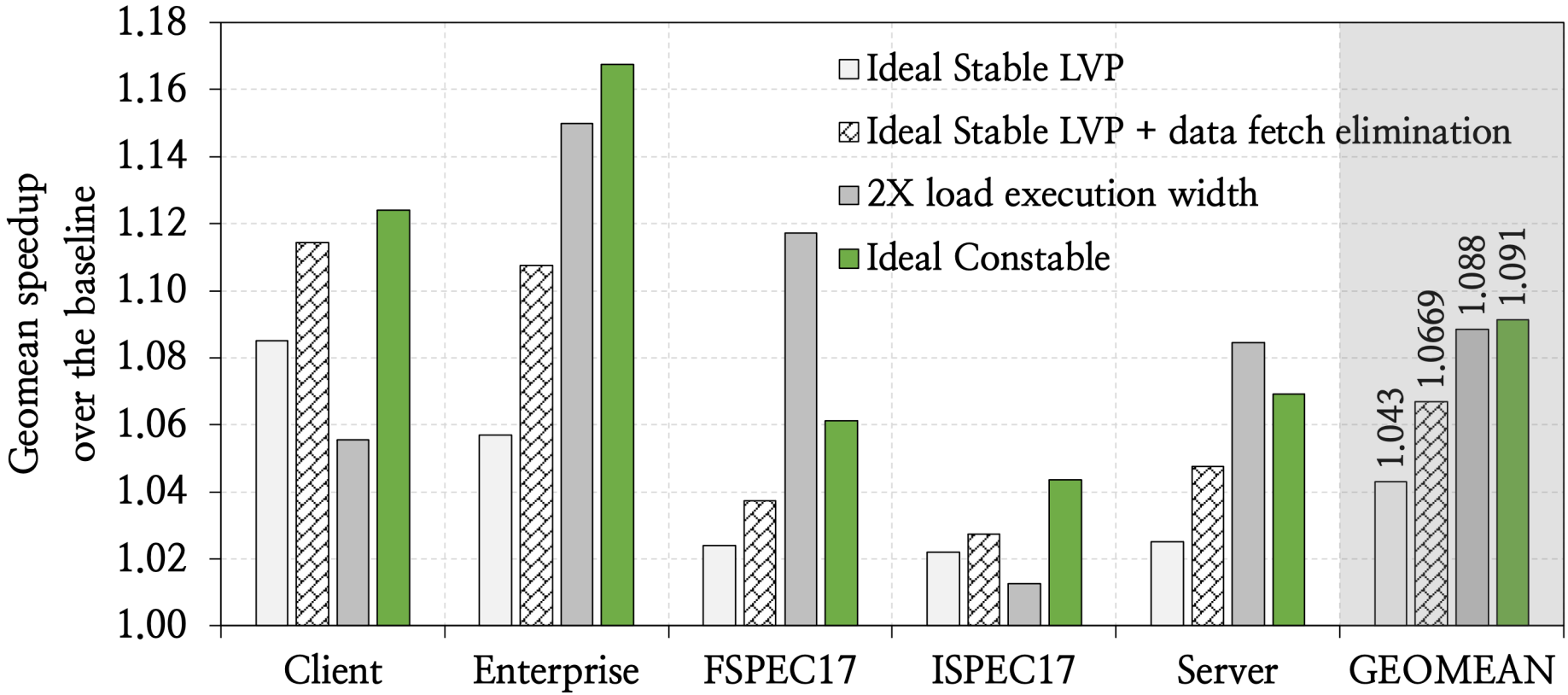
Resource Dependence by Global-Stable Loads



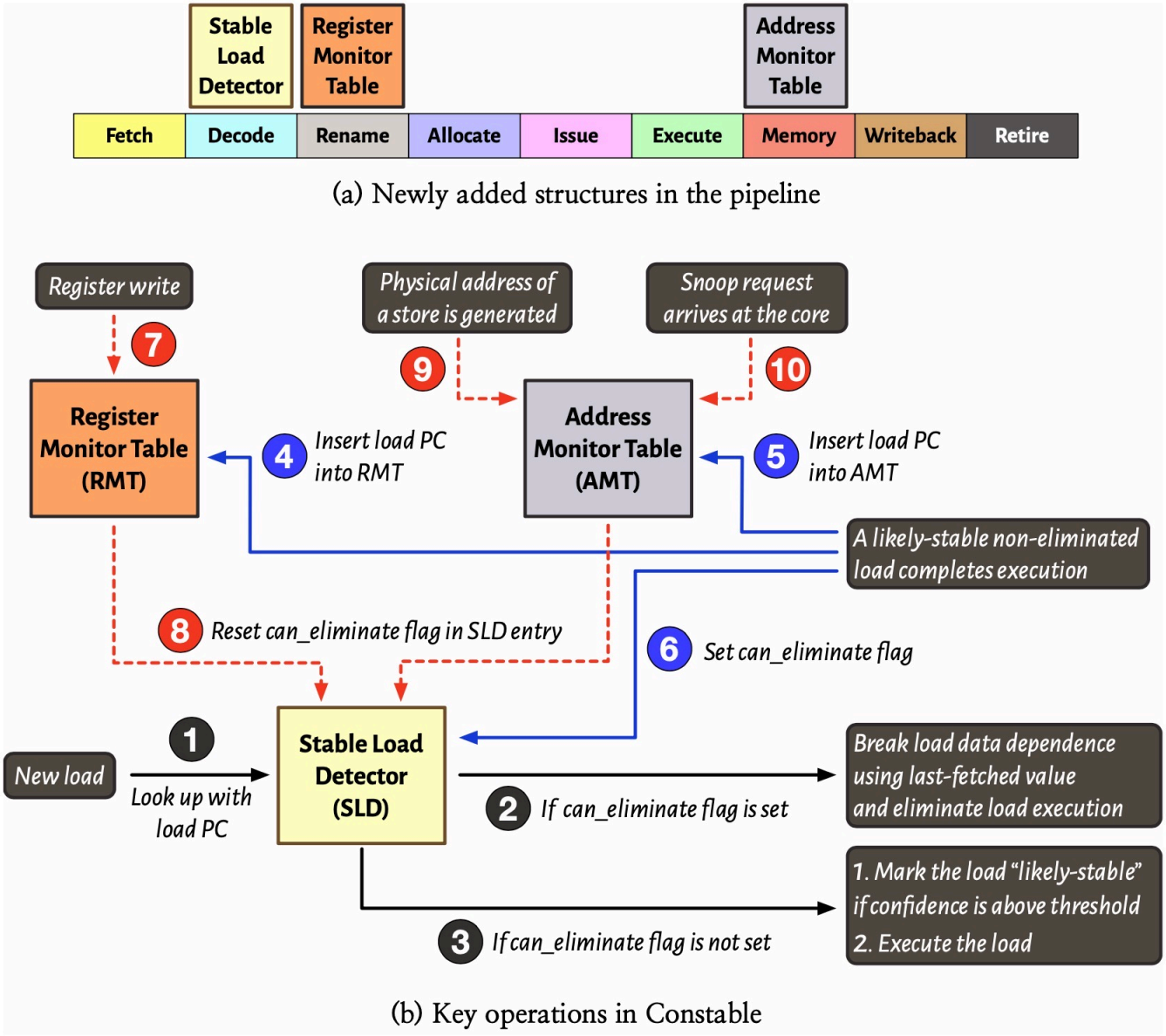


Global-stable loads cause significant resource dependence

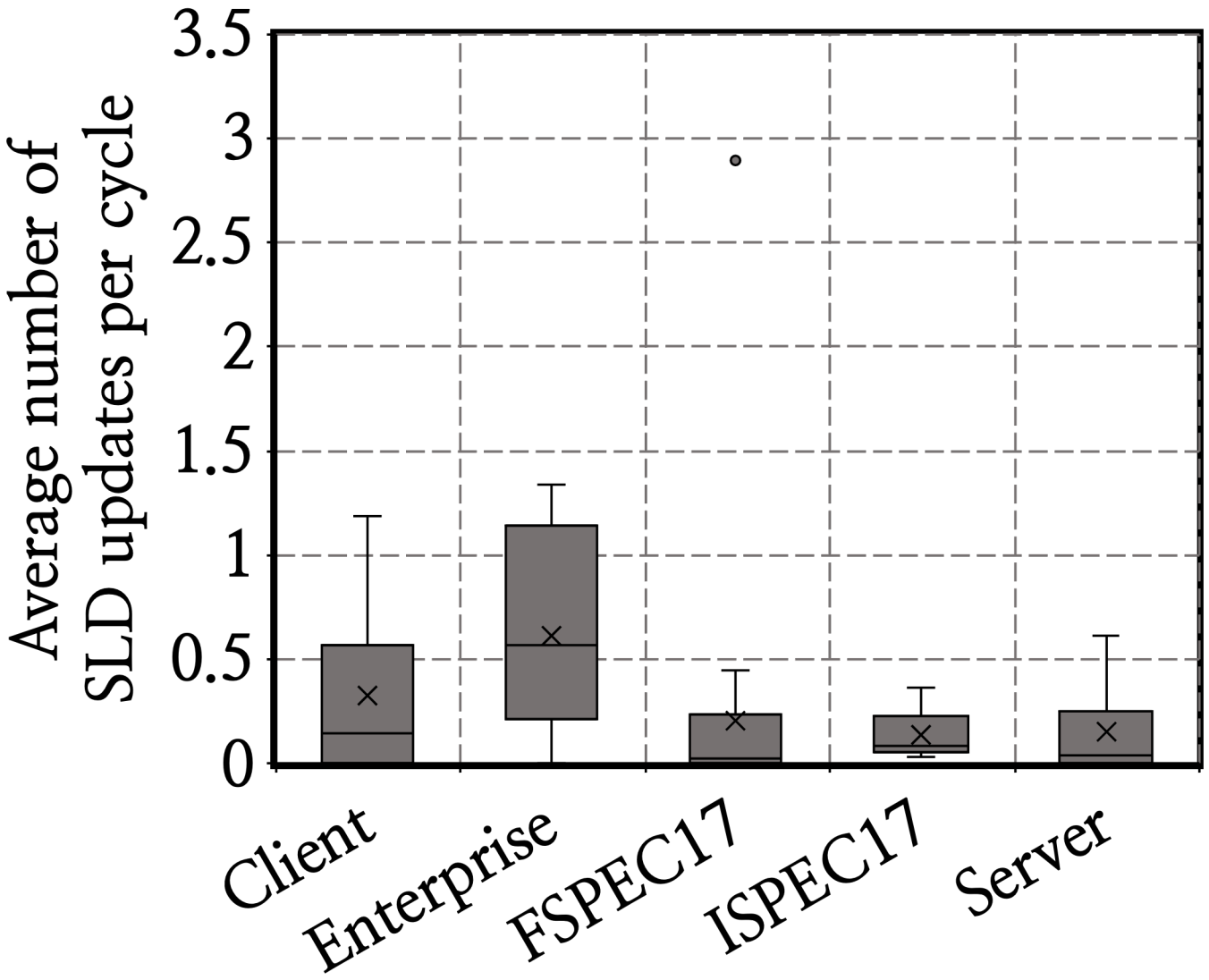
Performance Headroom Analysis



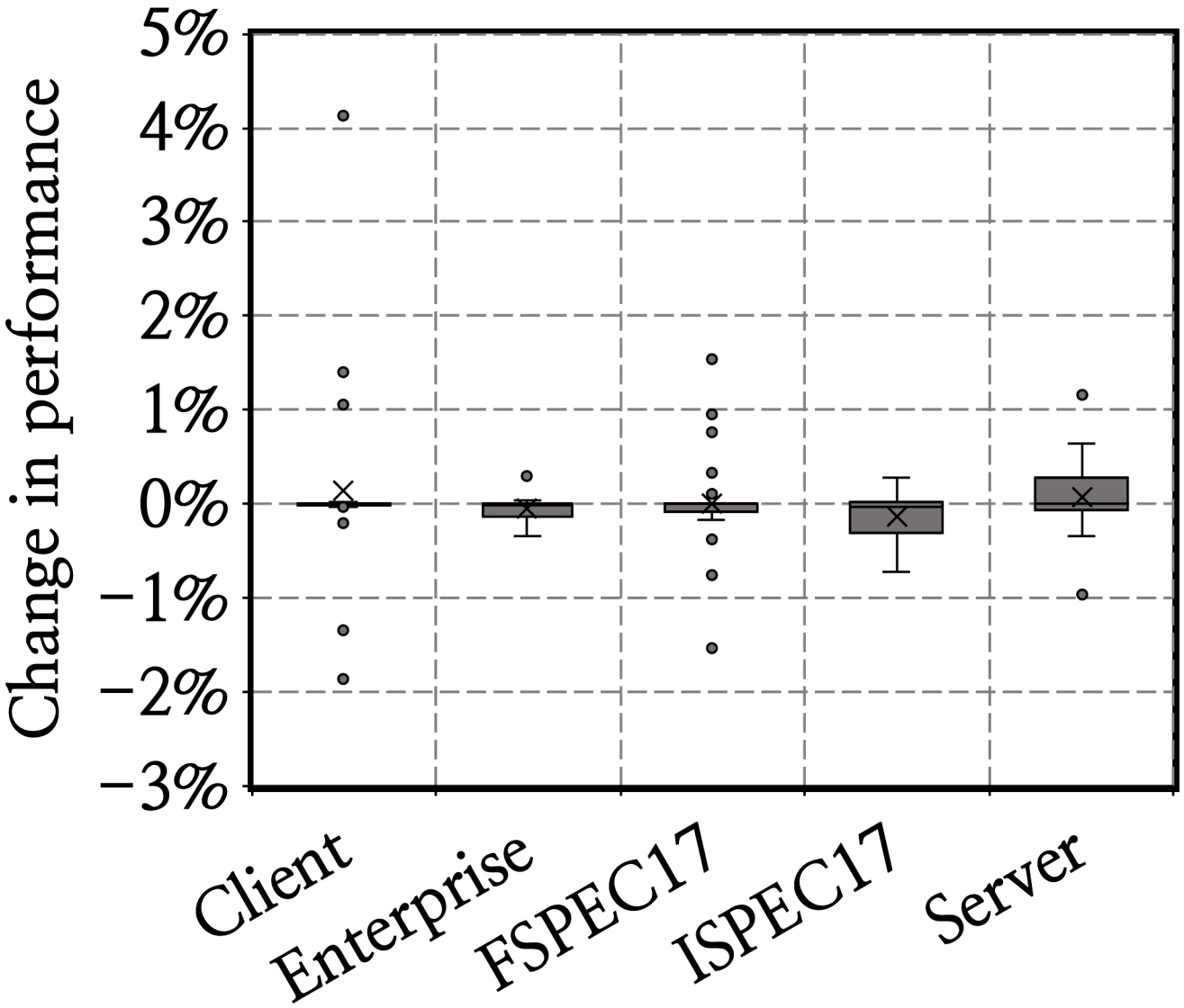
Constable Overview



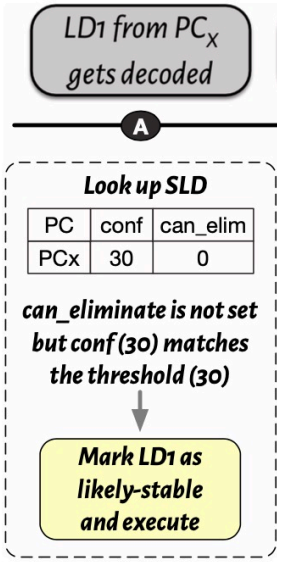
Architecting SLD



Effect of Wrong-Path Update

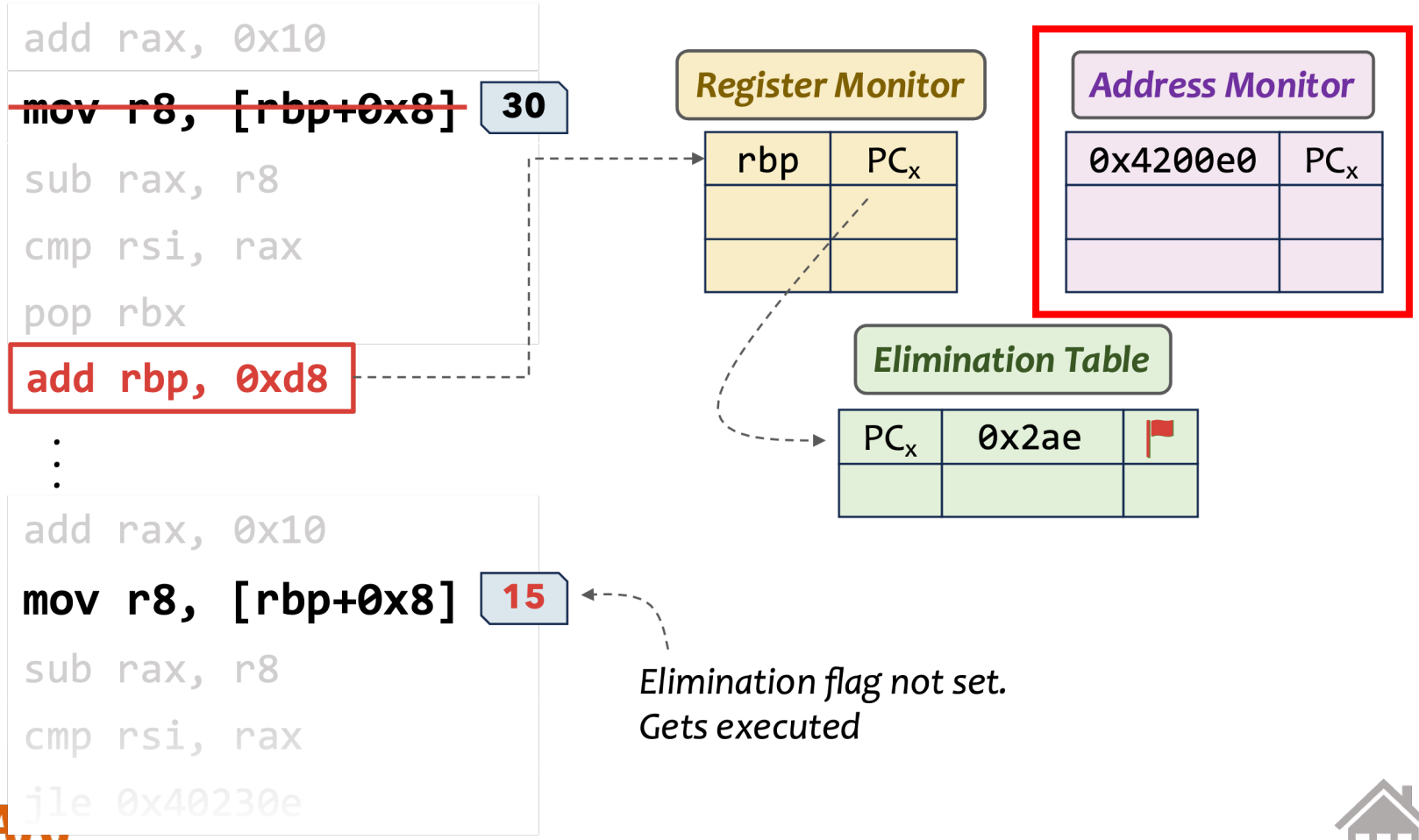


An Example of Constable's Operation



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

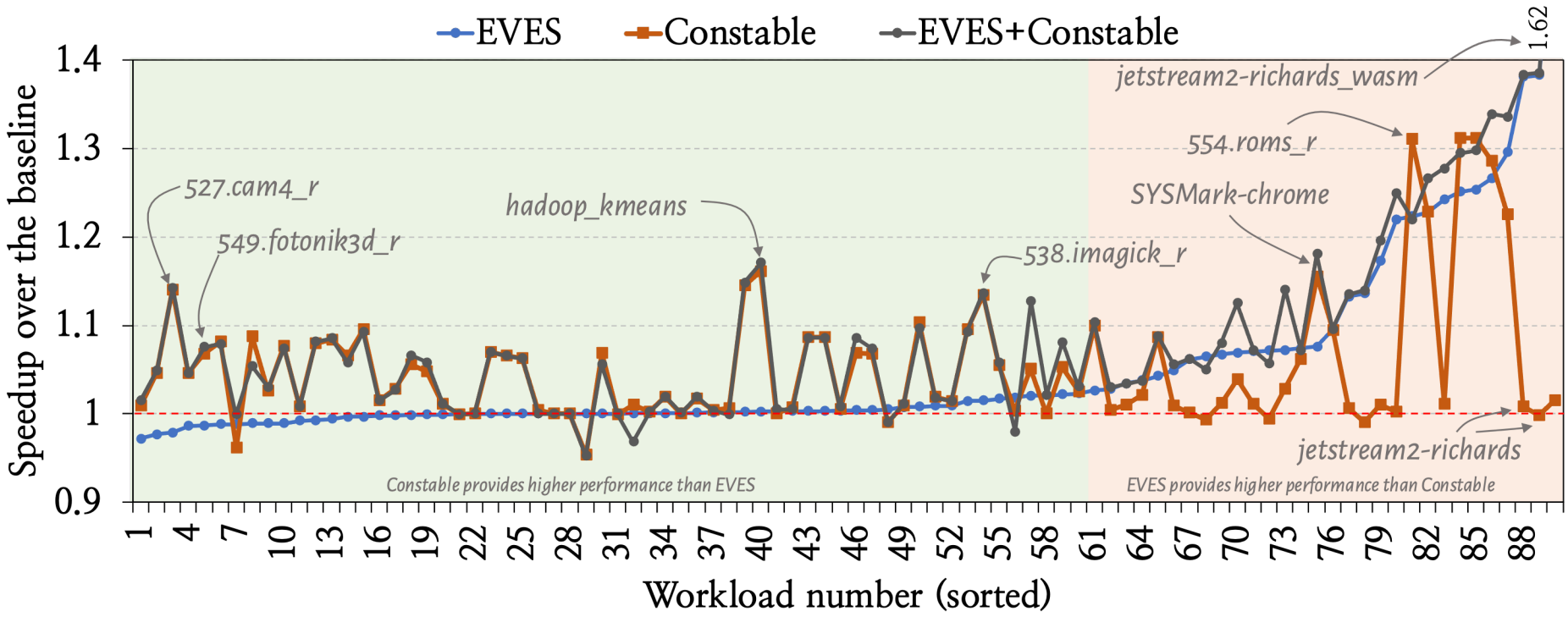
Simulated System Parameters

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**

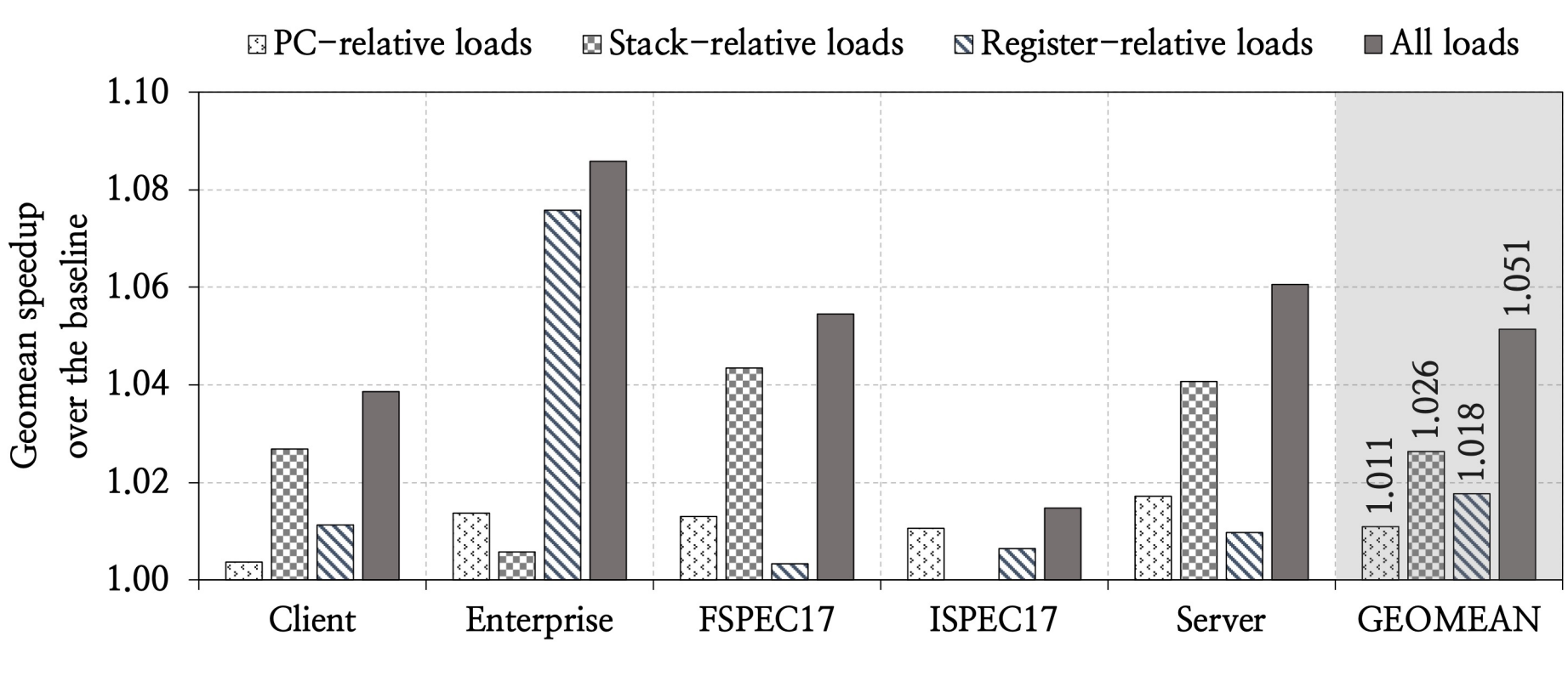
Evaluated Workloads

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]

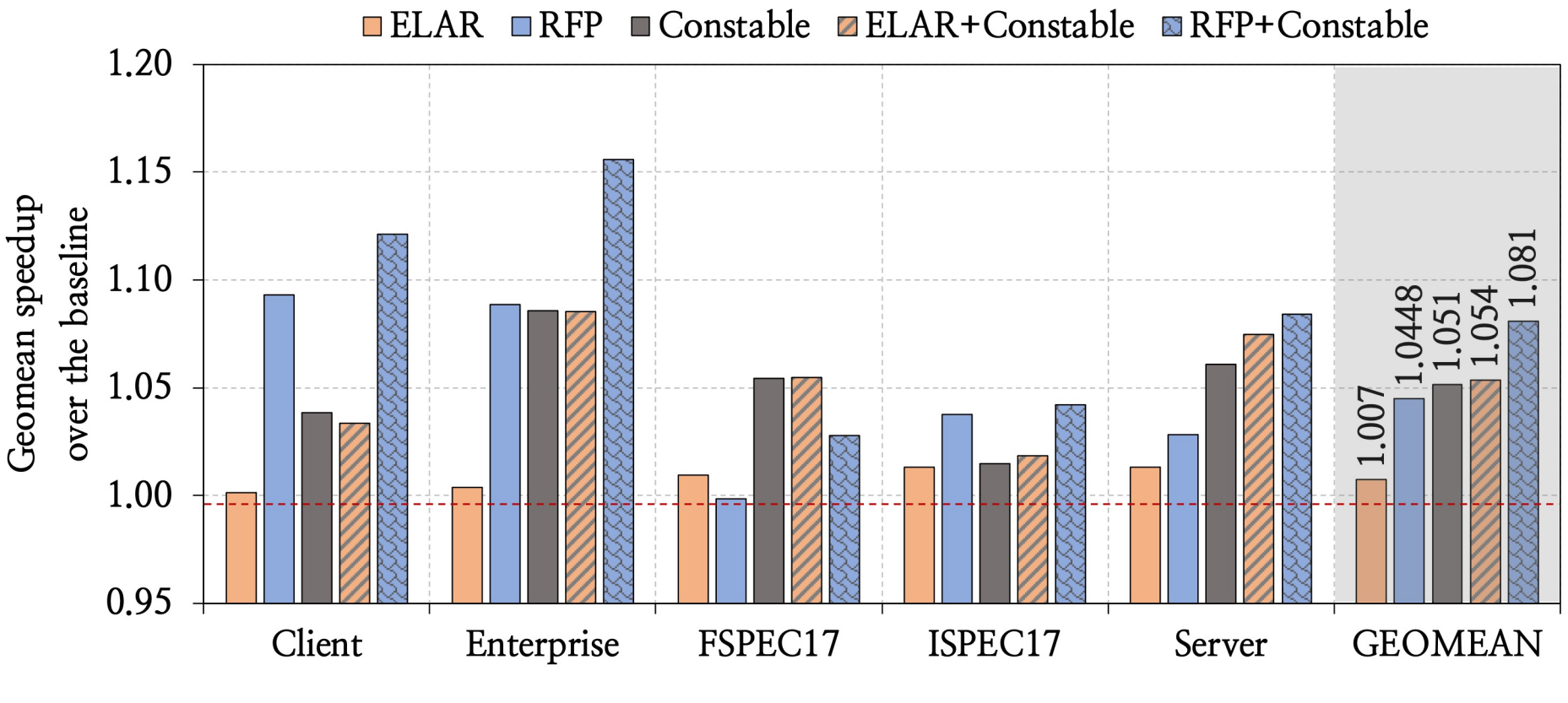
Workload-Wise Performance



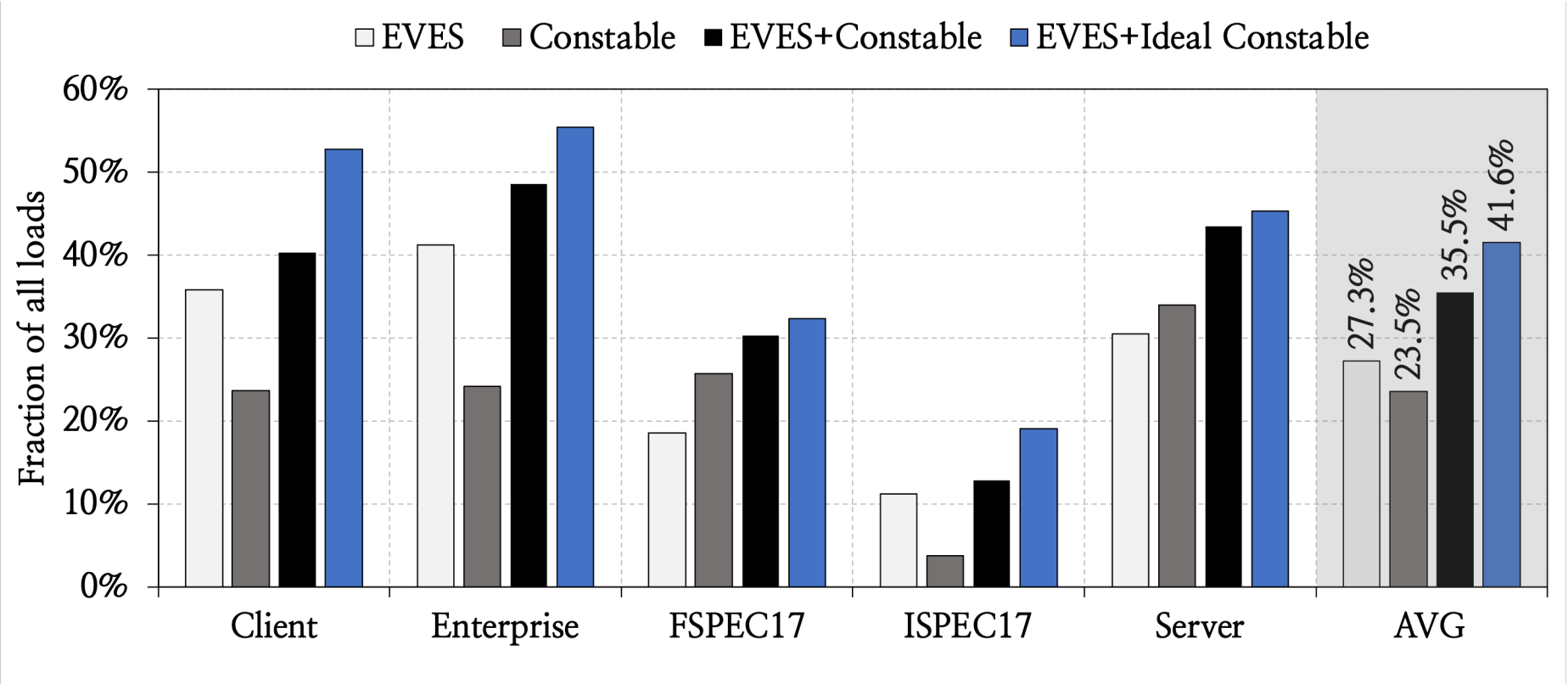
Load Category-Wise Performance



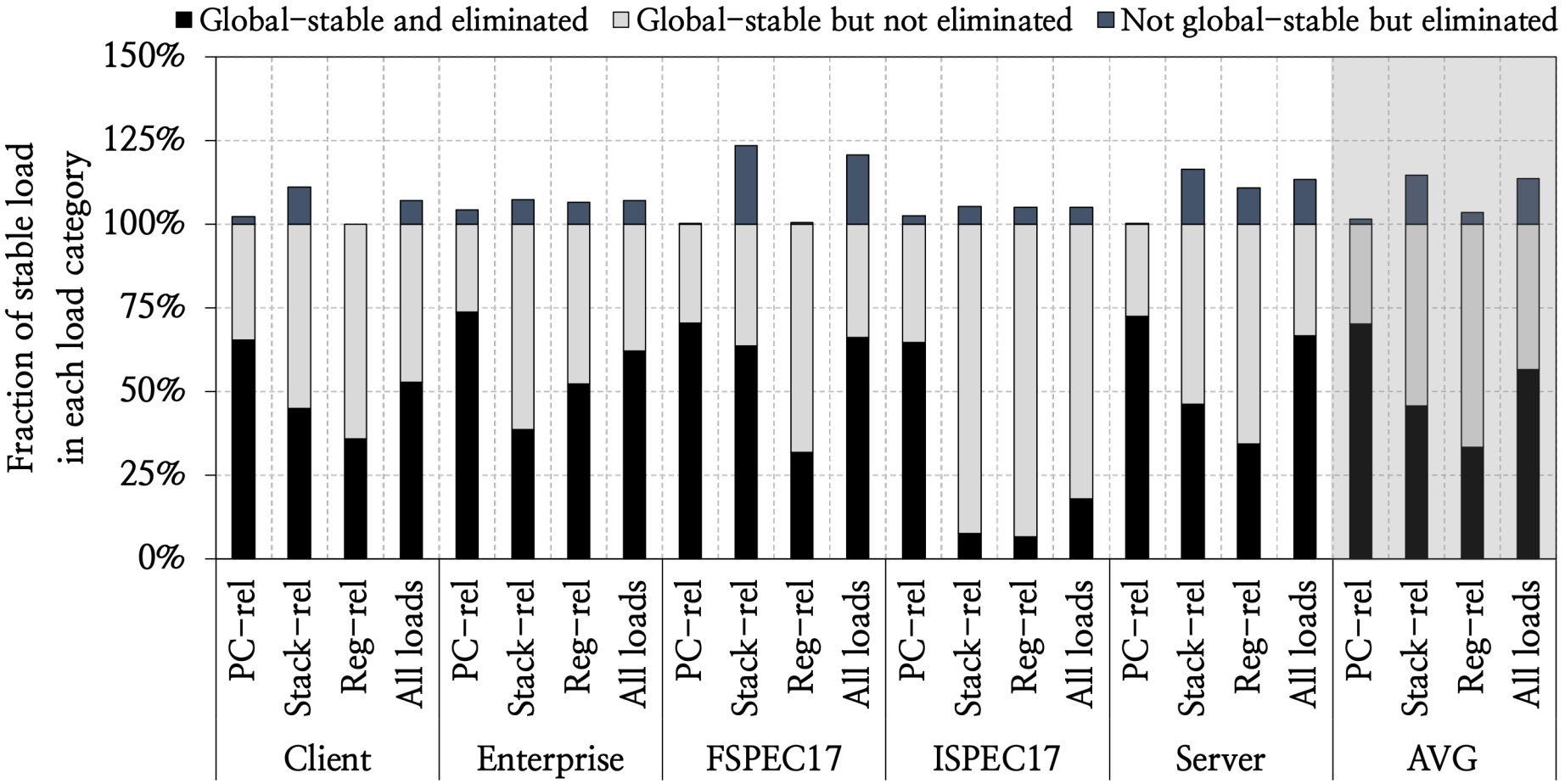
Performance Comparison with Prior Works



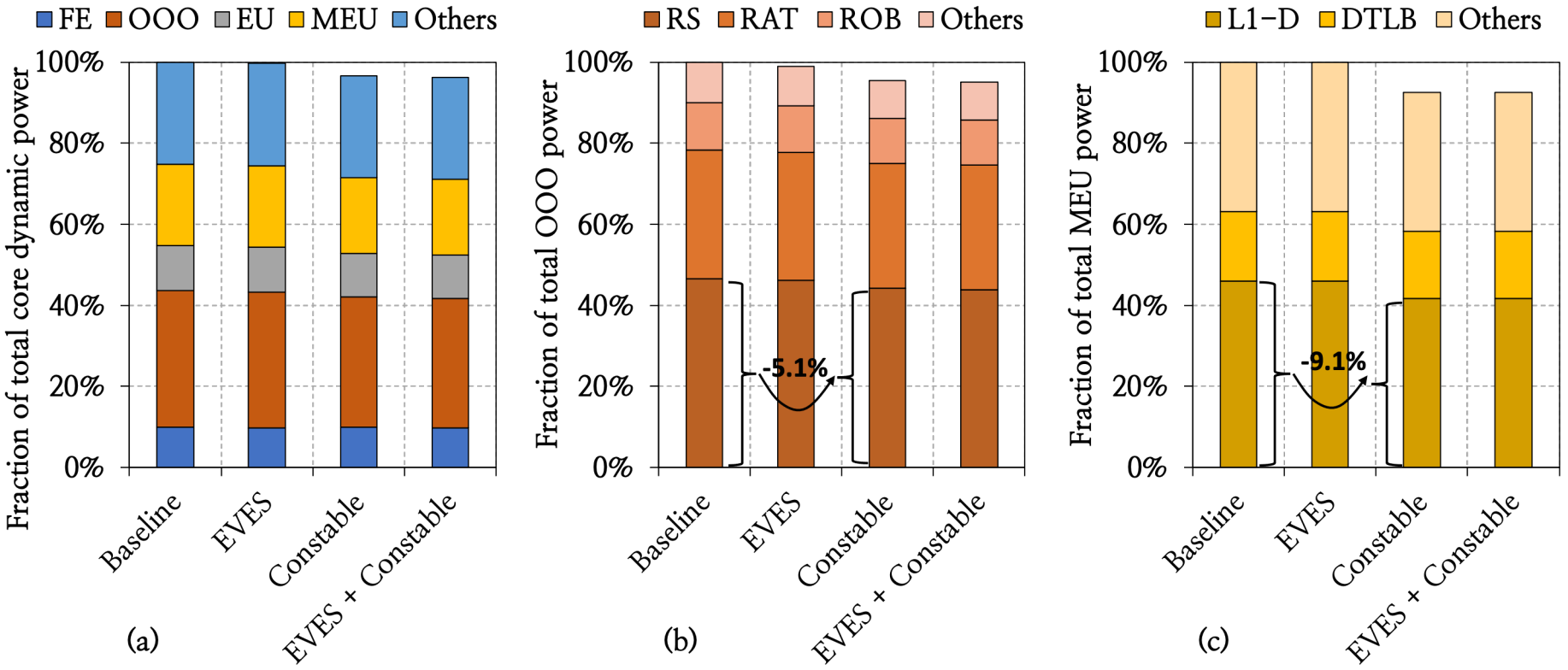
Elimination Coverage



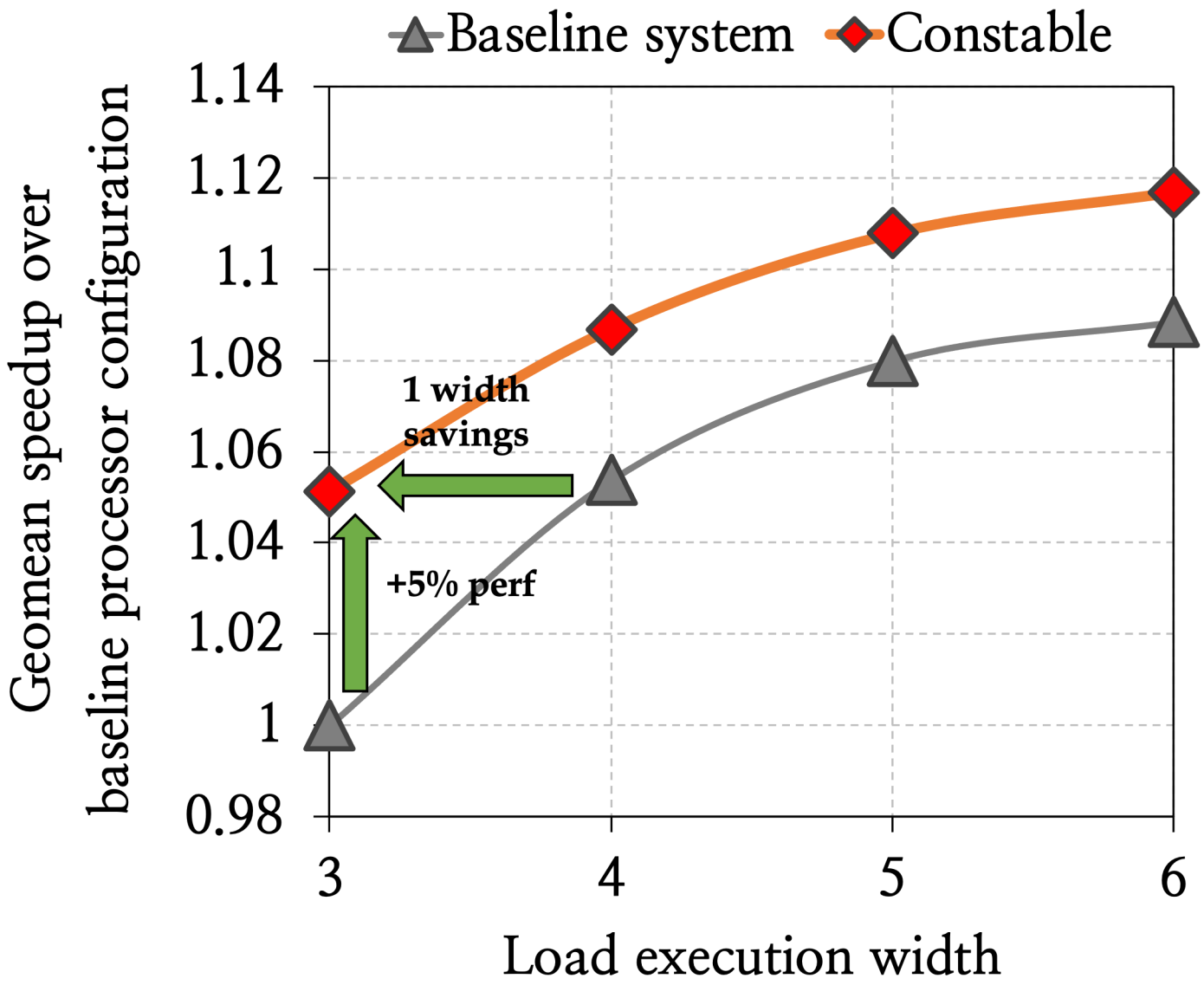
Coverage of Global-Stable Loads



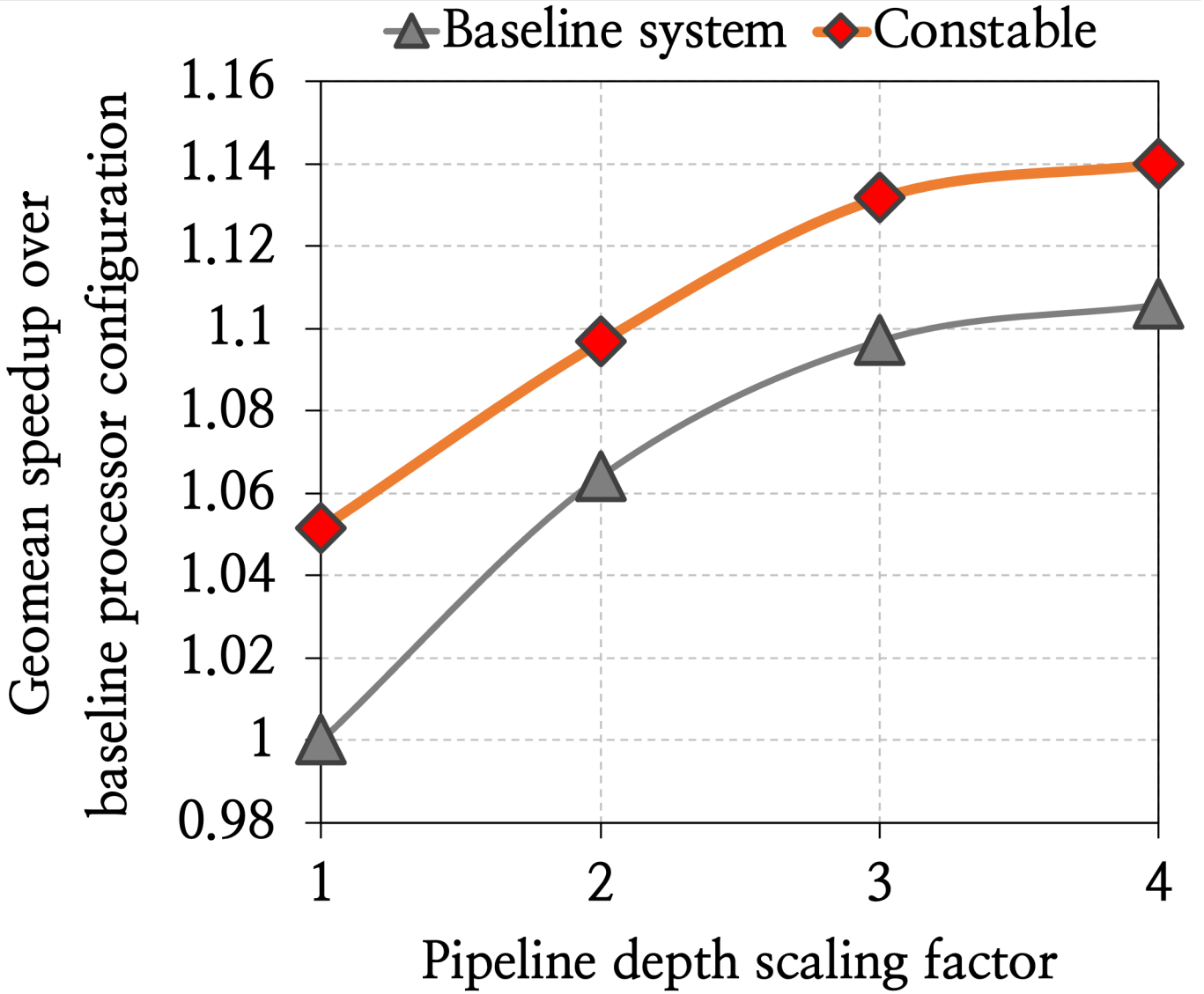
Reduction in Dynamic Power



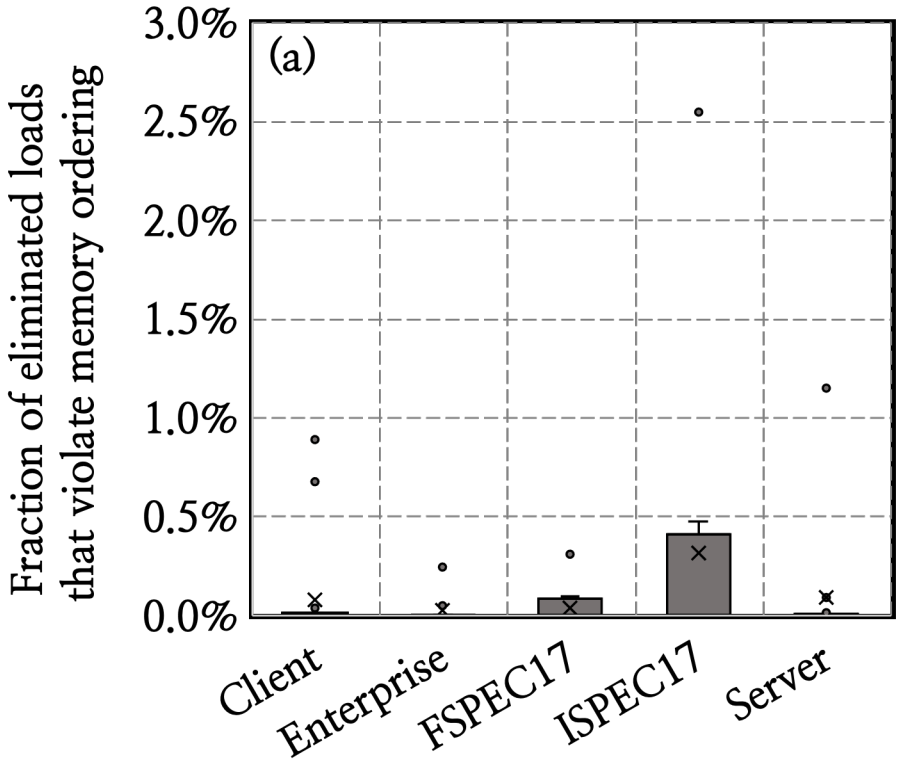
Performance Sensitivity to Load Execution Width Scaling



Performance Sensitivity to Pipeline Depth Scaling

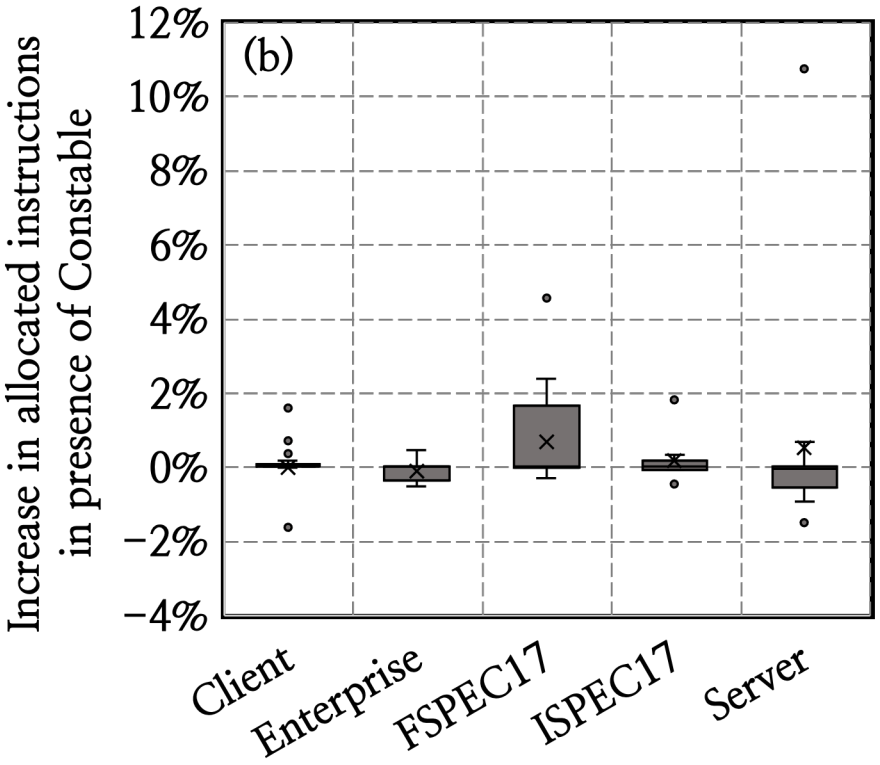


Eliminated Loads that Violates Memory Ordering



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%

Executive Summary

Key Findings

1. A significant fraction of loads always fetch the same data from same address throughout the entire workload
2. These loads cause significant resource dependence even when using value prediction and memory renaming
3. Mitigating their resource dependence has significant performance headroom

Key Mechanism

Constable

Key Idea: To mitigate both load data and load resource dependence by safely eliminating the entire execution of a load

Key Result: **Simultaneously improves performance and power** efficiency of a strong baseline OoO processor