# Concurrent GCs and Modern Java Workloads: A Cache Perspective

Maria Carpen-Amarie[1] Georgios Vavouliotis[1]
Konstantinos Tovletoglou[1] Boris Grot[1,2] Rene Mueller[1]

[1]Huawei Zurich Research Center

[2]University of Edinburgh

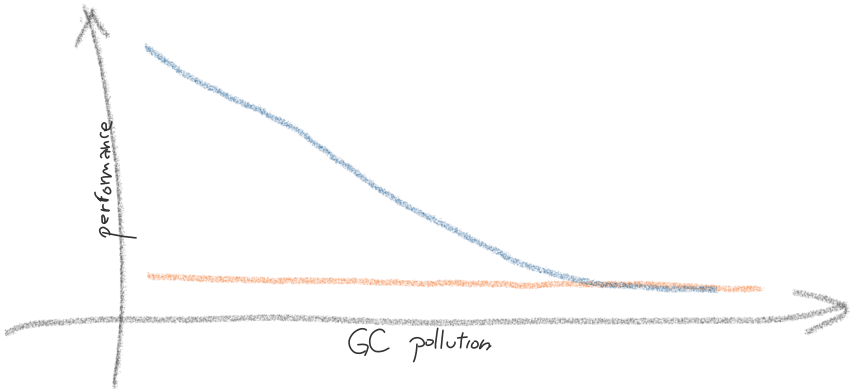
June 18, 2023

# There are two kinds of applications...

- Intuition: Concurrent garbage collection harms performance through cache pollution

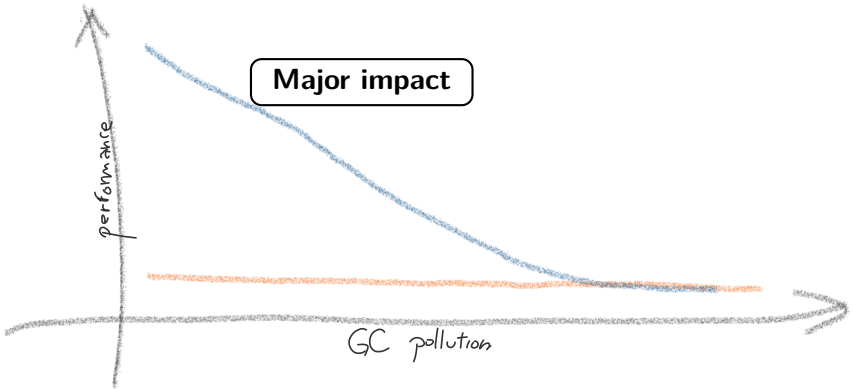
# There are two kinds of applications...

- Intuition: Concurrent garbage collection harms performance through cache pollution
- We found:



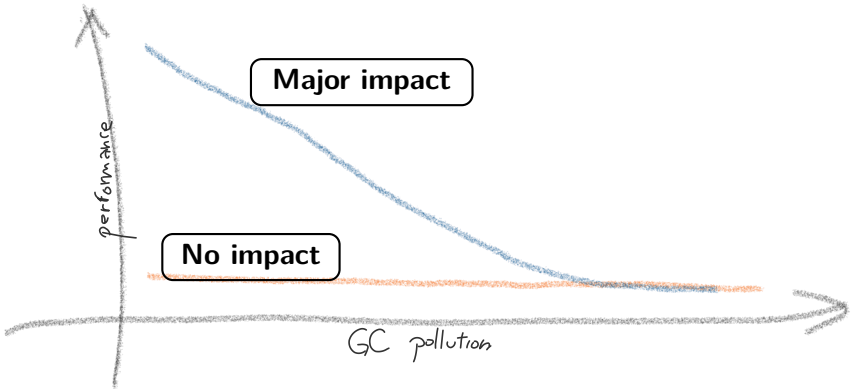
# There are two kinds of applications...

- Intuition: Concurrent garbage collection harms performance through cache pollution
- We found:



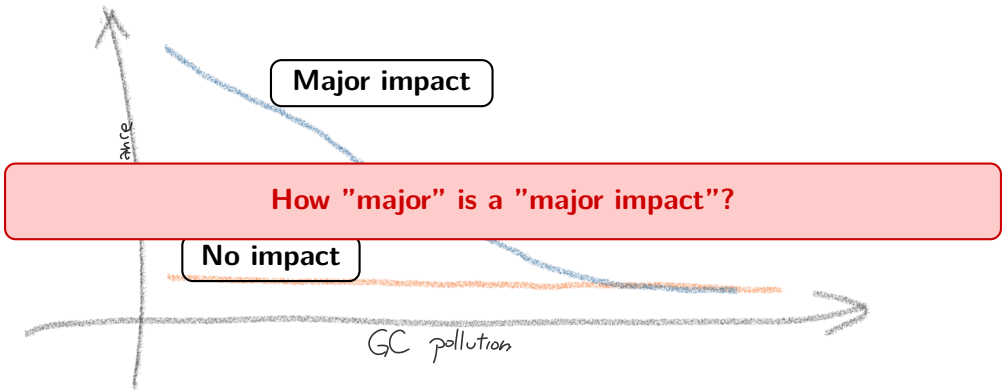
# There are two kinds of applications...

- Intuition: Concurrent garbage collection harms performance through cache pollution
- We found:



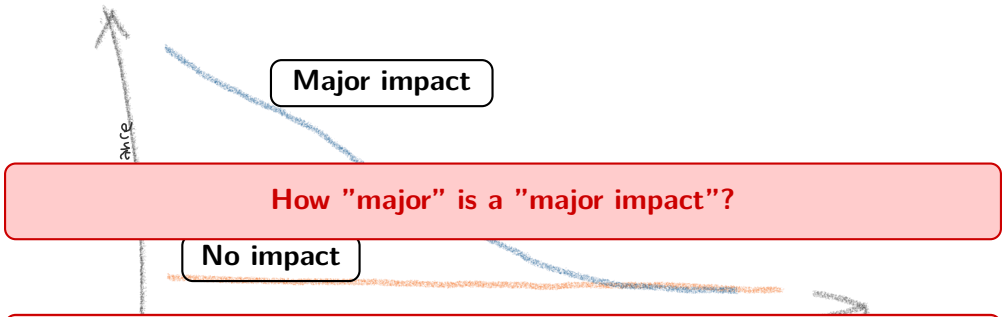
# There are two kinds of applications...

- Intuition: Concurrent garbage collection harms performance through cache pollution
- We found:



# There are two kinds of applications...

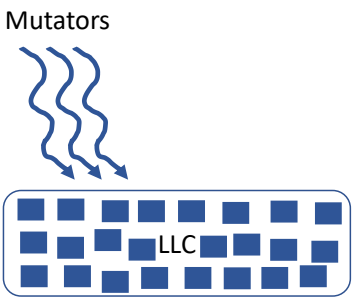
- Intuition: Concurrent garbage collection harms performance through cache pollution
- We found:



# Overview

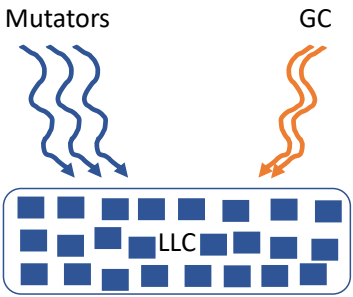
# Overview

# Motivation



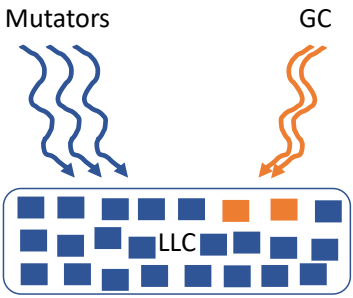

- Everything is multicore

# Motivation



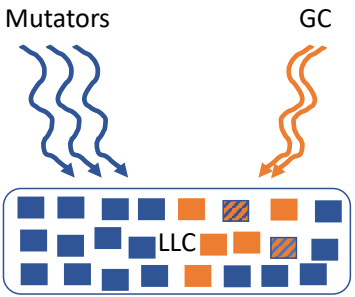

- Everything is multicore

# Motivation



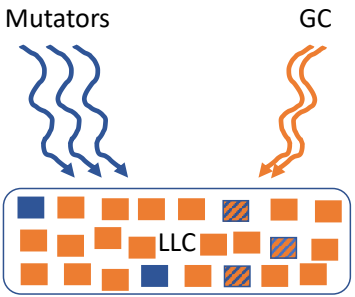

- Everything is multicore
- Last-level cache (LLC): contention point

# Motivation



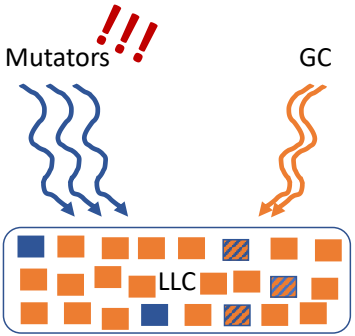

- Everything is multicore
- Last-level cache (LLC): contention point
- Application working-set != GC live-set

# Motivation



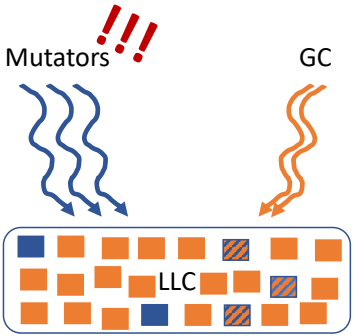

- Everything is multicore
- Last-level cache (LLC): contention point
- Application working-set != GC live-set

# Motivation




- Everything is multicore
- Last-level cache (LLC): contention point
- Application working-set != GC live-set

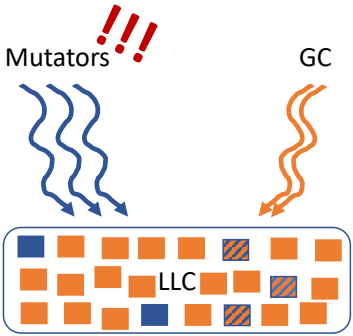
# Motivation




- Everything is multicore
- Last-level cache (LLC): contention point
- Application working-set != GC live-set

**Hypothesis: There *must* be a (measurable) impact from GC cache pollution.**

# Motivation




- Everything is multicore
- Last-level cache (LLC): contention point
- Application working-set != GC live-set

**Hypothesis: There *must* be a (measurable) impact from GC cache pollution.**

...was what we initially thought

# Not Easy to Quantify Though

**Challenges**

- The GC cannot be simply turned off
- Concurrent $\Rightarrow$ cannot isolate it either
- Mutators stick their noses in GC work
- Benchmarks – not ideal

# Not Easy to Quantify Though

**Challenges**

- The GC cannot be simply turned off
- Concurrent ⇒ cannot isolate it either
- Mutators stick their noses in GC work
- Benchmarks – not ideal

**Methodology**

- Choose benchmarks with iterations

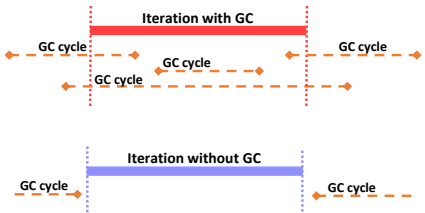
# Not Easy to Quantify Though

## Challenges

- The GC cannot be simply turned off
- Concurrent ⇒ cannot isolate it either
- Mutators stick their noses in GC work
- Benchmarks – not ideal

## Methodology

- Choose benchmarks with iterations



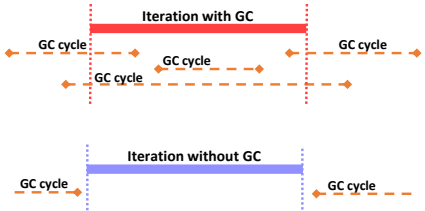
# Not Easy to Quantify Though

## Challenges

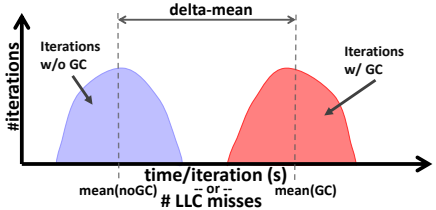
- The GC cannot be simply turned off
- Concurrent ⇒ cannot isolate it either
- Mutators stick their noses in GC work
- Benchmarks – not ideal

## Methodology

- Choose benchmarks with iterations

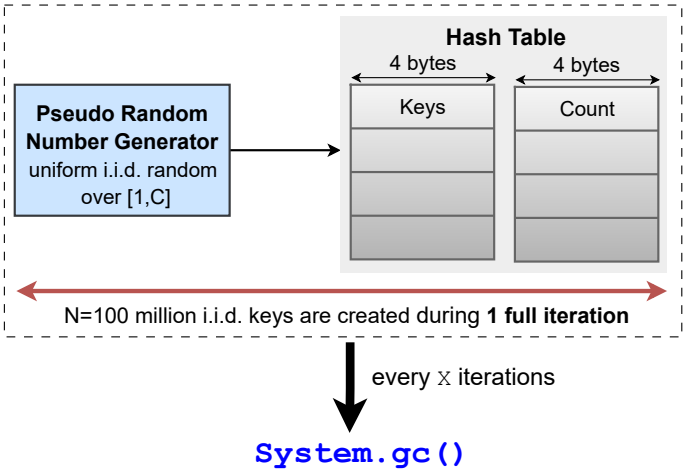



- Quantify based on iteration distribution

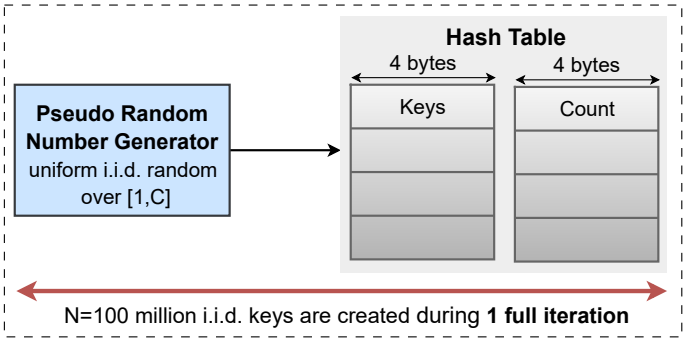


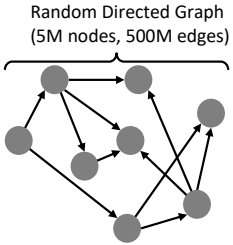
# Overview

# Micro-Benchmark: Controlled Environment

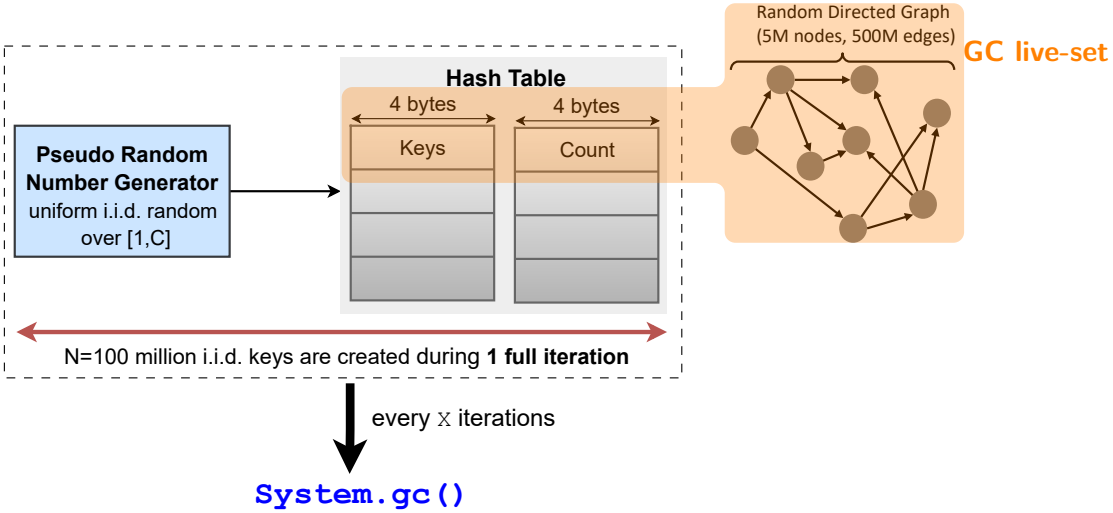


# Micro-Benchmark: Controlled Environment

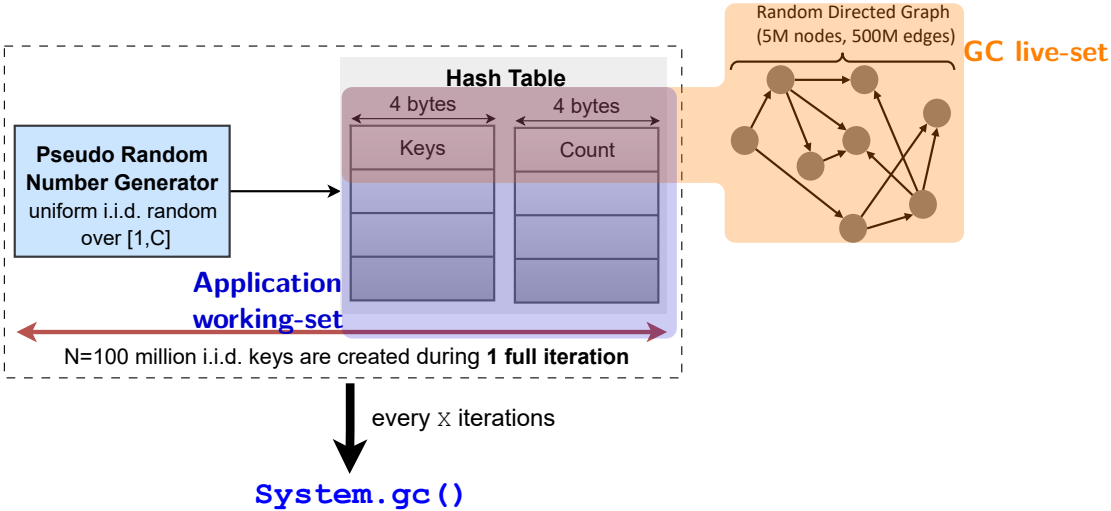


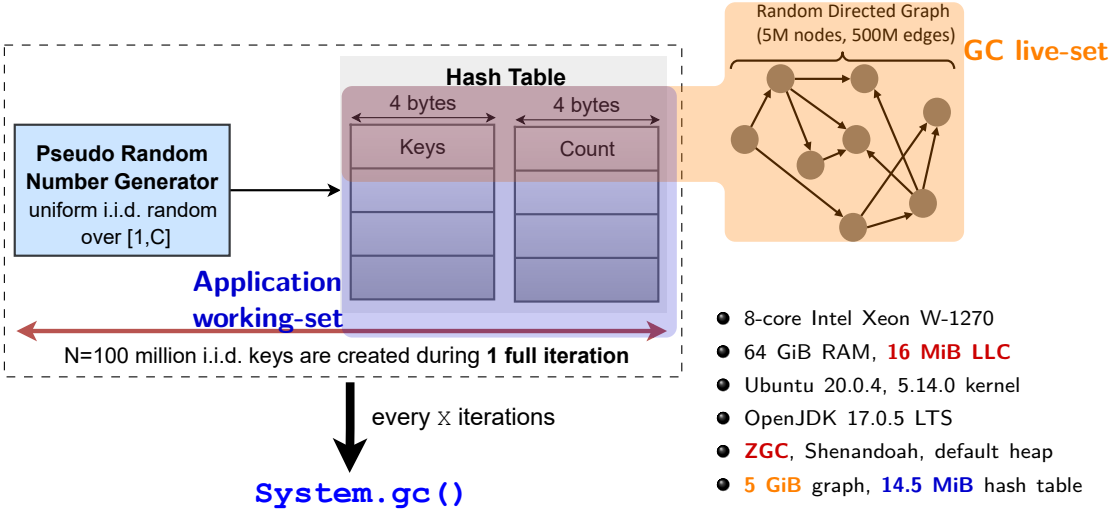
# Micro-Benchmark: Controlled Environment



# Micro-Benchmark: Controlled Environment

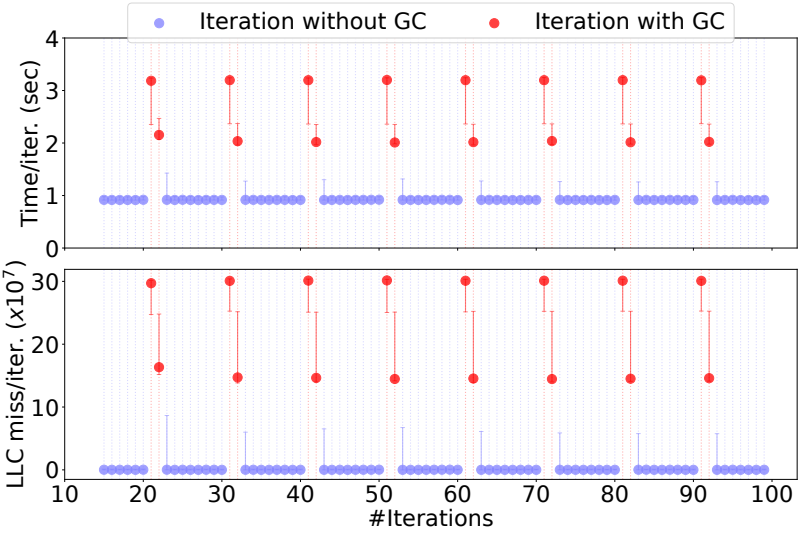


# Micro-Benchmark: Controlled Environment

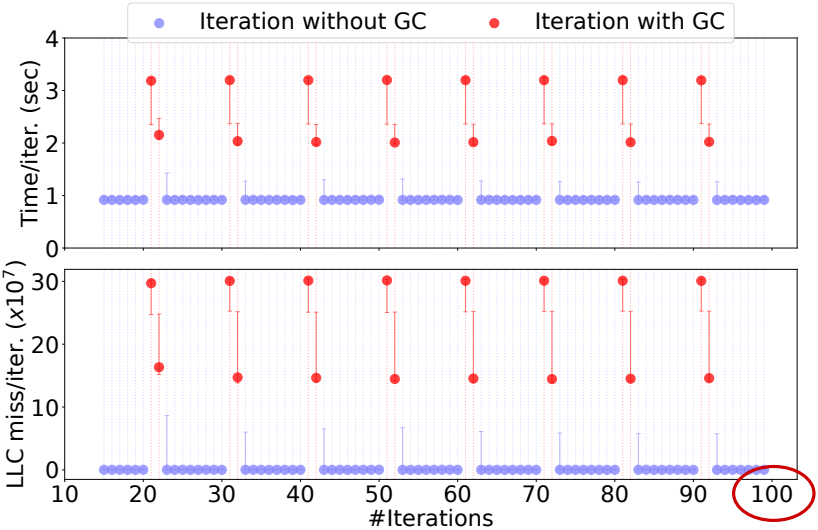



- 8-core Intel Xeon W-1270
- 64 GiB RAM, **16 MiB LLC**
- Ubuntu 20.0.4, 5.14.0 kernel
- OpenJDK 17.0.5 LTS
- **ZGC**, Shenandoah, default heap
- **5 GiB** graph, **14.5 MiB** hash table

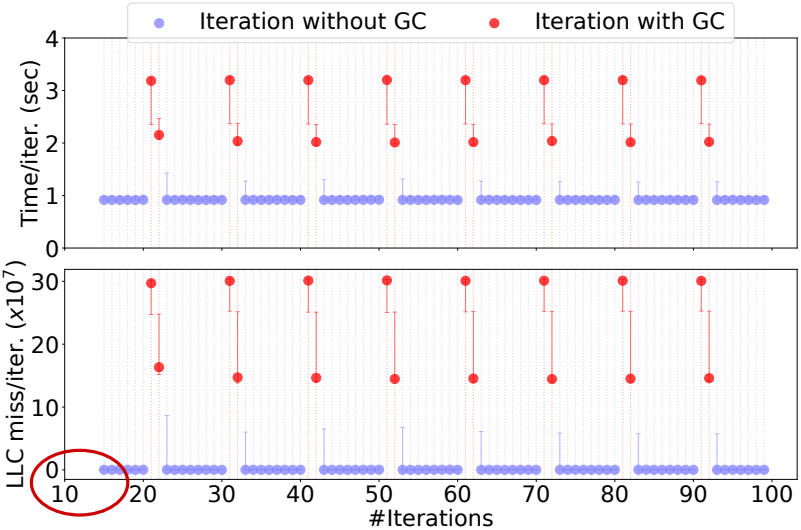
# Results


Iteration without GC
Iteration with GC
Time/iter. (sec)
LLC miss/iter. (x10^7)
#Iterations
0
1
2
3
4
10
20
30
40
50
60
70
80
90
100

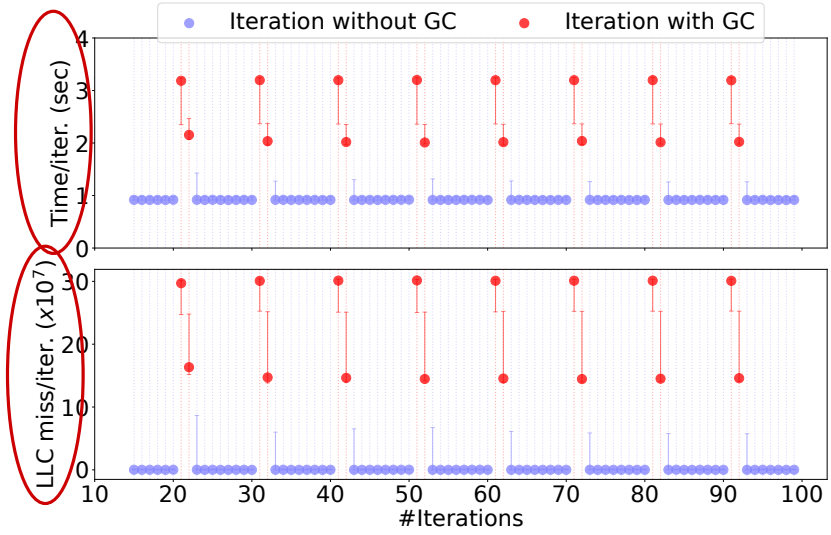
# Results


Iteration without GC
Iteration with GC
Time/iter. (sec)
LLC miss/iter. (x10^7)
#Iterations
0
1
2
3
4
10
20
30
40
50
60
70
80
90
100

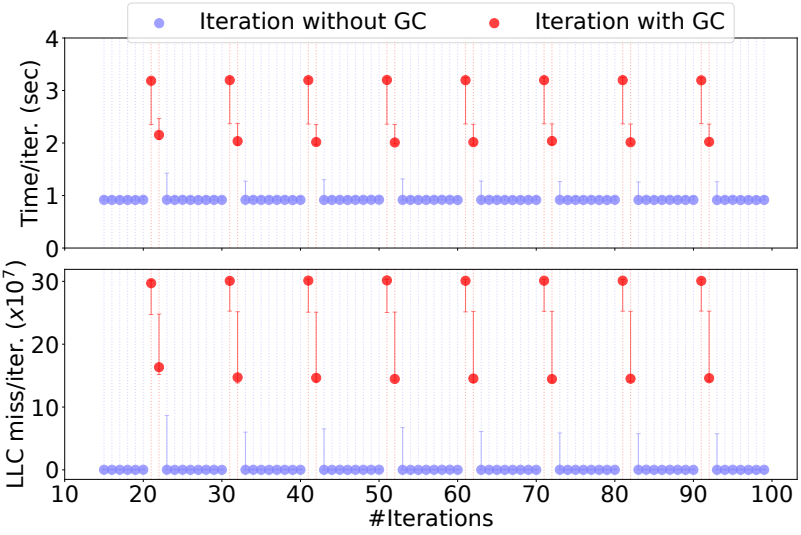
# Results


Iteration without GC
Iteration with GC
Time/iter. (sec)
LLC miss/iter. (x10^7)
#Iterations
0
1
2
3
4
0
10
20
30
10
20
30
40
50
60
70
80
90
100

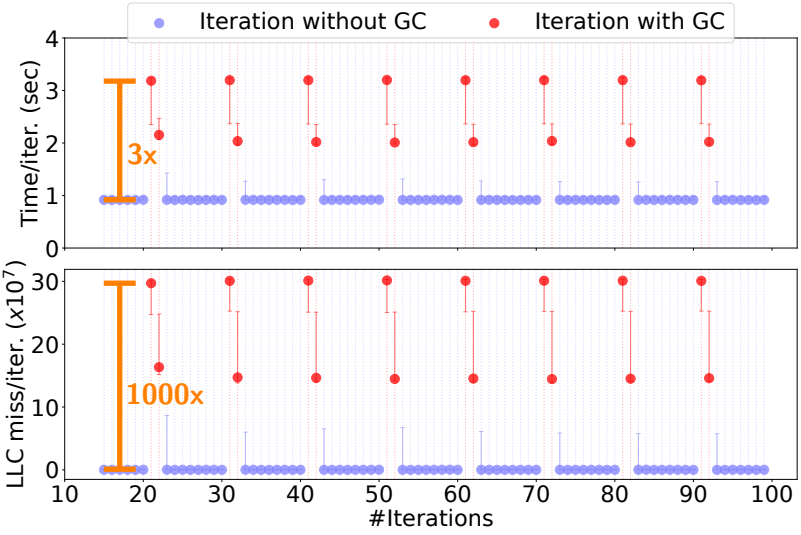
# Results


Iteration without GC
Iteration with GC
Time/iter. (sec)
LLC miss/iter. (x10^7)
#Iterations

# Results


Iteration without GC
Iteration with GC
Time/iter. (sec)
LLC miss/iter. (x10^7)
#Iterations
0
1
2
3
4
10
20
30
40
50
60
70
80
90
100

# Results


Iteration without GC
Iteration with GC
Time/iter. (sec)
3x
LLC miss/iter. (x10^7)
1000x
#Iterations

# Results


**Major impact on both LLC misses and execution time, consistent across runs.**

# Overview

# Benchmark Selection and Setup

**Renaissance**

- 23 applications
- Representative for modern workloads
- Variety: spark, concurrency, database, functional, scala, web
- Iteration-based benchmarks

**Experimental Setup**

- Disable proactive GC
- Disable explicit GC between iterations
- 10-minute runs
- Drop first half of iterations

# Benchmark Selection and Setup

**Renaissance**

- 23 applications
- Representative for modern workloads
- Variety: spark, concurrency, database, functional, scala, web
- Iteration-based benchmarks

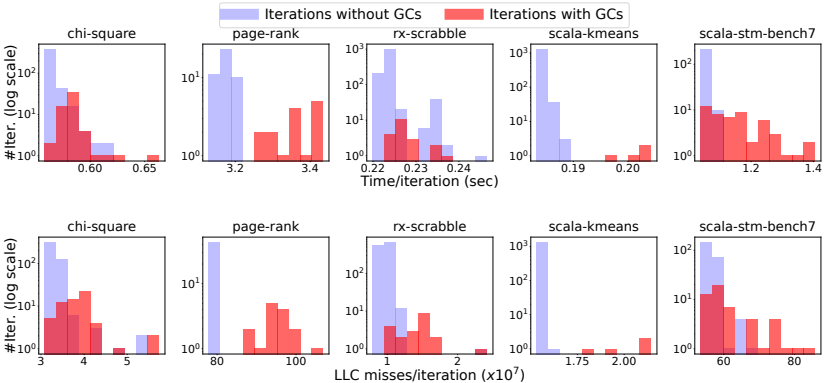
**Experimental Setup**

- Disable proactive GC
- Disable explicit GC between iterations
- 10-minute runs
- Drop first half of iterations

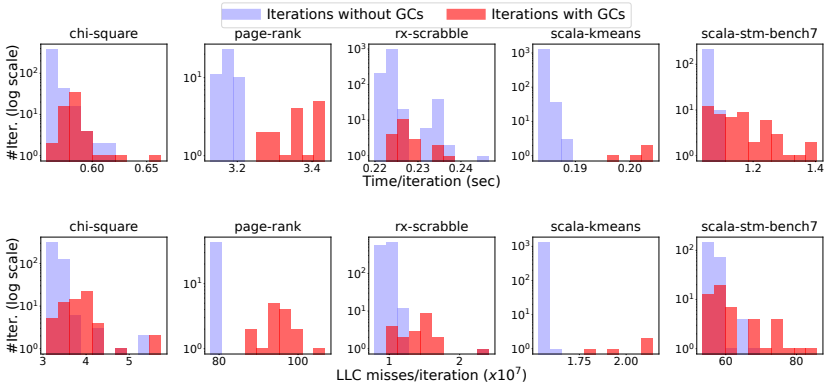
**Some stats:**
**3/23** benchmarks: $< 2\%$ of iterations overlap with GC cycles
**2/23** benchmarks: $> 98\%$ of iterations overlap with GC cycles
**5/23** benchmarks: **correlation** between GC cache pollution and performance drop*

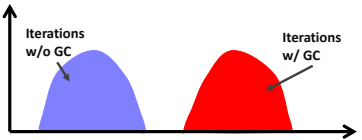
*More details on the statistical analysis in the paper.

# Results: And the Winners Are...

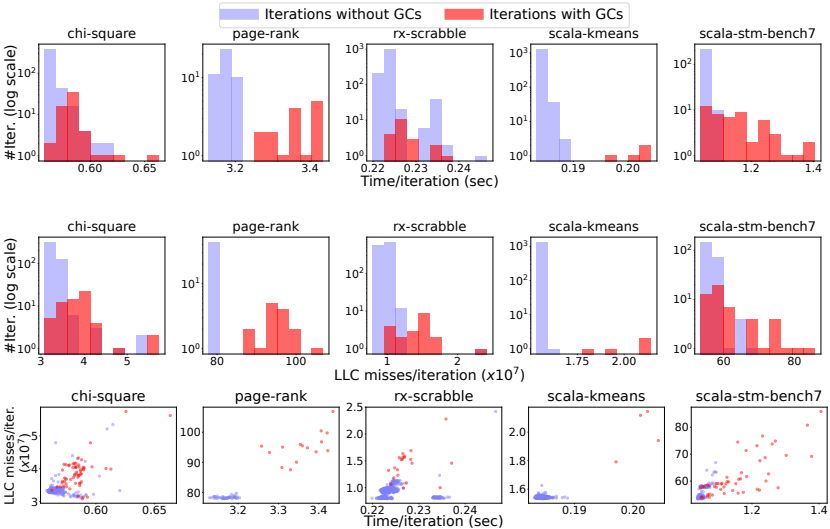


# Results: And the Winners Are...

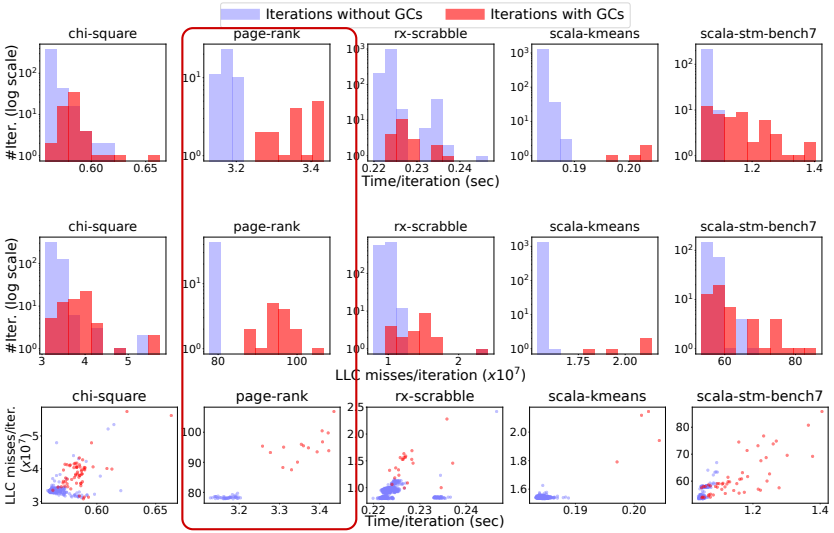


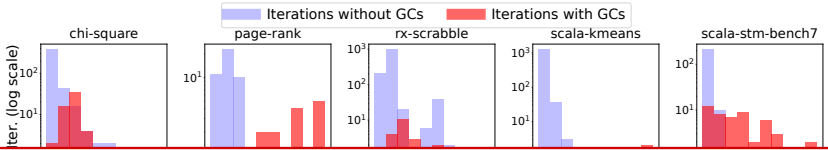
# Results: And the Winners Are...



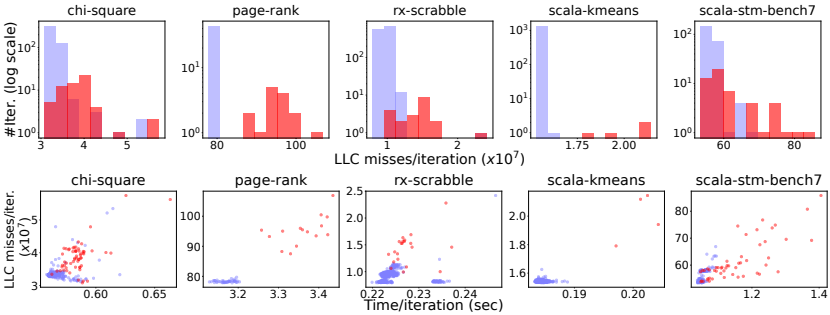
# Results: And the Winners Are...



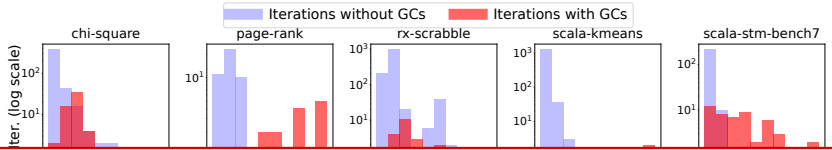
# Results: And the Winners Are...



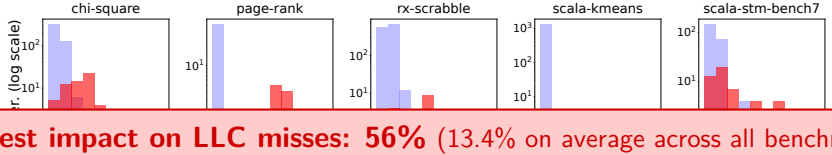

**Highest performance overhead: 10%** (2.8% on average across all benchmarks)



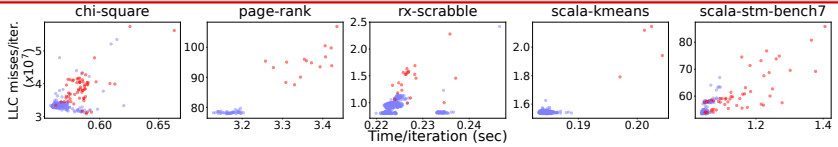
# Results: And the Winners Are...




**Highest performance overhead: 10%** (2.8% on average across all benchmarks)

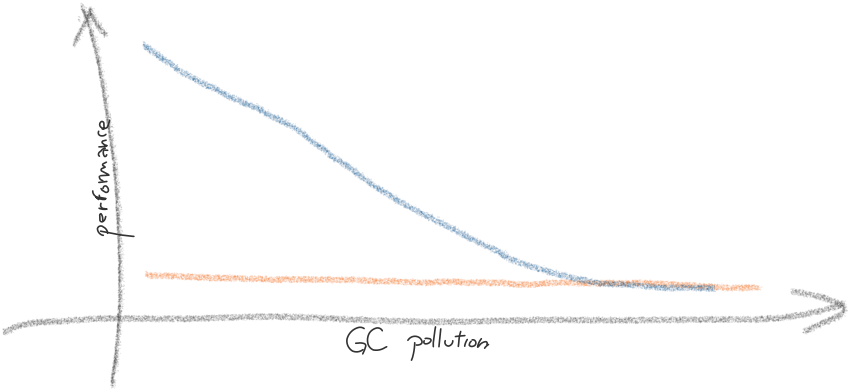



**Highest impact on LLC misses: 56%** (13.4% on average across all benchmarks)

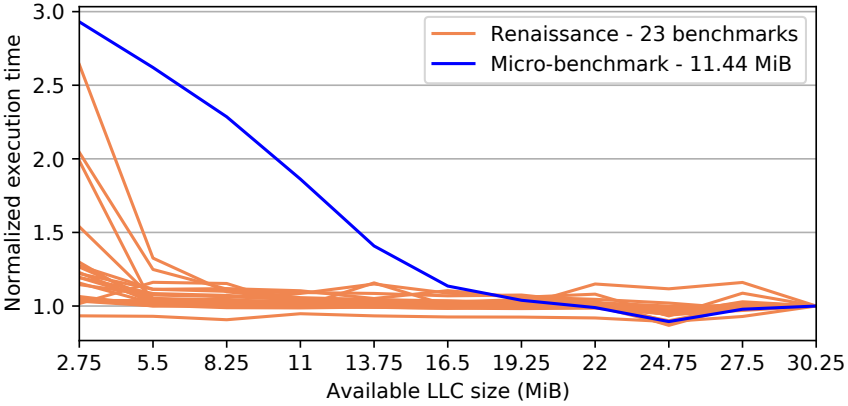


# Overview

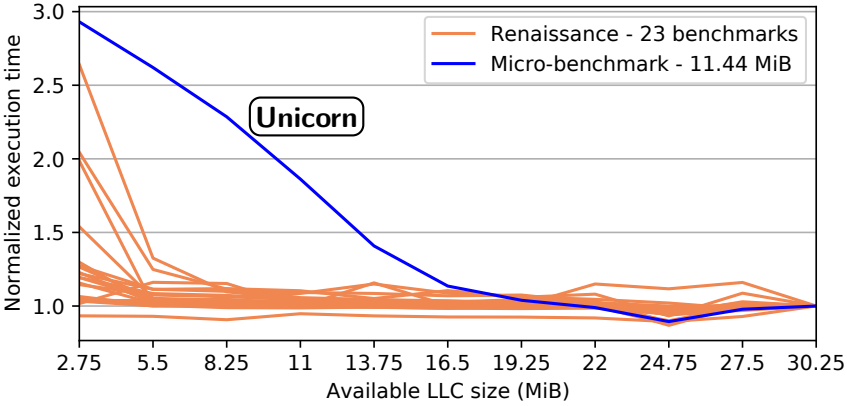
# What's the Catch?



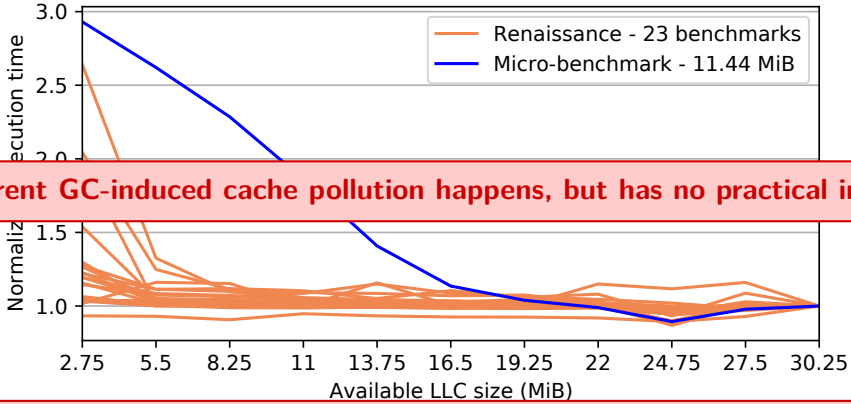
# What's the Catch?



# What's the Catch?



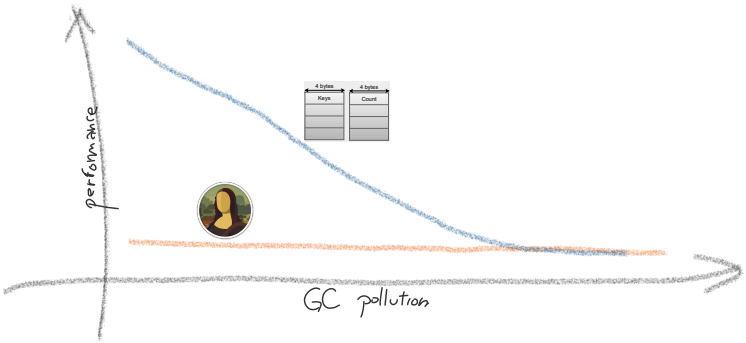
# What's the Catch?




**Concurrent GC-induced cache pollution happens, but has no practical impact...**

**...because real-world Java workloads are typically not cache-sensitive**

# Thank you!




Any questions?