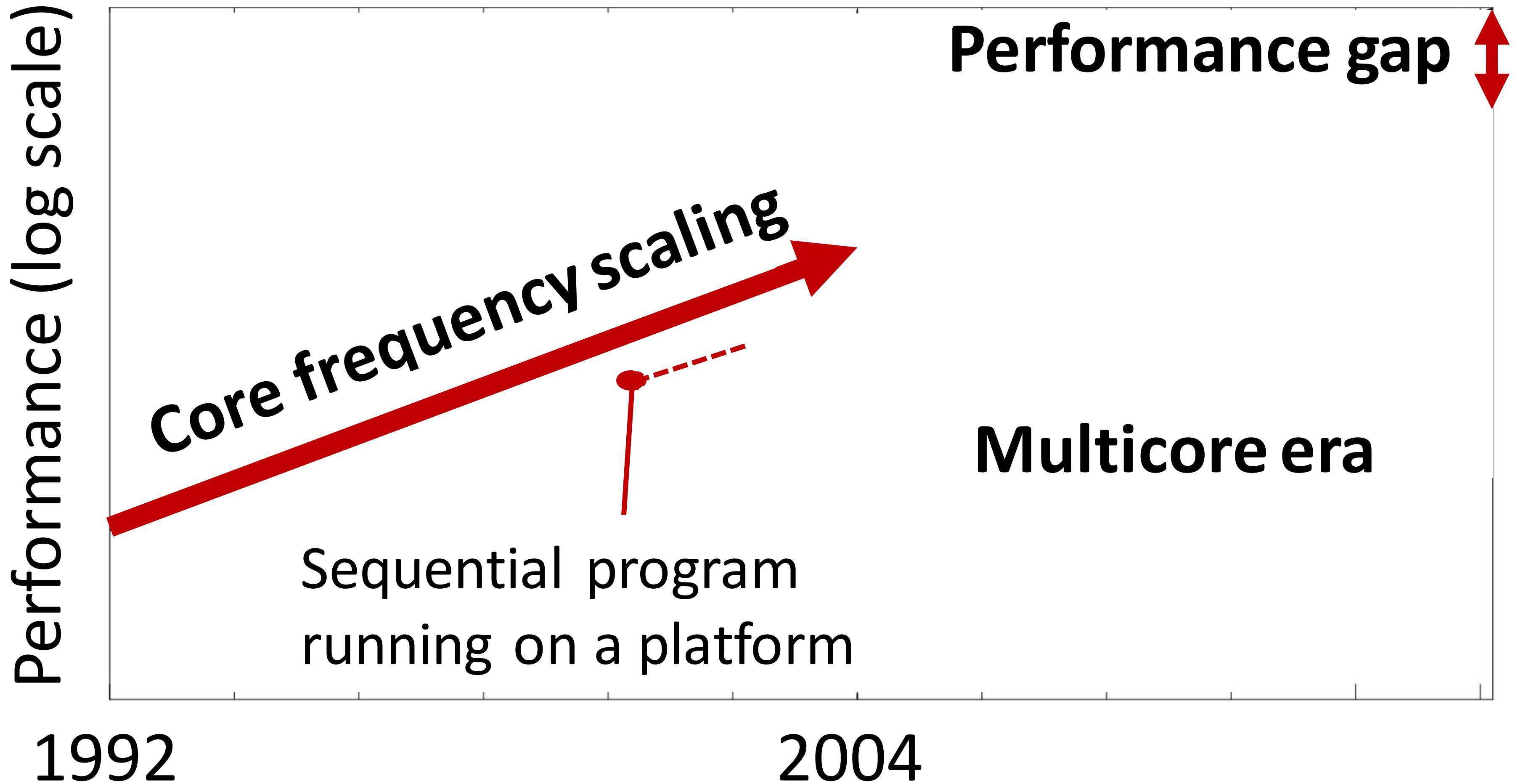
# Liberating Threads from Non-Numerical Programs with an Architecture-Compiler Co-Design

Simone Campanoni

UNIVERSITY OF
CAMBRIDGE

PRINCETON
UNIVERSITY

# Sequential programs are not accelerating like they used to



# Multicores are underutilized

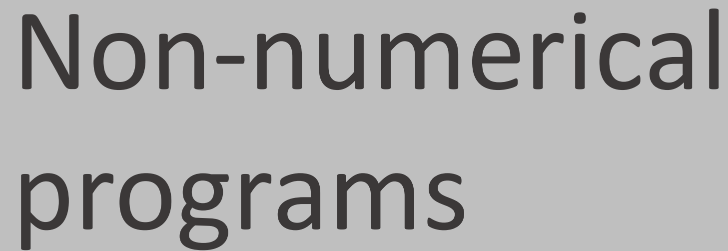
**Single application:**
Not enough explicit parallelism

- Developing parallel code is hard
- Sequentially-designed code is still ubiquitous

**Multiple applications:**
Only a few CPU-intensive applications running concurrently in client devices

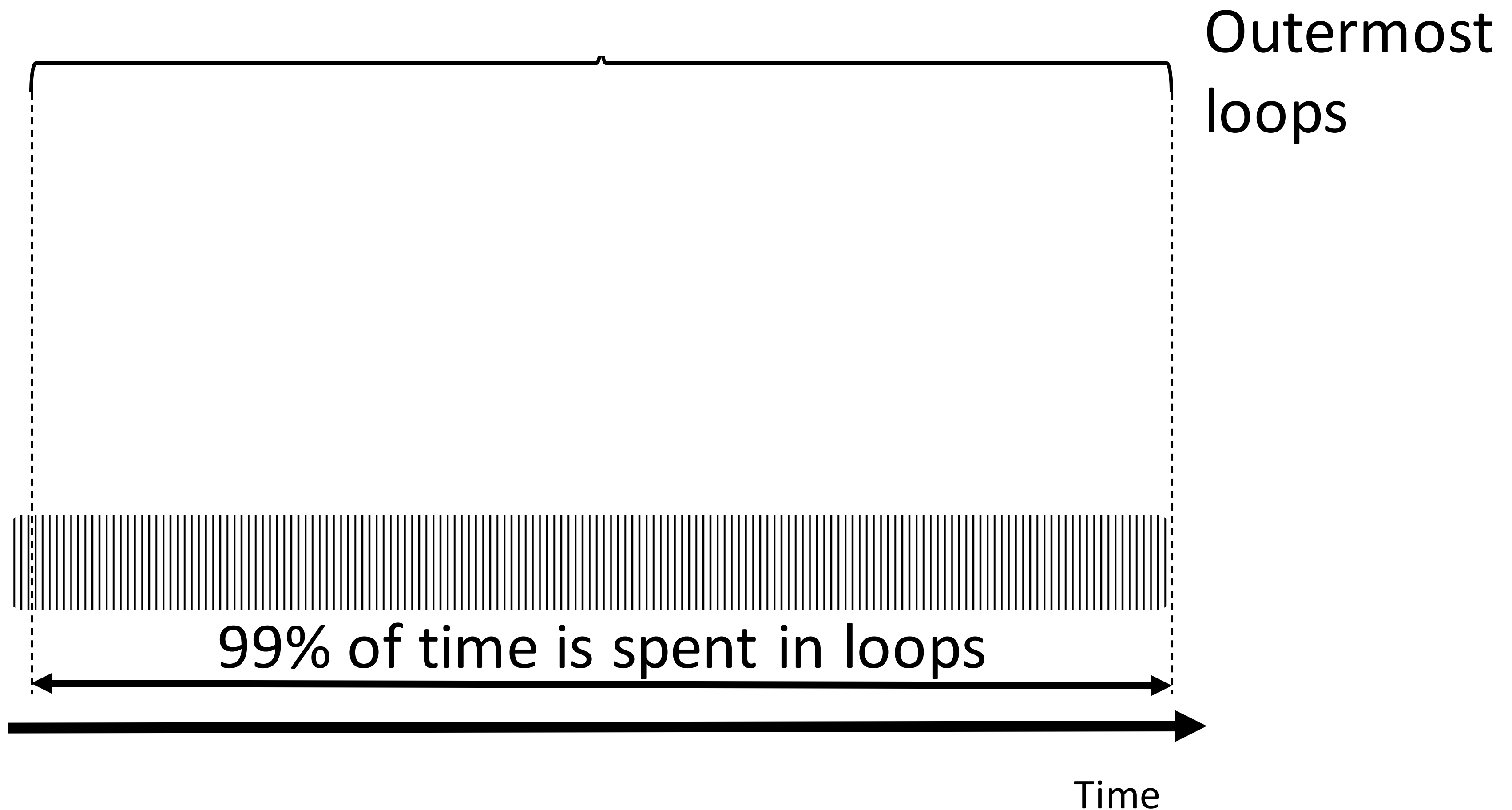
**Parallelizing compiler:**
Exploit unused cores
to accelerate
sequential programs

# Non-numerical programs need to be parallelized

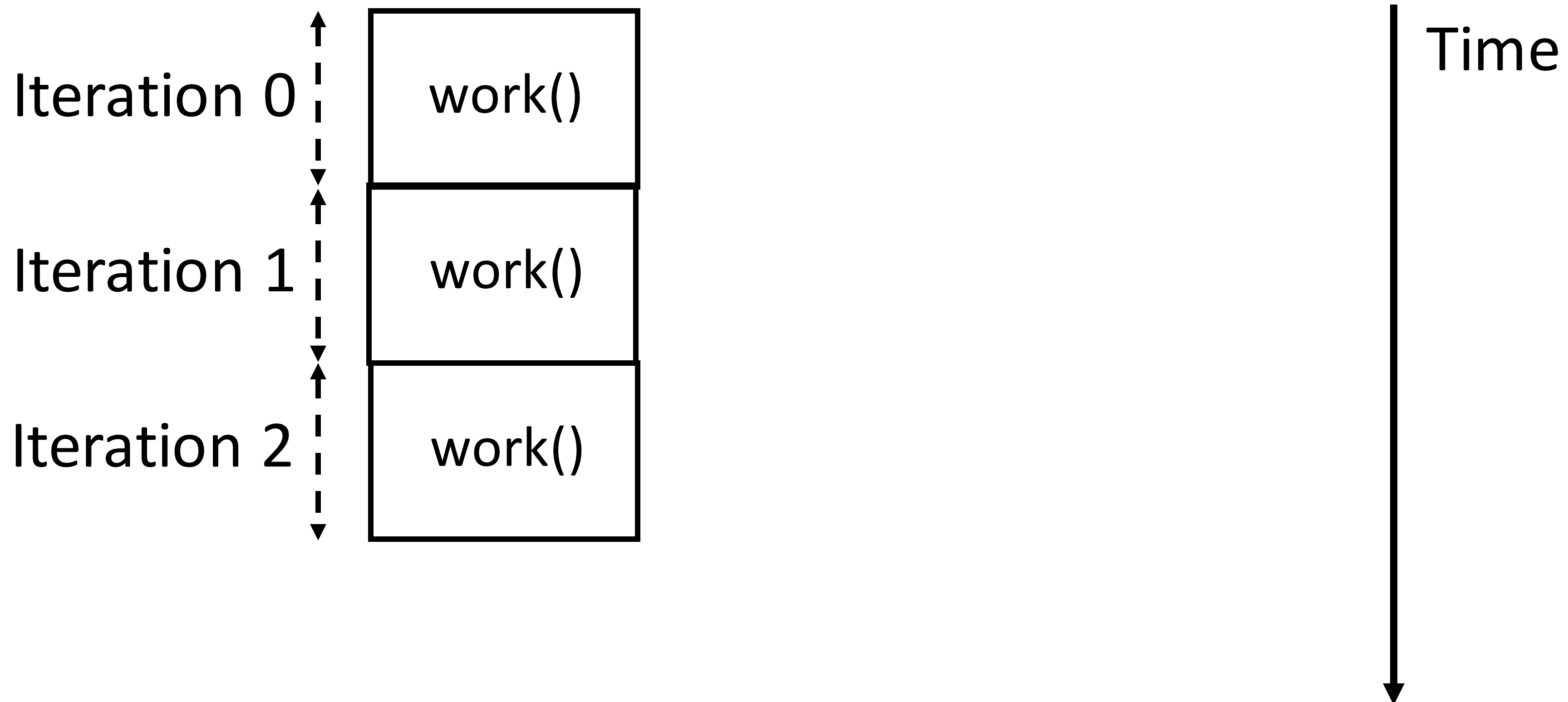



Numerical
programs

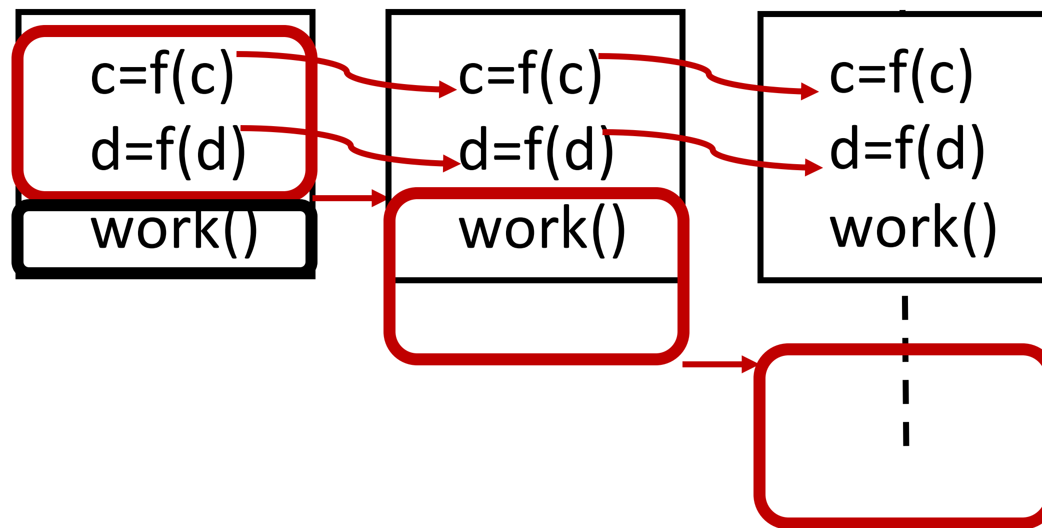
# Parallelize loops to parallelize a program



# DOALL parallelism

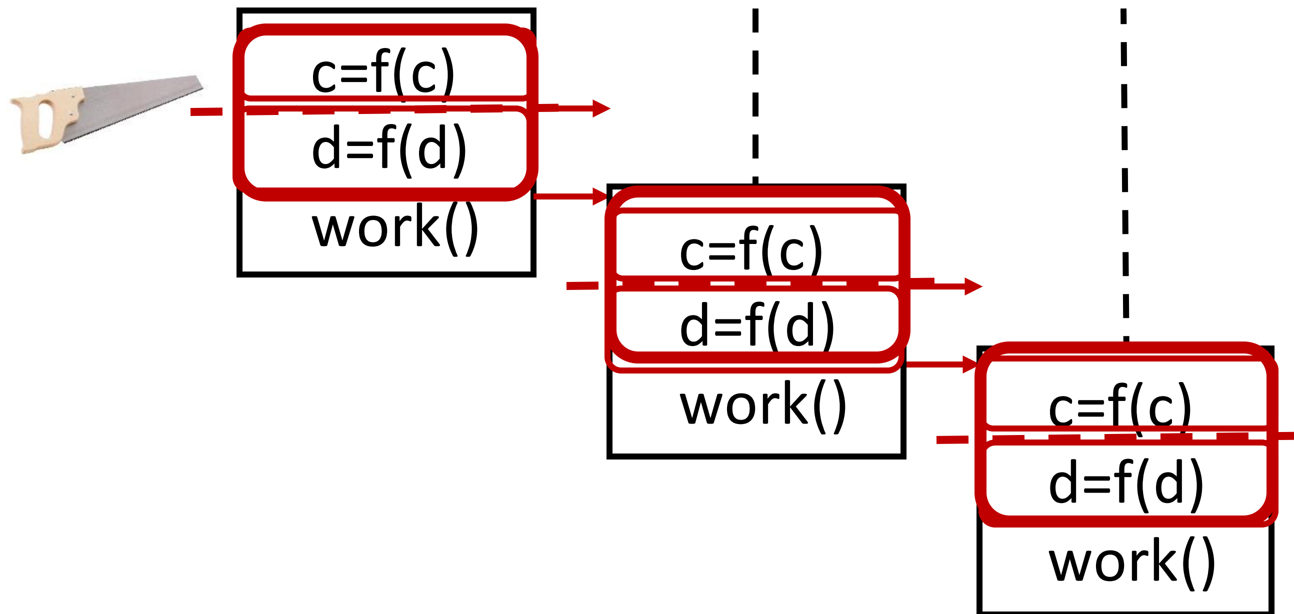


# DOACROSS parallelism



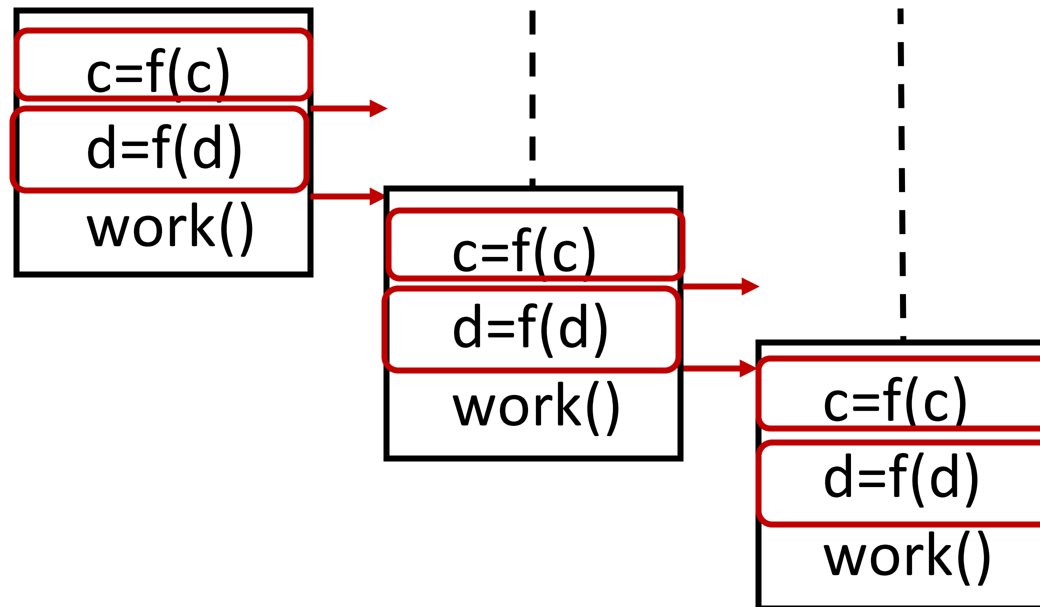
# HELIX: DOACROSS for multicore

[*CGO 2012, DAC 2012, IEEE Micro 2012*]



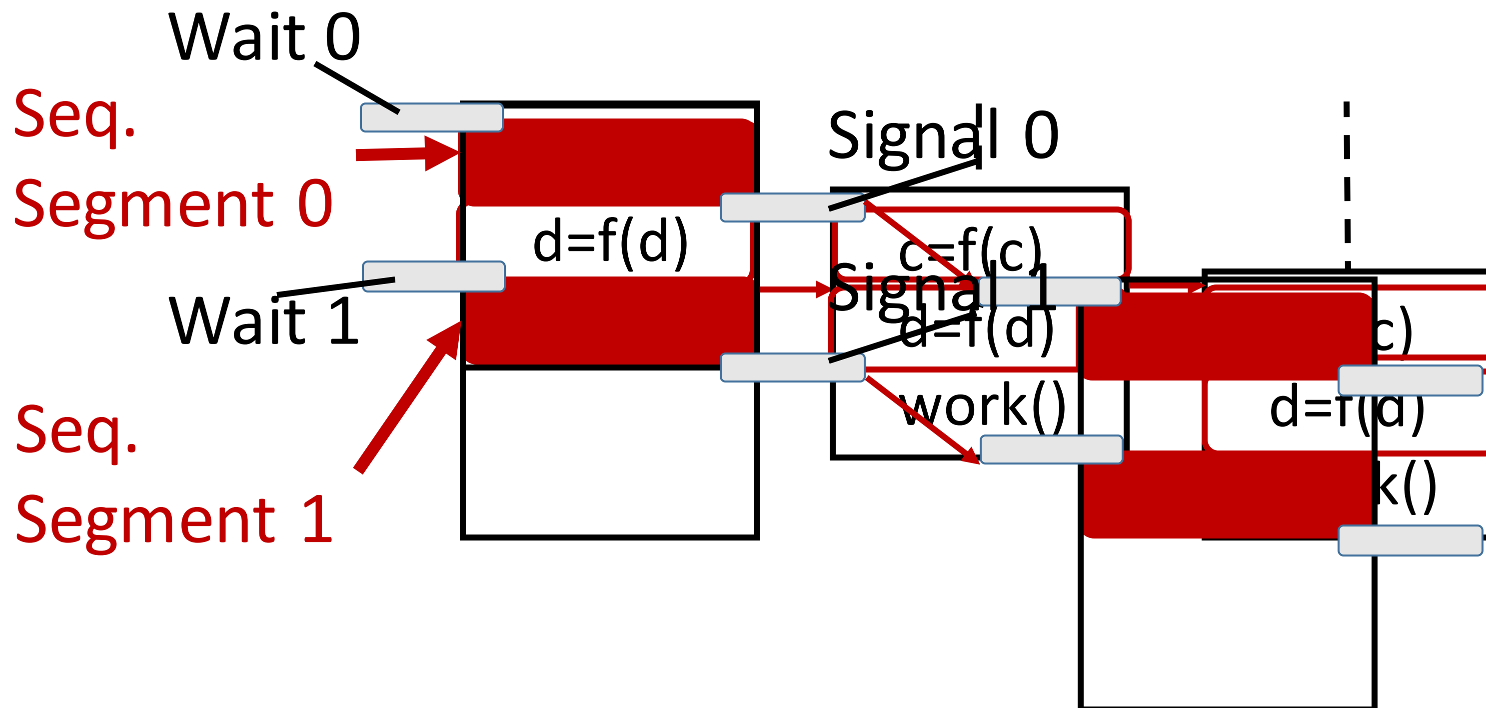
# HELIX: DOACROSS for multicore

[*CGO 2012, DAC 2012, IEEE Micro 2012*]



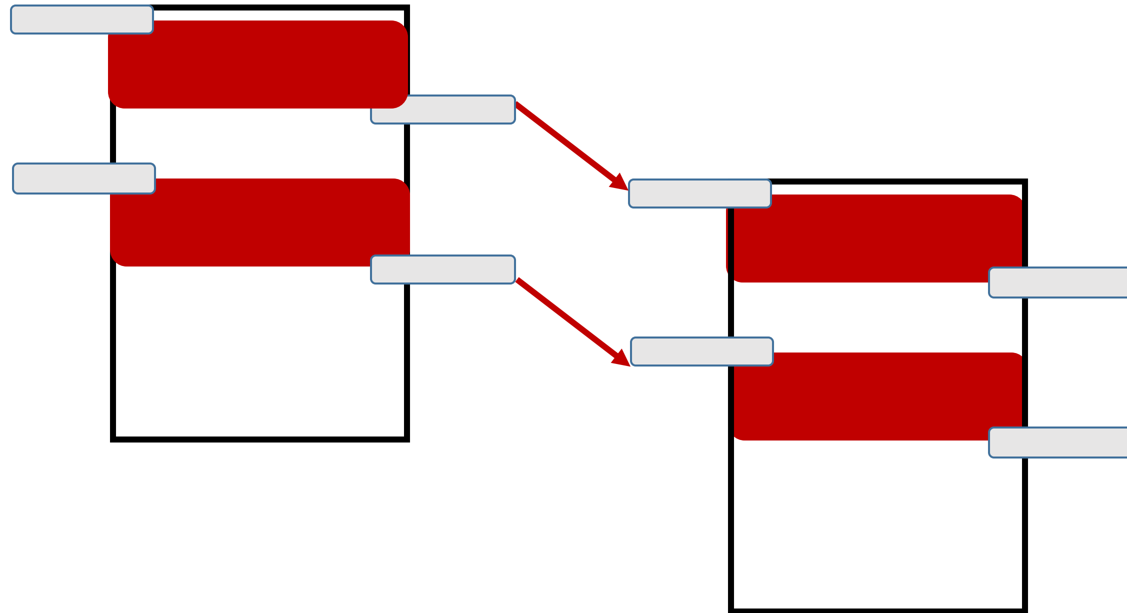
# HELIX: DOACROSS for multicore

[*CGO 2012, DAC 2012, IEEE Micro 2012*]



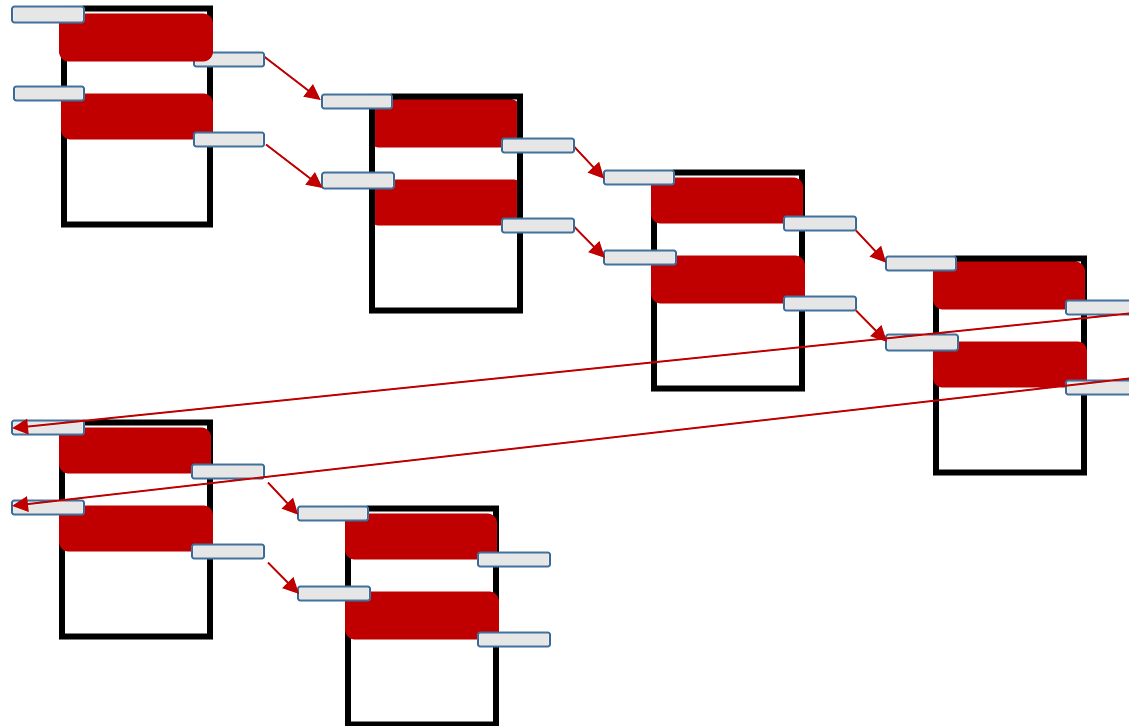
# HELIX: DOACROSS for multicore

[*CGO 2012, DAC 2012, IEEE Micro 2012*]

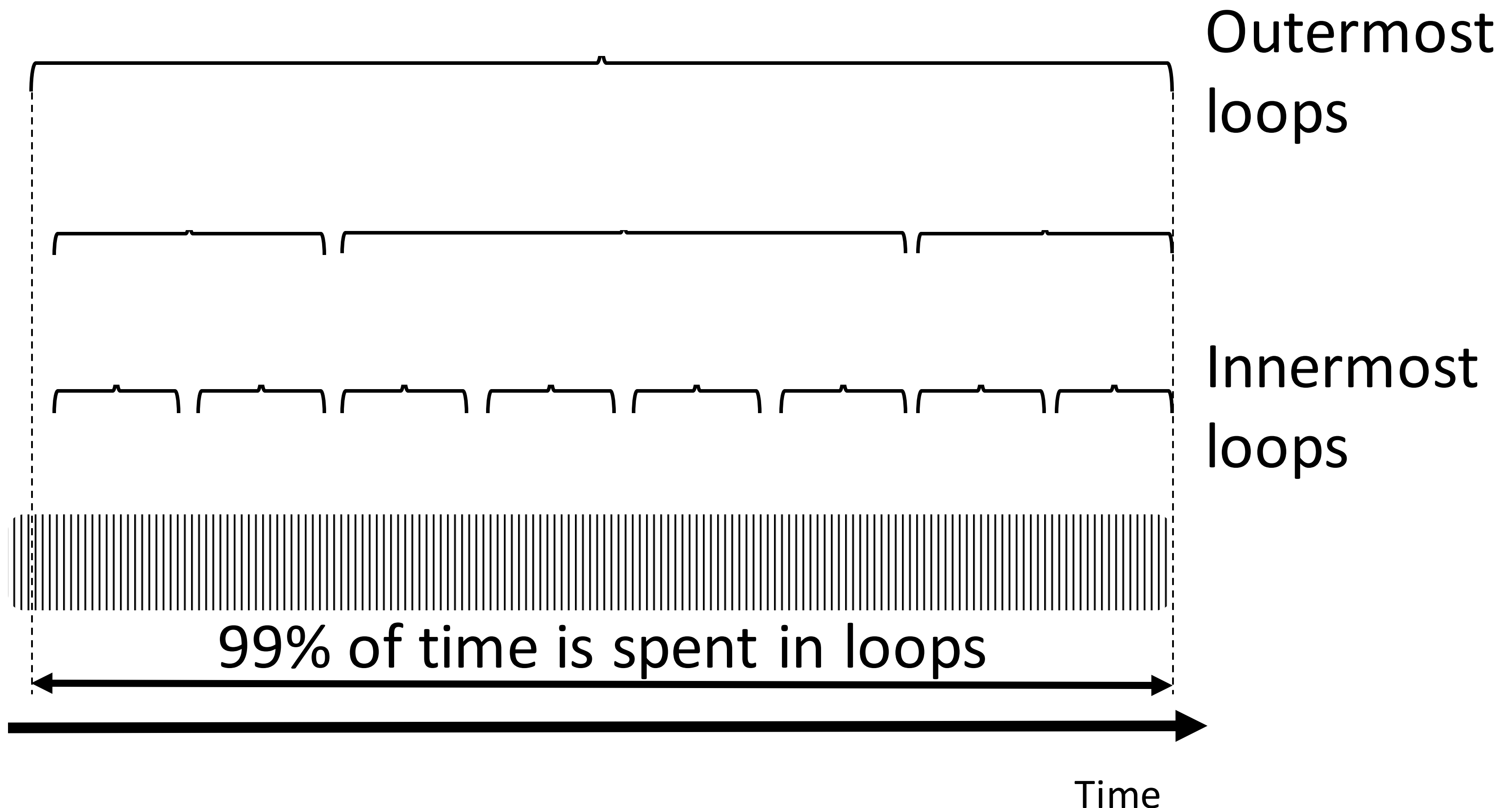


# HELIX: DOACROSS for multicore

[*CGO 2012, DAC 2012, IEEE Micro 2012*]



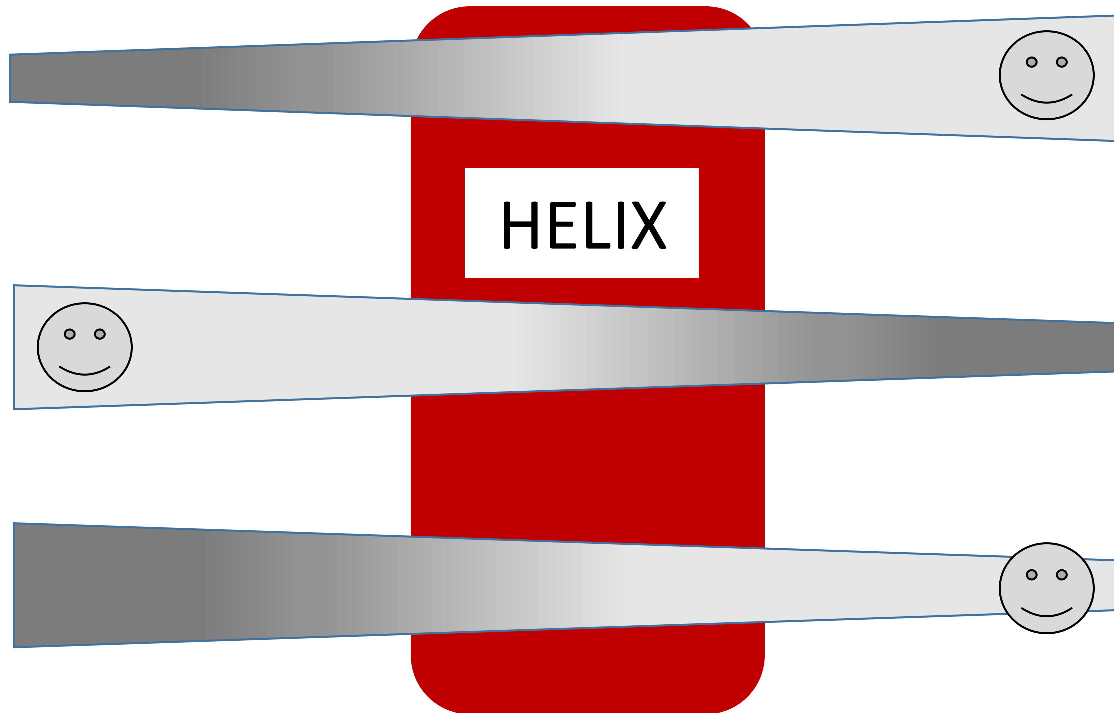
# Parallelize loops to parallelize a program



# Parallelize loops
# to parallelize a program

Innermost loops

Outermost loops




Coverage

Ease of analysis

Communication

# HELIX: DOACROSS for multicore

*[CGO 2012, DAC 2012, IEEE Micro 2012]*

16

Innermost loops

Outermost loops

Coverage

HELIX

HELIX-RC
HELIX-UP

Easy of analysis

Loop Parallelism

Studio DOACROSS

Communication

Speedup

4

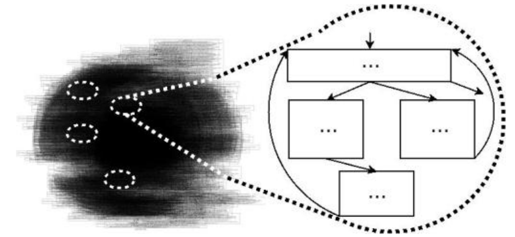
1

baseline

4-core Intel Nehalem

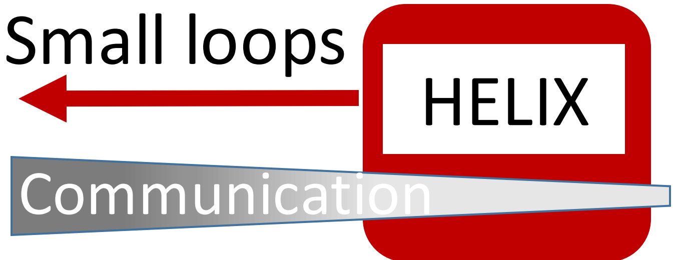
## Small Loop Parallelism and HELIX

[*CGO 2012*
*DAC 2012,*
*IEEE Micro 2012*]



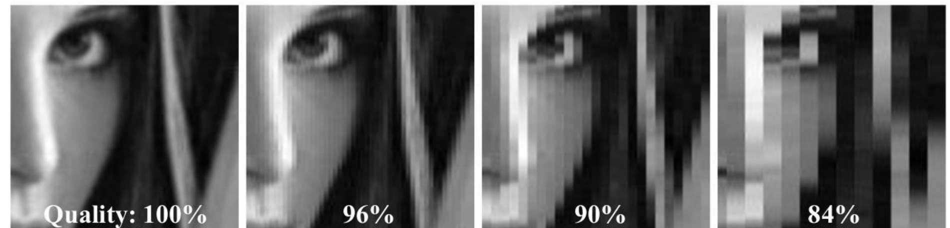
## HELIX-RC: Architecture/Compiler Co-Design

[*ISCA 2014,*
*IEEE Top Picks honorable mention 2014,*
*ACM research highlight 2017*]

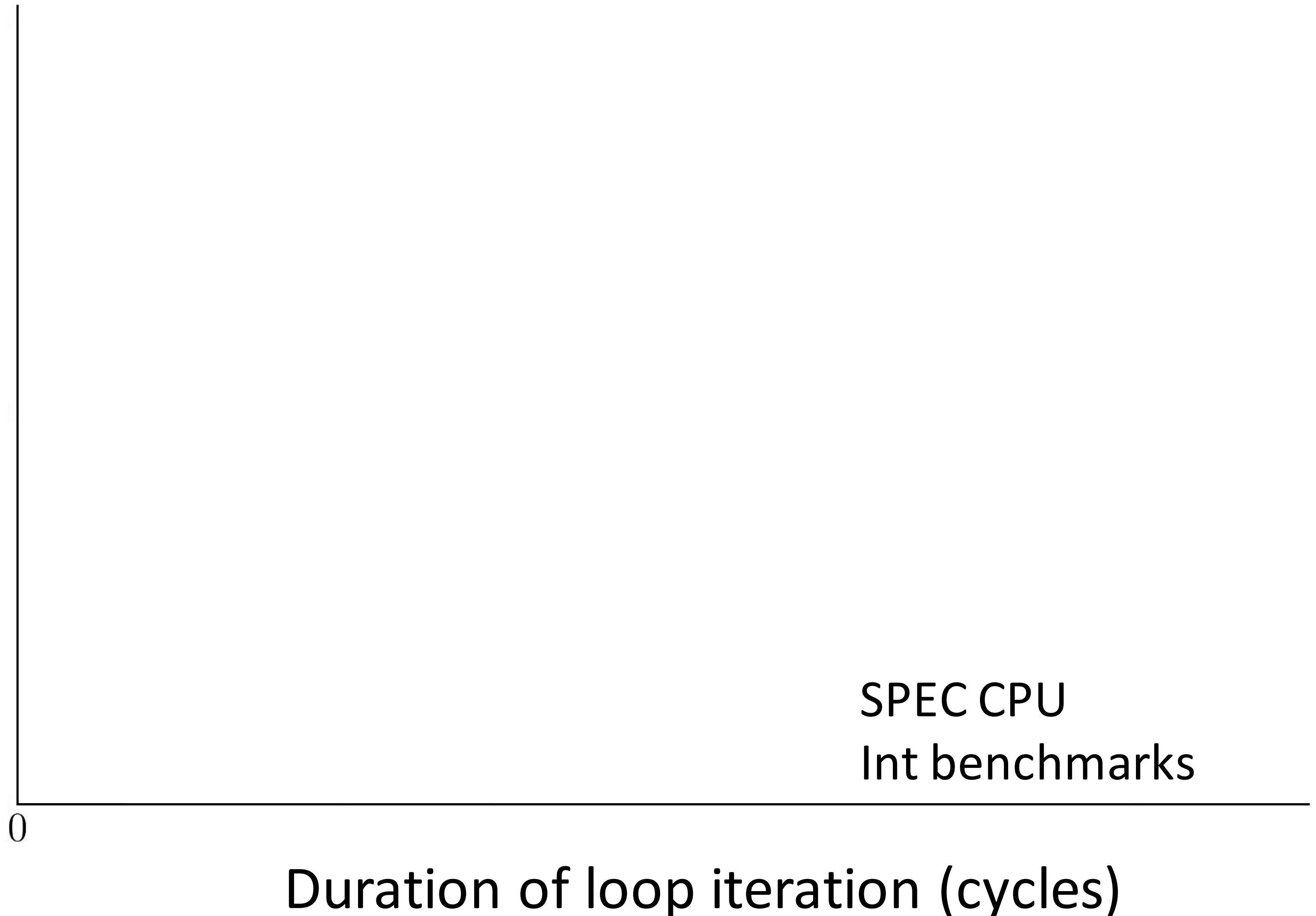



## HELIX-UP: Unleash Parallelization

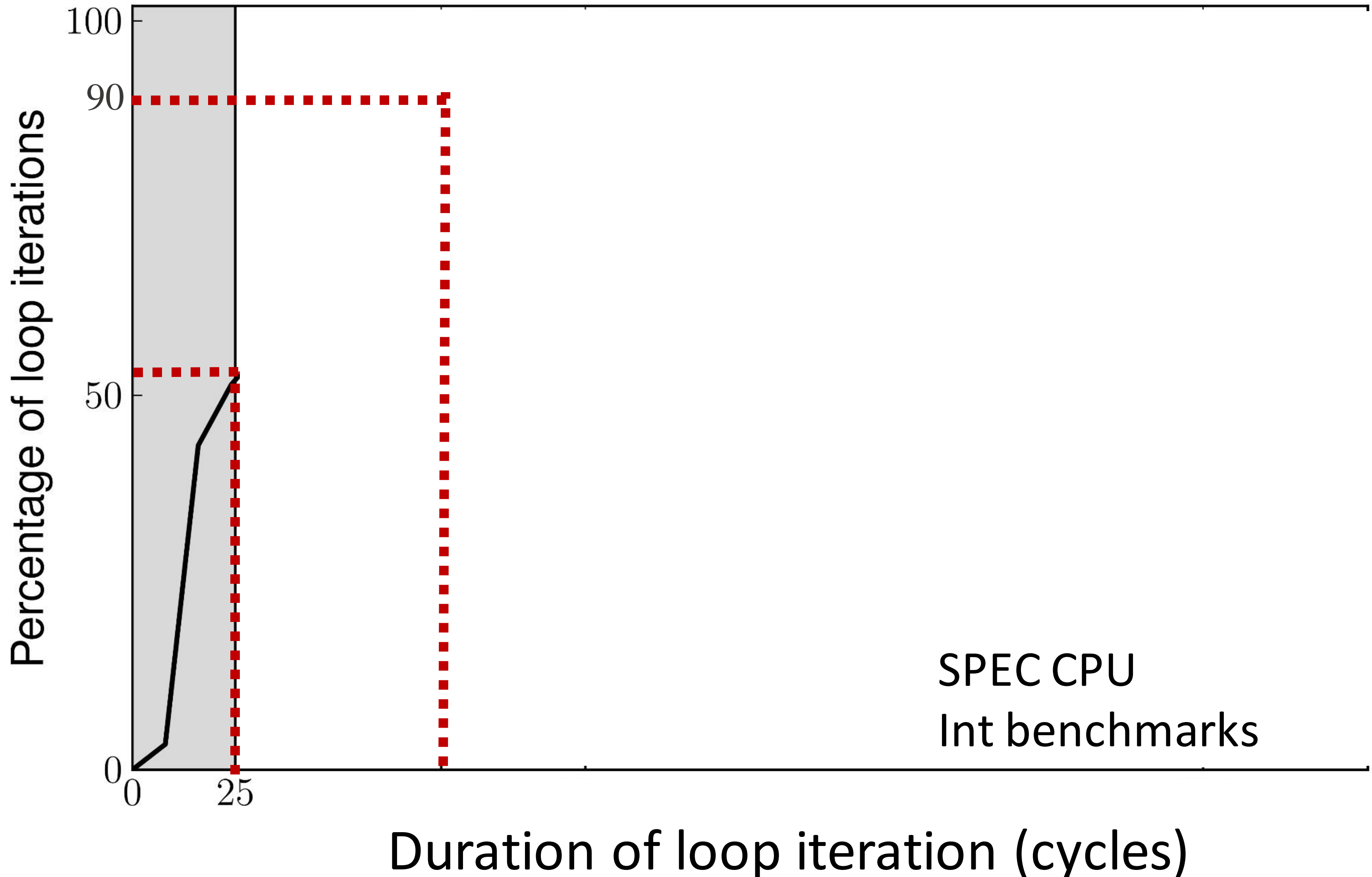
[*CGO 2015*]



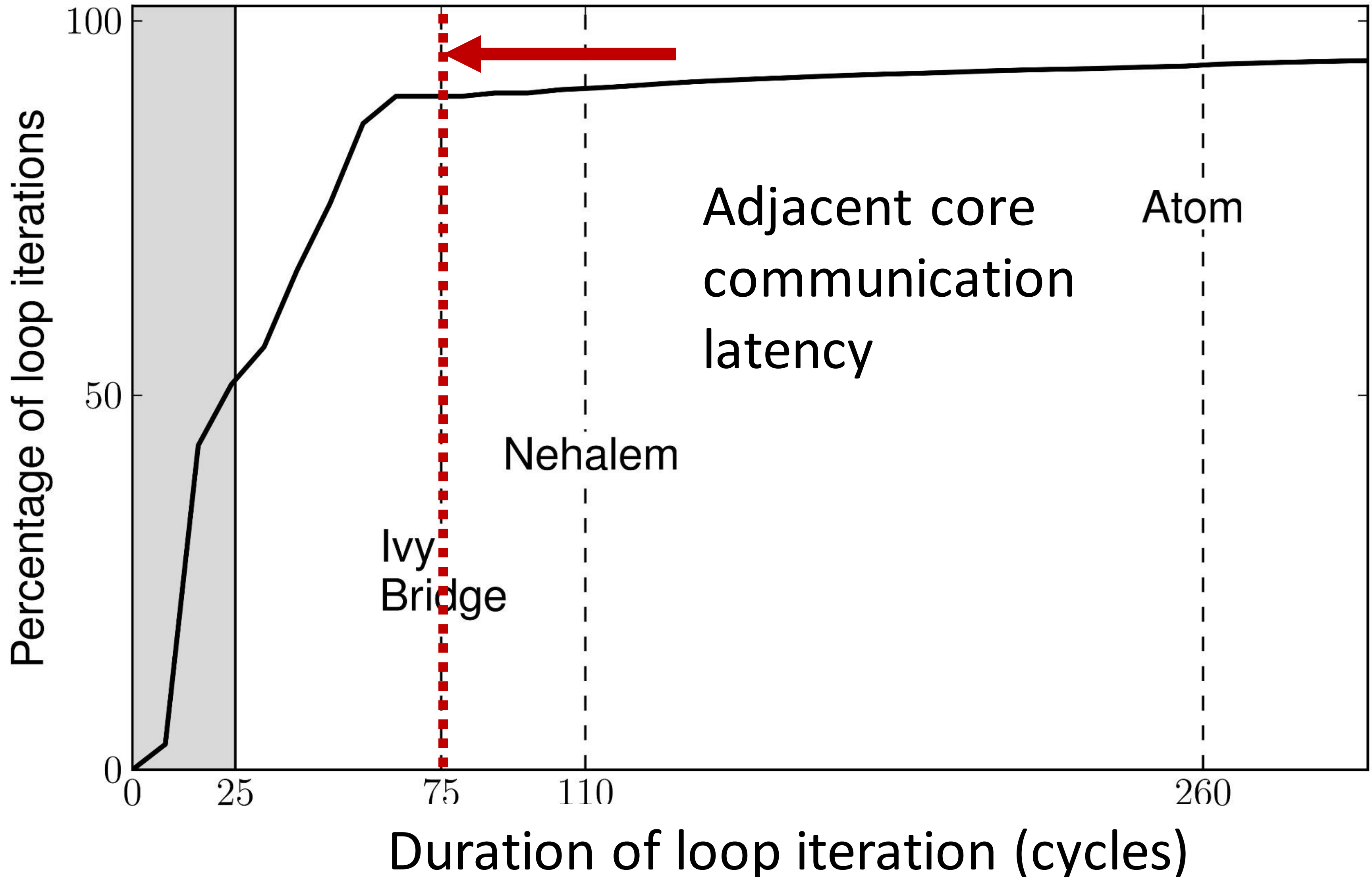
# SLP challenge: short loop iterations



# SLP challenge: short loop iterations



# SLP challenge: short loop iterations

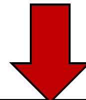


# A compiler-architecture co-design to efficiently execute short iterations

## Compiler

- Identify latency-critical code in each small loop
  - Code that generates shared data
- Expose information to the architecture

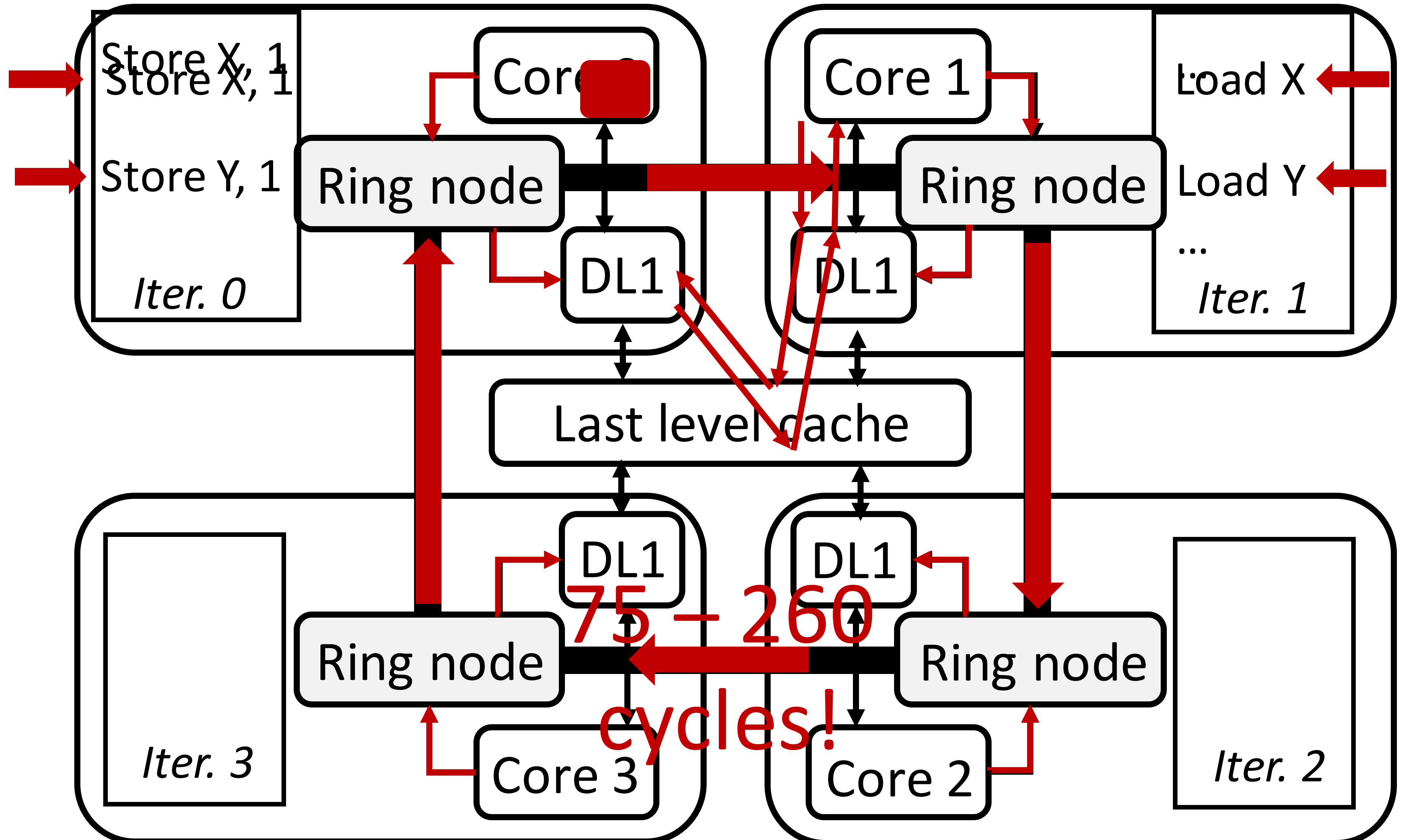
Wait 0



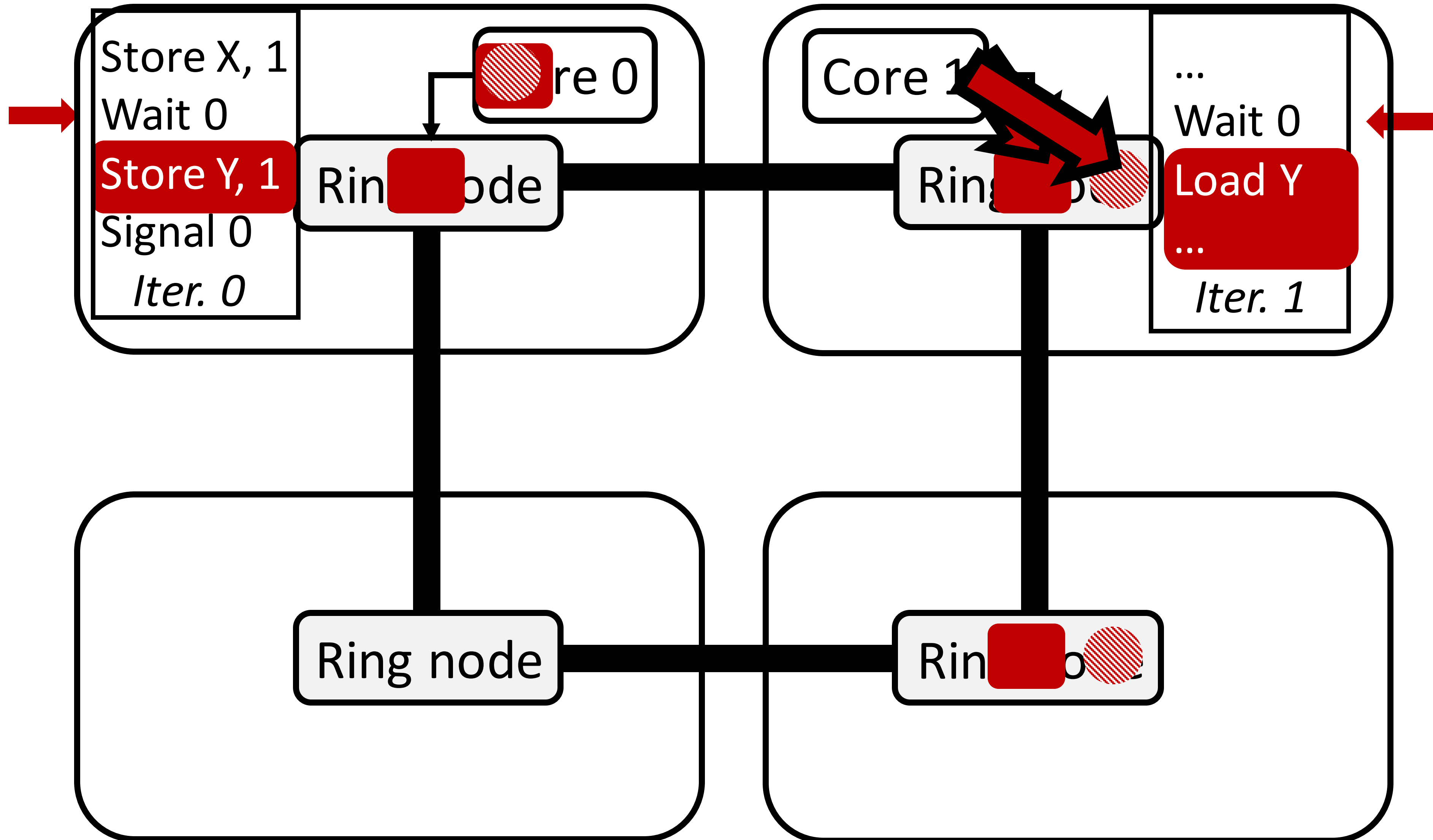
## Architecture: Ring Cache

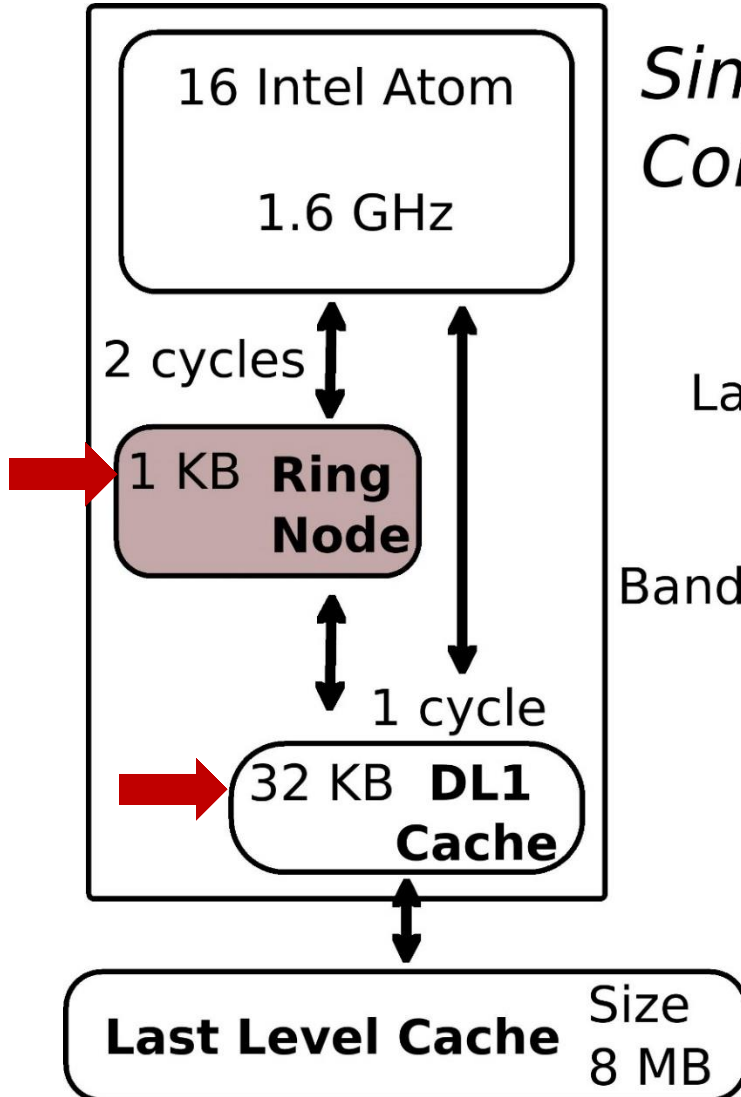
- Reduce the communication latency on the critical path

# Light-weight enhancement of today's multicore architecture



# Light-weight enhancement of today's multicore architecture





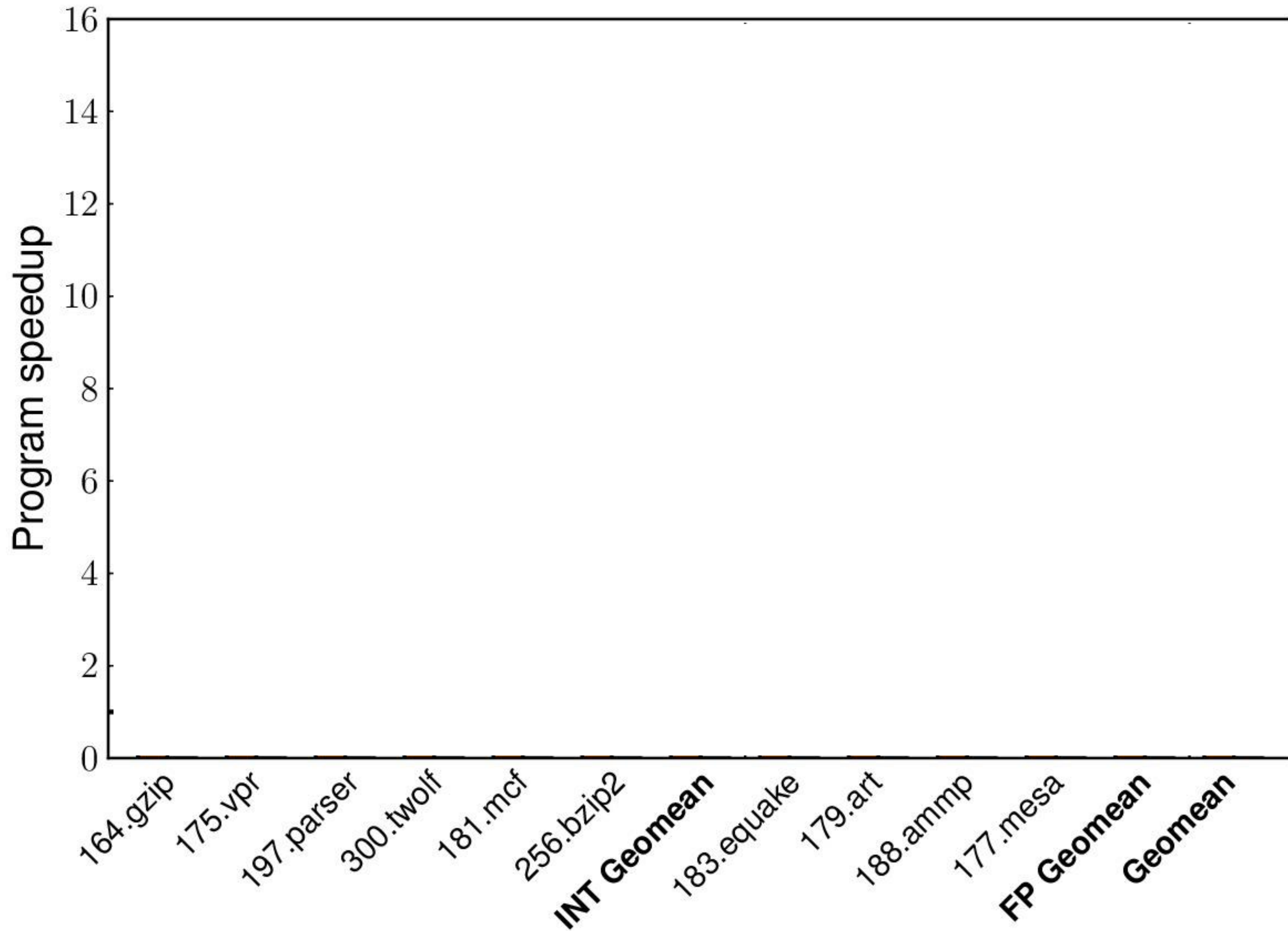

*Simulator: XIOSim, DRAMSim*
*Compiler : ILDJIT (LLVM)*

Latency: 1 cycle

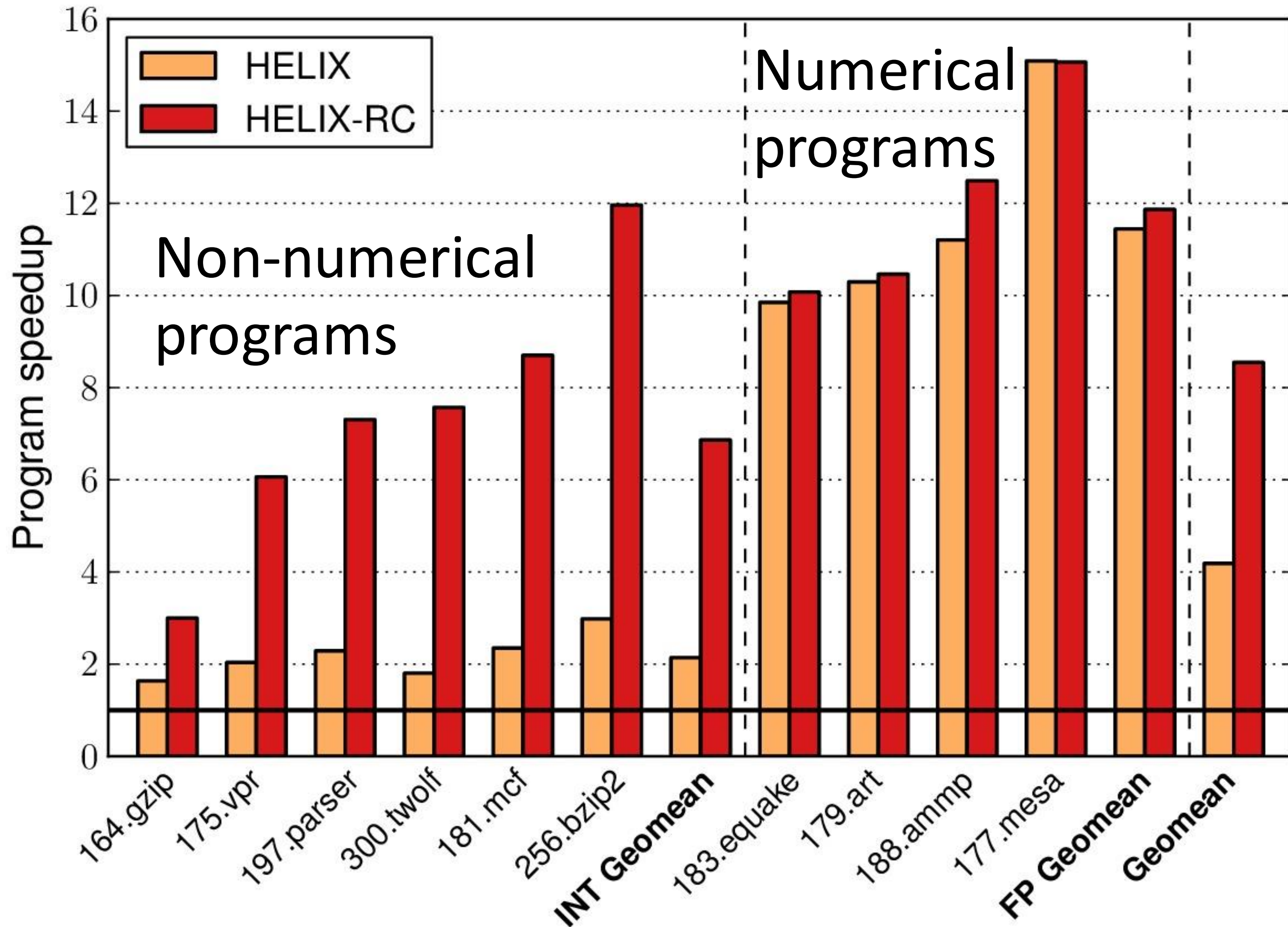
Bandwidth: 70 bits for signals
68 bits for data

98% hit rate

# The importance of HELIX-RC



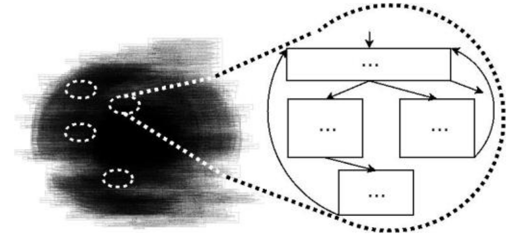
# The importance of HELIX-RC



Thank you!

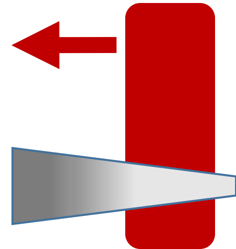
## Small Loop Parallelism and HELIX

- *Parallelism hides in small loops*



## HELIX-RC: Architecture/Compiler Co-Design

- *Irregular programs require low latency*



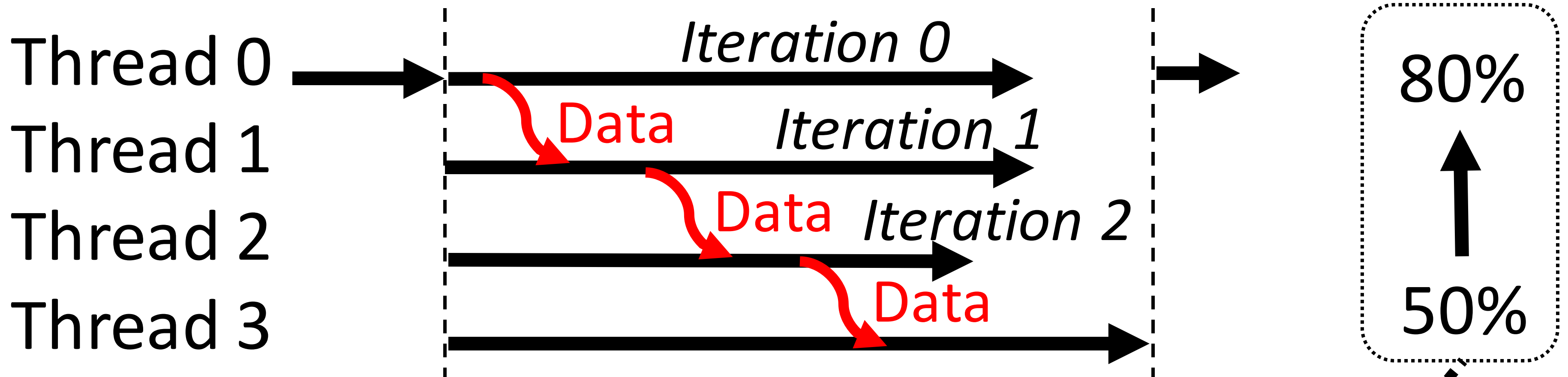
---

## HELIX-UP: Unleash Parallelization

- *Tolerating distortions boosts parallelization*

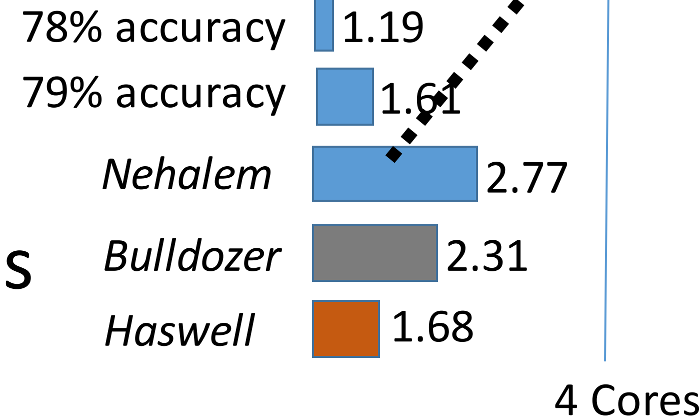
BACKUP

# HELIX and its limitations




Performance:

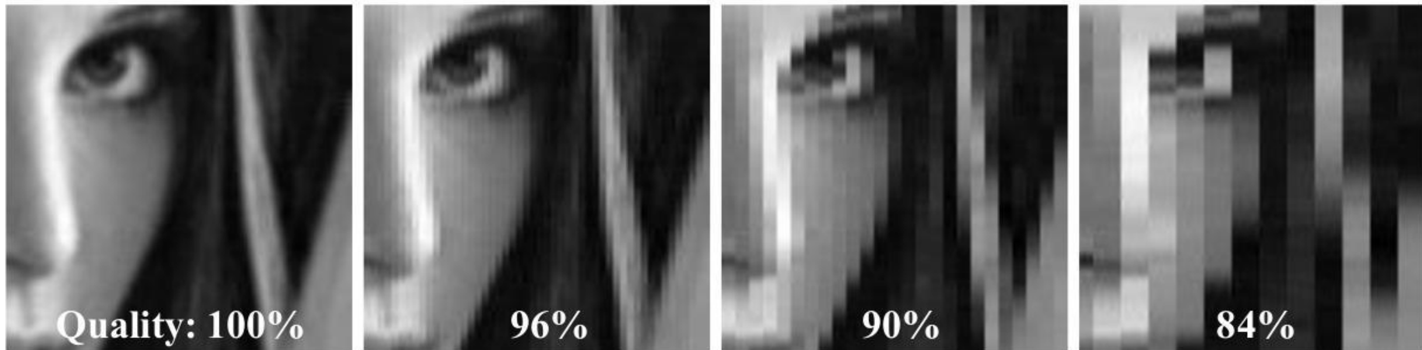
- Lower than you would like
- Inconsistent across architectures
- Sensitive to dependence analysis accuracy




**What can we do to improve it?**

# Opportunity: relax program semantics

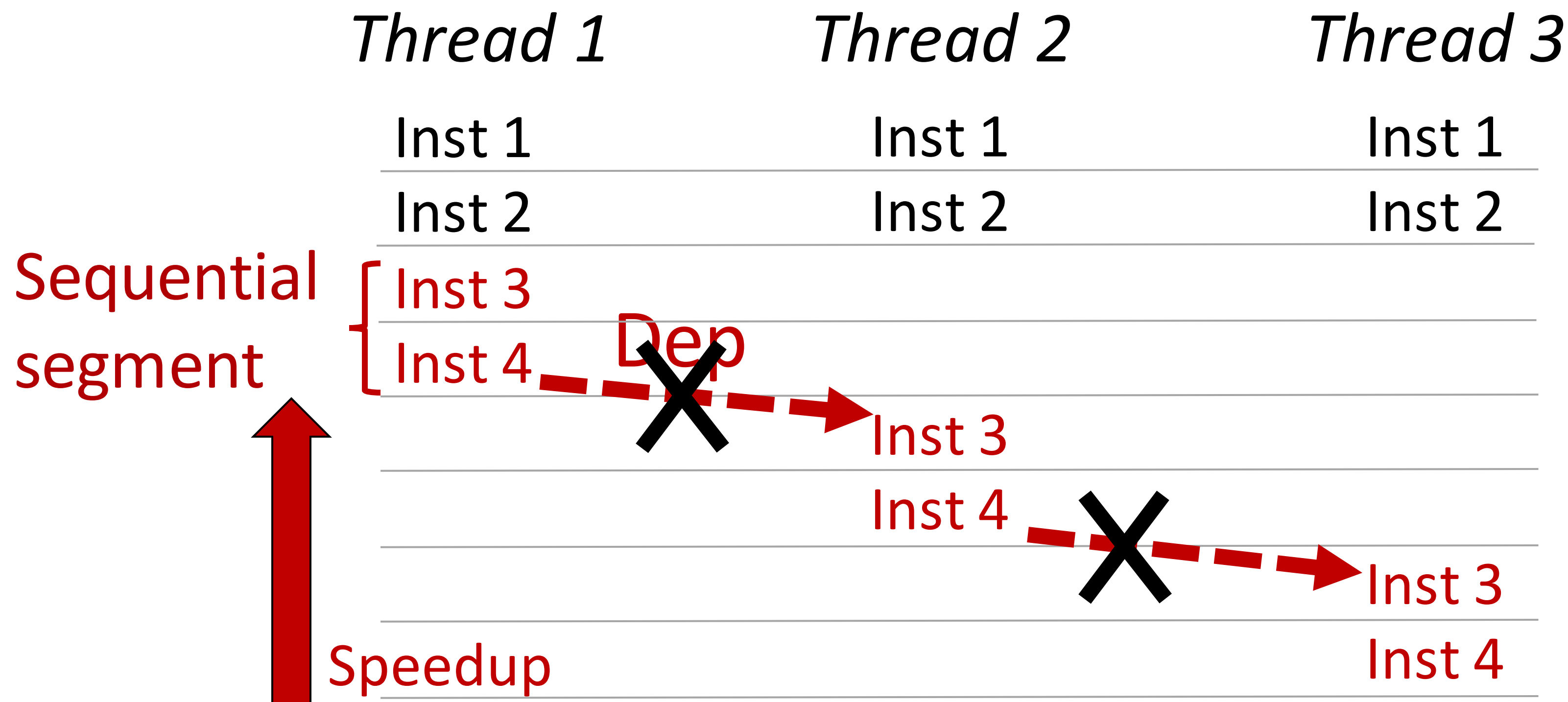
- Some workloads tolerate output distortion




- Output distortion is workload-dependent

# Relaxing transformations remove performance bottlenecks

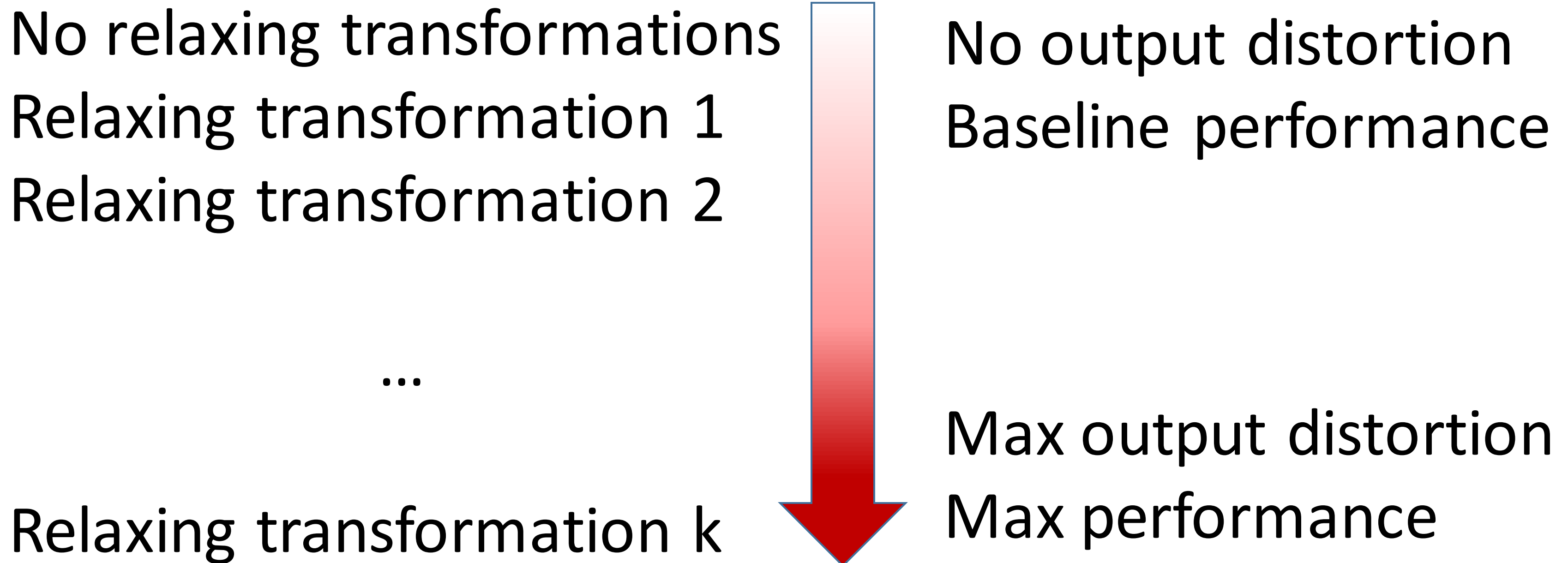
- Sequential bottleneck



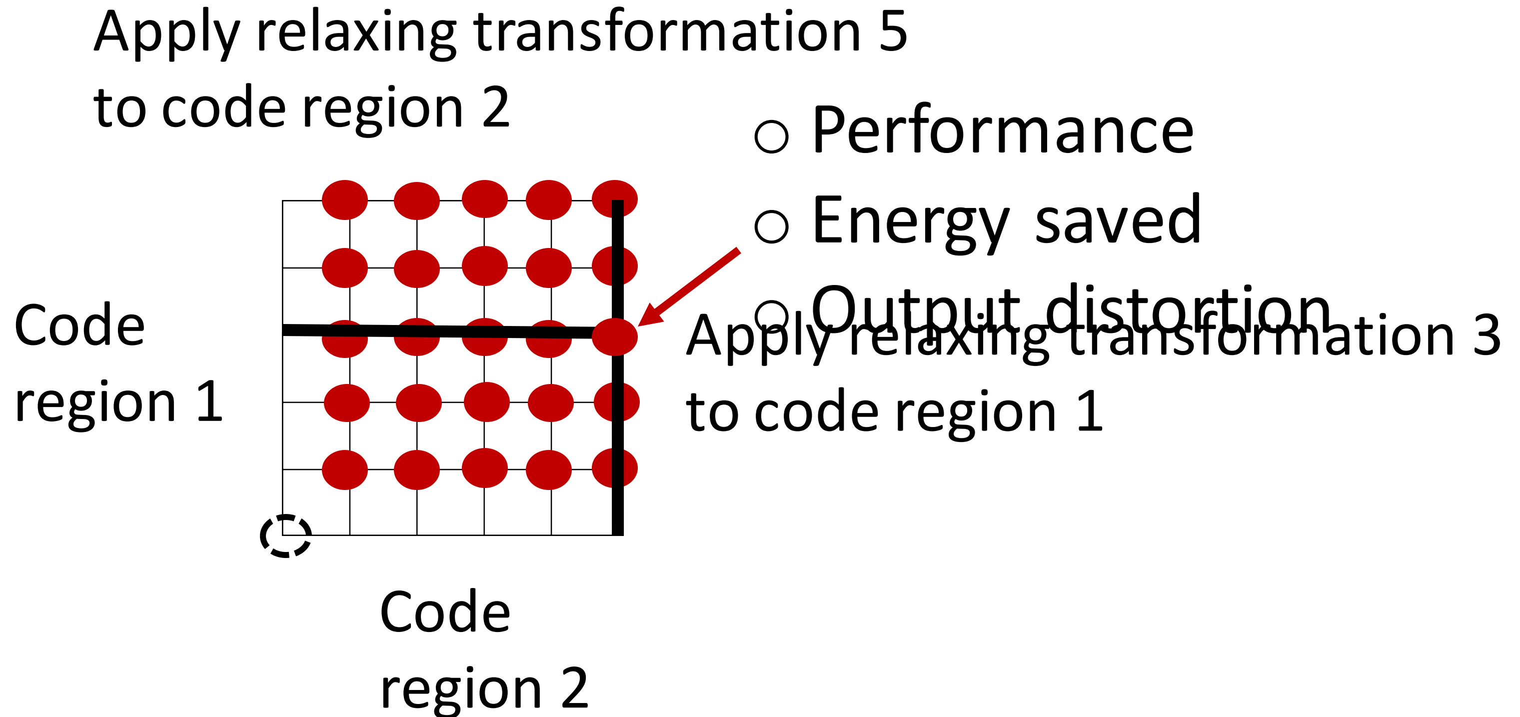
# Relaxing transformations remove performance bottlenecks

- Sequential bottleneck
- Communication bottleneck
- Data locality bottleneck

# Relaxing transformations remove performance bottlenecks



# Design space of HELIX-UP




1) User provides output distortion limits
2) System finds the best configuration
3) Run parallelized code with that configuration

# Pruning the design space

**Empirical observation:**
Transforming a code region
affects only the loop it belongs to

50 loops, 2 code regions per loop
2 transformations per code region

Complete space $= 2^{100}$
Pruned space $= 50 * (2^2) = 200$

**How well does HELIX-UP perform?**

# HELIX-UP unblocks extra parallelism with small output distortions

# HELIX: no relaxing transformations

HELIX | Static HELIX-UP

Hardware threads

Cores

Output distortion: 3.8%

Program Speedup

0 2 4 6 8 10 12

177.mesa 179.art 183.equake 256.bzip2 blackscholes swaptions Geomean

Nehalem 6 cores

2 threads per core

# HELIX-UP unblocks extra parallelism with small output distortions

Hardware threads

HELIX | Static HELIX-UP

Program Speedup

0 2 4 6 8 10 12

Output distortion

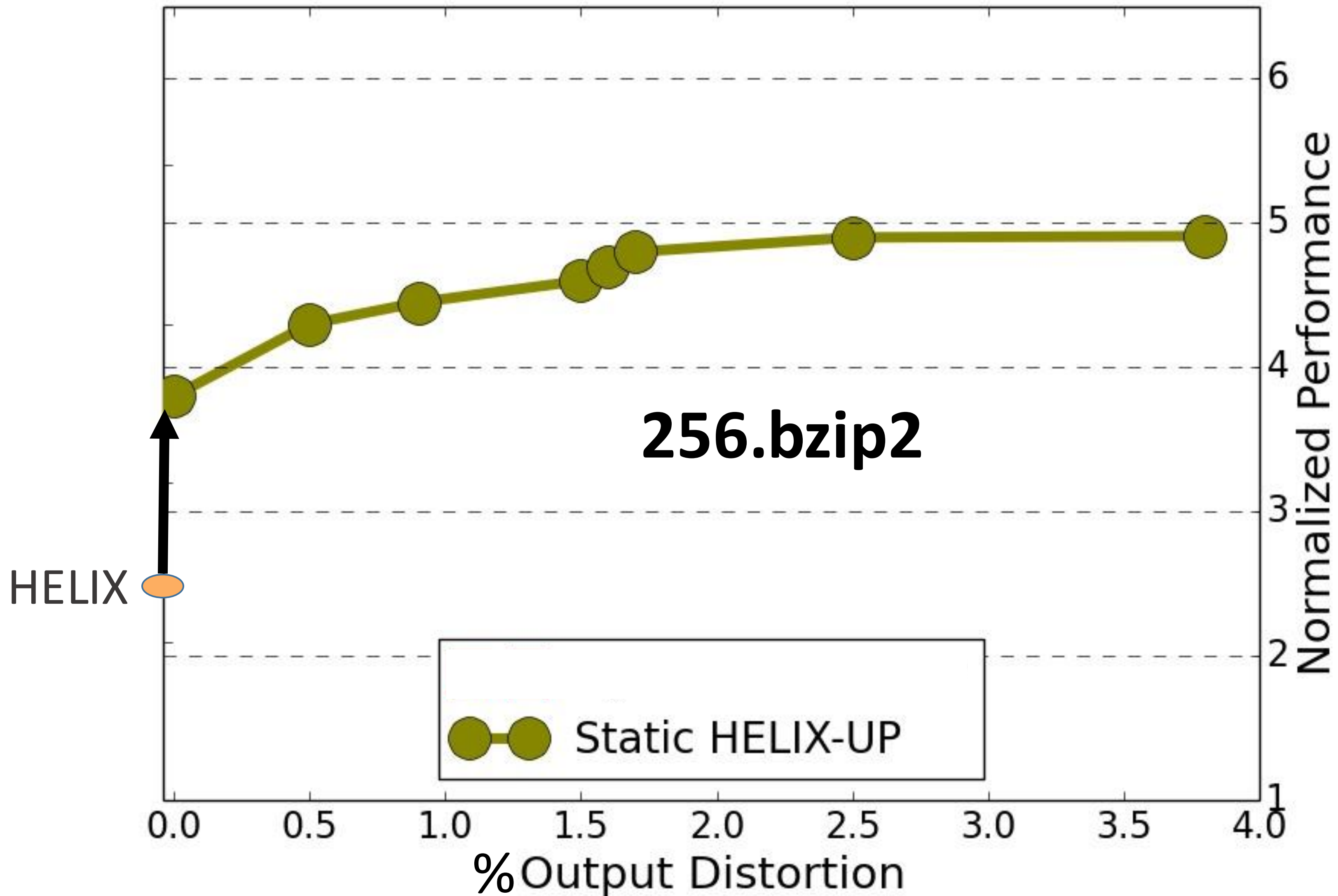
2.1% 3% 6.9% 3.8% 0% 2%

Cores

177.mesa 179.art 183.equake 256.bzip2 blackscholes swaptions Geomean

Nehalem 6 cores
2 threads per core

# Performance/distortion tradeoff

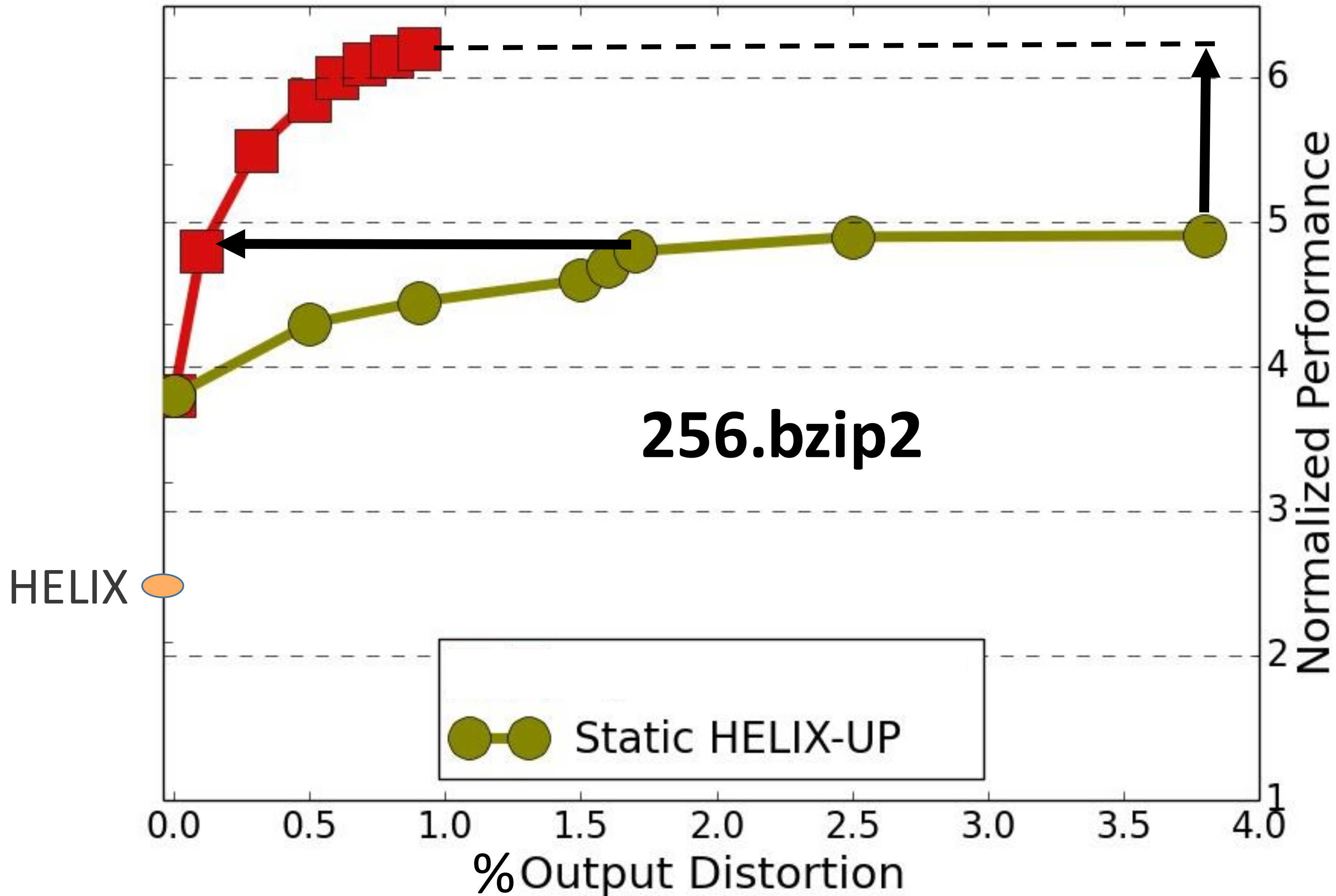


# Run time code tuning

- Static HELIX-UP decides
  how to transform the code based on
  profile data averaged over inputs

- The runtime reacts to transient bottlenecks
  by adjusting code accordingly

# Adapting code at run time unlocks more parallelism

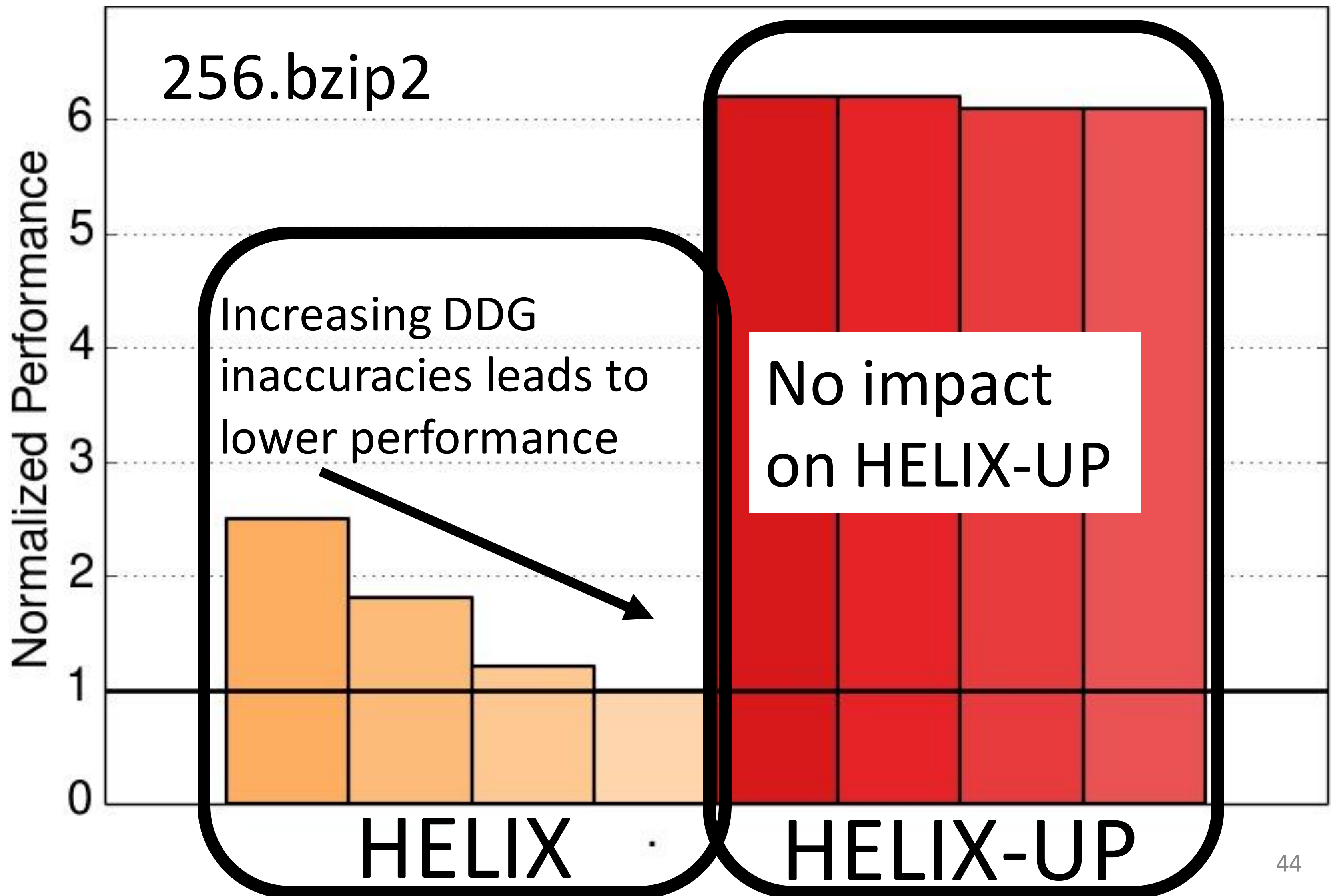


# HELIX-UP improves more than just performance

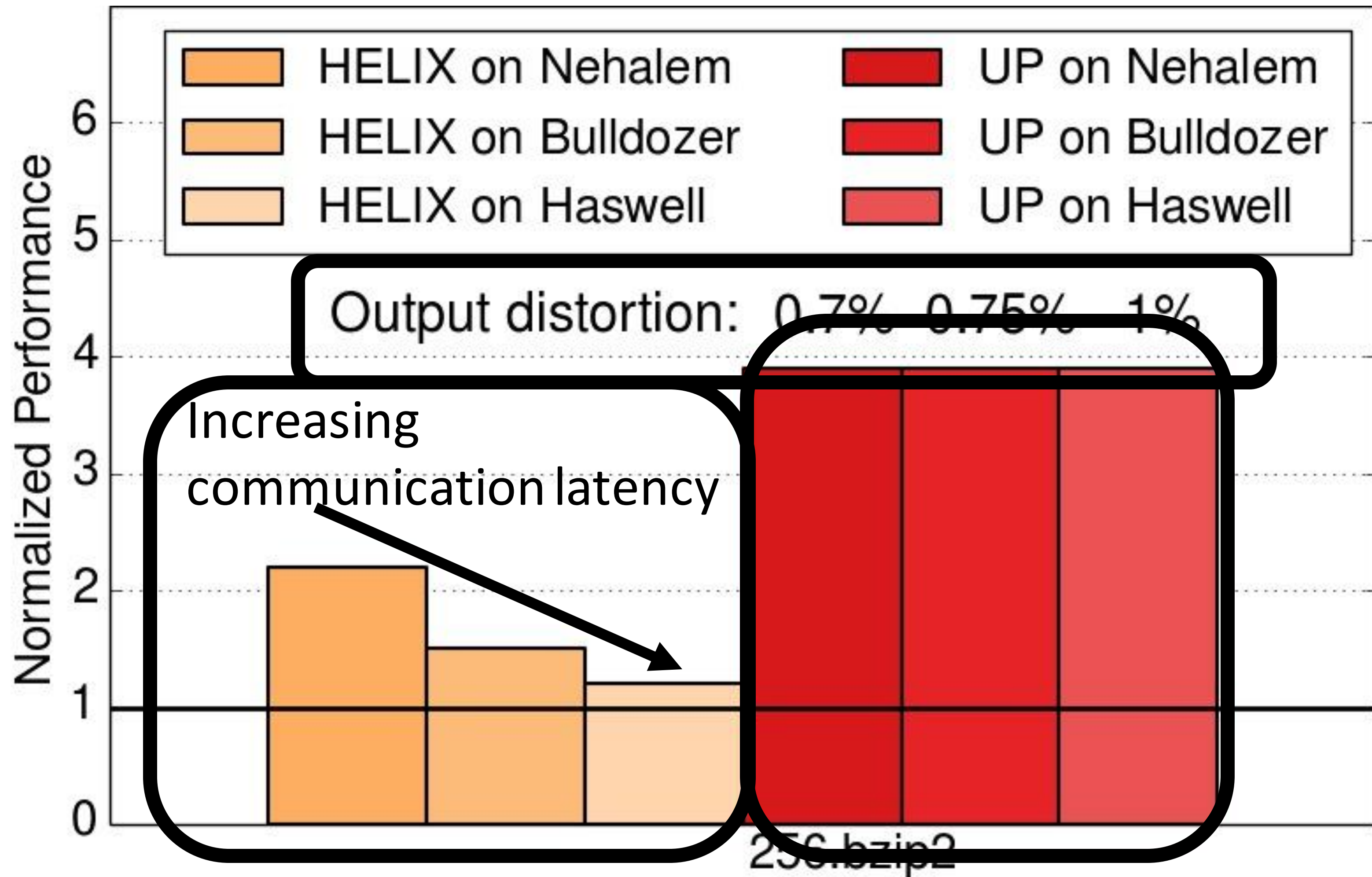
- Robustness to DDG inaccuracies

- Consistent performance across platforms

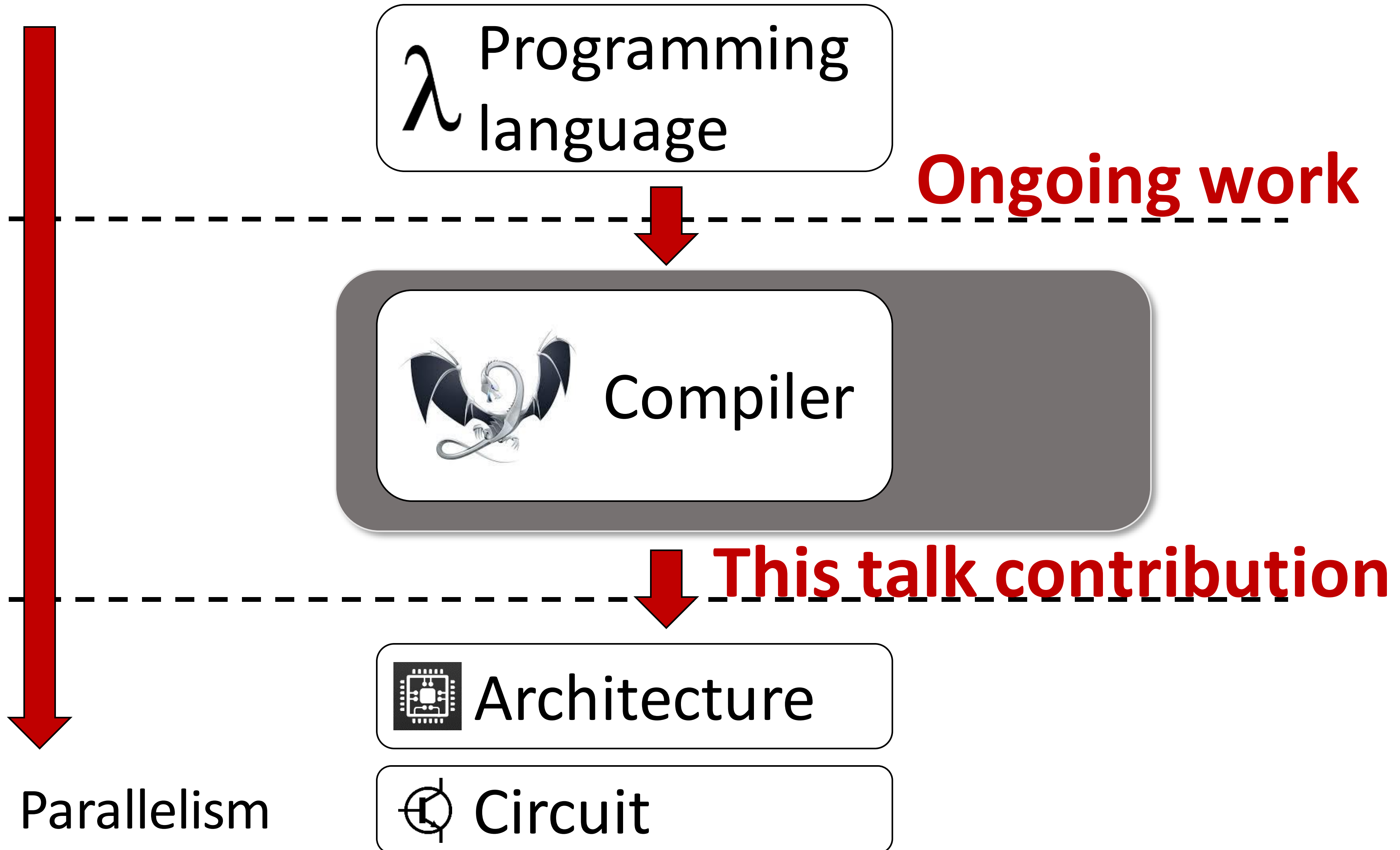
# Relaxed transformations to be robust to DDG inaccuracies



# Relaxed transformations for consistent performance



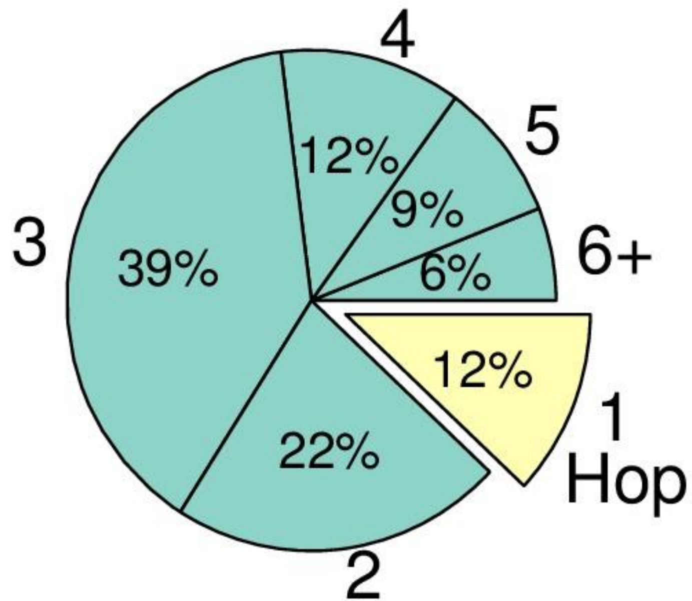
# Ongoing work: application-specific



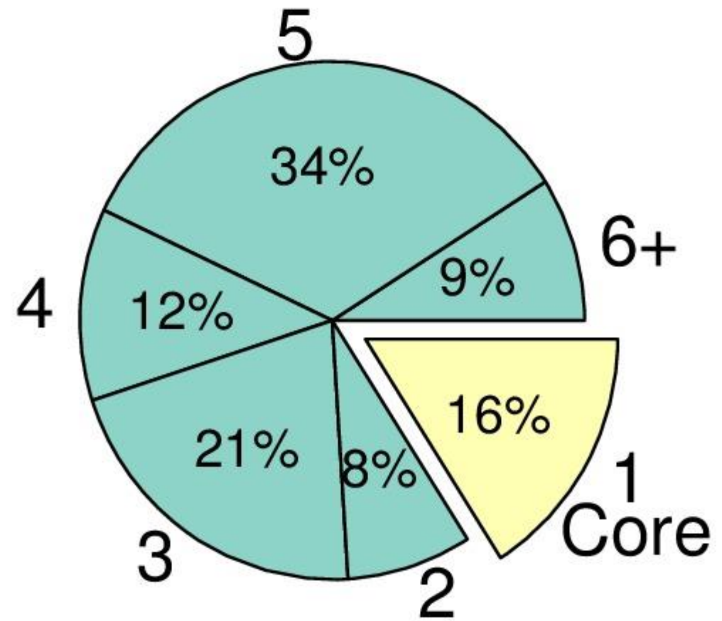

Parallelism

- Communication
- HW/compiler interface

# Irregular data consumption




Distance of consumers




Number of consumers

**Proactively broadcast shared data**

# Subsequent work

- Real system evaluation of HELIX-RC

- Mu

- Sp

# Small Loop Parallelism opportunity

Code complexity

- Control flow
- Data flow

Dependences to satisfy

- Actual
- Apparent

**Prior works**

- Thread Level Speculation (TLS)

Apparent

- TLS overhead big loops (10x more dependences!)

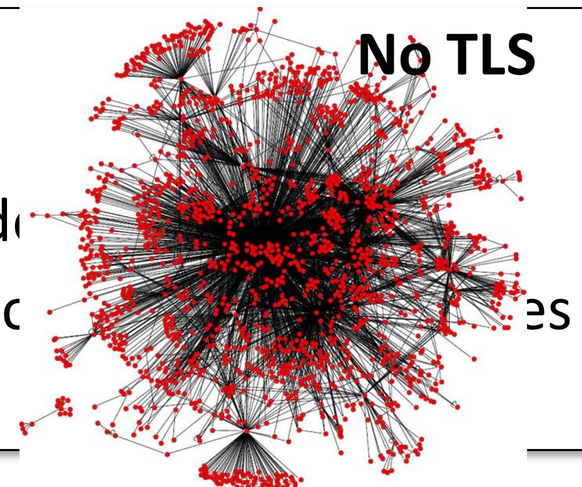
**Benefits of small (hot) loops**

- Code complexity

Apparent (only 1.2x more d

- Enable code transformations to reco …es

Actual



# HELIX-UP unblocks extra parallelism

HELIX: no relaxing transformations

Bzip2 has 2 outputs → Compressed file 100%
→ Statistics 0 <-> 100%

$$\frac{|correct - output| + 1}{correct + 1}$$

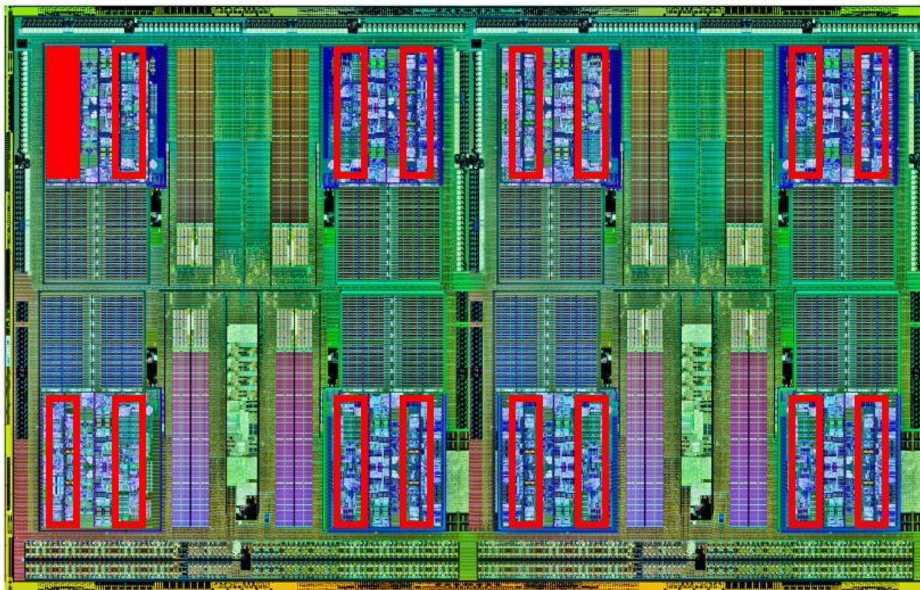
Program Speedup chart:

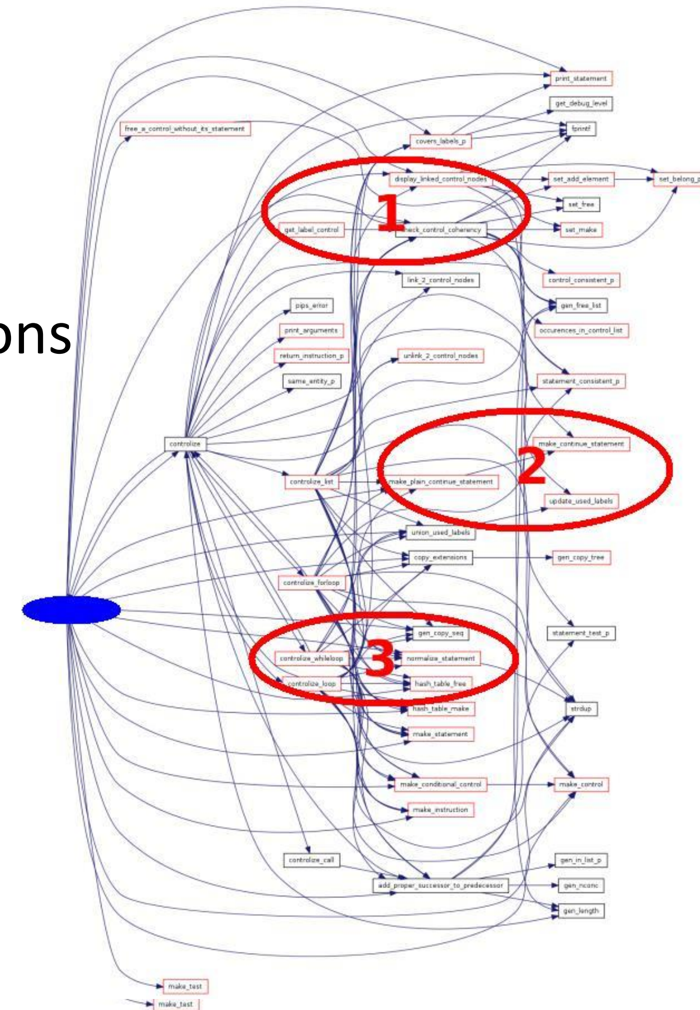


Nehalem 6 cores
2 threads per core

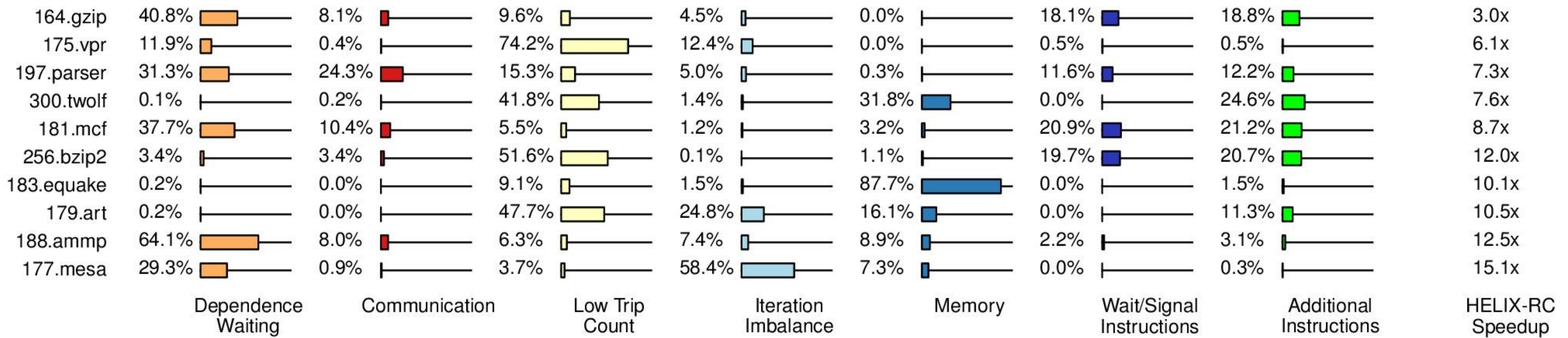
# Compiler: HCC

- Identify and analyze small loops
- For each small loop
    - Identify code that *may* generate data to be shared between iterations
    - Shape the code to minimize dependence cost



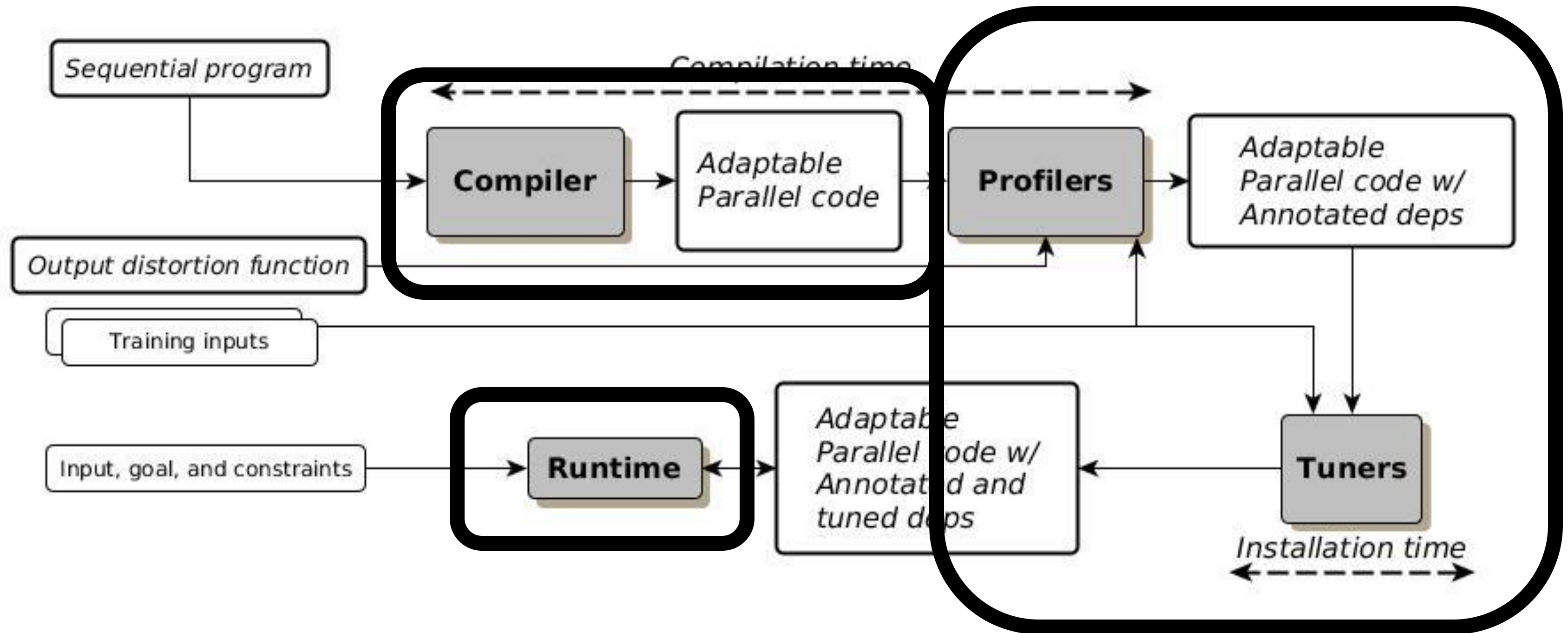


# Breakdown of overhead left



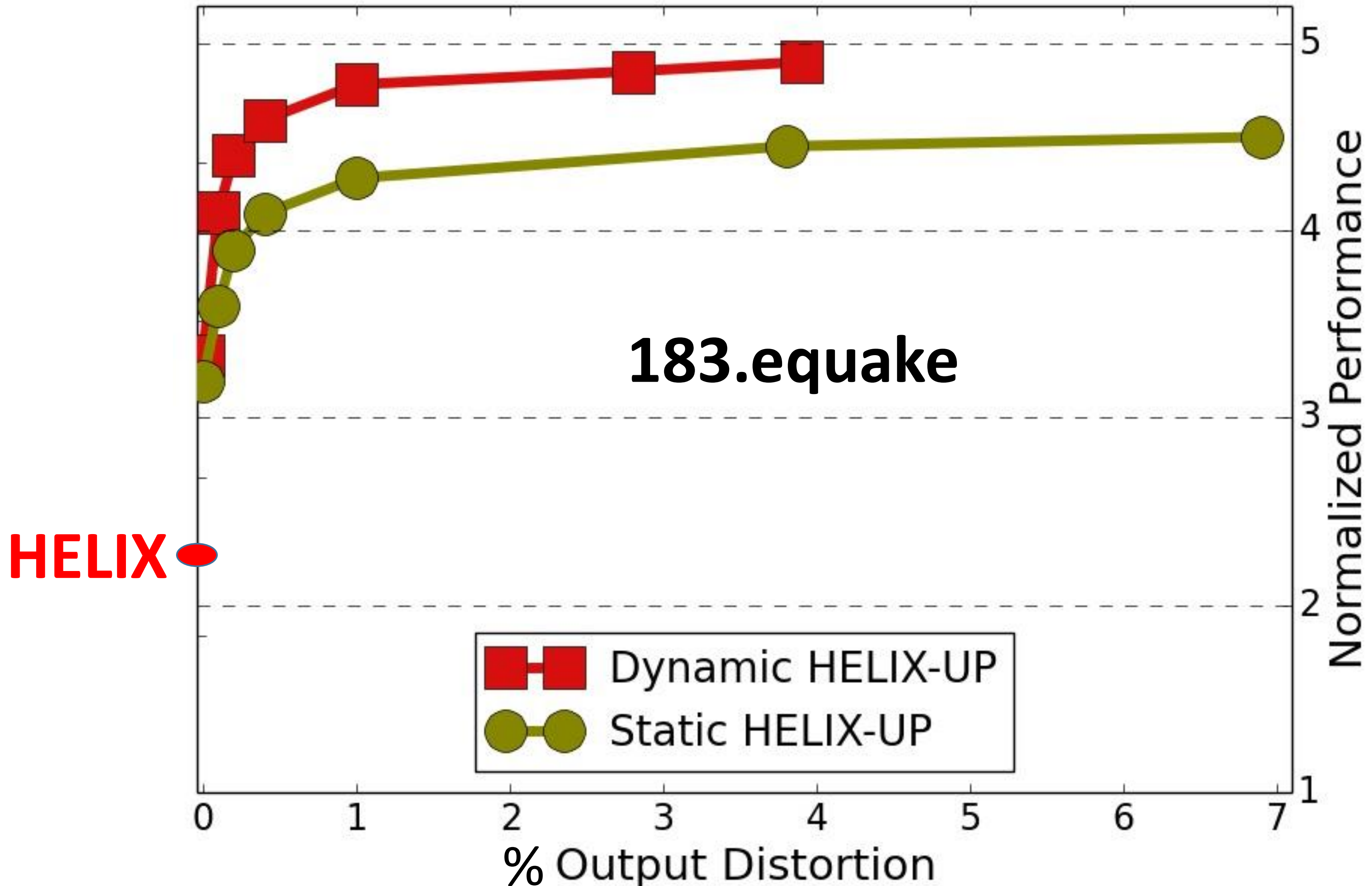
# How to adapt code?

# Which code to adapt?

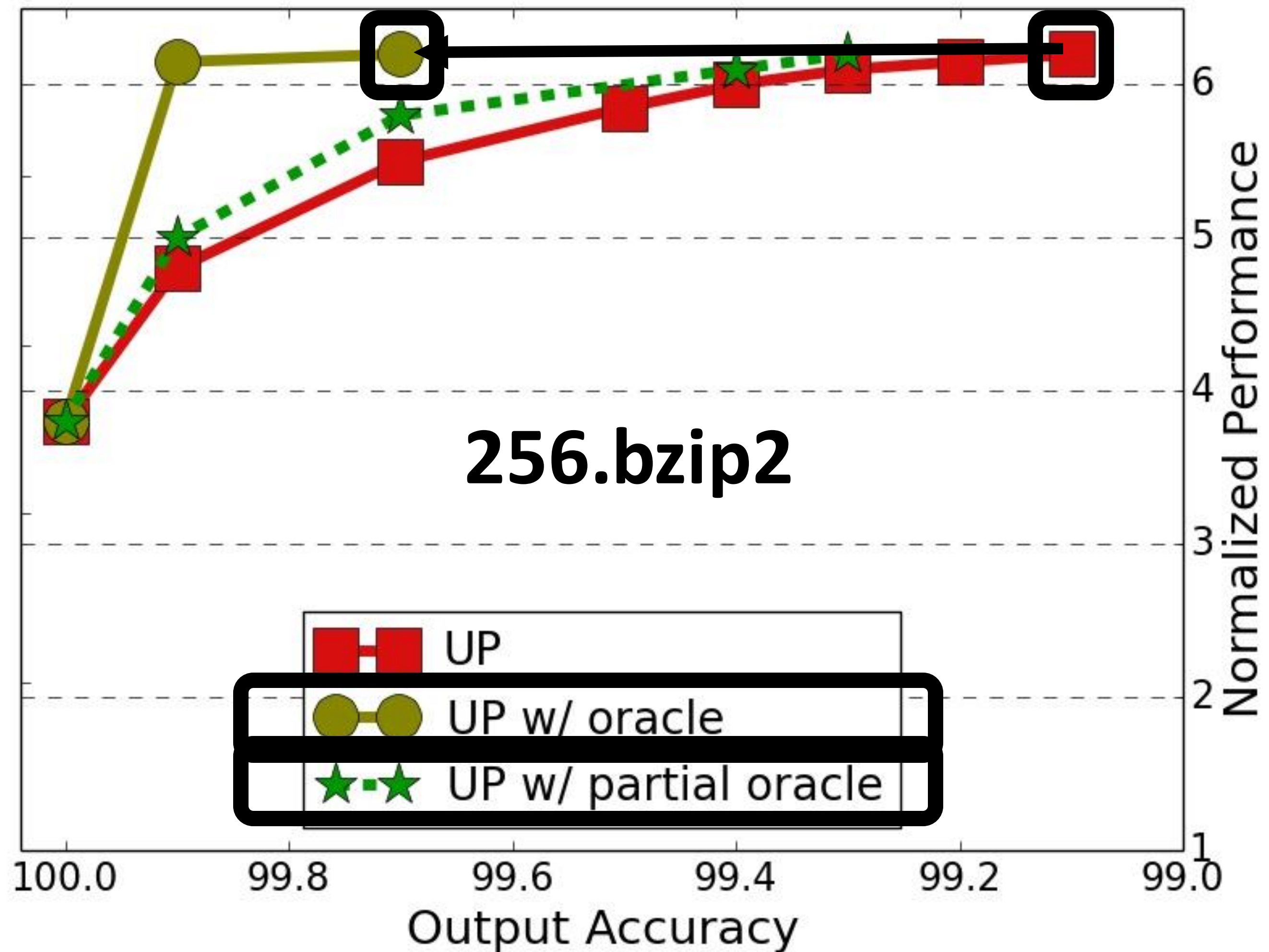



# When to adapt code?

# Setting knobs statically is good enough for regular workload



# HELIX-UP runtime is "good enough"



# HELIX-UP unblocks extra parallelism with small output distortions

# HELIX: no relaxing transformations

Hardware threads

HELIX

Static HELIX-UP

Cores

Output distortion: 3.8%

Program Speedup

0 2 4 6 8 10 12

177.mesa 179.art 183.equake 256.bzip2 blackscholes swaptions Geomean

Nehalem 6 cores
2 threads per core

# HELIX-UP unblocks extra parallelism with small output distortions

HELIX | Static HELIX-UP

Hardware threads

Cores

Program Speedup

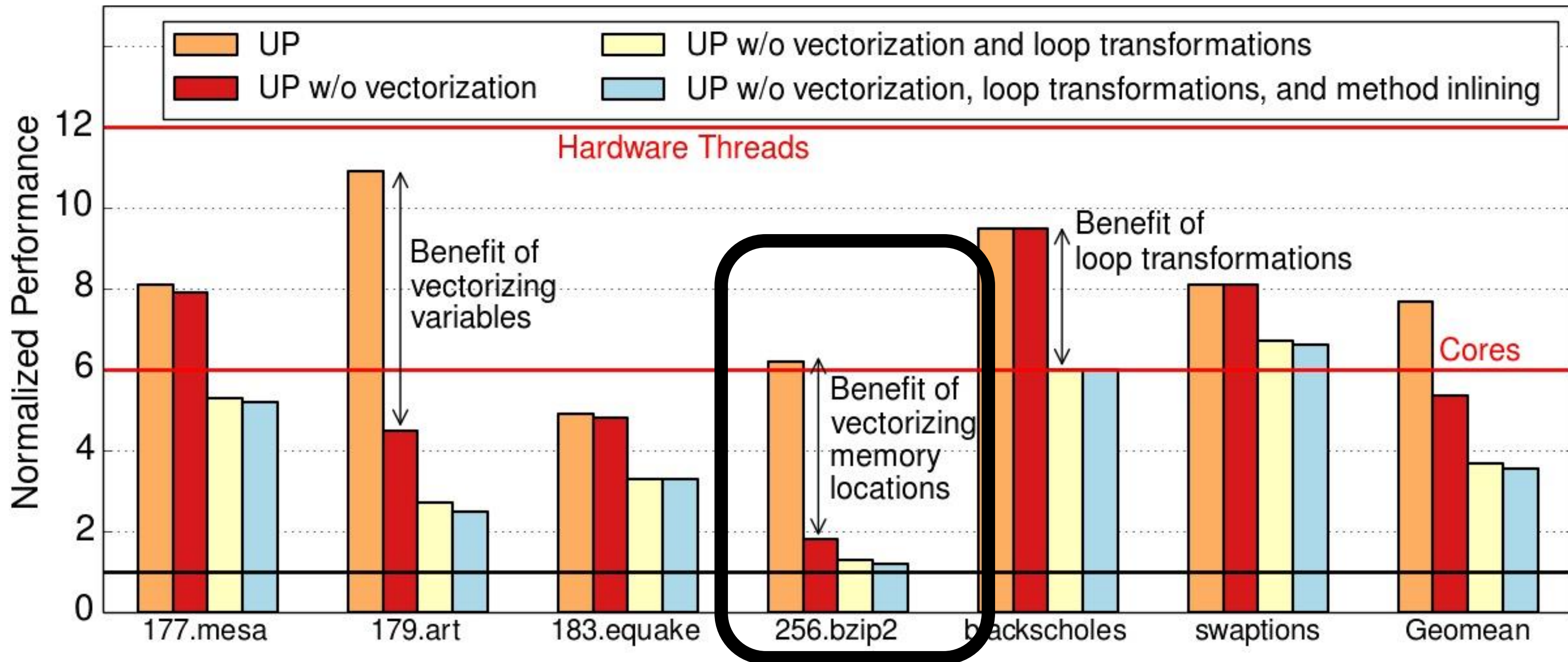
0 2 4 6 8 10 12

Output distortion

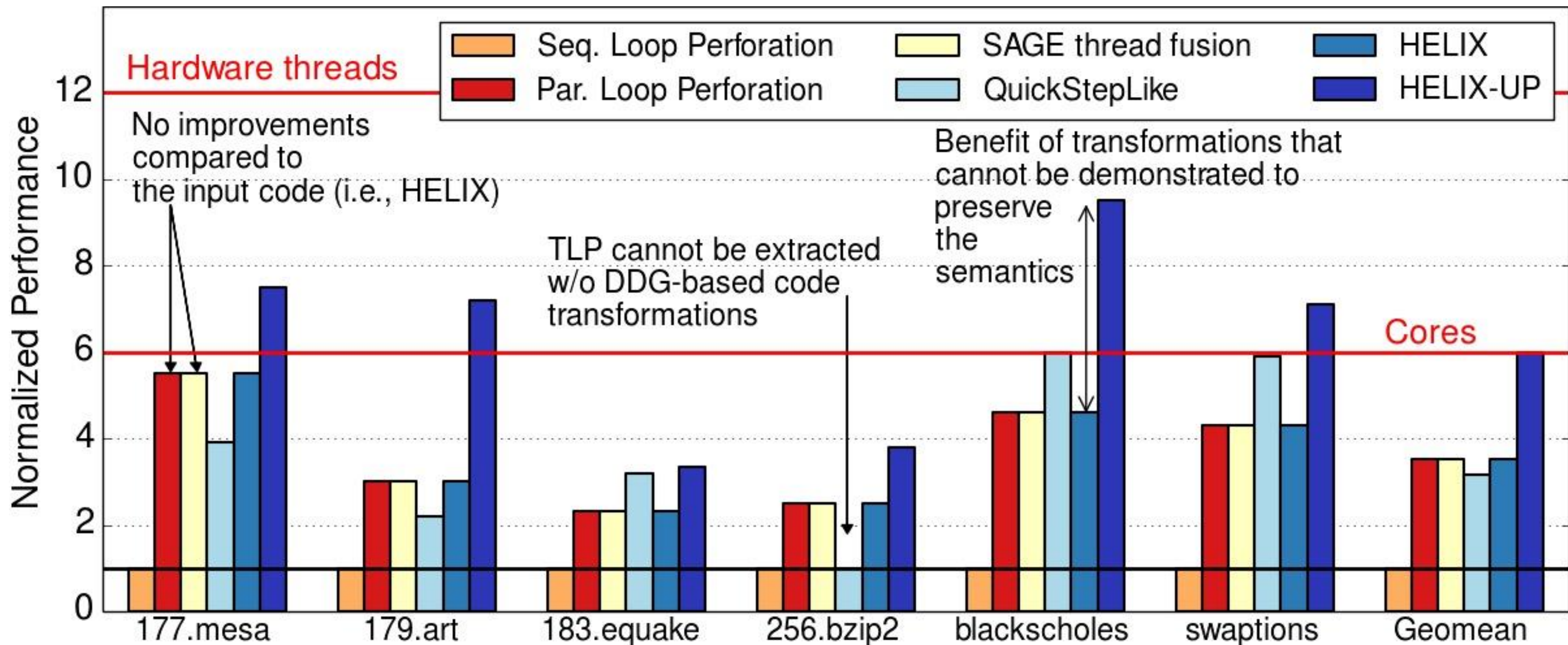
177.mesa: 2.1%
179.art: 3%
183.equake: 6.9%
256.bzip2: 3.8%
blackscholes: 0%
swaptions: 2%
Geomean

Nehalem 6 cores
2 threads per core

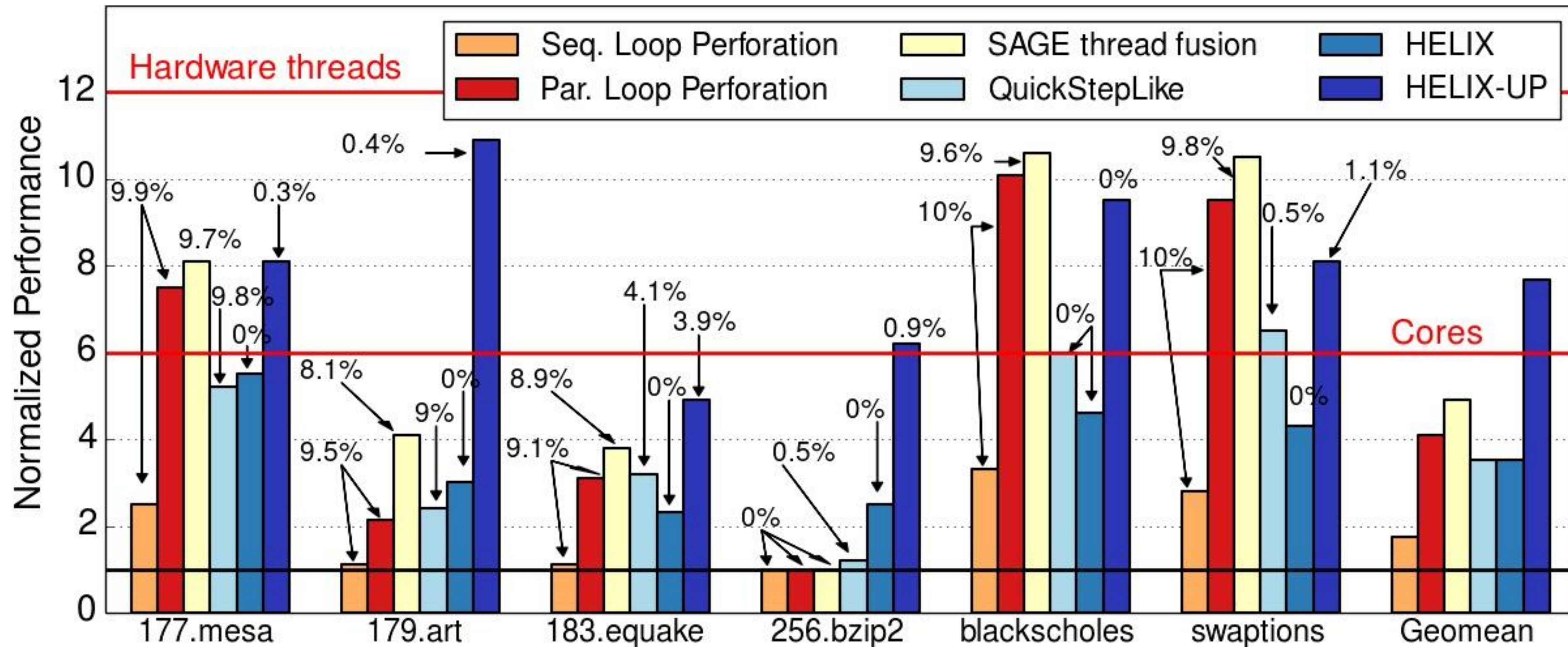
# Conventional transformations are still important



# HELIX-UP and Related Work with no output distortion



# HELIX-UP and Related Work with output distortion

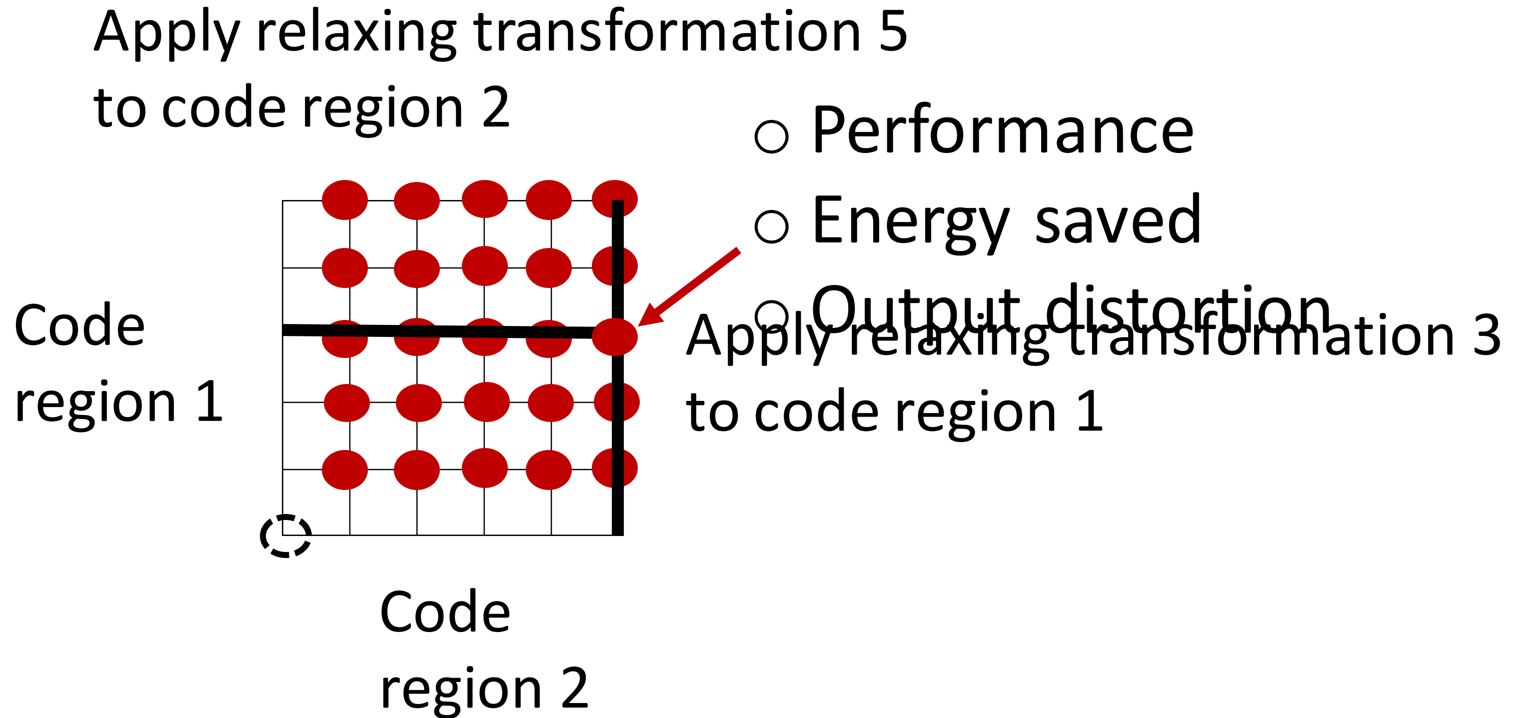


# Relaxed transformations remove performance bottlenecks

- Sequential bottleneck
  - A code region executed sequentially

  - A knob for each sequential code region

# Design space of HELIX-UP




1) User provides output distortion limits
2) System finds the best configuration
3) Run parallelized code with that configuration

# Pruning the design space

**Empirical observation:**
Transforming a code region
affects only the loop it belongs to

50 loops, 2 code regions per loop
2 transformations per code region

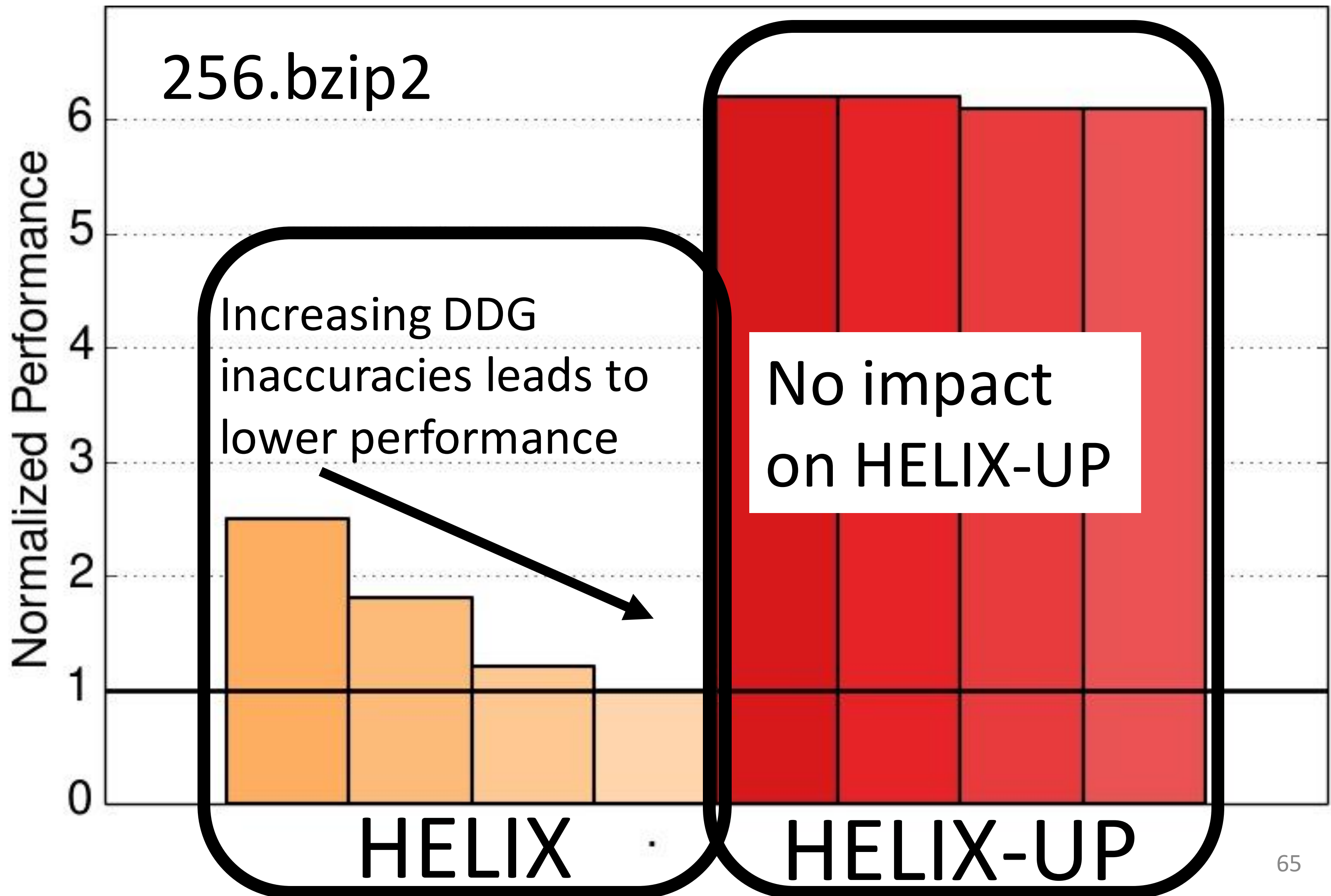
Complete space $= 2^{100}$
Pruned space $= 50 * (2^2) = 200$

**How well does HELIX-UP perform?**

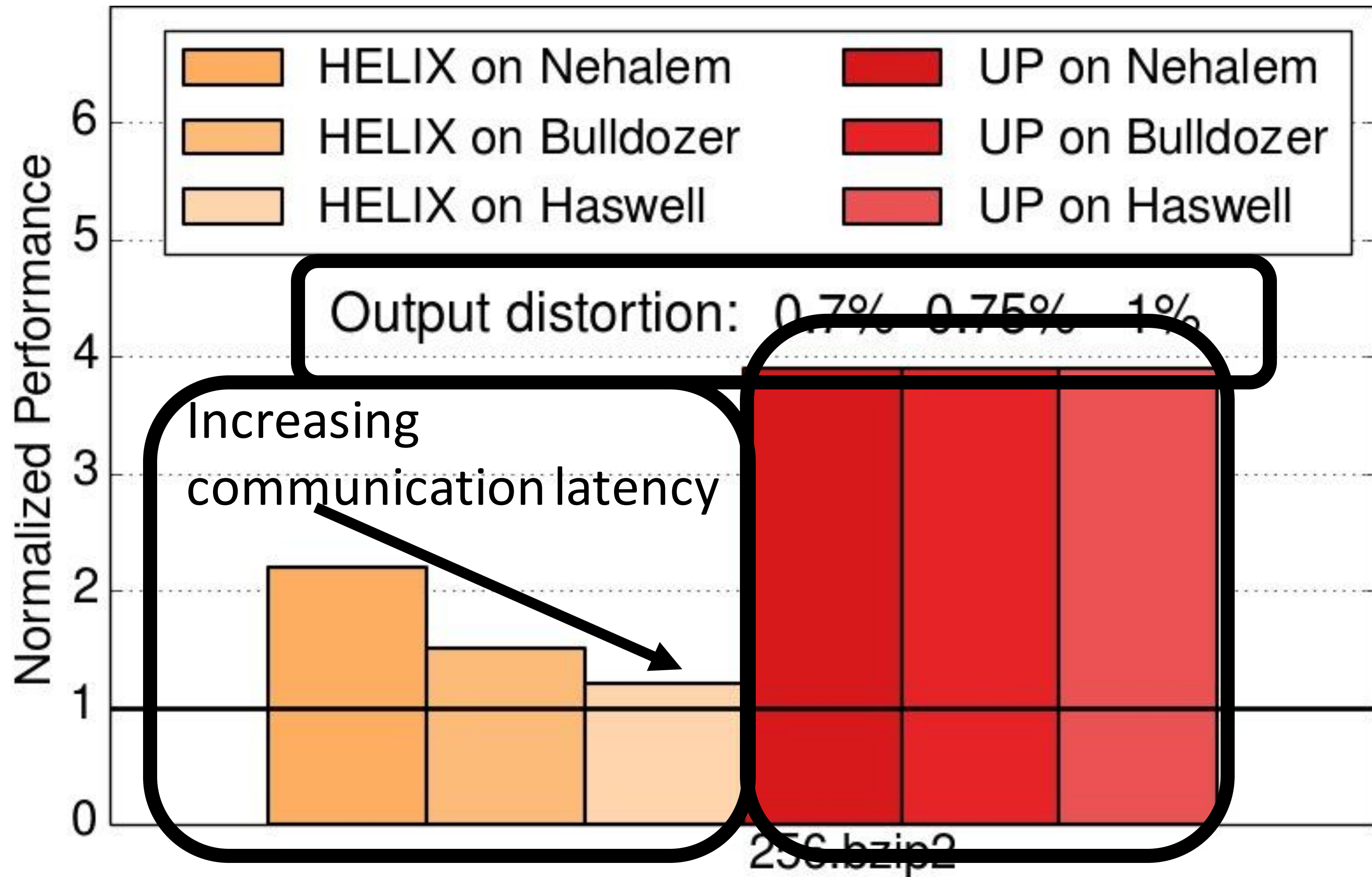
# HELIX-UP improves more than just performance

- Robustness to DDG inaccuracies
- Consistent performance across platforms

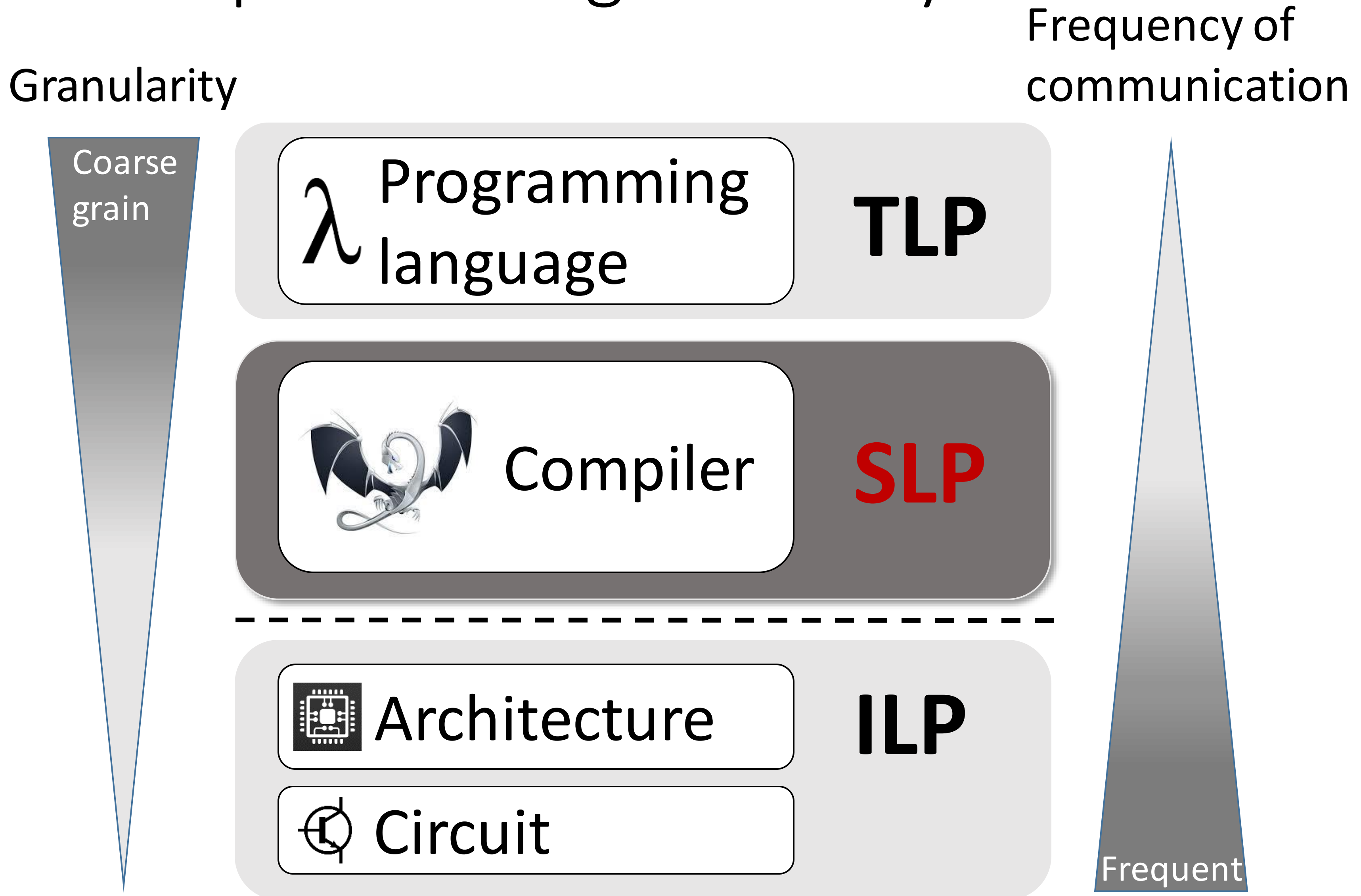
# Relaxed transformations to be robust to DDG inaccuracies



# Relaxed transformations for consistent performance

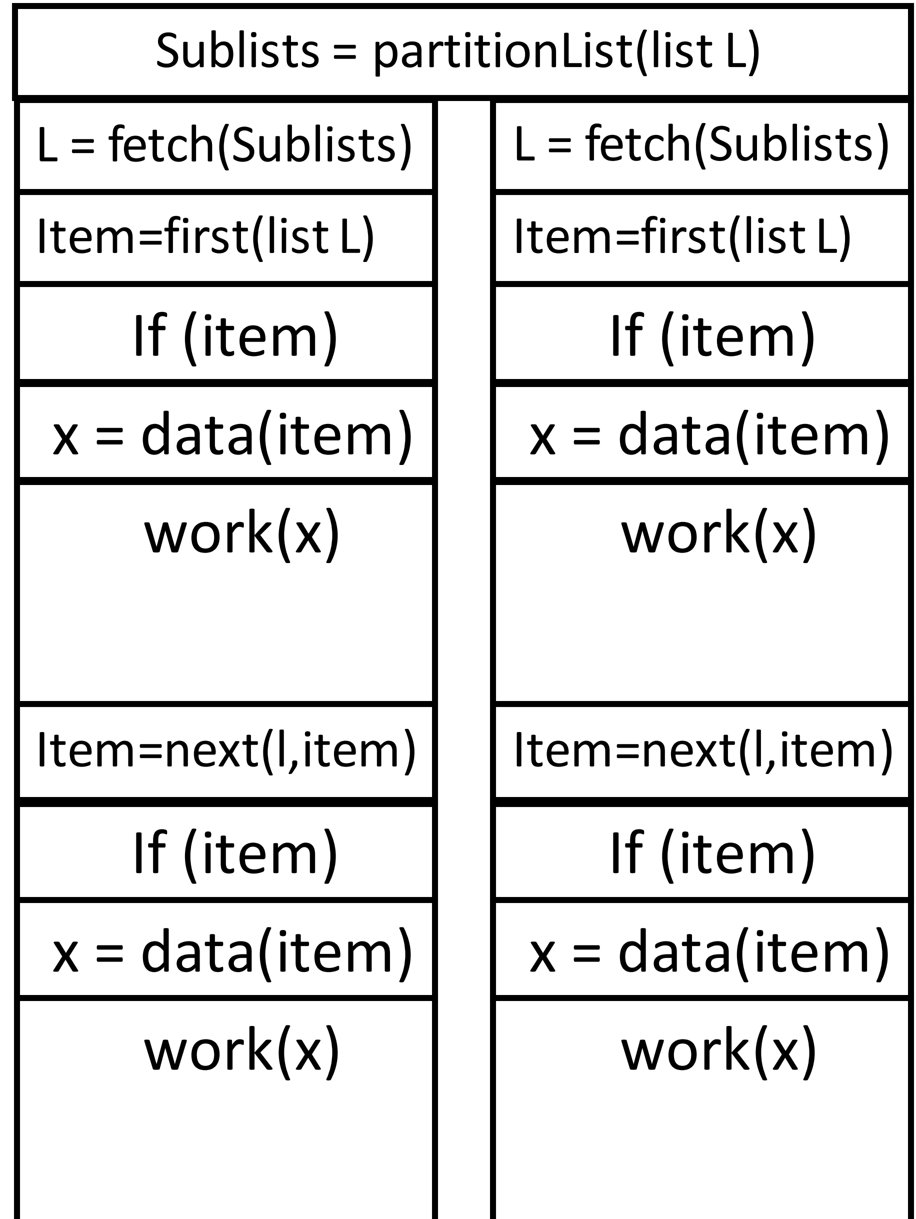


# Small Loop Parallelism:
# a compiler-level granularity



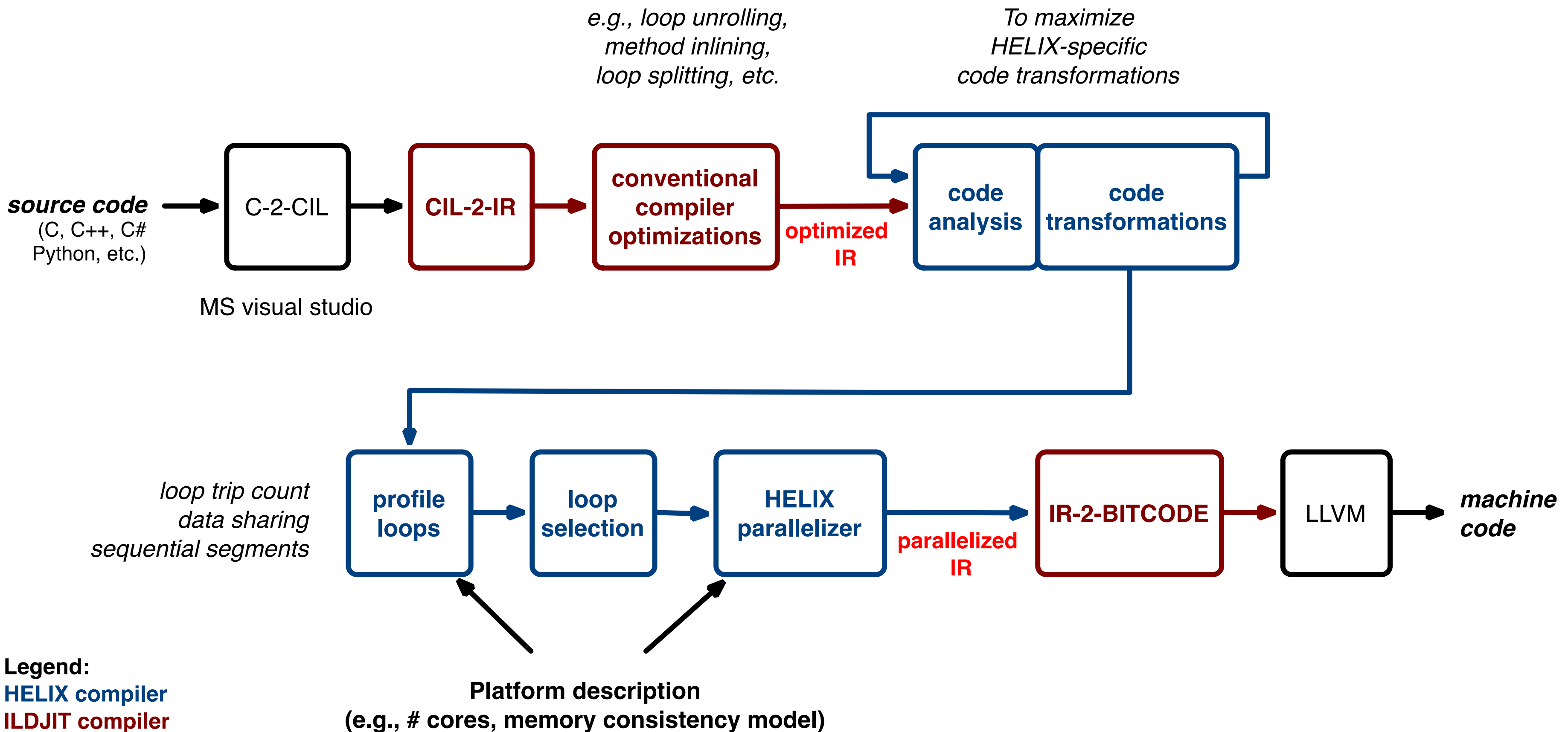
# Future work

```
Item = first(list L)
while (item){
  x = data(item)
  work(x)
  item = next(l, item)
}
```

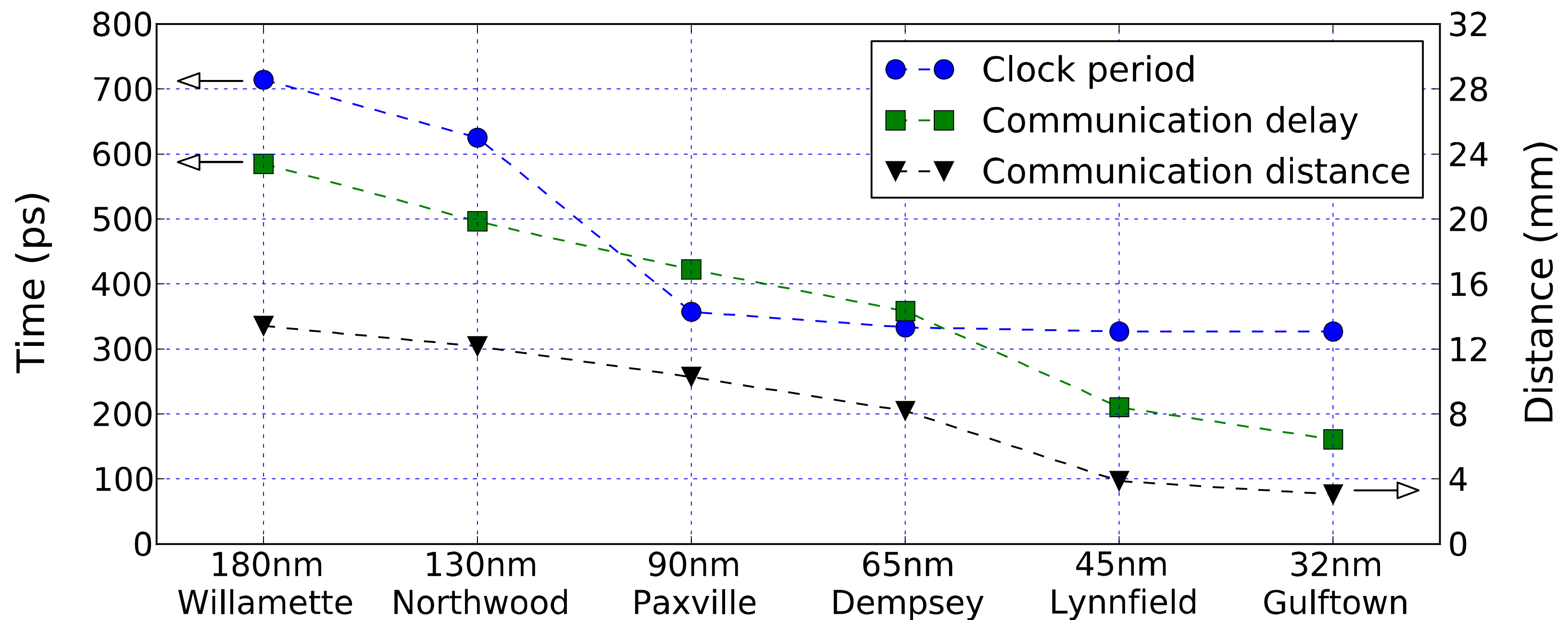


# The HELIX flow

**We've made many enhancements to existing code analysis, e.g., DDG, induction variables, etc.**

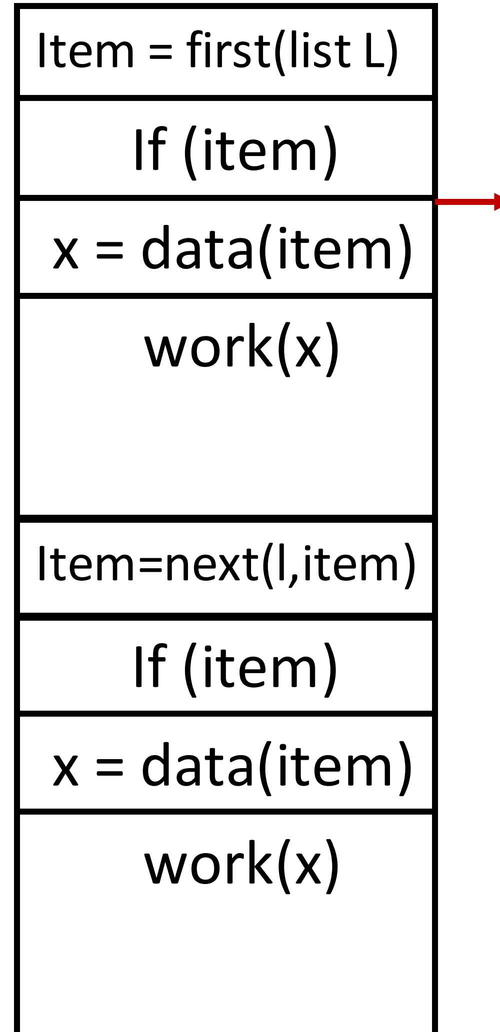


# Sims show single-cycle latency is possible for adjacent-core communication



# Future work: data-centric compiler

Item = first(list L)



# Simulator overview and accuracy

Workload (parallel x86)

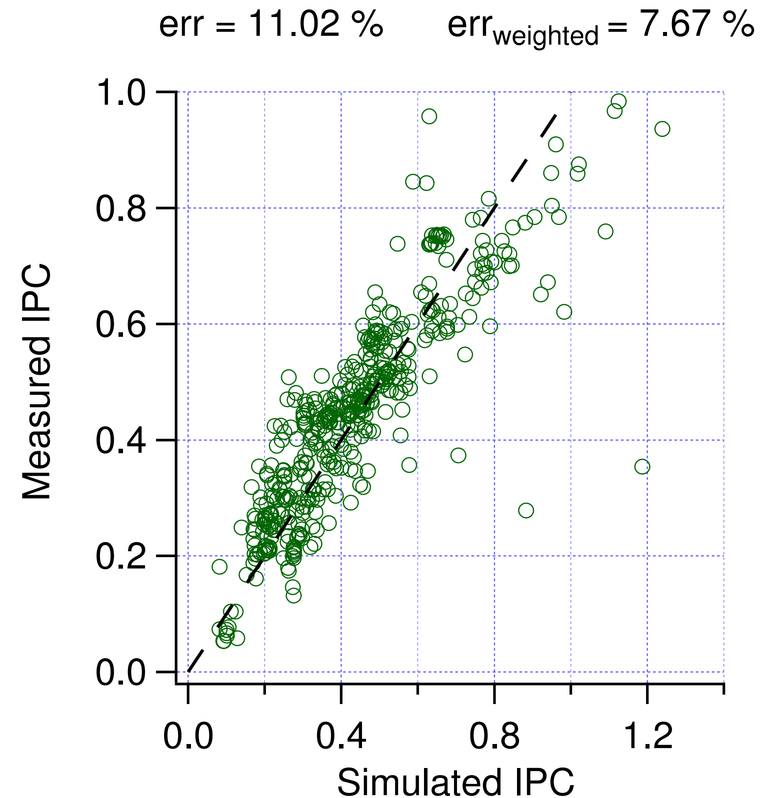
Pin VM

Host / OS

Core

RC

Un-core



# Brief description of SPEC2K benchmarks used in our experiments

**Non-numerical**

- gzip & bzip2
  - compression
- vpr & twolf
  - place & route CAD
- parser
  - Syntactic parser of English
  - XML?
- mcf
  - Combinatorial optimization; network flow, scheduling

**Numerical**

- art
  - Neural network simulation; image recognition; machine learning
- equake
  - Seismic wave propagation
- mesa
  - Emulates graphics on CPU
- ammp
  - Computational chemistry (e.g., atoms moving)

# Characteristics of parallelized benchmarks

| Benchmark | Phases | Parallel loop coverage | | |
|---|---|---|---|---|
| | | HELIX+RC | HELIX+ | HELIX |
| Integer benchmarks | | | | |
| 164.gzip | 12 | 98.2% | 42.3% | 42.3% |
| 175.vpr | 28 | 99.2% | 55.1% | 55.1% |
| 197.parser | 19 | 98.7% | 60.2% | 60.2% |
| 300.twolf | 18 | 99.99% | 62.4% | 62.4% |
| 181.mcf | 19 | 99.99% | 65.3% | 65.3% |
| 256.bzip2 | 23 | 99.99% | 72.3% | 72.1% |
| Floating point benchmarks | | | | |
| 183.equake | 7 | 99.3% | 99.3% | 77.1% |
| 179.art | 11 | 99.99% | 99.99% | 84.1% |
| 188.ammp | 23 | 99.99% | 99.99% | 60.2% |
| 177.mesa | 8 | 99.99% | 99.99% | 64.3% |