Elzar

Triple Modular Redundancy using Intel AVX

Dmitrii Kuvaiskii

Oleksii Oleksenko Pramod Bhatotia Christof Fetzer Pascal Felber

TECHNISCHE UNIVERSITÄT DRESDEN

UNIVERSITÉ DE NEUCHÂTEL

Hardware Errors in the Wild

- Online services run in **huge data centers**

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
- These faults can lead to **arbitrary state corruptions**

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
- These faults can lead to **arbitrary state corruptions**

„…There were a handful of messages that had **a single bit corrupted** such that the message was **still intelligible**, but the system state information was **incorrect**…“

Hardware Errors in the Wild

- Online services run in **huge data centers**
- **Hardware faults** are the norm rather than the exception
- These faults can lead to **arbitrary state corruptions**

„…There were a handful of messages that had **a single bit corrupted** such that the message was **still intelligible**, but the system state information was **incorrect**…“

Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing

Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan
Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal
Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones
Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, Divyakant Agrawal
Google, Inc.

„…Corruption can occur **transiently in CPU** or RAM. Guarding against such corruptions is an **important goal** in Mesa’s overall design…“

Protecting Against Data Corruptions

Protecting Against Data Corruptions

Byzantine Fault Tolerance

Protecting Against Data Corruptions

Byzantine Fault Tolerance

☺ Tolerates arbitrary faults

☹ Pessimistic fault model

☹ High resource overheads

Protecting Against Data Corruptions

Byzantine Fault Tolerance

- ☺ Tolerates arbitrary faults
- ☹ Pessimistic fault model
- ☹ High resource overheads

Checksums / Assertions

Protecting Against Data Corruptions

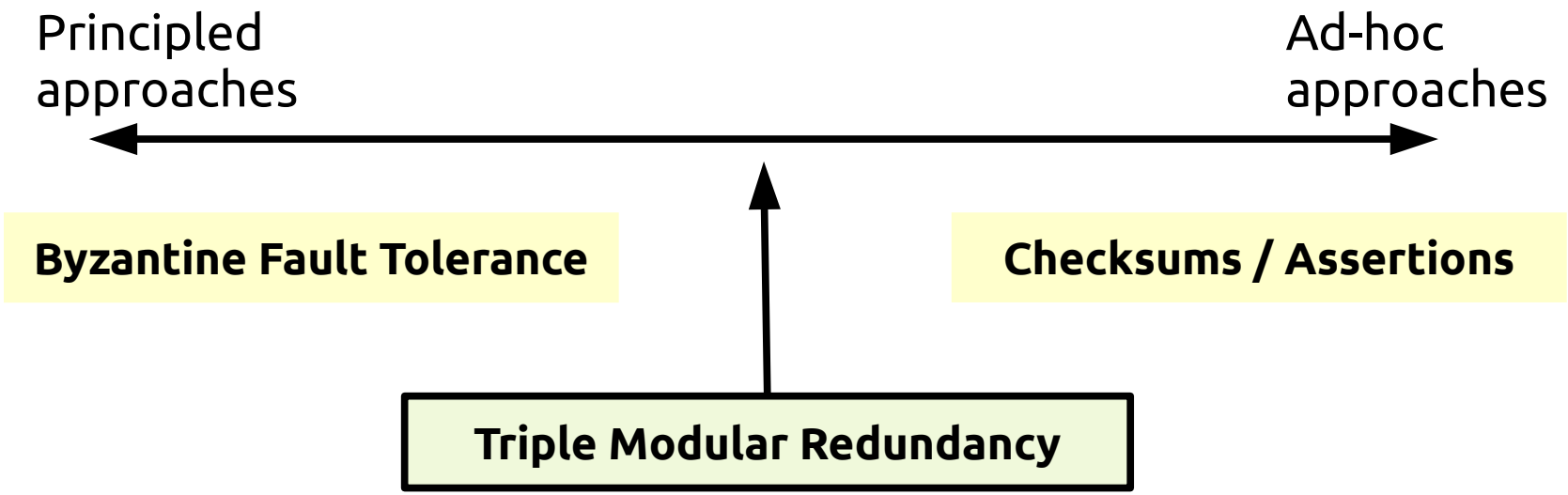
Byzantine Fault Tolerance

- ☺ Tolerates arbitrary faults
- ☹ Pessimistic fault model
- ☹ High resource overheads

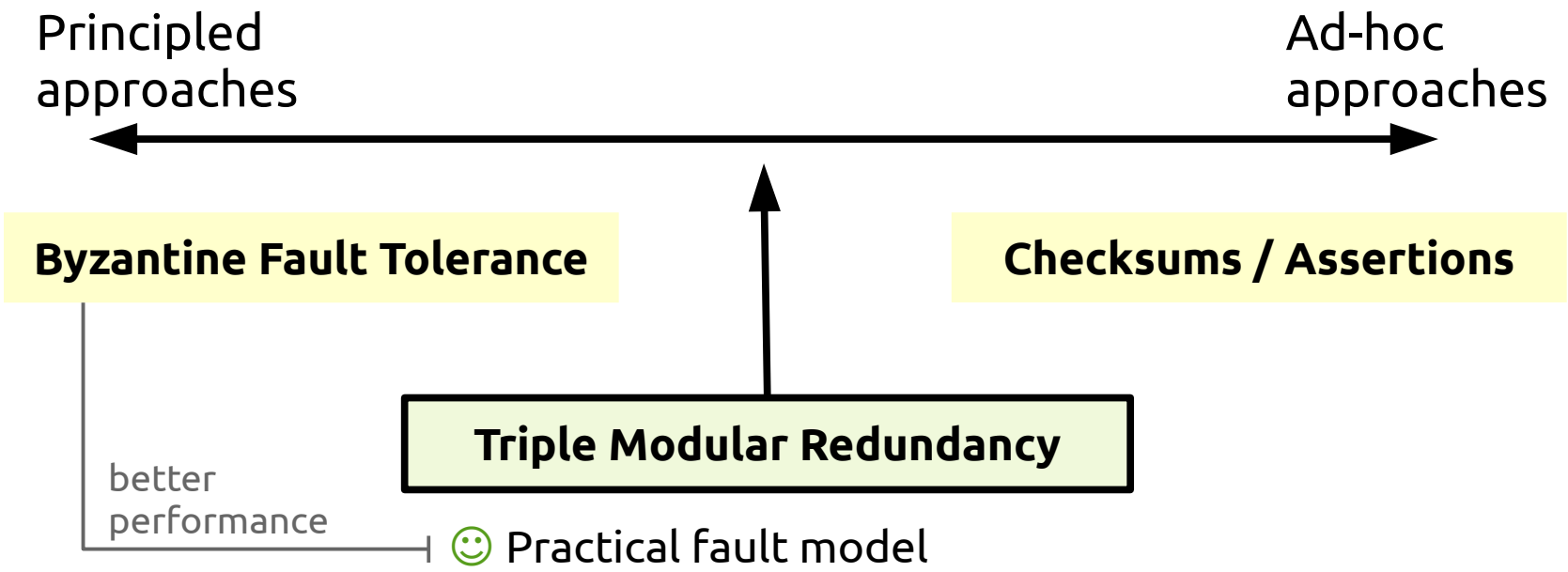
Checksums / Assertions

- ☺ Low performance overheads
- ☹ Only anticipated faults
- ☹ Manual and error-prone

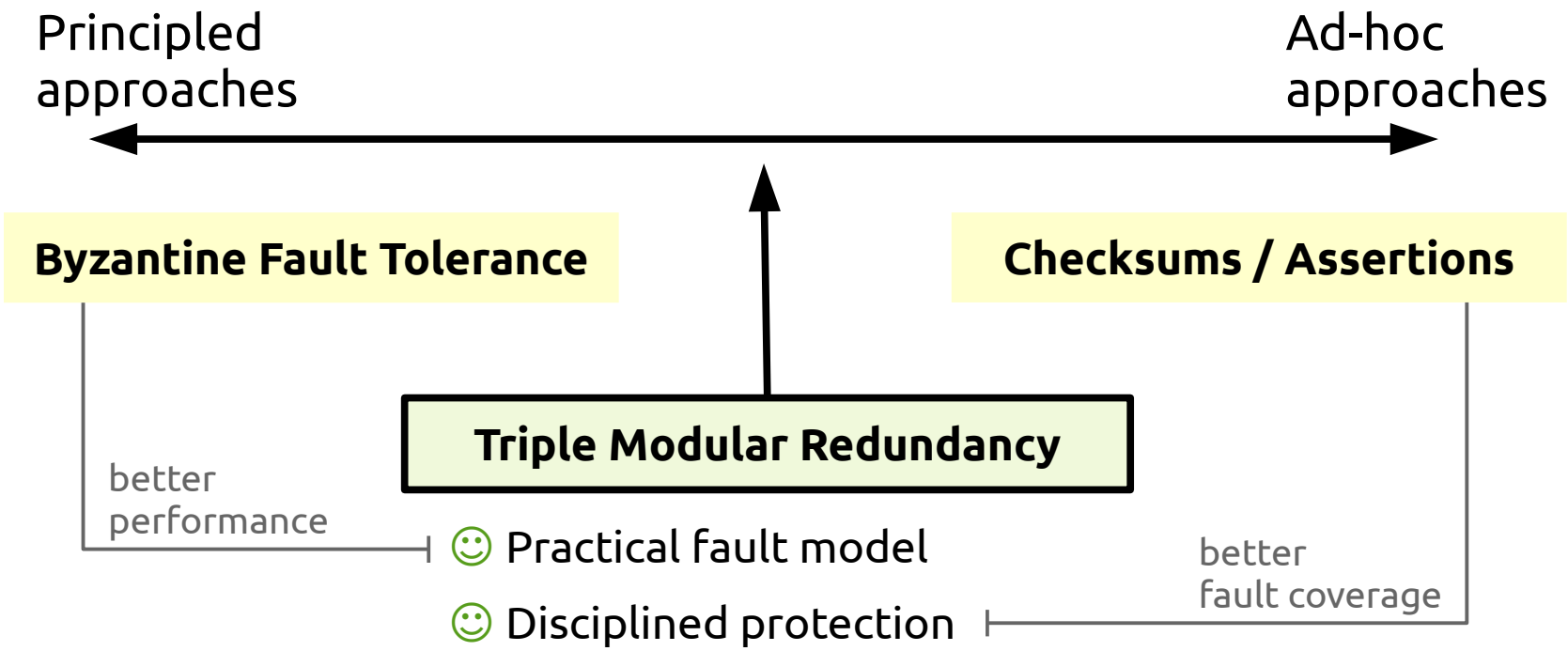
Protecting Against Data Corruptions



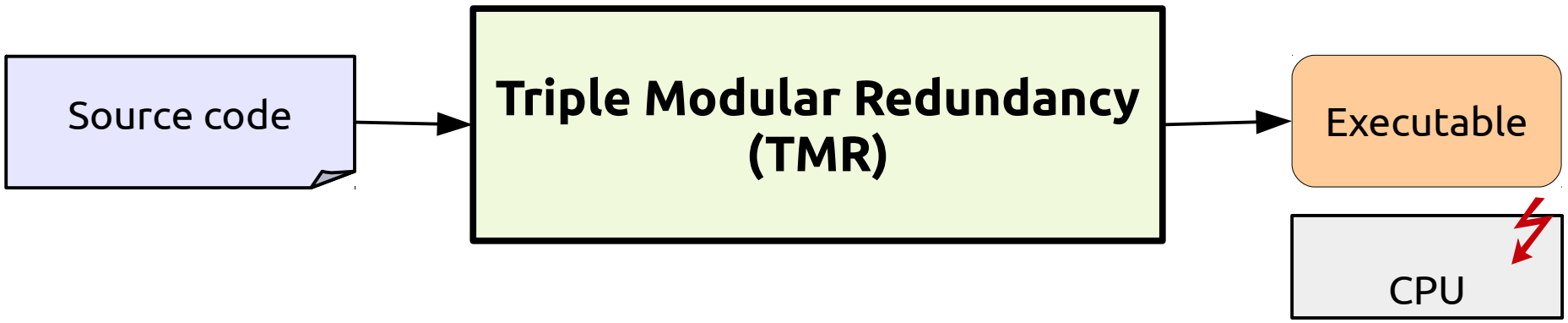
Protecting Against Data Corruptions



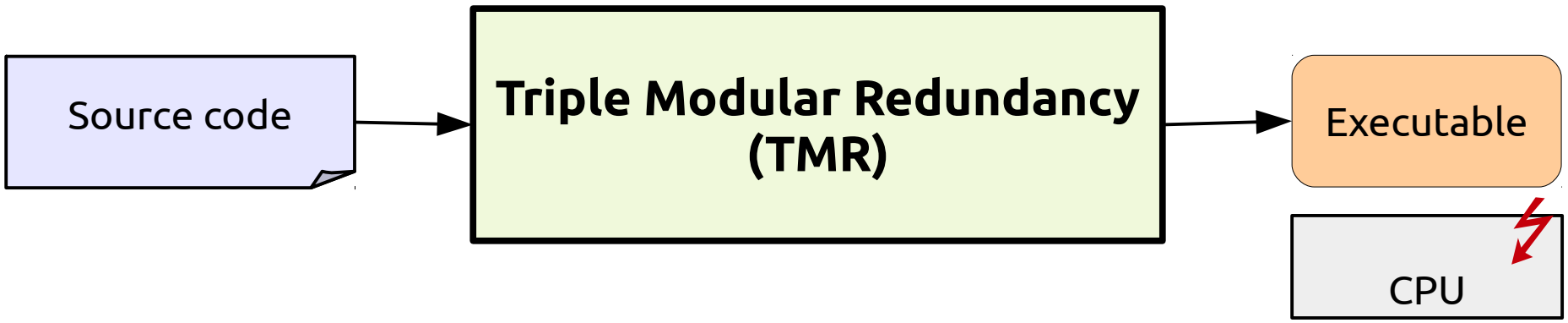
Protecting Against Data Corruptions



Triple Modular Redundancy



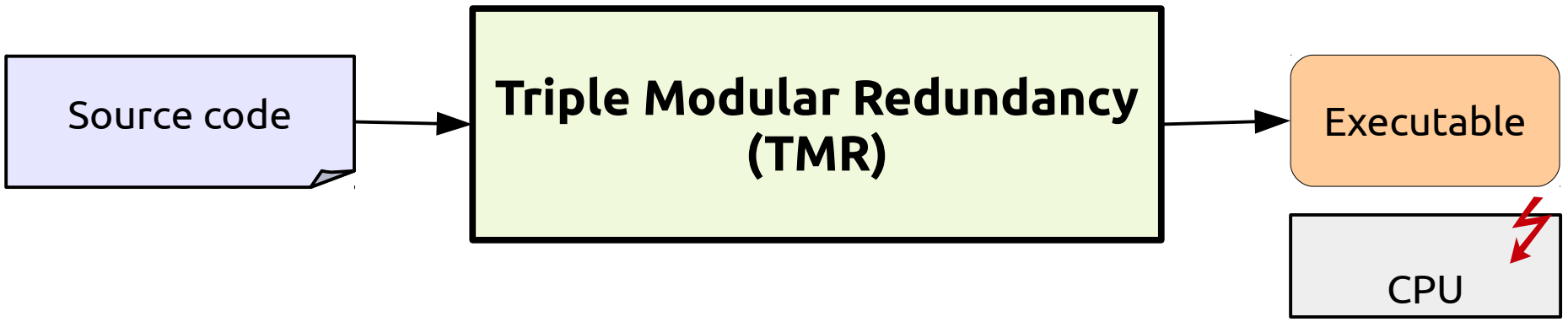
Triple Modular Redundancy



Native

```
z   = add x, y
```

Triple Modular Redundancy



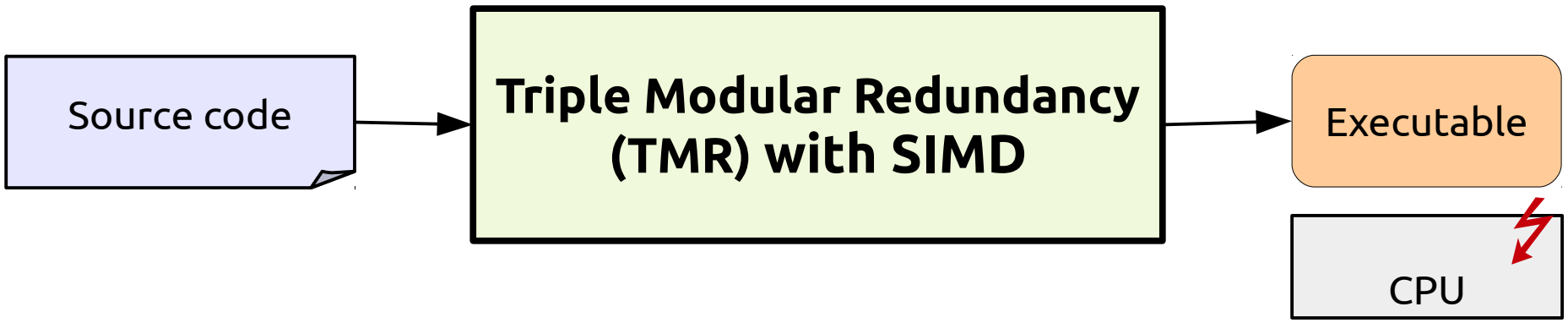
Native

```
z   = add x, y
```

TMR

```
z   = add x,   y
z2  = add x2,  y2
z3  = add x3,  y3
```

Triple Modular Redundancy



Native

```
z   = add x, y
```

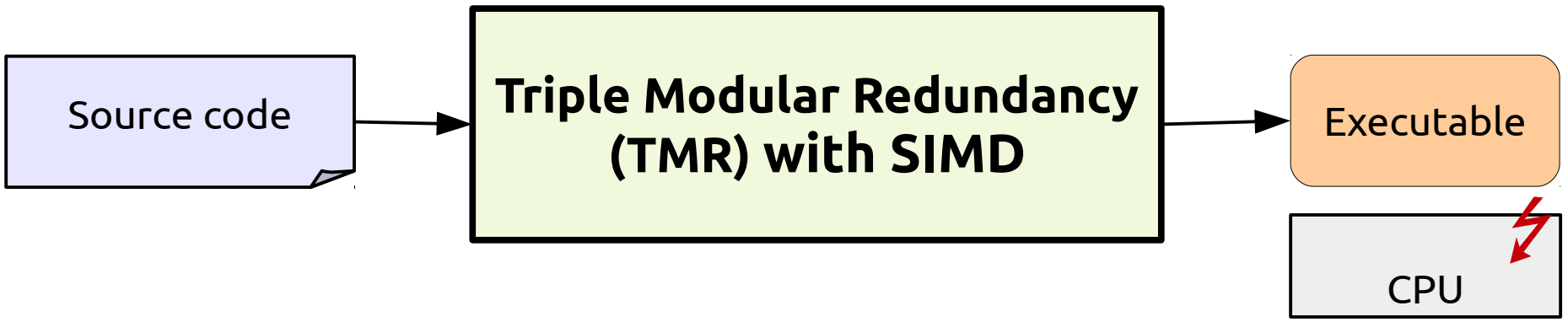
TMR

```
z   = add x,   y
z2  = add x2,  y2
z3  = add x3,  y3
```

TMR with SIMD

```
zwide = add xwide, ywide
```

Triple Modular Redundancy



Native

```
z   = add x, y
```

TMR

```
z   = add x,   y
z2  = add x2,  y2
z3  = add x3,  y3
```

TMR with SIMD

```
zwide = add xwide, ywide
```

Idea **SIMD** uses less instructions → **less perf overhead?**

Research Question

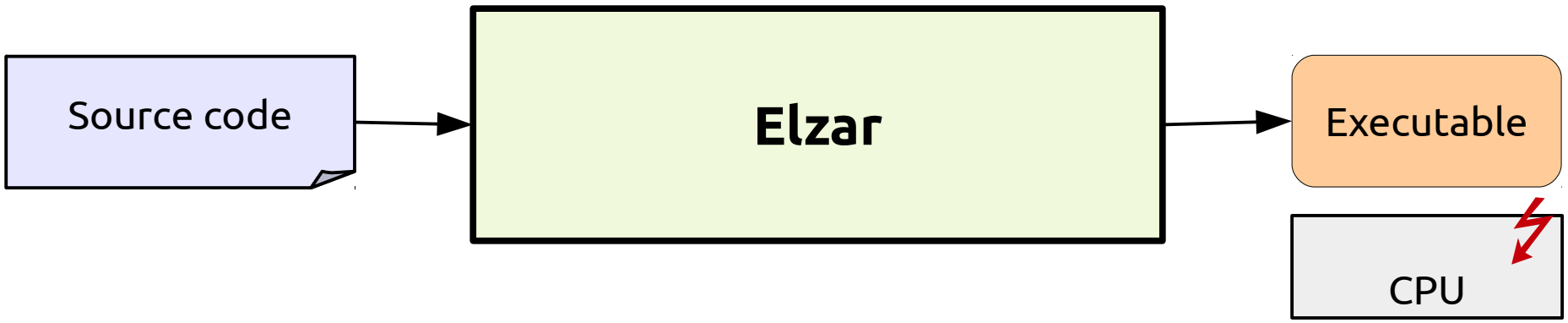
Can we use **SIMD** for **efficient fault tolerance**?

Research Question

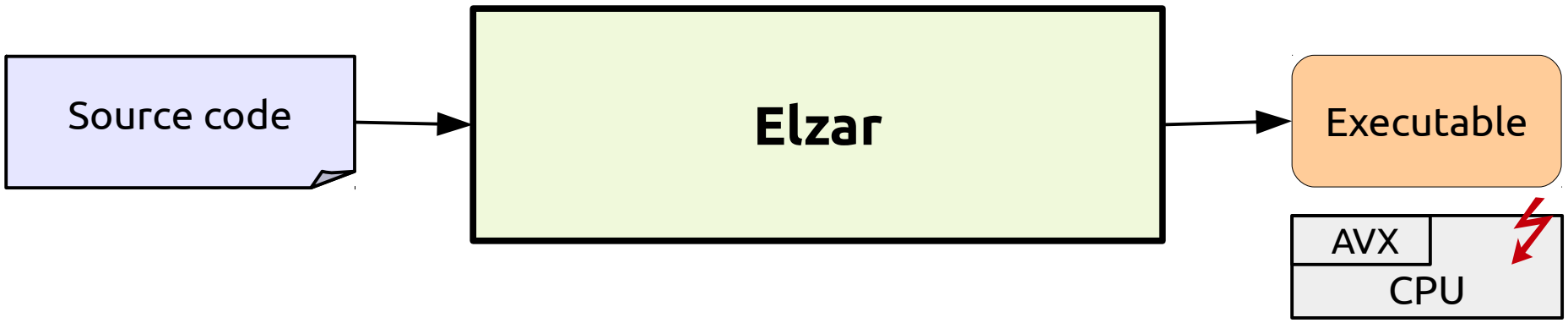
Can we use **SIMD** for **efficient fault tolerance**?

Implementation using **Intel AVX**

Elzar

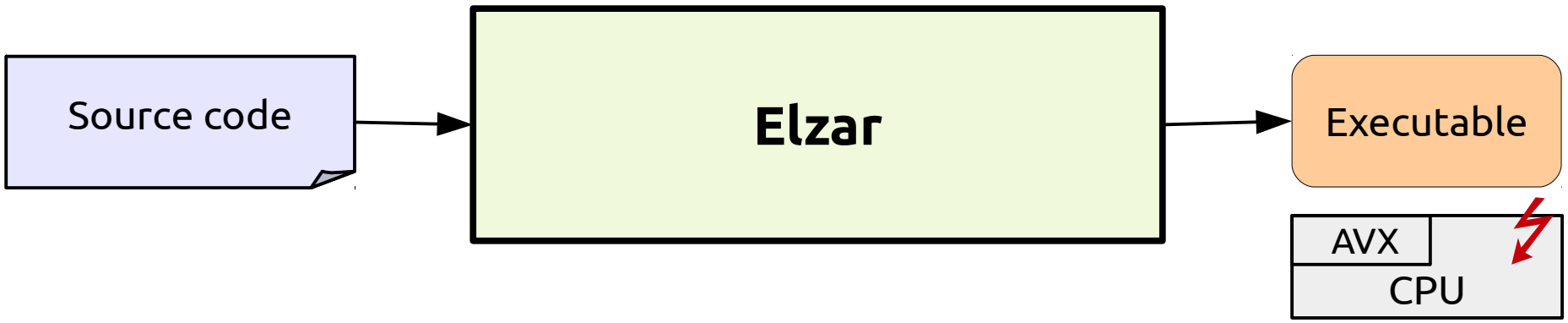


Elzar



Implementation **Intel AVX** for triple modular redundancy

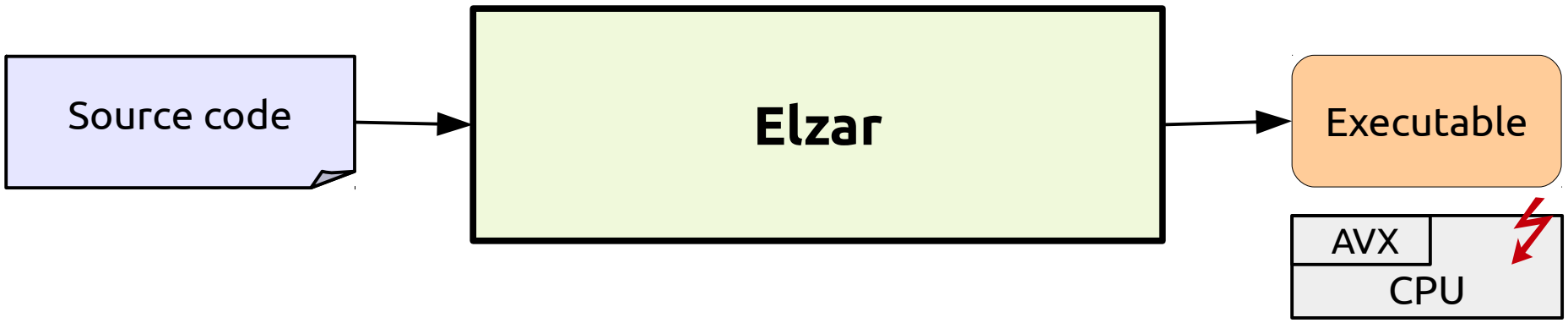
Elzar



Implementation **Intel AVX** for triple modular redundancy

Outcome **Mixed results** ☹ (AVX not general-purpose)

Elzar



Implementation **Intel AVX** for triple modular redundancy

Outcome **Mixed results** ☹ (AVX not general-purpose)

Investigation **Two bottlenecks** in current AVX

☑ Motivation

☐ **Intel AVX**

☐ Design

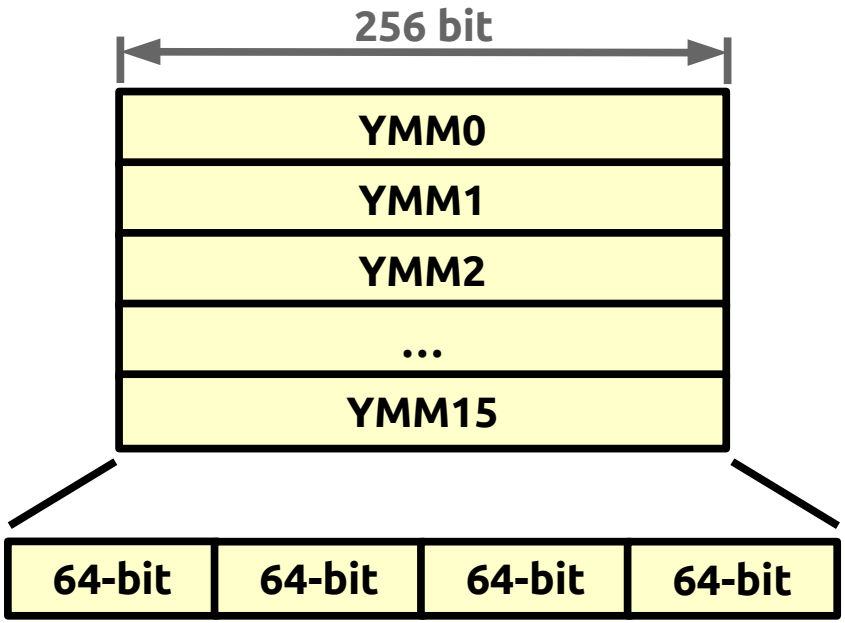
☐ Evaluation

☐ Discussion

Intel AVX Background

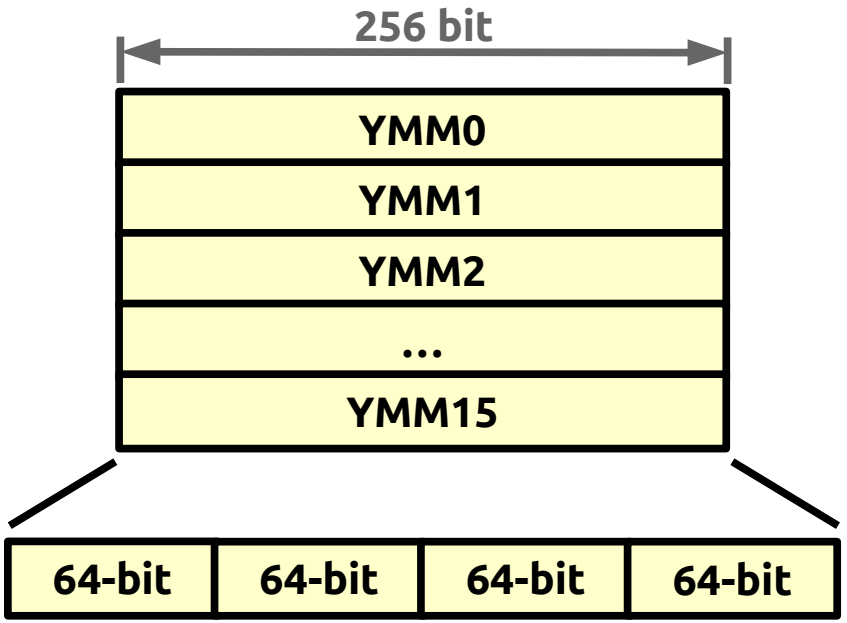
Intel AVX Background

New registers

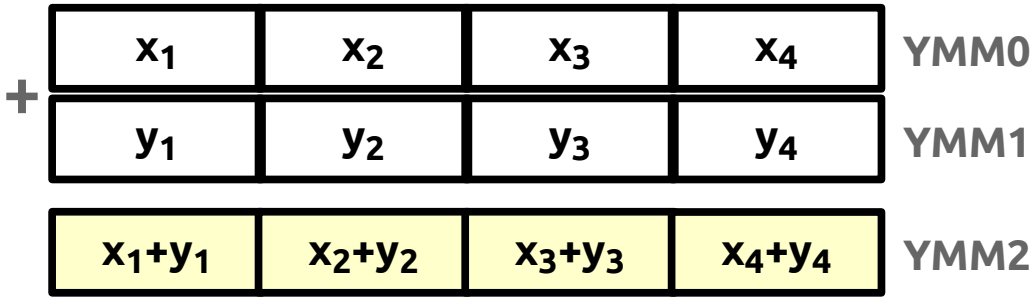


Intel AVX Background

New registers



New instructions

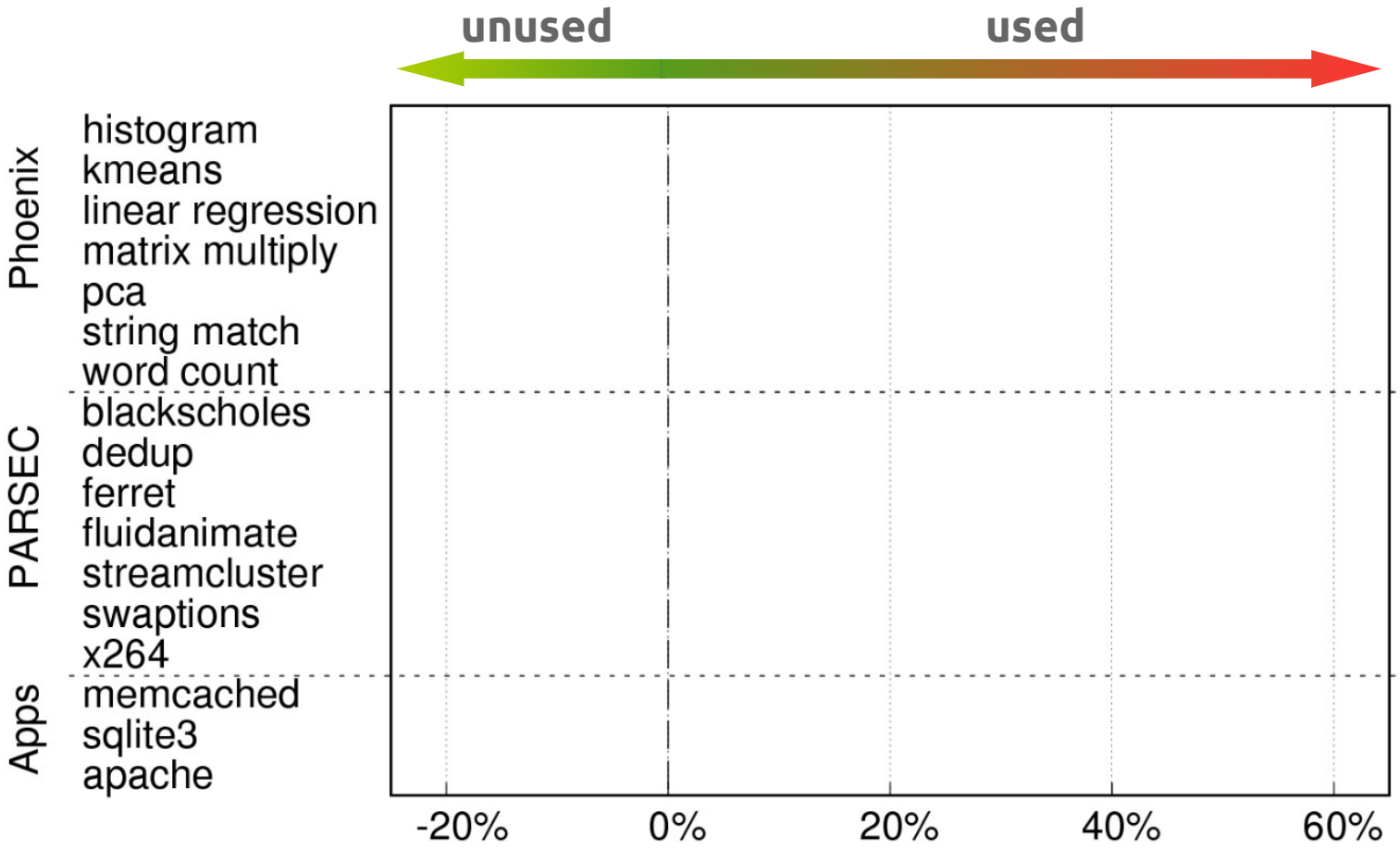


AVX as free resource?

AVX is not actively used?

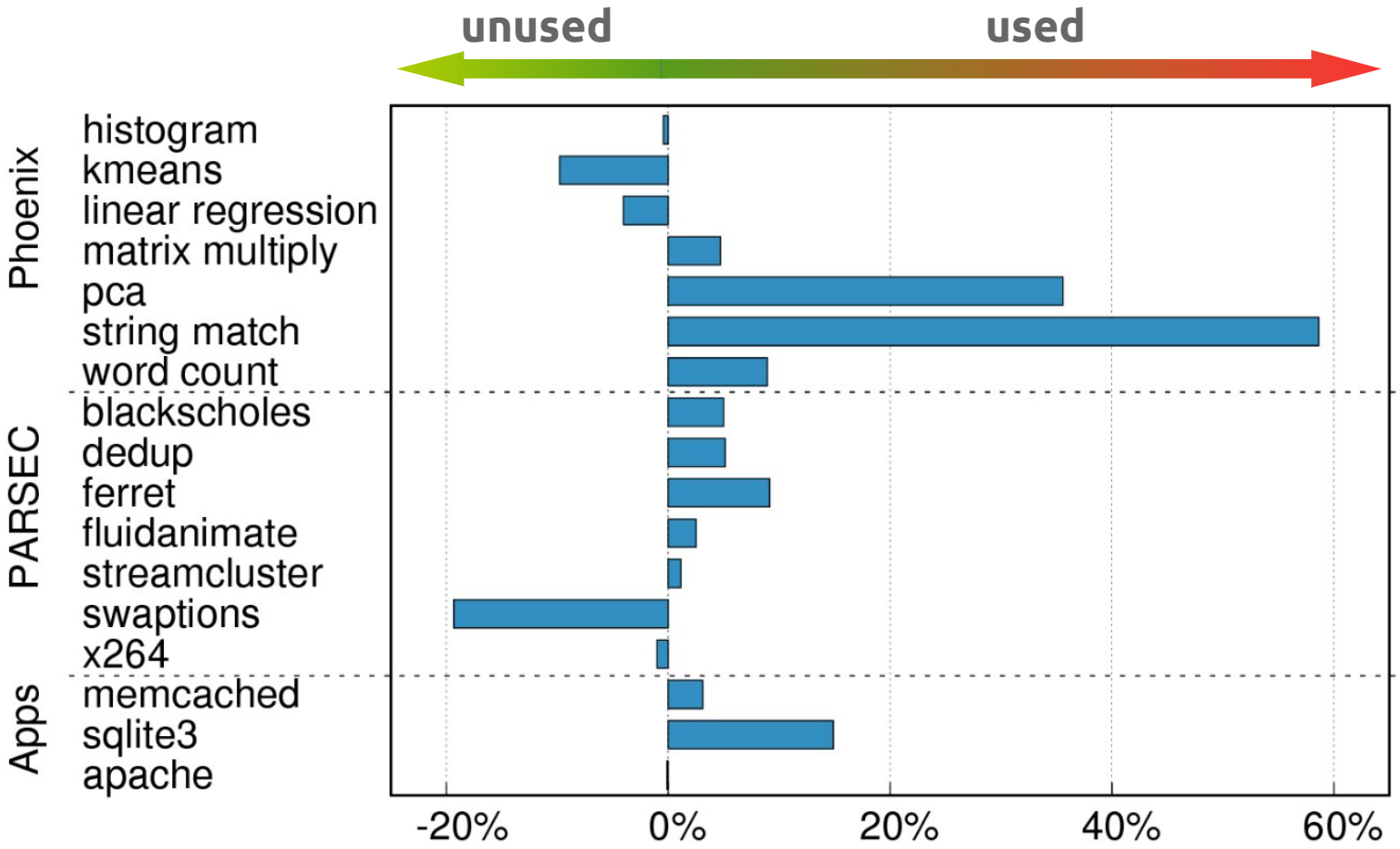
AVX as free resource?

AVX is not actively used?



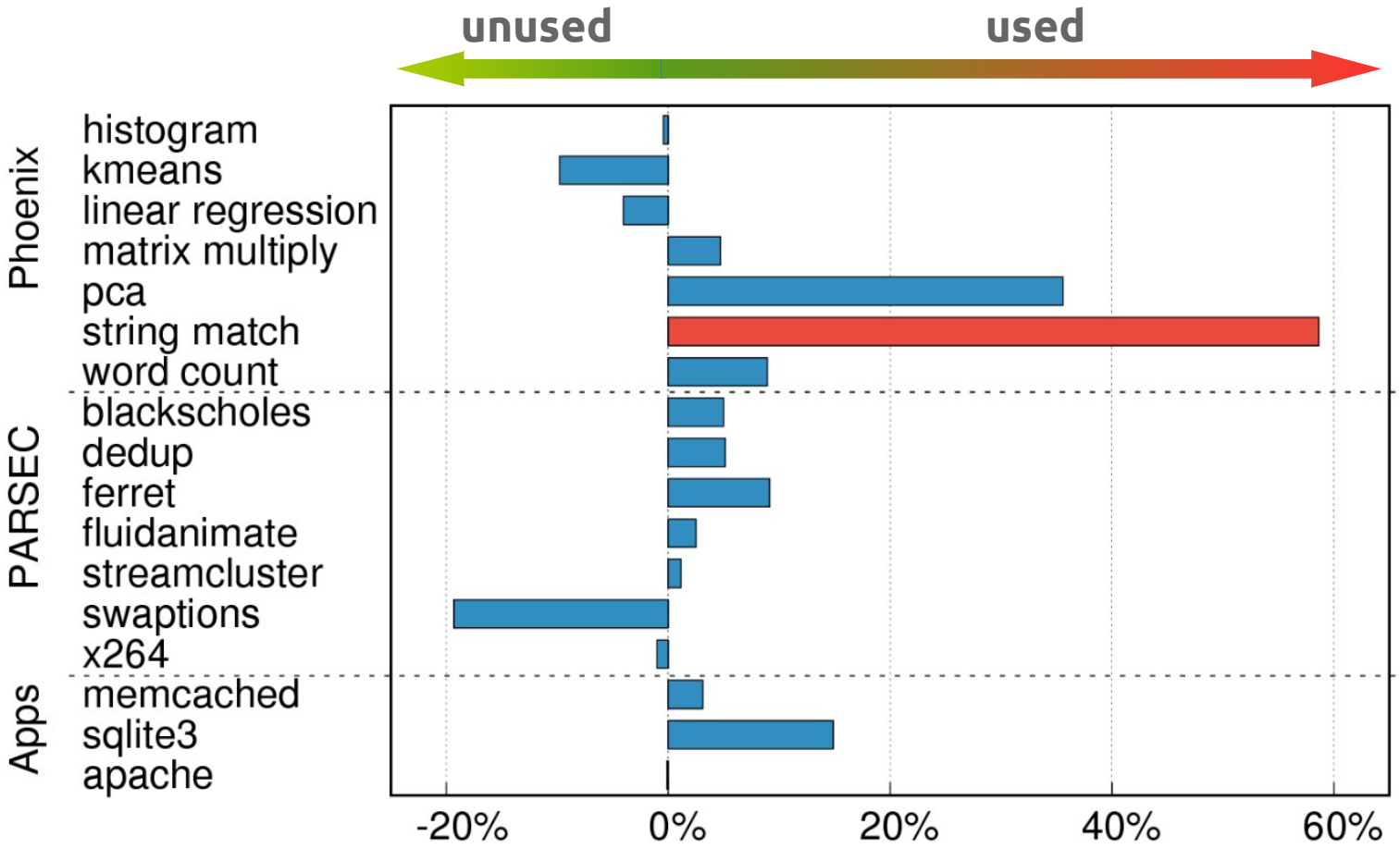
AVX as free resource?

AVX is not actively used?



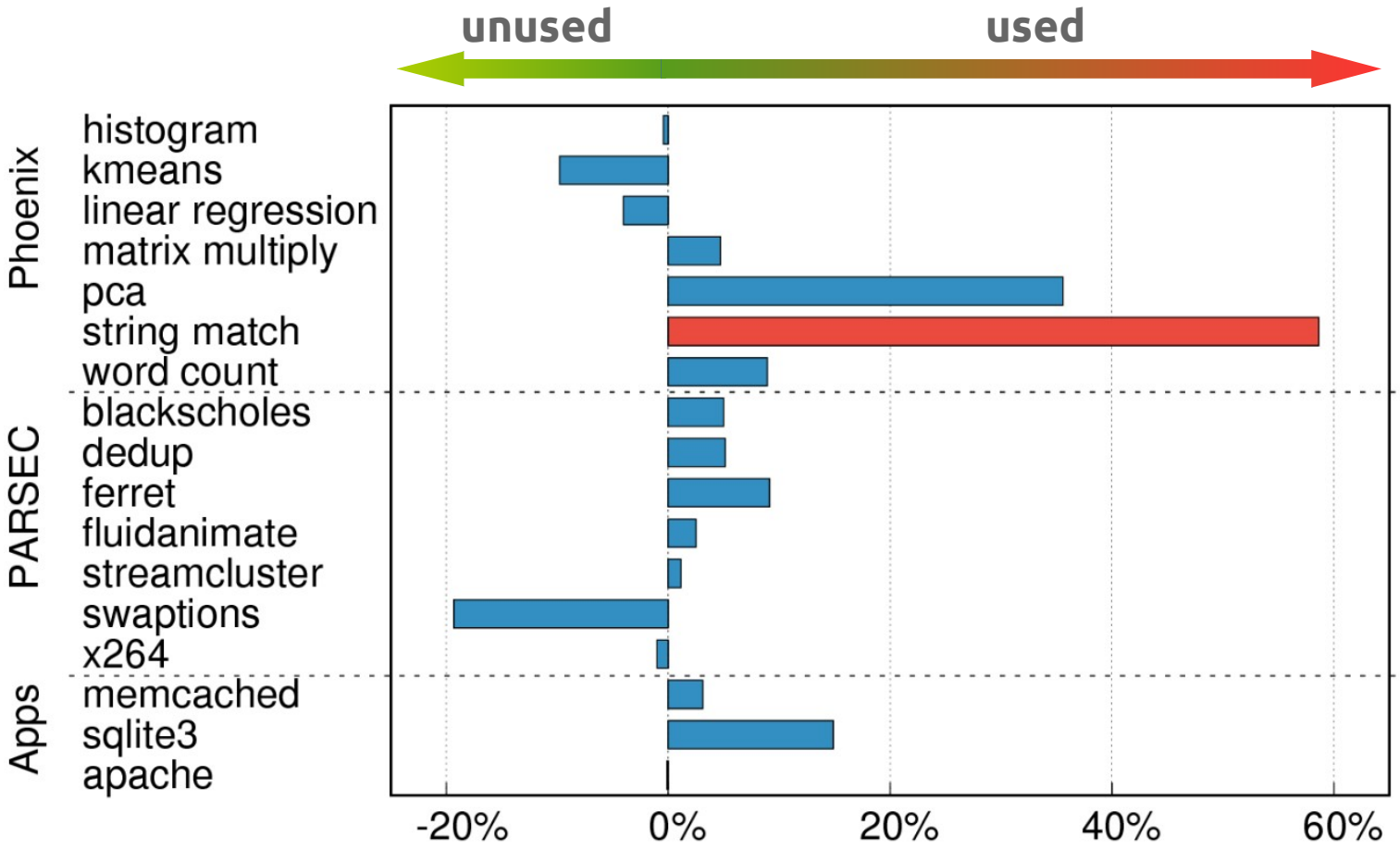
AVX as free resource?

AVX is not actively used?



AVX as free resource?

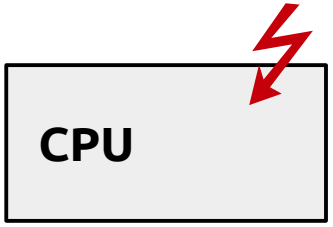
AVX is not actively used? → **Reuse for fault tolerance!**



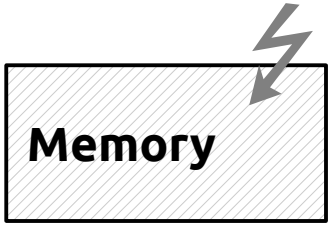
- ☑ Motivation
- ☑ Intel AVX
- ☐ **Design**
- ☐ Evaluation
- ☐ Discussion

Fault Model

- Protect against **transient faults in CPU**
  - corruptions in CPU registers
  - miscomputations in CPU execution units



- Memory is **protected by other means**
  - DRAM protected by ECC
  - CPU caches protected by ECC and parity



Elzar by Example

Native

```
x  = load a
z  = add x, 1
```

Elzar by Example

Native

```
x = load a
z = add x, 1
```

TMR

```
x  = load a
z  = add x,  1
z2 = add x2, 1
z3 = add x3, 1
majority(z, z2, z3)
```

Elzar by Example

Native

```
x  = load a
z  = add x, 1
```

TMR

```
x  = load a
z  = add x,  1
z2 = add x2, 1
z3 = add x3, 1
majority(z, z2, z3)
```

Elzar

```
x  = avx-load a
z  = avx-add x,  1

avx-check(z)
```

Elzar by Example

Native

```
x  = load a
z  = add x, 1
```

TMR

```
x  = load a
z  = add x,  1
z2 = add x2, 1
z3 = add x3, 1
majority(z, z2, z3)
```

Elzar

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```

Common instructions introduce lower overhead ☺

Elzar by Example

Native

```
x = load a
z = add x, 1
```

TMR

```
x  = load a
z  = add x,  1
z2 = add x2, 1
z3 = add x3, 1
majority(z, z2, z3)
```

Elzar

```
x = avx-load a
z = avx-add x,  1
avx-check(z)
```

Bottleneck 1: Memory accesses require wrappers ☹

Elzar by Example

Native

```
x = load a
z = add x, 1
```

TMR

```
x  = load a
z  = add x,  1
z2 = add x2, 1
z3 = add x3, 1
majority(z, z2, z3)
```

Elzar

```
x = avx-load a
z = avx-add x,  1
avx-check(z)
```

Bottleneck 2: Checks are expensive ☹

Bottleneck 1: Memory accesses

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```

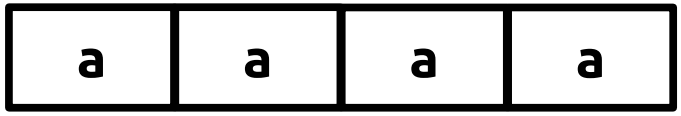
Bottleneck 1: Memory accesses

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```

No such thing as avx-load! ☹

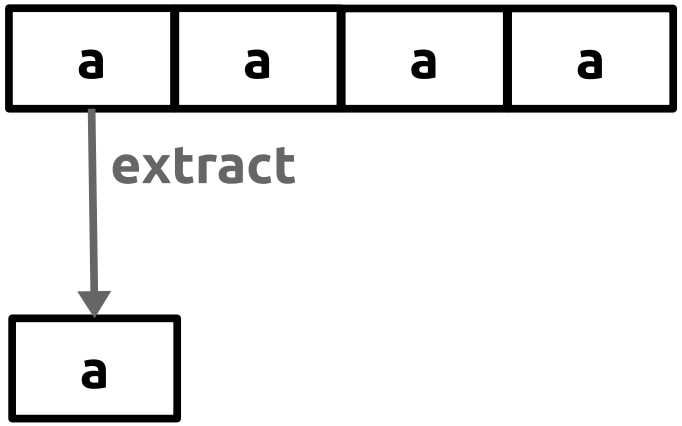
Bottleneck 1: Memory accesses

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



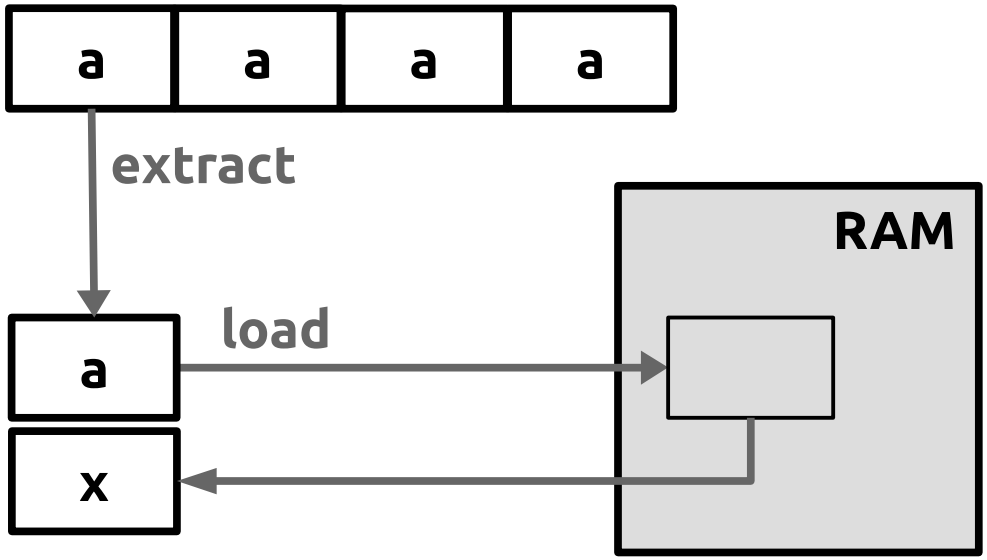
Bottleneck 1: Memory accesses

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



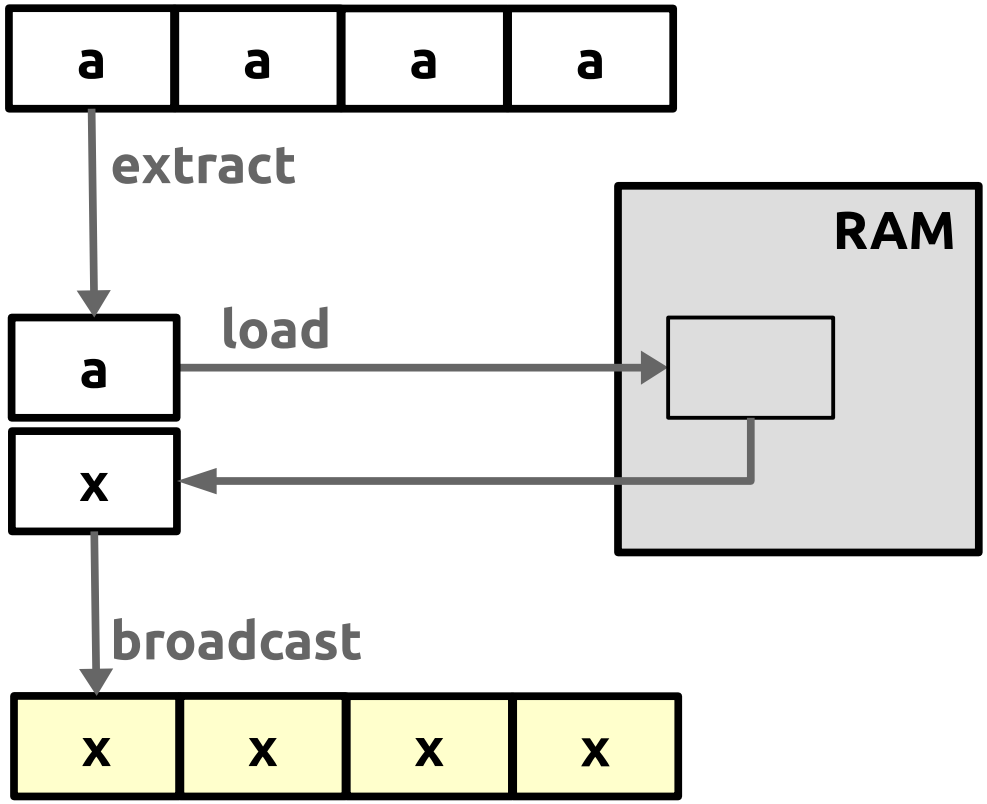
Bottleneck 1: Memory accesses

```
x   = avx-load a
z   = avx-add x,  1
avx-check(z)
```



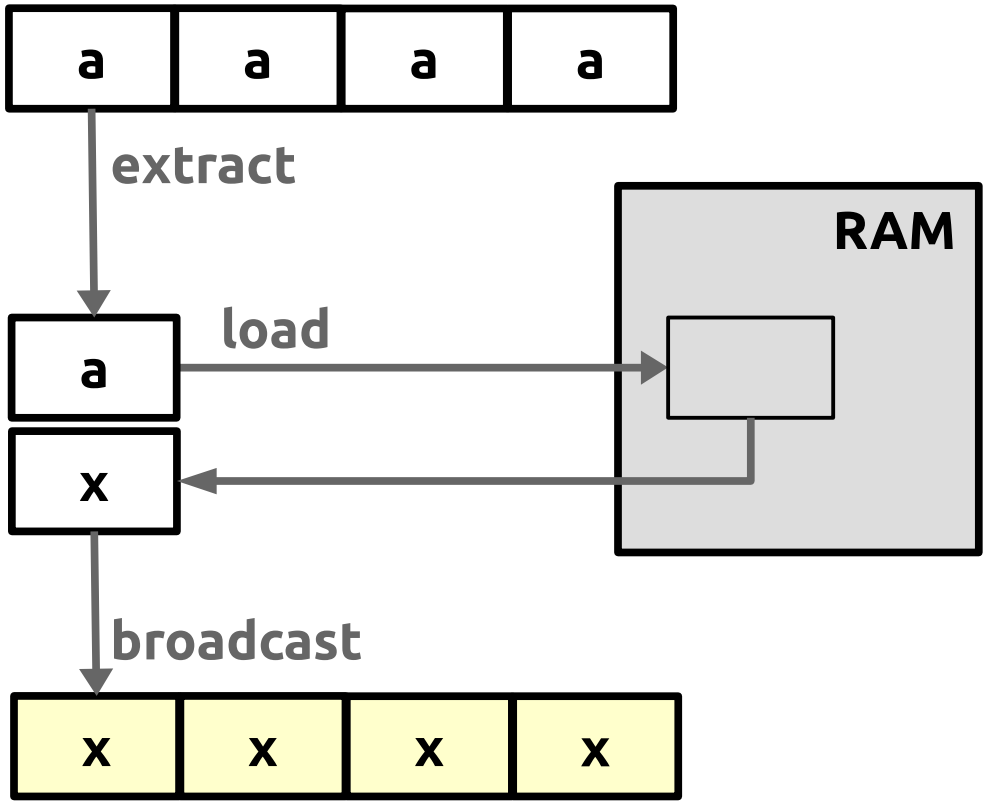
Bottleneck 1: Memory accesses

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



Bottleneck 1: Memory accesses

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



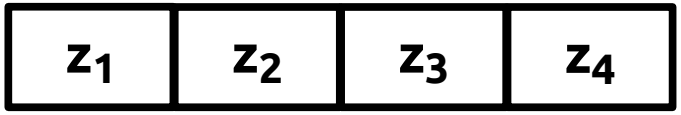
Memory accesses require wrappers ☹

Bottleneck 2: Checks

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```

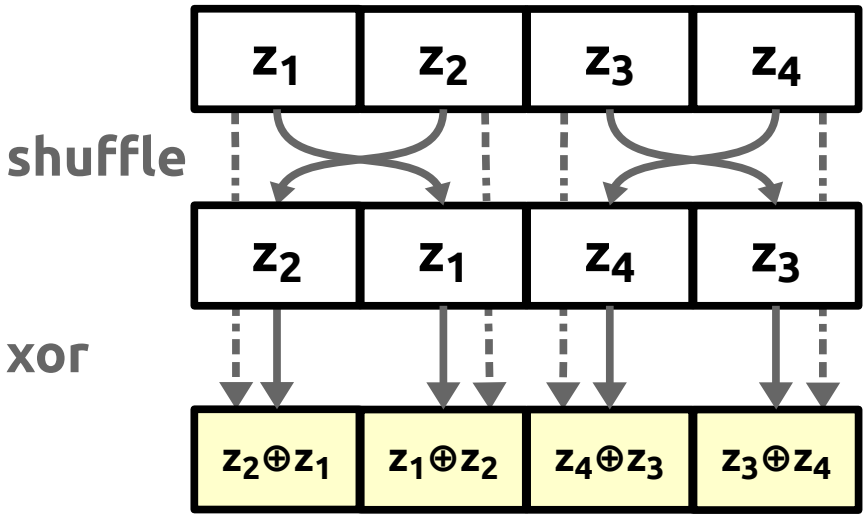
Bottleneck 2: Checks

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



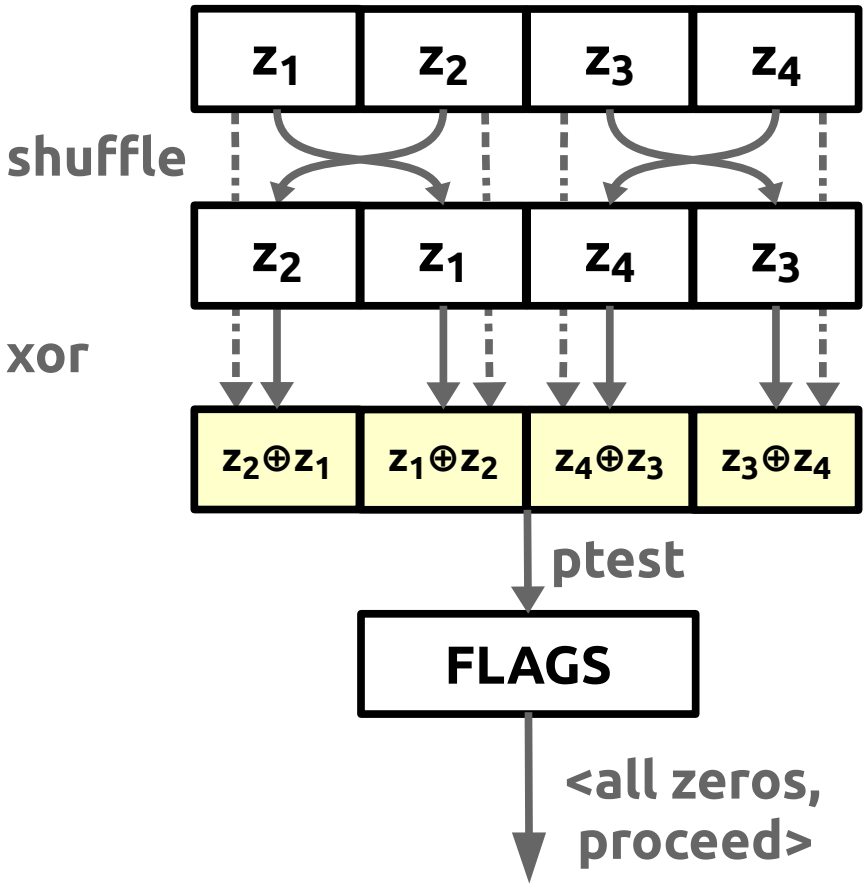
Bottleneck 2: Checks

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



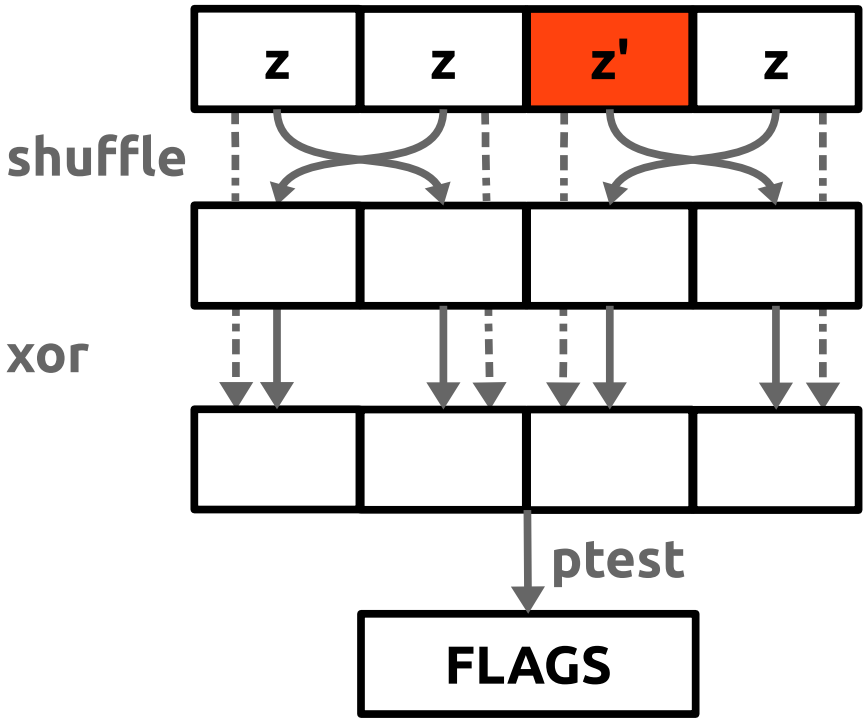
Bottleneck 2: Checks

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



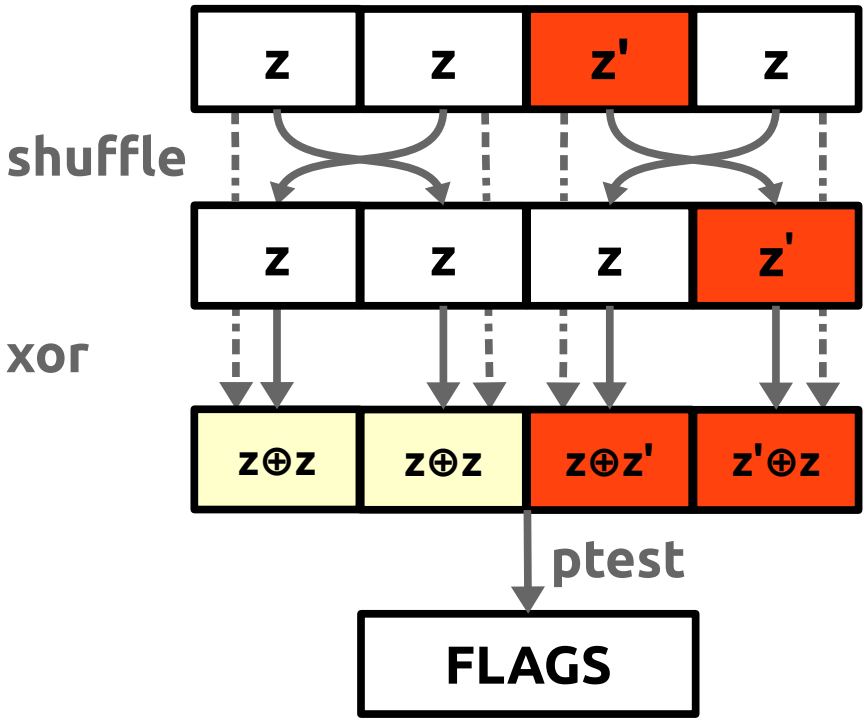
Bottleneck 2: Checks

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



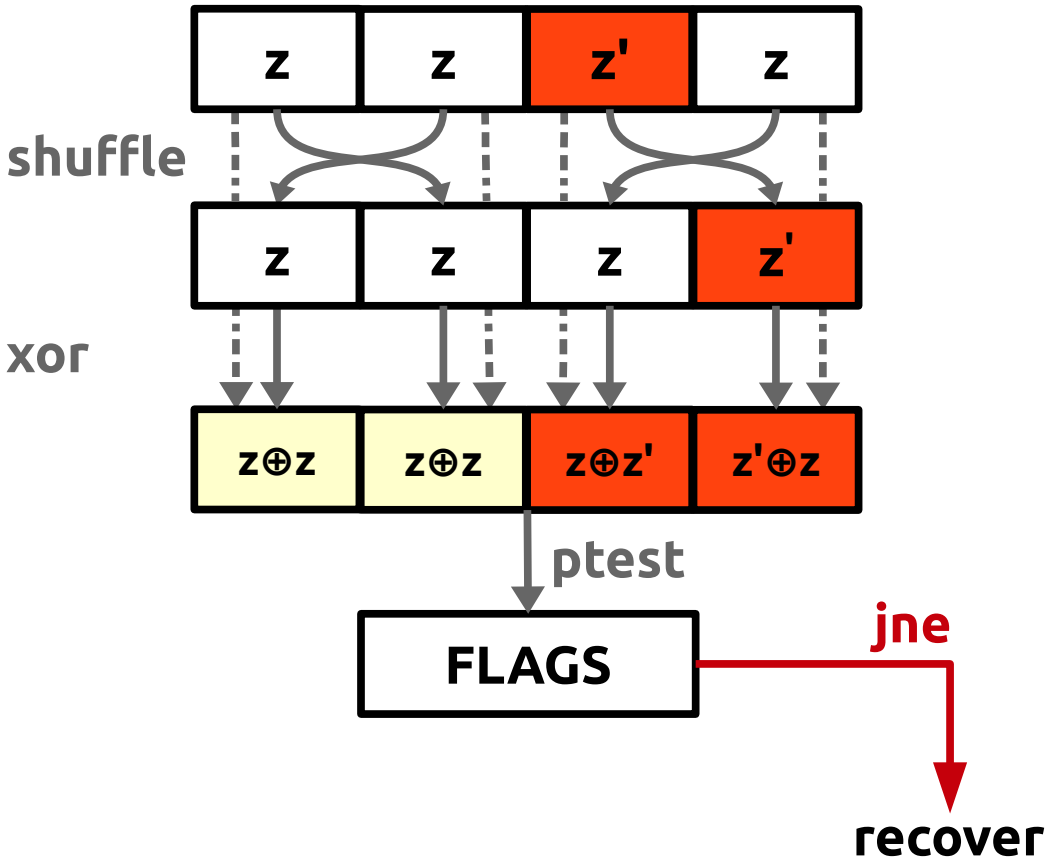
Bottleneck 2: Checks

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



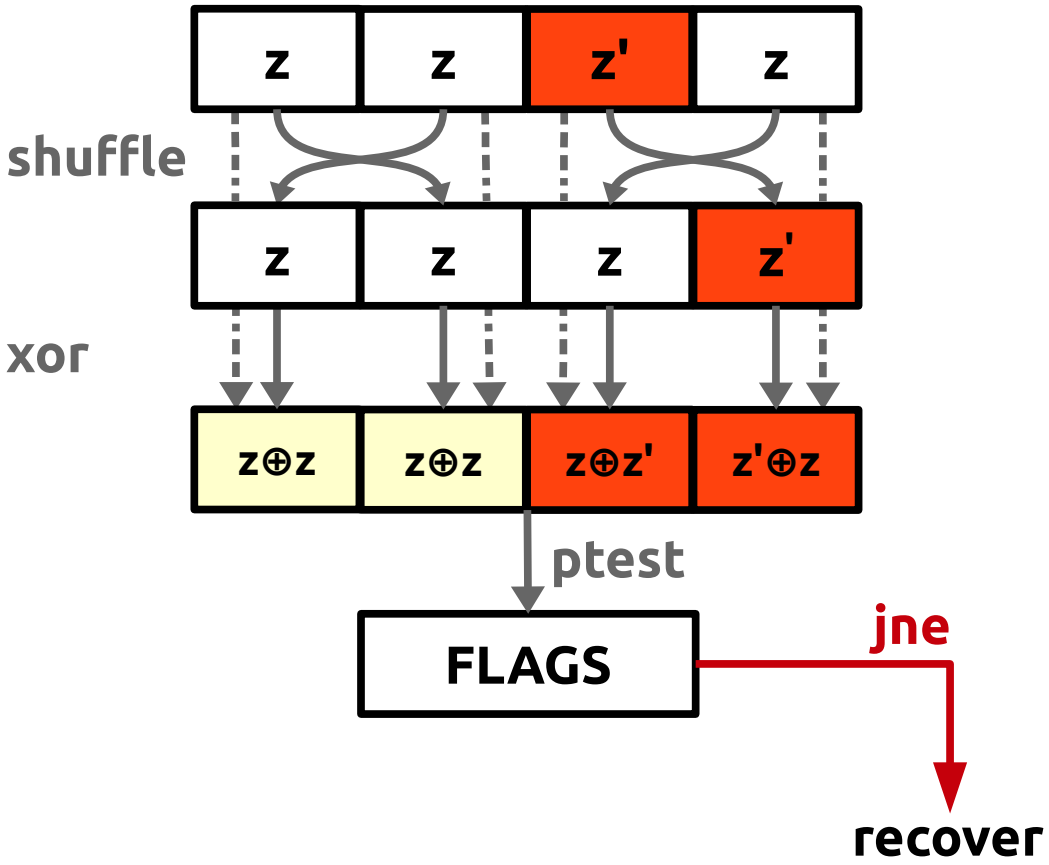
Bottleneck 2: Checks

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```



Bottleneck 2: Checks

```
x  = avx-load a
z  = avx-add x,  1
avx-check(z)
```

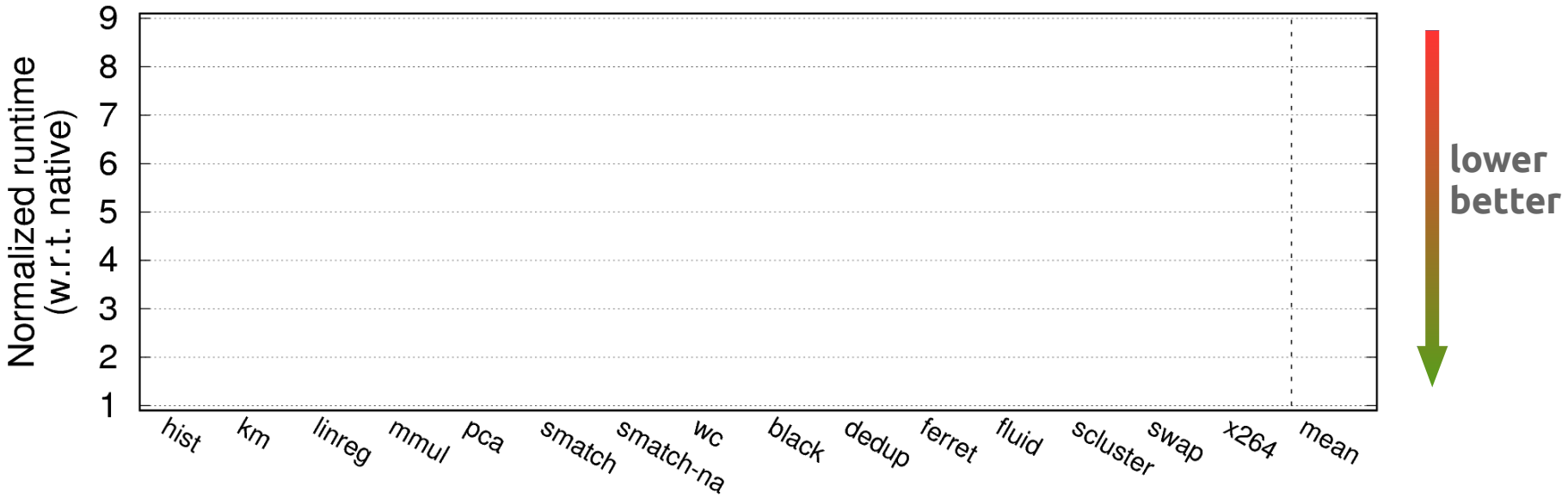


Checks introduce additional **55%** overhead ☹

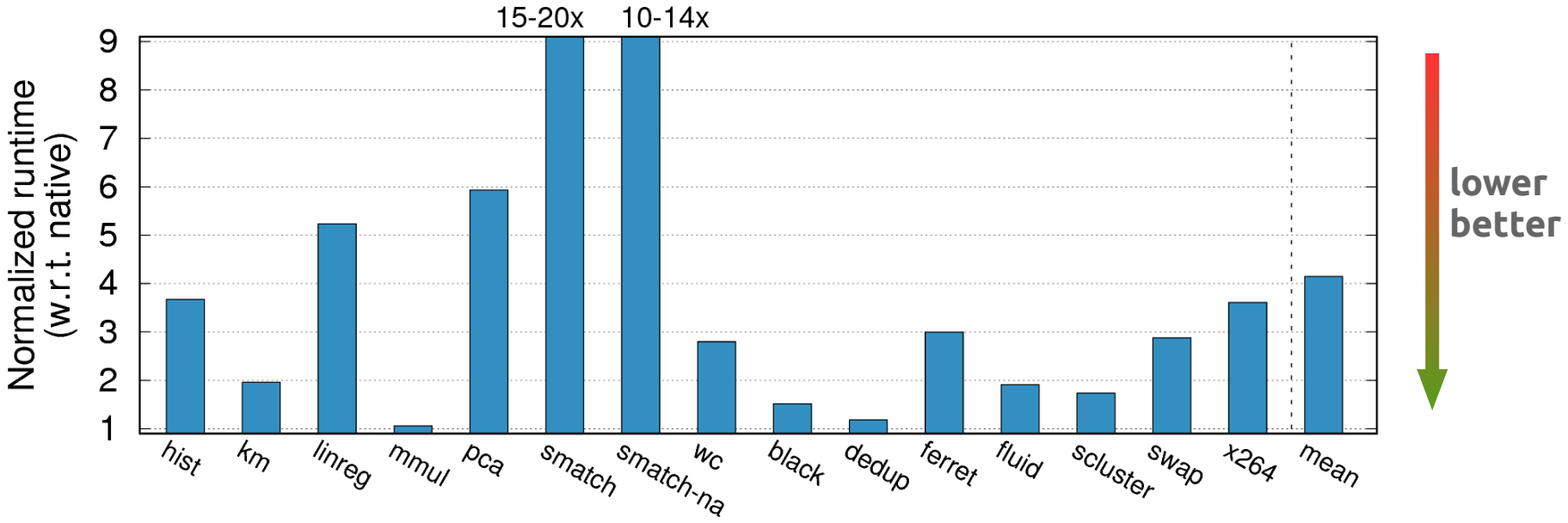
- ☑ Motivation
- ☑ Intel AVX
- ☑ Design
- ☐ **Evaluation**
- ☐ Discussion

Performance overheads

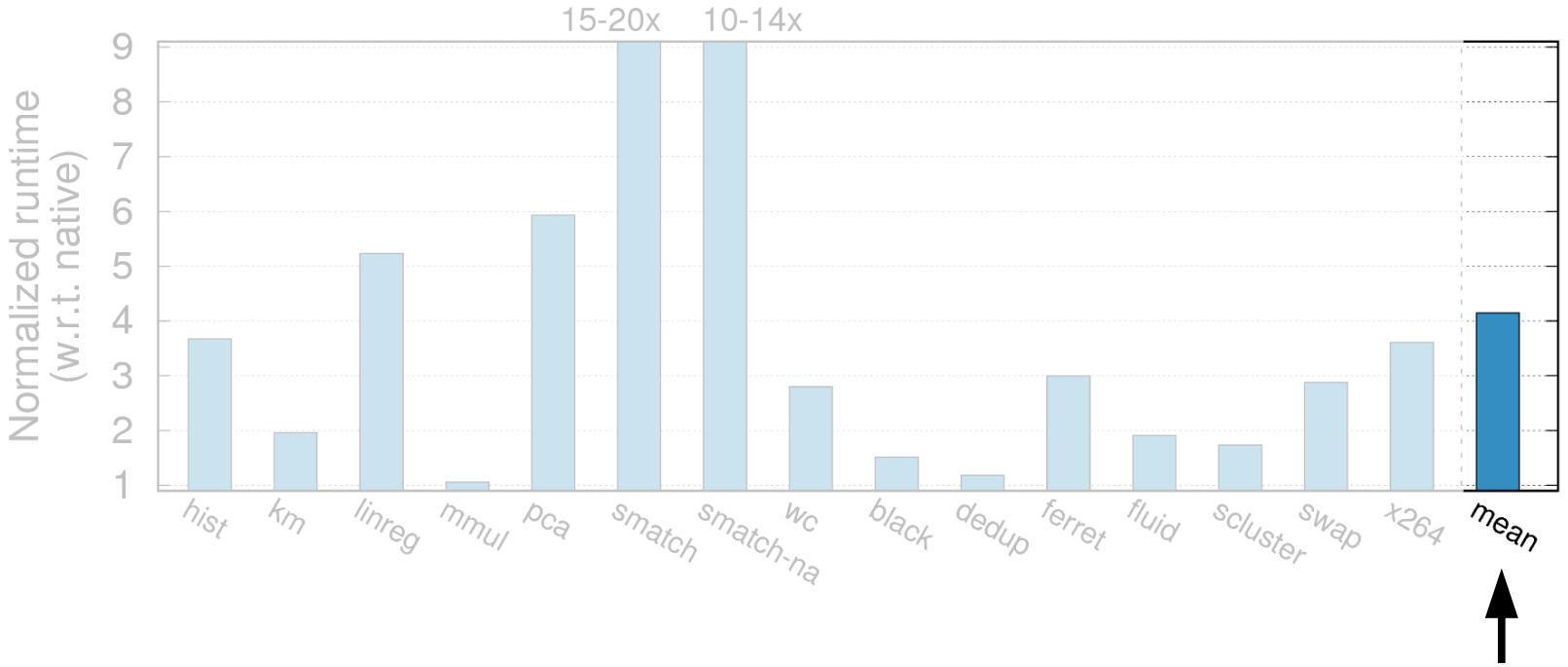
Performance overheads



Performance overheads

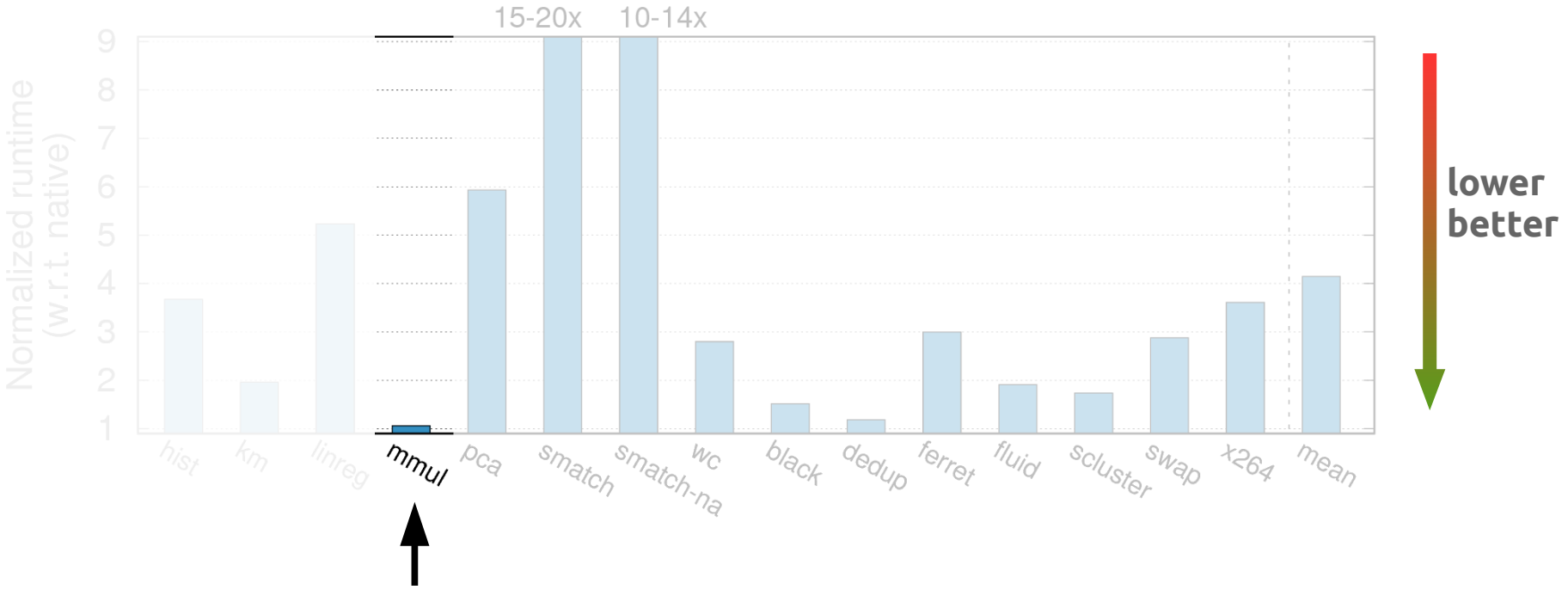


Performance overheads



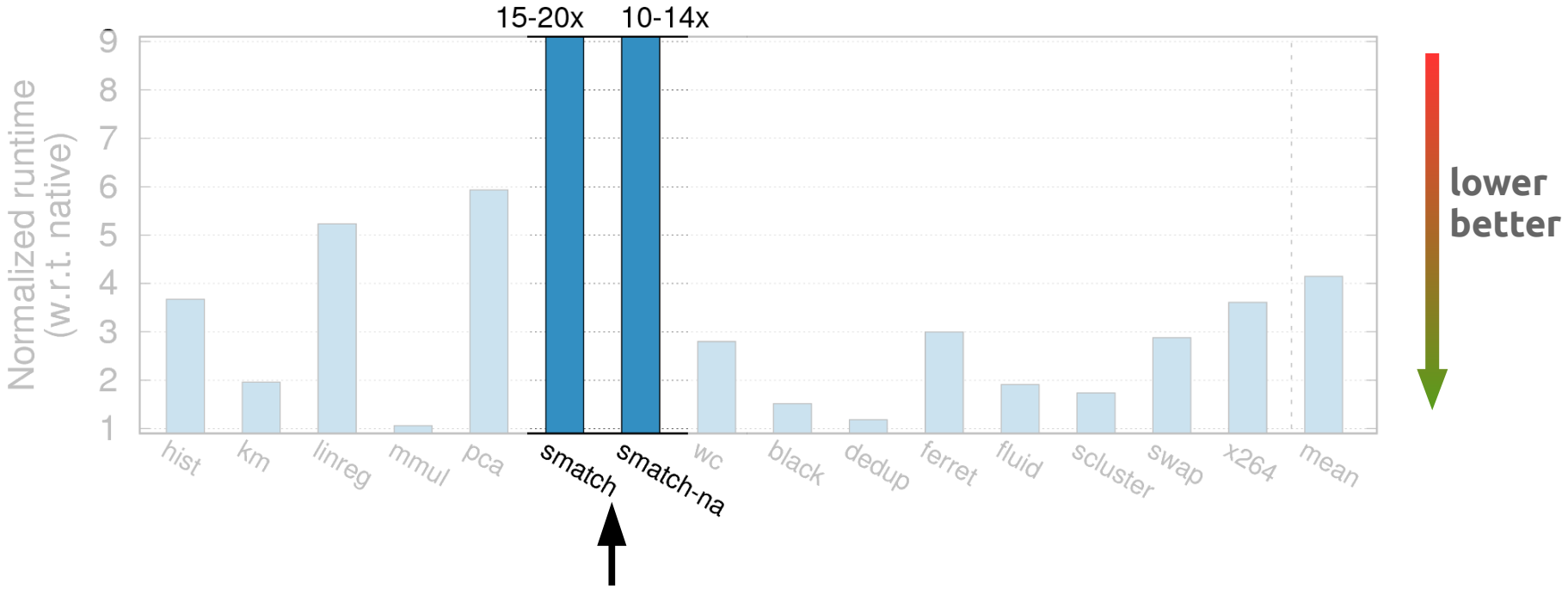
Average performance overhead is **4×**

Performance overheads



Amortized by very poor memory access pattern

Performance overheads

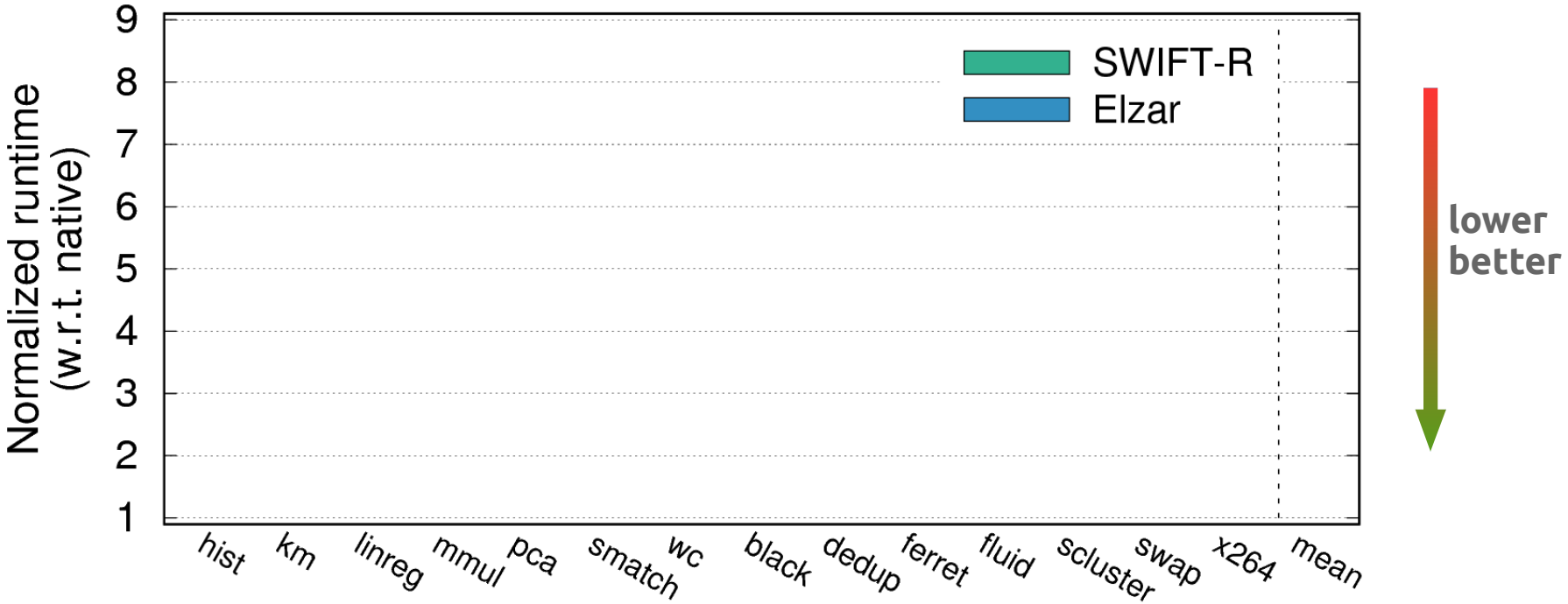


(1) Native benefits from AVX vectorization

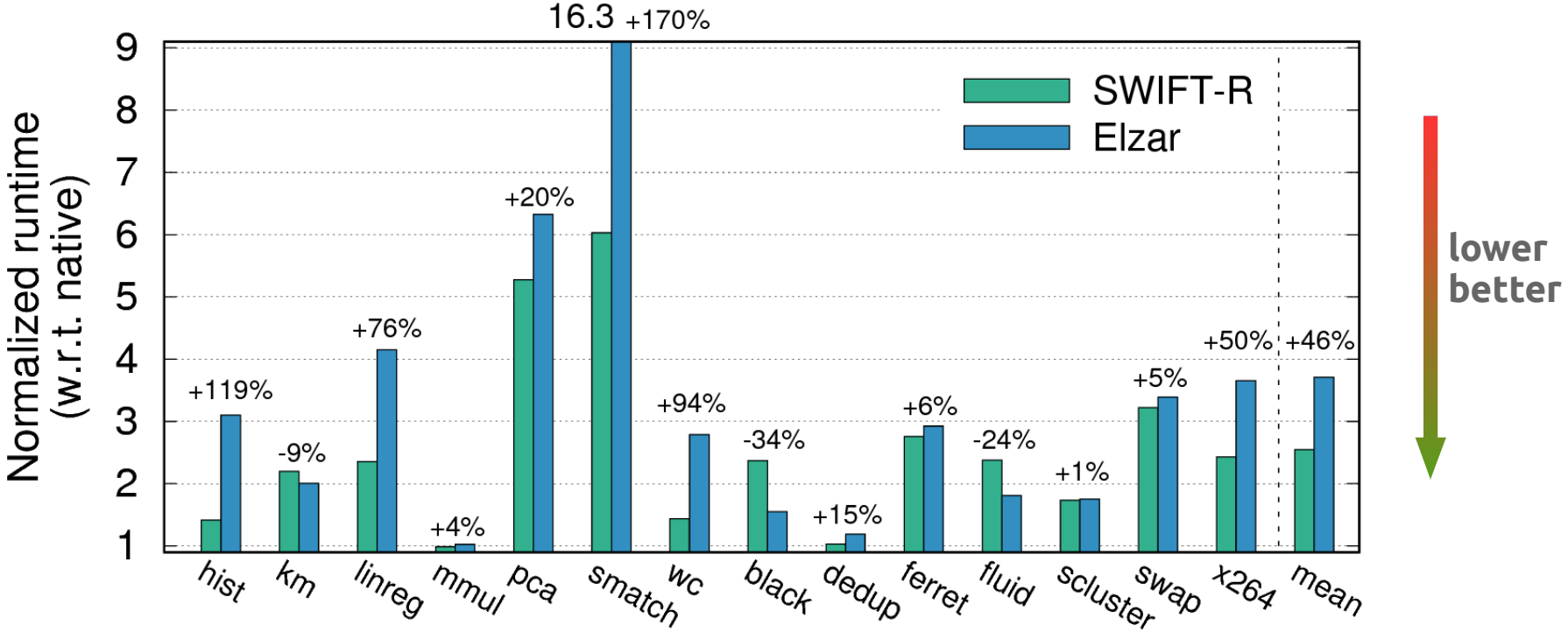
(2) Most of time spent in mem-intensive `bzero()`

Comparison with SWIFT-R (state-of-the-art TMR approach)

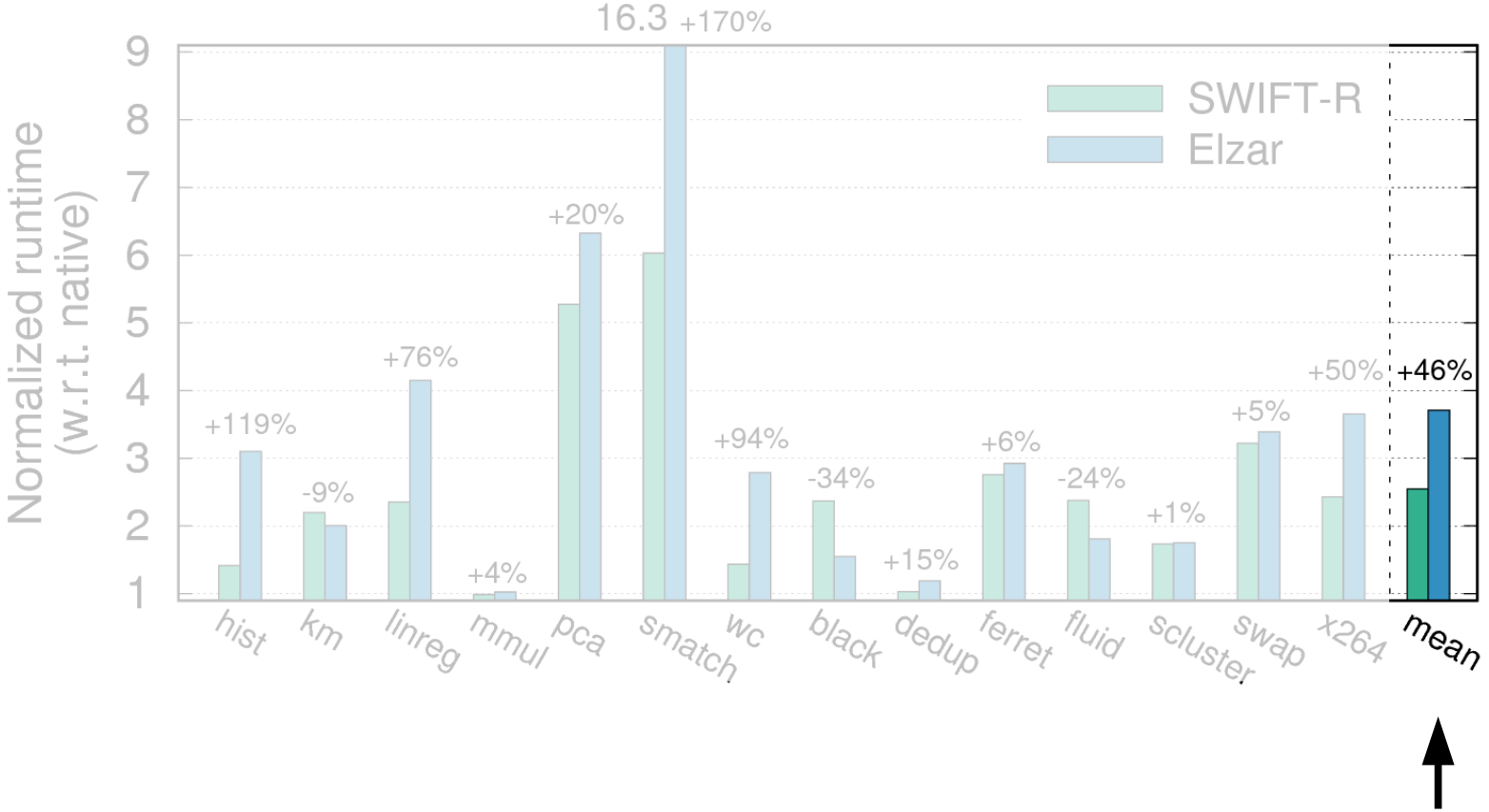
Comparison with SWIFT-R (state-of-the-art TMR approach)



Comparison with SWIFT-R (state-of-the-art TMR approach)

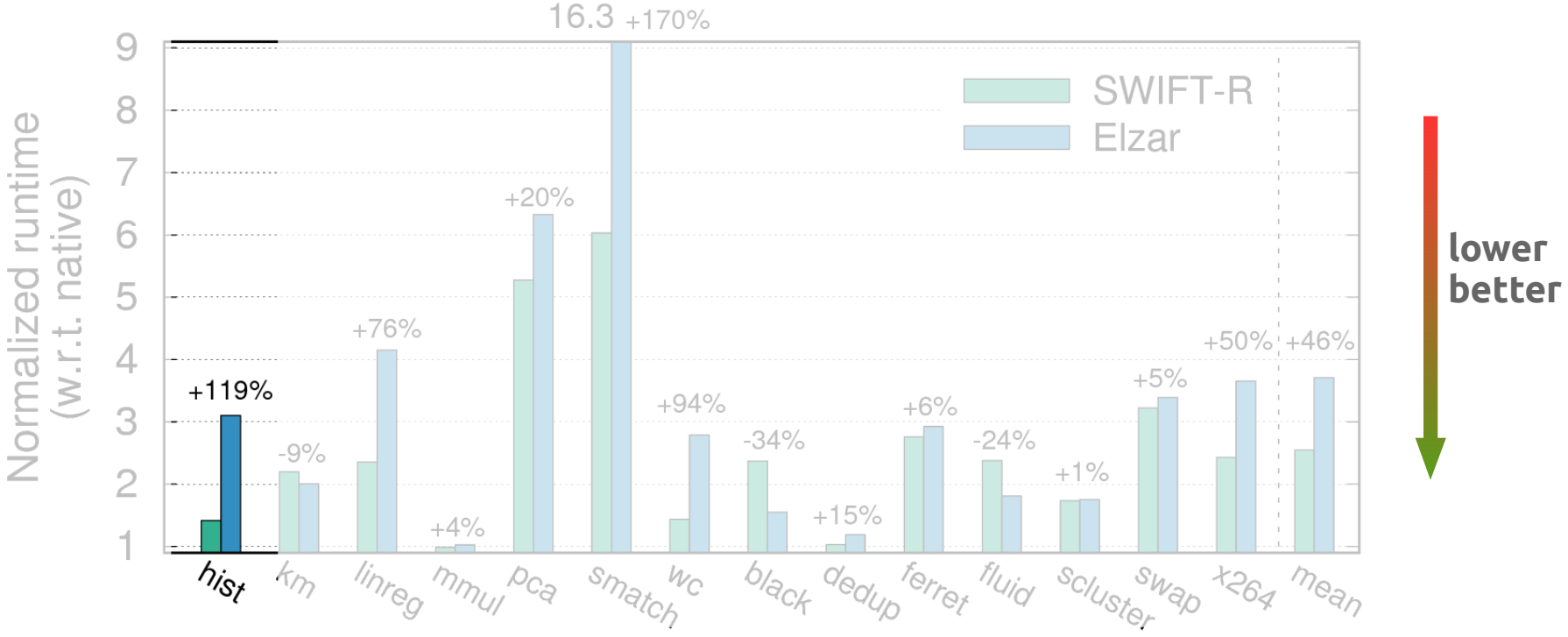


Comparison with SWIFT-R (state-of-the-art TMR approach)



Elzar performs **46%** worse than SWIFT-R on average ☹

Comparison with SWIFT-R (state-of-the-art TMR approach)



Dominated by memory accesses, Elzar inserts many wrappers ☹

Comparison with SWIFT-R (state-of-the-art TMR approach)

Elzar produces **3x less** instructions than SWIFT-R ☺

- ☑ Motivation
- ☑ Intel AVX
- ☑ Design
- ☑ Evaluation
- ☐ **Discussion**

Bottlenecks and Proposed Solution

Problem

AVX lacks certain instructions

- Need wrappers for memory accesses
- Need `shuffle-xor-ptest` for checks

Bottlenecks and Proposed Solution

Problem — **AVX** lacks certain instructions

- Need wrappers for memory accesses
- Need `shuffle-xor-ptest` for checks

Solution — Offload to **FPGA accelerator**

- Intel's Xeon-FPGA pairing

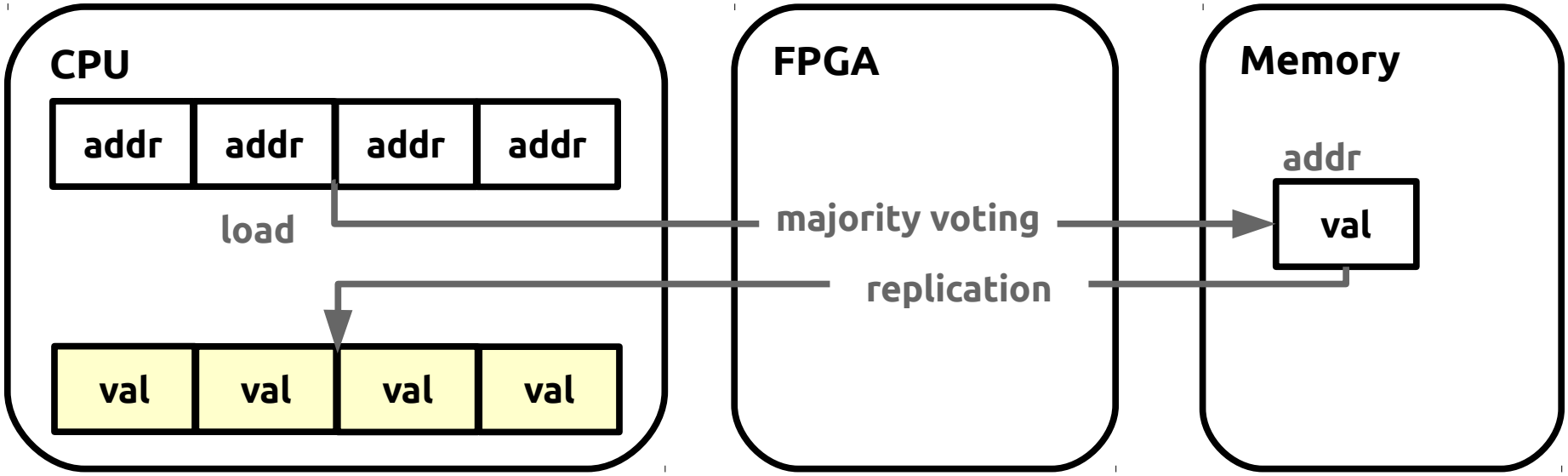
Bottlenecks and Proposed Solution

Problem **AVX** lacks certain instructions

- Need wrappers for memory accesses
- Need `shuffle-xor-ptest` for checks

Solution Offload to **FPGA accelerator**

- Intel's Xeon-FPGA pairing



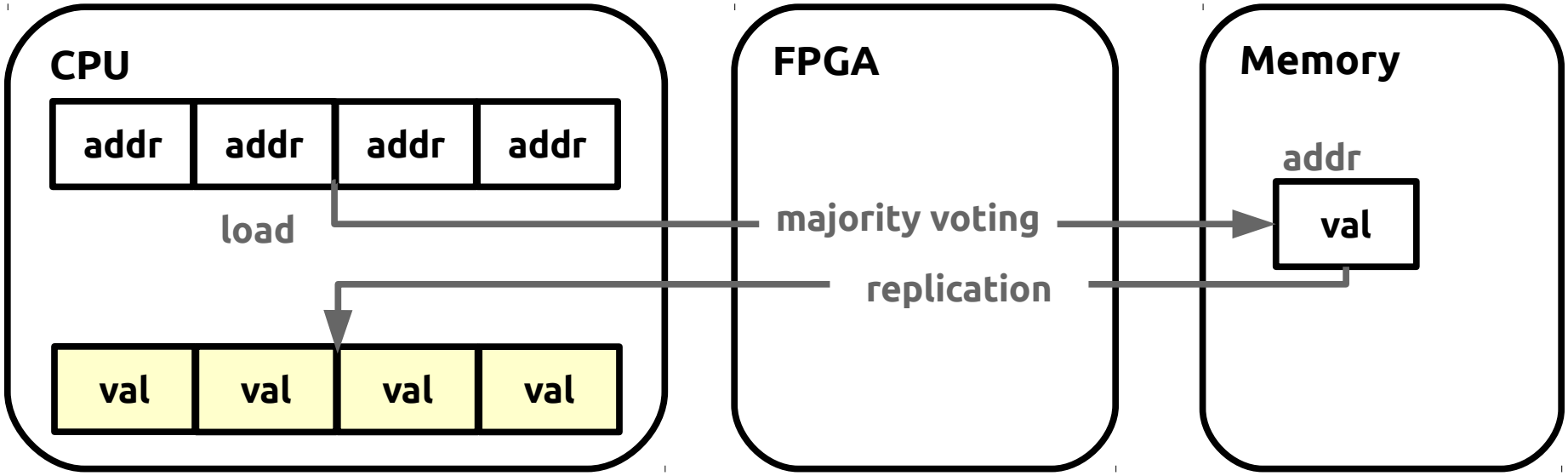
Bottlenecks and Proposed Solution

Problem **AVX** lacks certain instructions

- Need wrappers for memory accesses
- Need `shuffle-xor-ptest` for checks

Solution Offload to **FPGA accelerator**

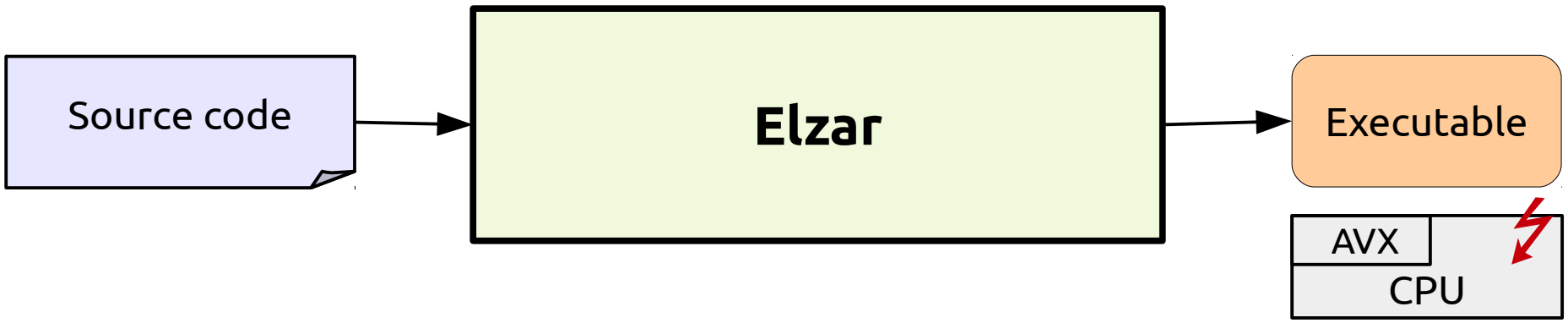
- Intel's Xeon-FPGA pairing



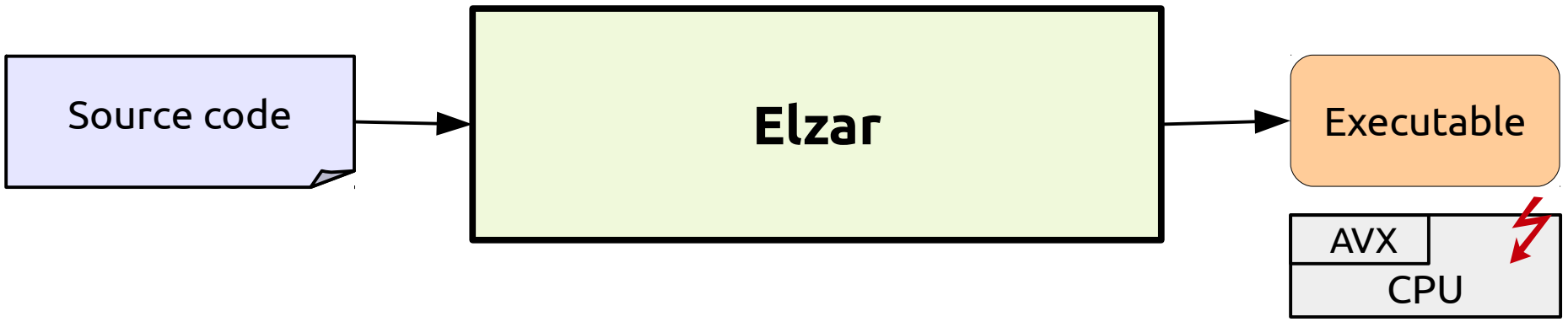
Potentially **71% better** than SWIFT-R

Conclusion

Conclusion

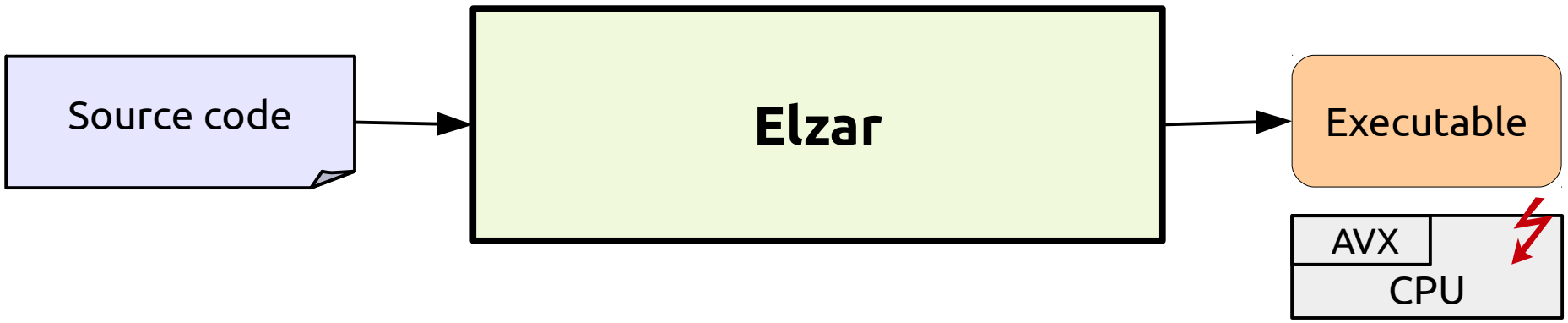


Conclusion



Implementation **Intel AVX** for triple modular redundancy

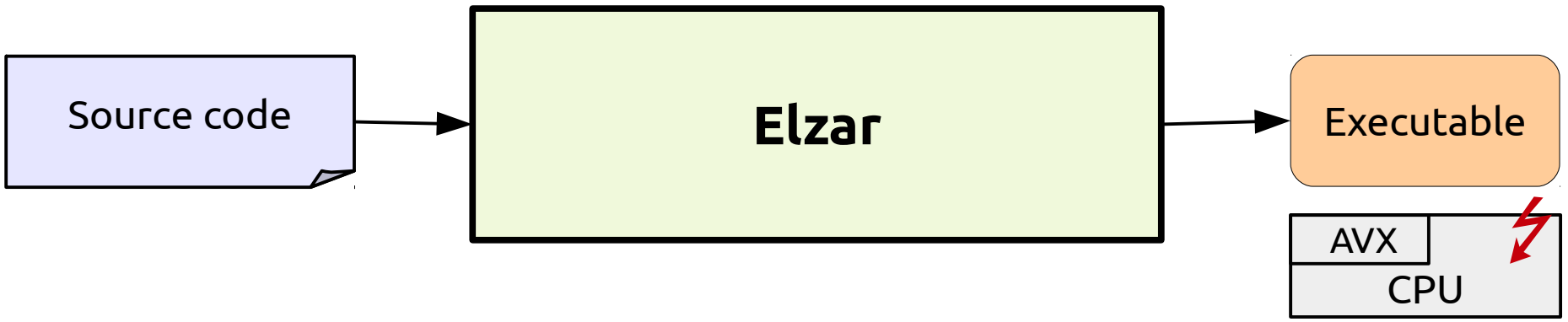
Conclusion



Implementation **Intel AVX** for triple modular redundancy

Hypothesis **Less** instructions → **less** perf overhead

Conclusion

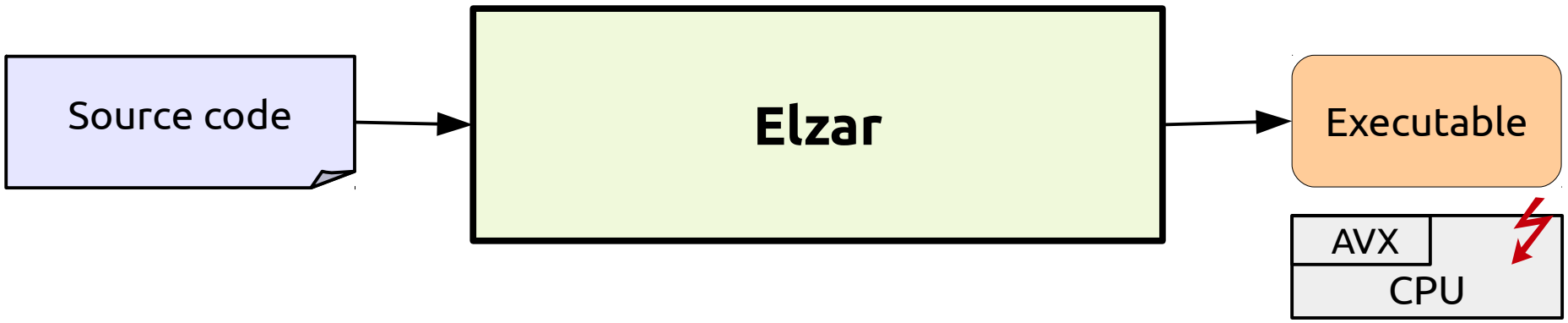


Implementation **Intel AVX** for triple modular redundancy

Hypothesis **Less** instructions → **less** perf overhead

Outcome **46%** worse than SWIFT-R

Conclusion



Implementation	**Intel AVX** for triple modular redundancy
Hypothesis	**Less** instructions → **less** perf overhead
Outcome	**46%** worse than SWIFT-R
Discussion	With FPGA, **~71%** better than SWIFT-R

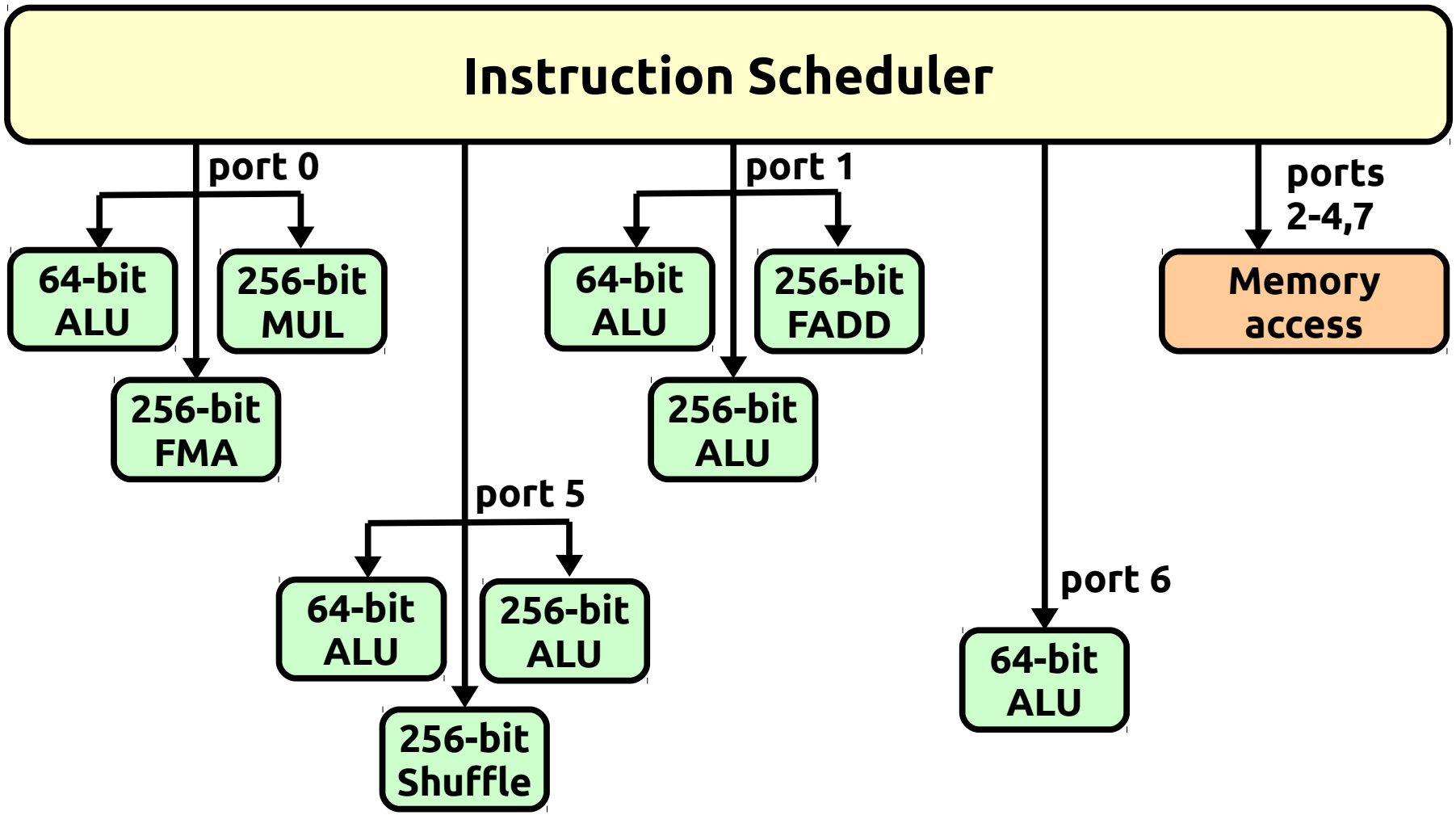
Thank you!

dmitrii.kuvaiskii@tu-dresden.de

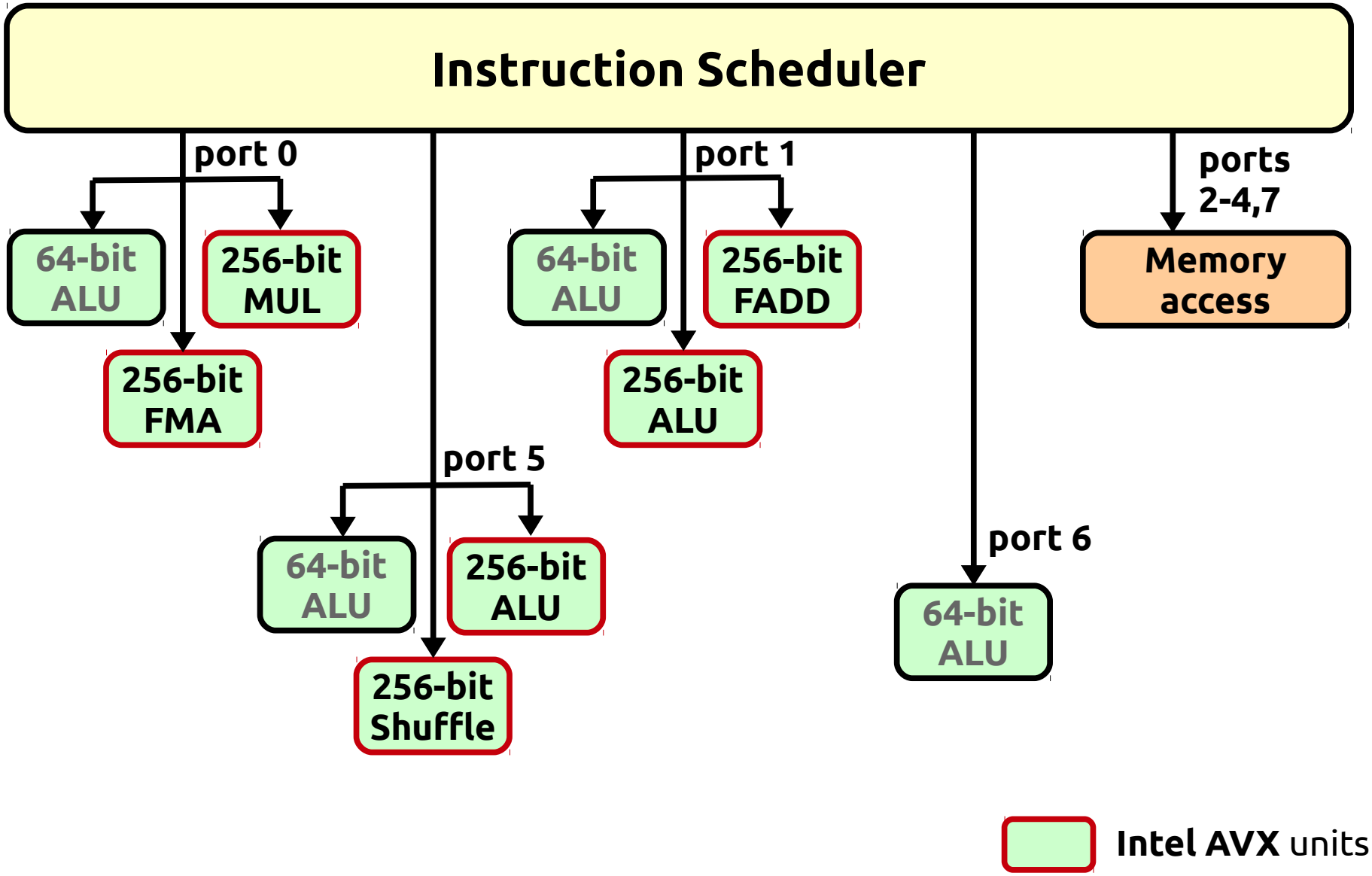
GitHub repo: https://github.com/tudinfse/elzar

backup slides

Intel Haswell CPU microarchitecture



Intel Haswell CPU microarchitecture



Elzar: Check on Branch

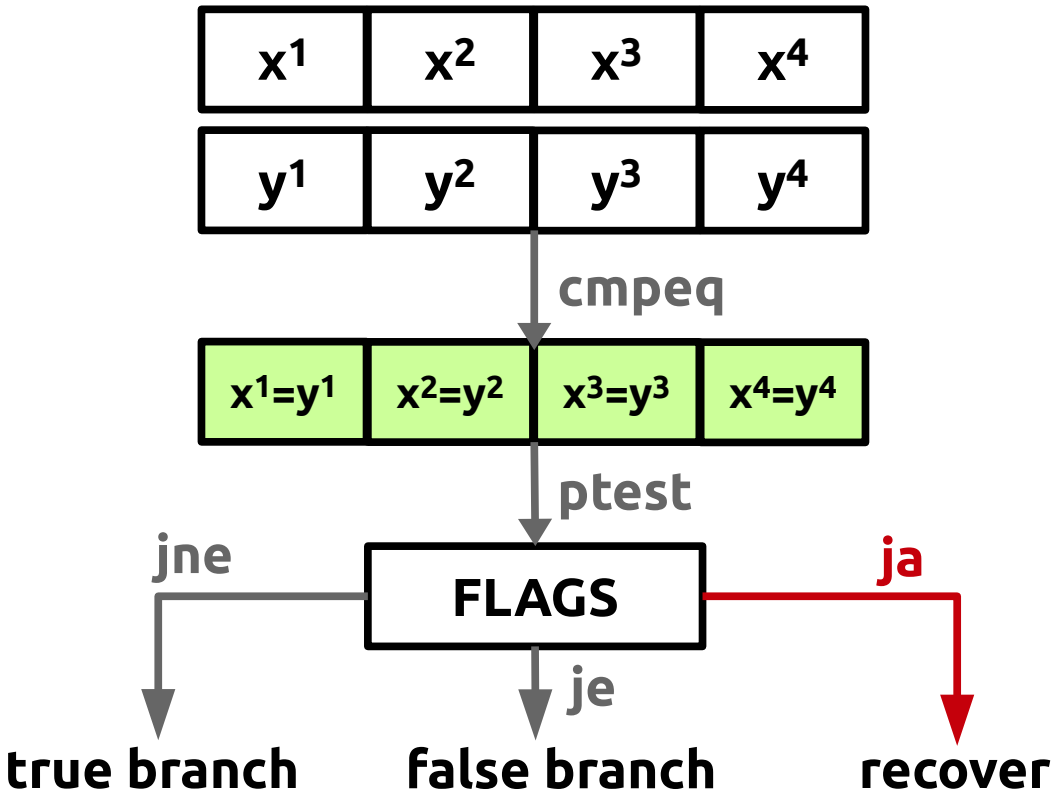
Native

```
cmp x, y
jne truebranch
```

Elzar: Check on Branch

Native

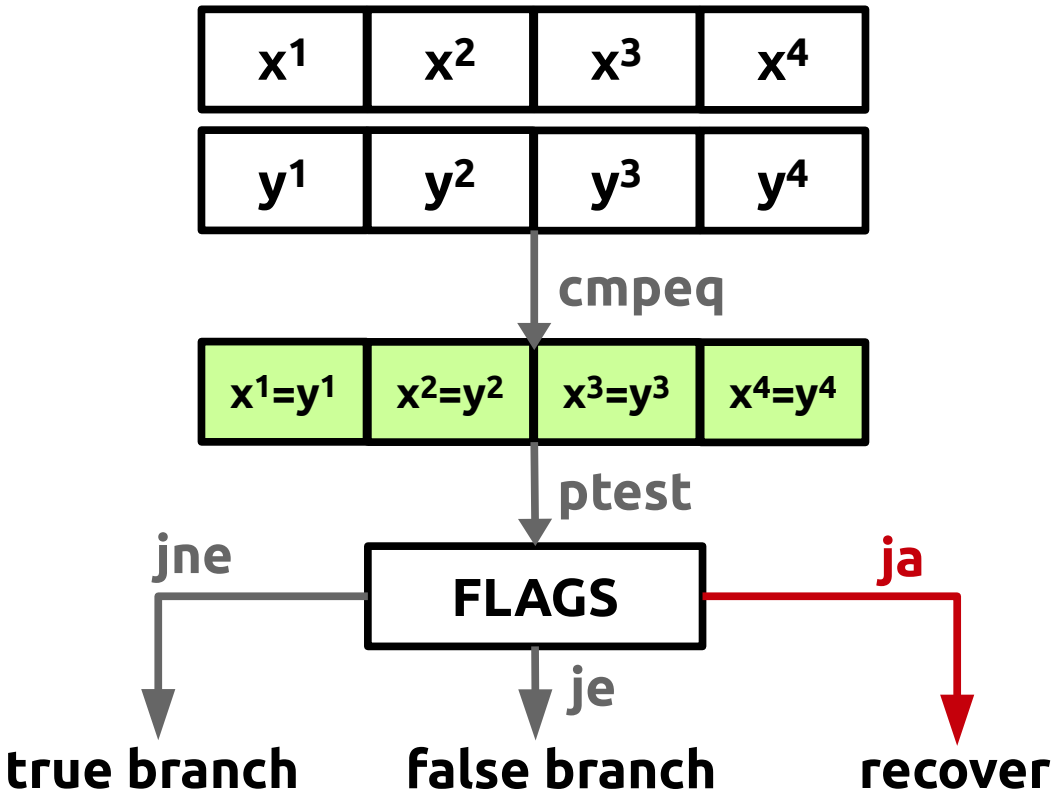
```
cmp x, y
jne truebranch
```



Elzar: Check on Branch

Native

```
cmp x, y
jne truebranch
```



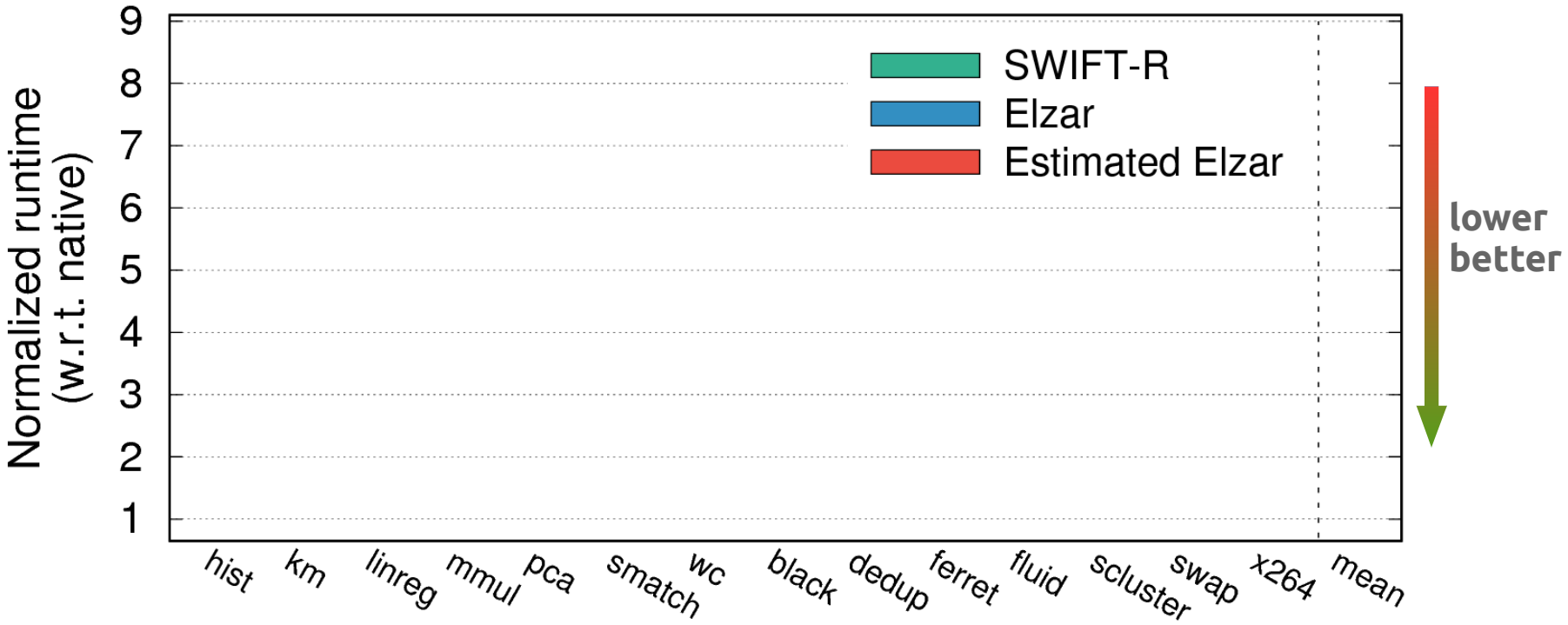
Impact **Checks on branches** introduce only **4%** overhead ☺

Main Bottlenecks

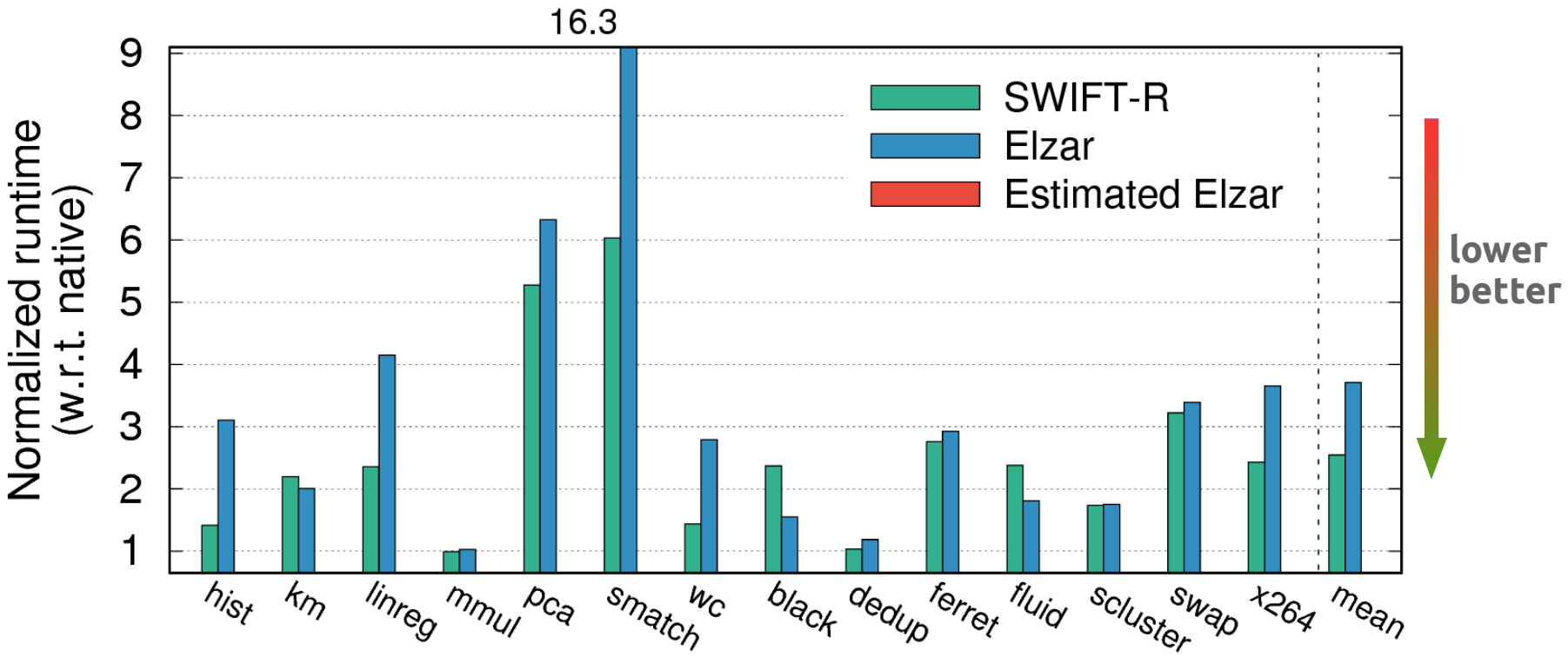
Problem **AVX** instruction set lacks certain instructions

- Loads/stores require extracting AVX-replicated address
  - Elzar creates expensive wrappers around loads/stores
  - AVX-512 introduces promising **`gather/scatter`** instructions
- AVX has only one control-flow instruction `ptest`
  - Elzar has to insert **`ptest`** for each control-flow decision
  - AVX could add **`cmp`** instructions directly affecting control flow
- AVX misses integer division, integer truncation, etc…
  - Elzar produces very ineffective code in these cases

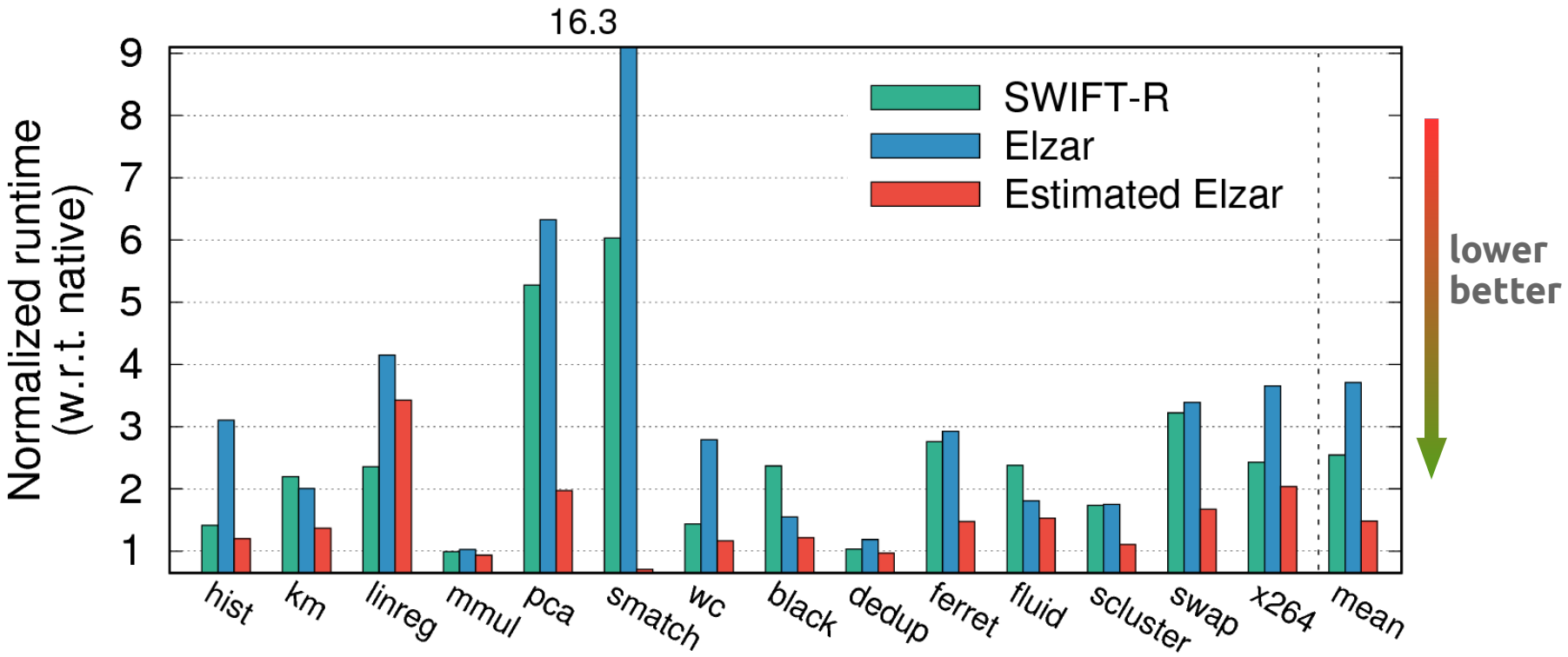
Estimation of Proposed Changes



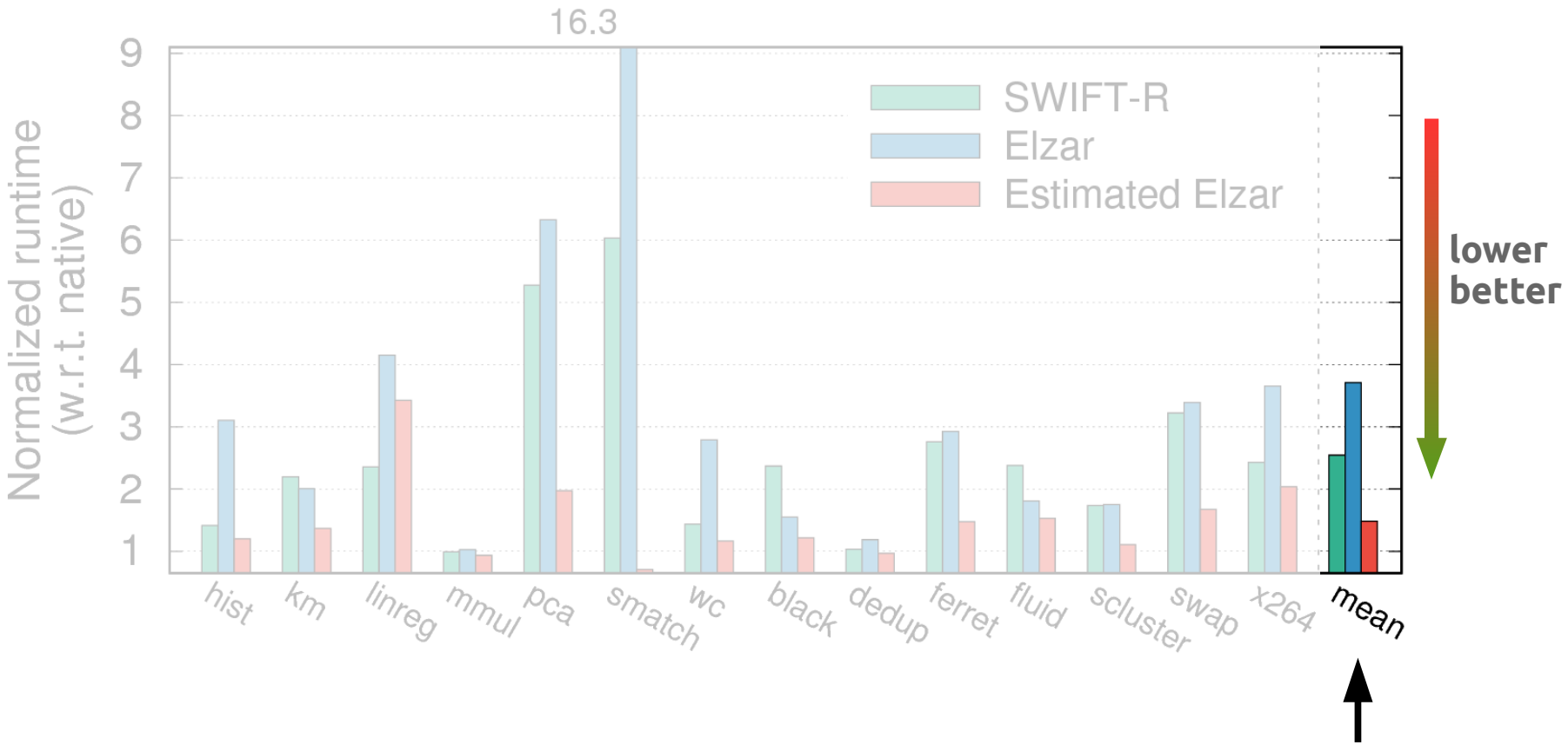
Estimation of Proposed Changes



Estimation of Proposed Changes



Estimation of Proposed Changes



Estimated average overhead would be **48%**
(**71%** improvement over SWIFT-R)