Geekbang> 极客邦科技 InfoQ

QCon 全球软件开发大会【北京站】2016

OS-caused Long JVM Pauses - Deep Dive and Solutions

Zhenyun Zhuang

LinkedIn Corp., Mountain View, California, USA
https://www.linkedin.com/in/zhenyun
Zhenyun@gmail.com

International Software Development Conference

QCon

2016.10.20~22
上海·宝华万豪酒店

全球软件开发大会2016

[上海站]

购票热线: 010-64738142
会务咨询: qcon@cn.infoq.com
赞助咨询: sponsor@cn.infoq.com
议题提交: speakers@cn.infoq.com
在线咨询（QQ）: 1173834688

团·购·享·受·更·多·优·惠

7折 优惠（截至06月21日）
现在报名，立省2040元/张

Outline

- Introduction
- Background
- Scenario 1: startup state
- Scenario 2: steady state with memory pressure
- Scenario 3: steady state with heavy IO
- Lessons learned

Introduction

- Java + Linux
  - Java is popular in production deployments
  - Linux features interact with JVM operations
  - Unique challenges caused by concurrent applications
- Long JVM pauses caused by Linux OS
  - Production issues, in three scenarios
  - Root causes
  - Solutions
- References
  - *Ensuring High-performance of Mission-critical Java Applications in Multi-tenant Cloud Platforms*, IEEE Cloud 2014
  - *Eliminating Large JVM GC Pauses Caused by Background IO Traffic*, LinkedIn Engineering Blog, 2016 (Too many tweets bringing down a twitter server! :)

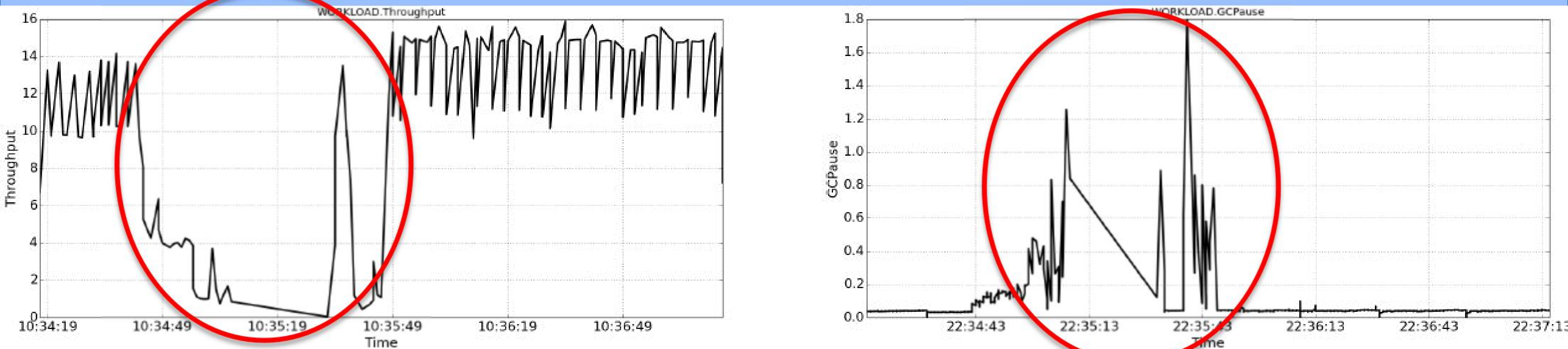
Background

- JVM and Heap
  - Oracle HotSpot JVM
- Garbage collection
  - Generations
  - Garbage collectors
- Linux OS
  - Paging (Regular page, Huge page)
  - Swapping (Anonymous memory)
  - Page cache writeback (Batched, Periodic)

Scenarios

- Three scenarios
  - Startup state
  - Steady state with memory pressure
  - Steady state with heavy IO
- Workload
  - Java application keeps allocating/de-allocating objects
  - Background applications taking memories or issuing disk IO
- Performance metrics
  - Application throughput (K allocations/sec)
  - Java GC pauses

Scenario 1: Startup State (App. Symptoms)

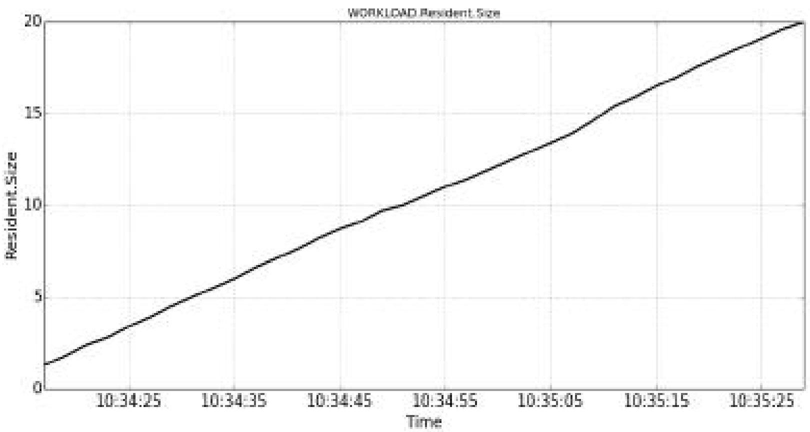


(a) JavaApp throughput (K allocations/sec)

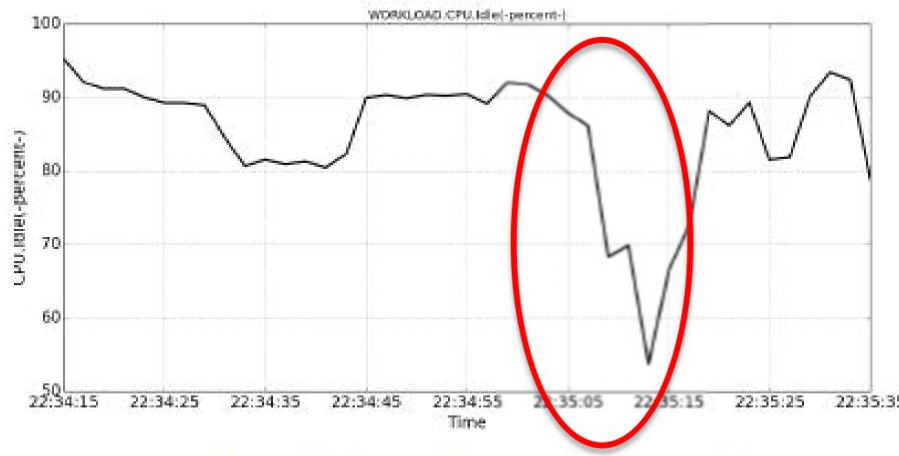
(b) JavaApp GC pauses (Second)

- ❑ When Java applications start
- ❑ Life is good in the beginning
- ❑ Then Java throughput drops sharply
- ❑ Java GC pauses spike during the same period

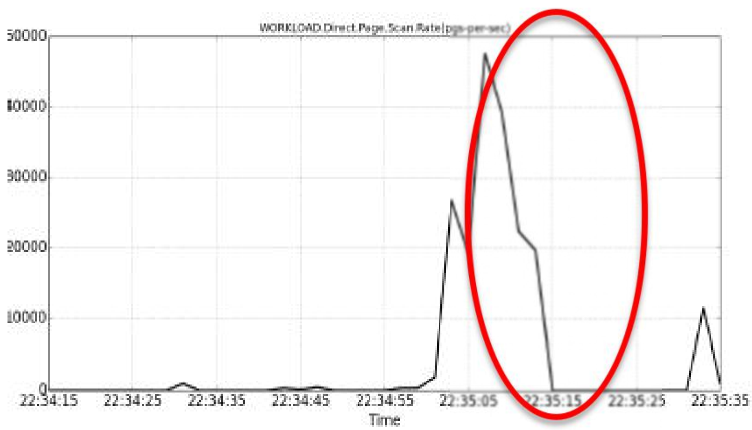
Scenario 1: Startup State (Investigations)



(a) JVM resident size (GB)



(b) Cpu idle usage (%)



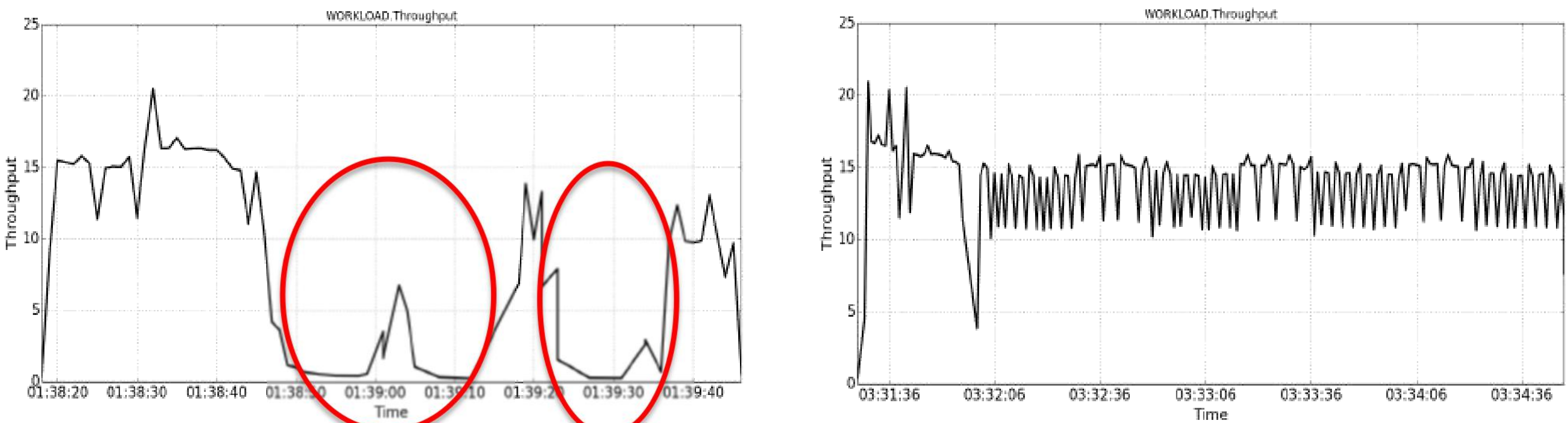
(c) Direct page scanning (pages per second)

- Java heap is gradually allocated
- Without enough memory, direct page scanning can happen
- Heap is swapped out and in
- It causes large GC

Solutions

- Pre-allocating JVM heap spaces
  - JVM “-XX:AlwaysPreTouch”
- Protecting JVM heap spaces from being swapped out
  - Swappoff command
  - Swappiness
    - =0 for kernel version before 2.6.32-303
    - =1 for kernel version from 2.6.32-303
  - Cgroup

Evaluations (Pre-allocating Heap)

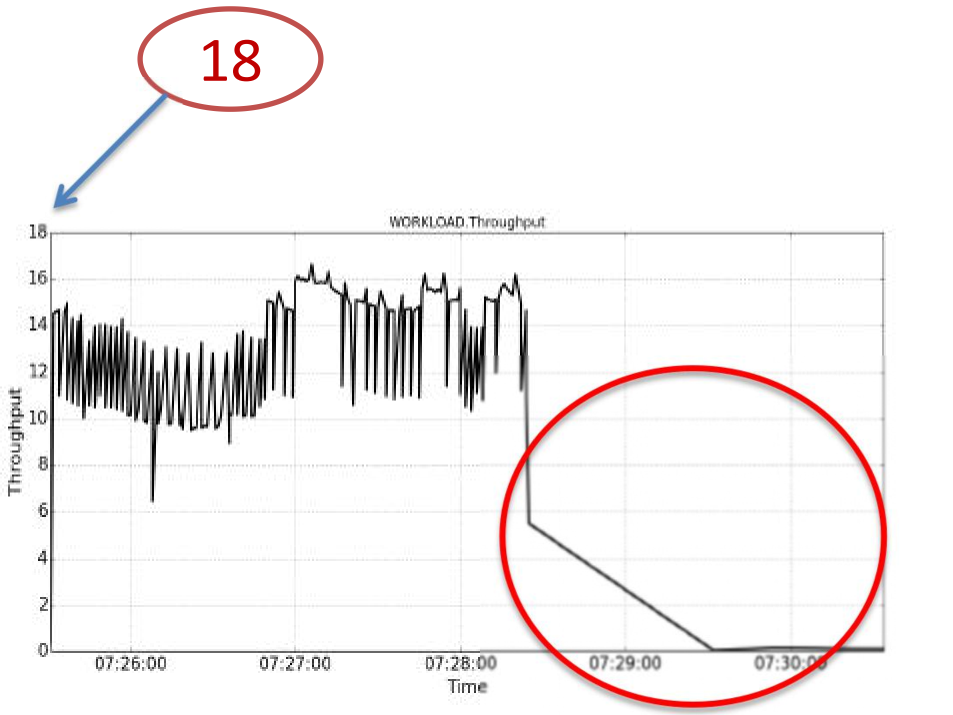


(a) Throughput without *AlwaysPreTouch* (b) Throughput with *AlwaysPreTouch*

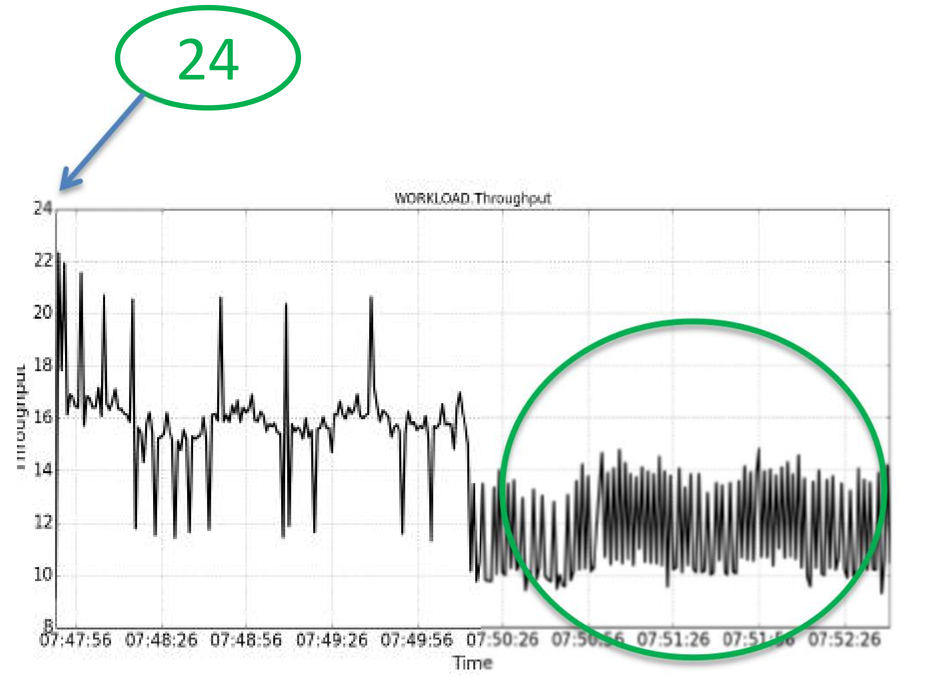
STARTUP DELAY OF PRE-ALLOCATING JVM HEAP SPACE

Heap size (GB)	1	2	4	10	20	30	40
Start delay (Sec)	0.5	0.8	1.6	2.5	7.3	11	15

Evaluations (Protecting Heap)

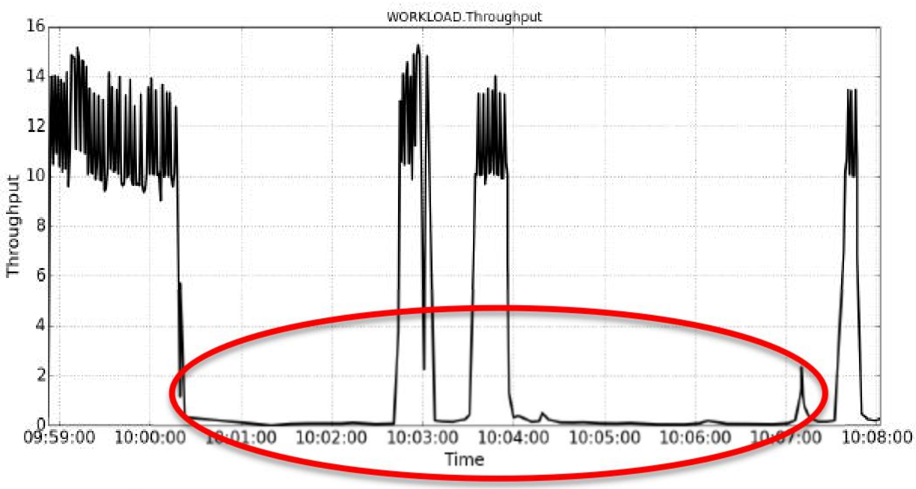


(a) Throughput when *swappiness*=100

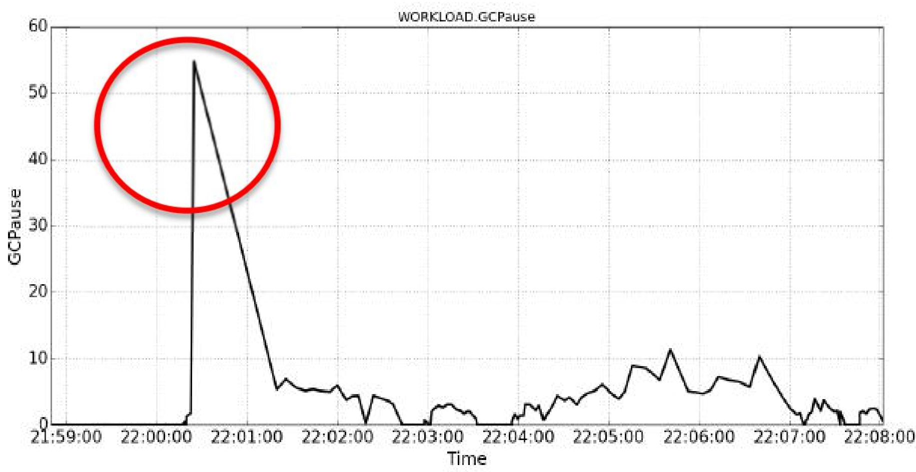


(b) Throughput when *swappiness*=0

Scenario 2: Steady State (App. Symptoms)



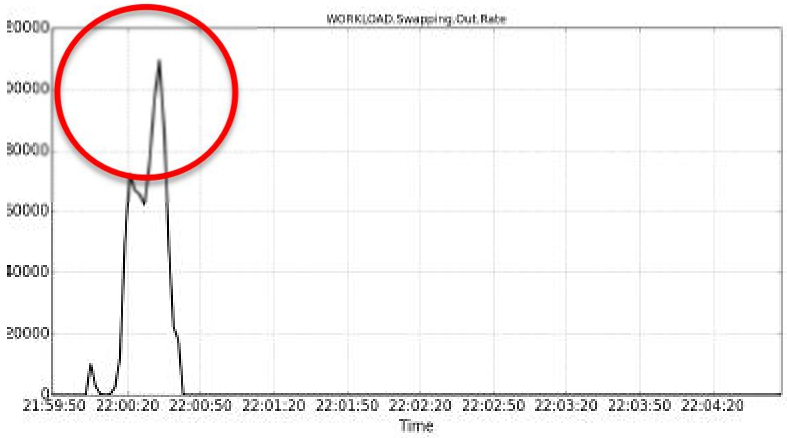
(a) JavaApp throughput (K allocations/sec)



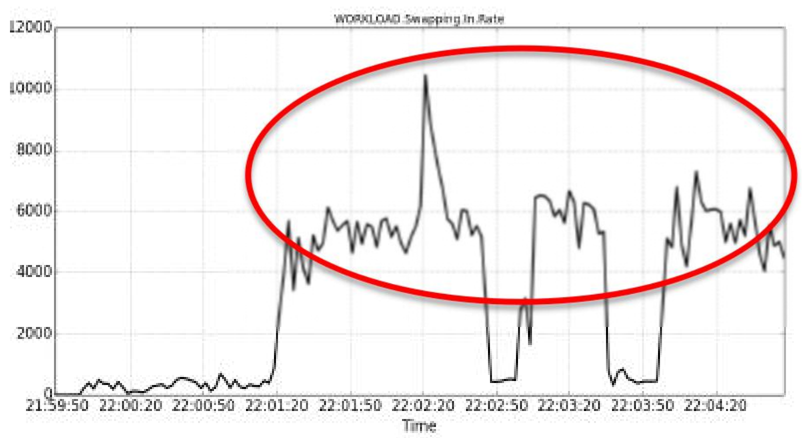
(b) JavaApp GC pauses (second)

- During steady state of a Java application, system memory stresses due to other applications
- Java throughput drops sharply and performs badly
- Java GC pauses spike

Scenario 2: Steady State (**Level-1** Investigations)



(a) Swapping out (pages/sec)

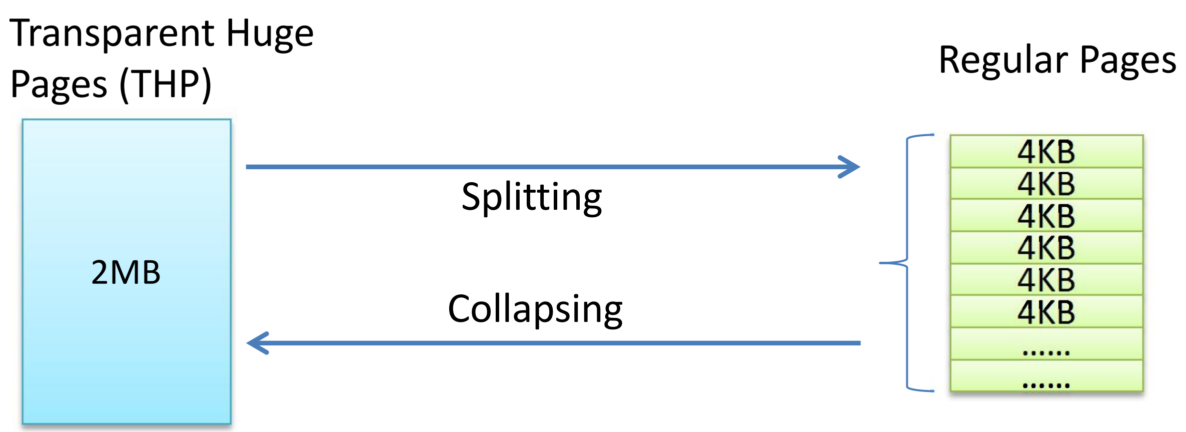


(b) Swapping in (pages/sec)

- During GC pauses, swapping activities persist
- Swapping in JVM pages causes GC pauses
- However, swapping is not enough
  - Excessive GC pauses (i.e., 55 seconds)
  - High sys-cpu usage (swapping is not sys-cpu intensive)

[Times: user=0.12 sys=54.67, real=54.83 secs]

Scenario 2: Steady State (**Level-2** Investigations)



- THP (Transparent Huge Pages)
  - Improved TLB cache-hits
- Bi-directional operations
  - THPs are allocated first, but split during memory pressure
  - Regular pages are collapsed to make THPs
  - CPU heavy, and thrashing!

Solutions

- Dynamically adjusting THP
  - Enable THP when no memory pressure
  - Disable THP during memory pressure period
  - Fine tuning of THP parameters

Evaluations (Dynamic THP)

Without memory pressure

- Dynamic THP delivers similar performance as THP is on

Mechanism	THP Off	THP On	Dynamic THP
Throughput (K allocations/sec)	12	15	15

With memory pressure

- Dynamic THP has some performance overhead
- Performance is less than THP-off
- But better than THP-on

Mechanism	THP Off	THP On	Dynamic THP
Throughput (K allocations/sec)	13	11	12

Scenario 3: Steady State (Heavy IO)

```
2016-01-14T22:08:28.028+0000: 312052.604: [GC (Allocation Failure) 312064.042: [ParNew
Desired survivor size 1998848 bytes, new threshold 15 (max 15)
- age   1:     1678056 bytes,     1678056 total
: 508096K->3782K(508096K), 0.0142796 secs] 1336653K->835675K(4190400K), 11.4521443 secs]
[Times: user=0.18 sys=0.01, real=11.45 secs]
2016-01-14T22:08:39.481+0000: 312064.058: Total time for which application threads were stopped:
11.4566012 seconds
```

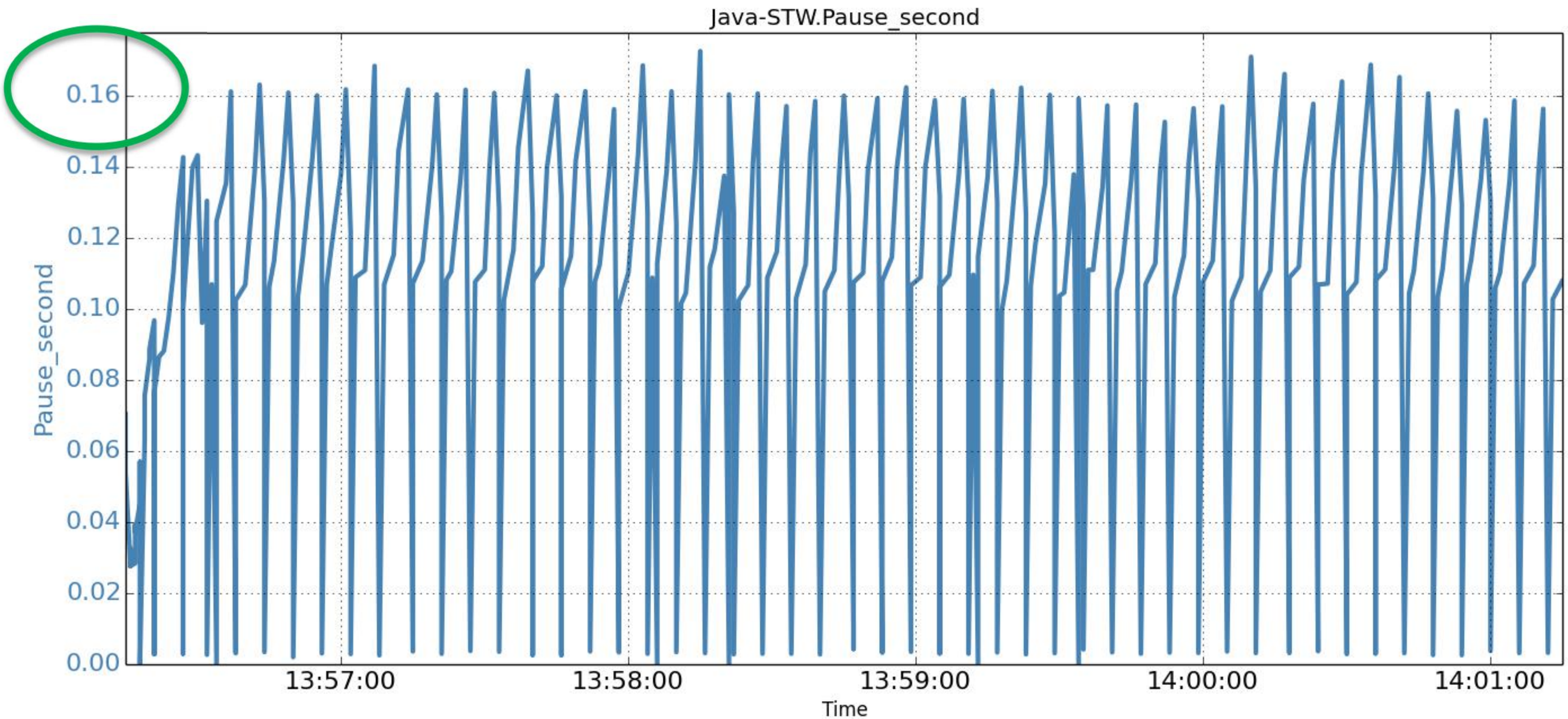
- Production issue
  - Online products
  - Applications have light workload
  - Both CMS and G1 garbage collectors
- Preliminary investigations
  - Examined many layers/metrics
  - The only suspect: disk IO occasionally is heavy
  - But all application IO are asynchronous

Reproducing the problem

- Workload
  - Simplified to avoid complex business logic
  - https://github.com/zhenyun/JavaGCworkload
- Background IO
  - Saturating HDD

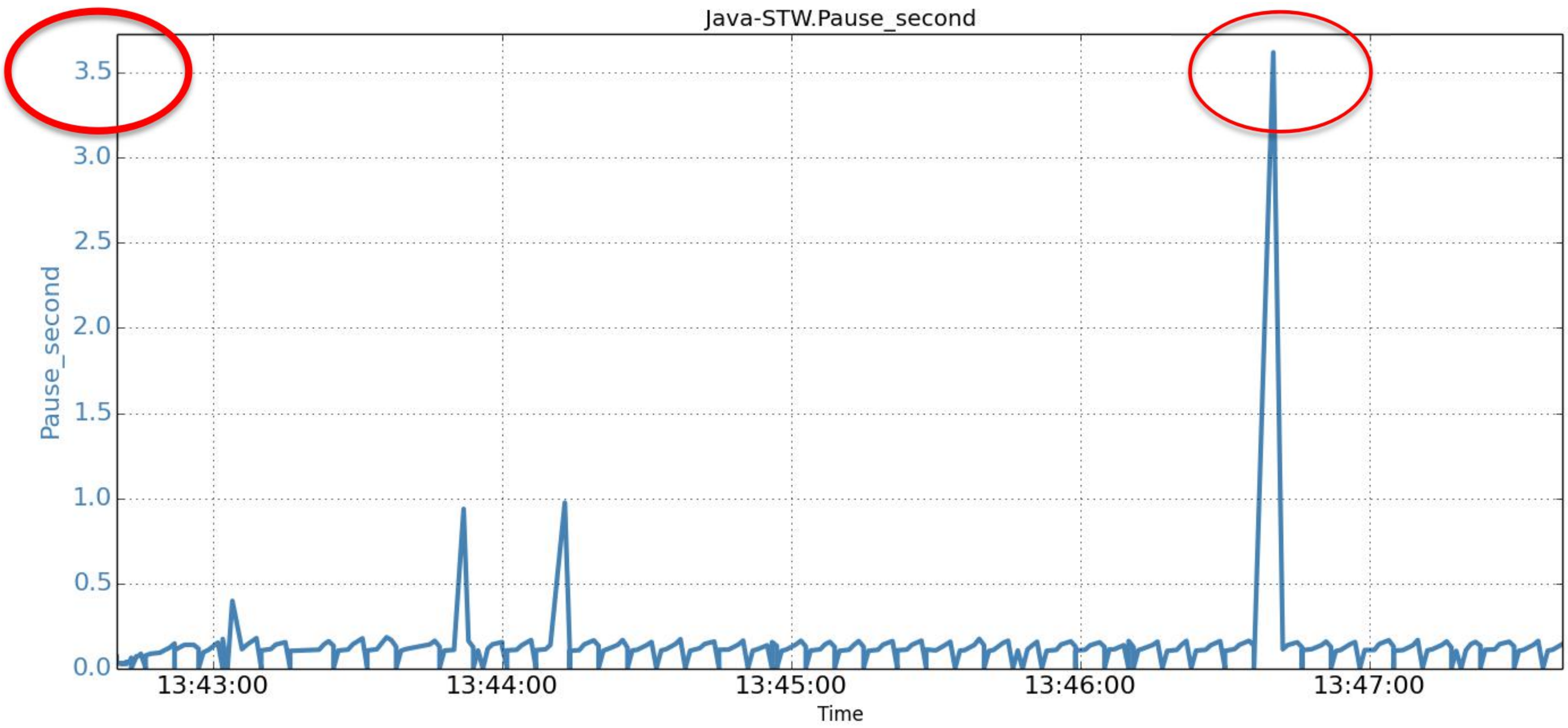
Setup	Values
Platform	HP Z620 Workstation with 1 socket of 12 Intel(R) Xeon(R) CPU E5-2620 2GHz hardware cpus.
OS	RHEL Linux 2.6.32-504.el6.x86_64
Hard Drives	Mirrored setup consisting of two SEAGATE ST3450857SS disks, SAS-connected.
File system	EXT4, with default mounting options.
JDK	Oracle HotSpot JDK-1_8_0_5
JVM options	-Xmx10g -Xms10g -XX:+UseG1GC -Xloggc:gc.log -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+PrintGCApplicationStoppedTime

Case I: Without background IO



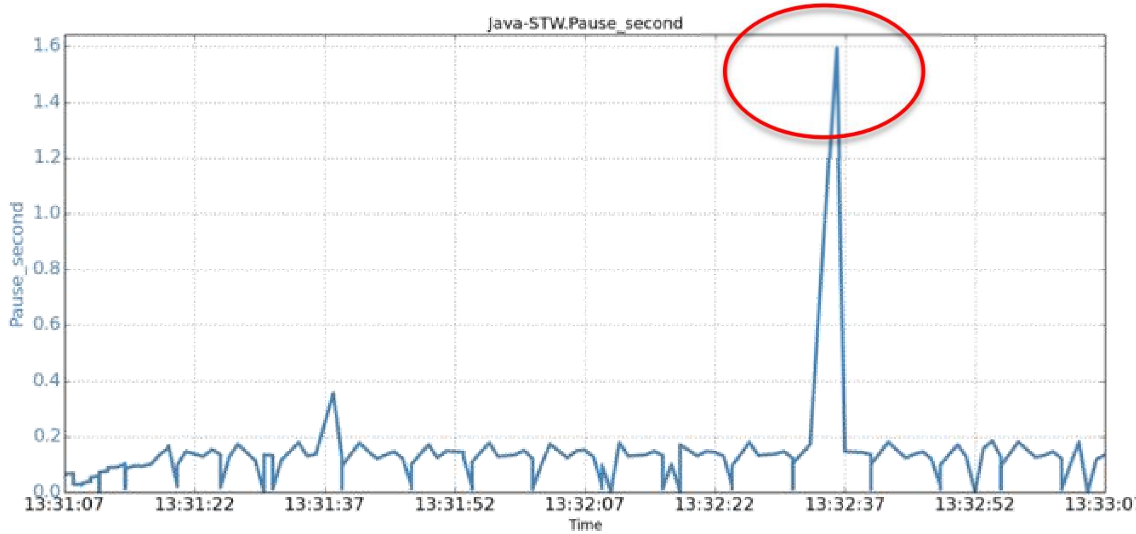
No single longer-than-200ms pause

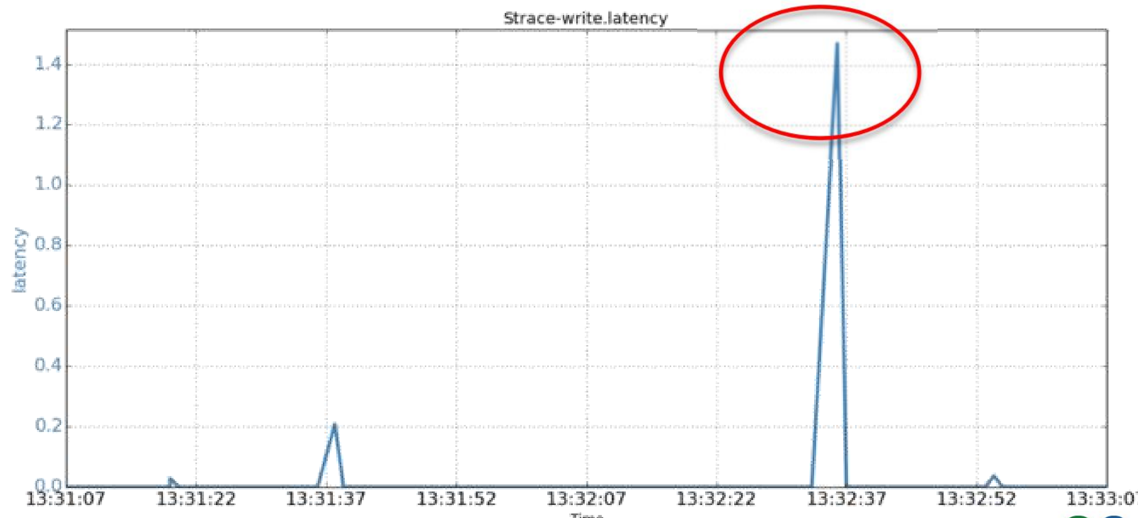
Case II: With background IO



Huge pause!

Investigations





Time lines

gc.log:

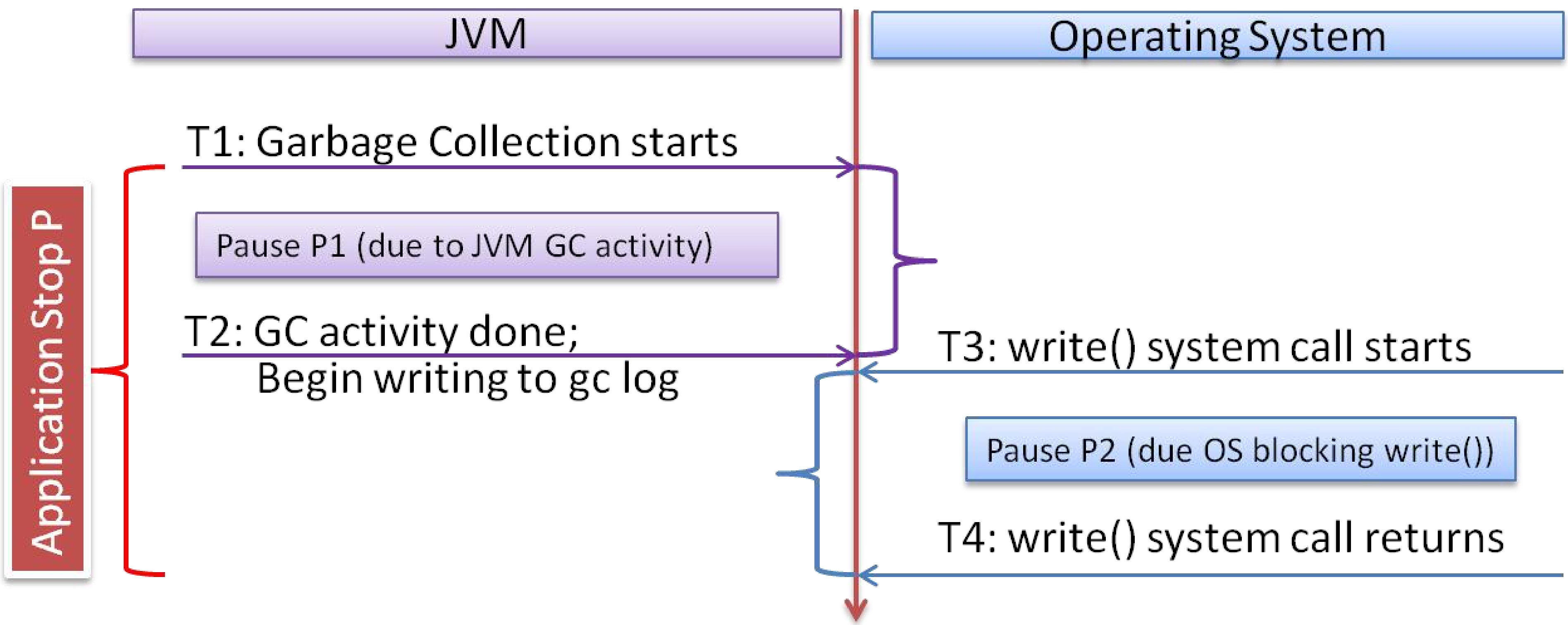
```
Line 1> 2015-12-21T13:32:33.736-0800: 86.268: Total time for which application threads were stopped: 0.1714770 seconds
Line 2> 2015-12-21T13:32:35.047-0800: 87.578: [GC pause (G1 Evacuation Pause) (young) 7814M->4114M(10G), 0.1227713 secs]
Line 3> 2015-12-21T13:32:36.641-0800: 89.172: Total time for which application threads were stopped: 1.5946271 seconds
```

strace output:

```
Line 4> [pid 11797] 13:32:35.169922 write(3, "2015-12-21T13:32:33.736-0800: 86"..., 224 <unfinished ...>
Line 5> [pid 11797] 13:32:36.640856 <... write resumed> ) = 224 <1.470906>
```

- At time 35.04 (line 2), a young GC starts and takes 0.12 seconds to complete.
- The young GC finishes at time 35.16 and JVM tries to output the young GC statistics to gc log file by issuing a write() system call (line 4).
- The write() call finishes at time 36.64 after being blocked for 1.47 seconds (line 5)
- When write() call returns to JVM, JVM records at time 36.64 this STW pause of 1.59 seconds (i.e., **0.12 + 1.47**) (line 3).

Interaction between JVM and OS



Non-blocking IO can be blocked

- Stable page write
  - For file-backed writing, OS writes to page cache first
  - OS has write-back mechanism to persist dirty pages
  - If a page is under write-back, the page is locked
- Journal committing
  - Journals are generated for journaling file system
  - When appending GC log files needs new blocks, journals need to be committed
  - Commitment might need to wait

Background IO activities

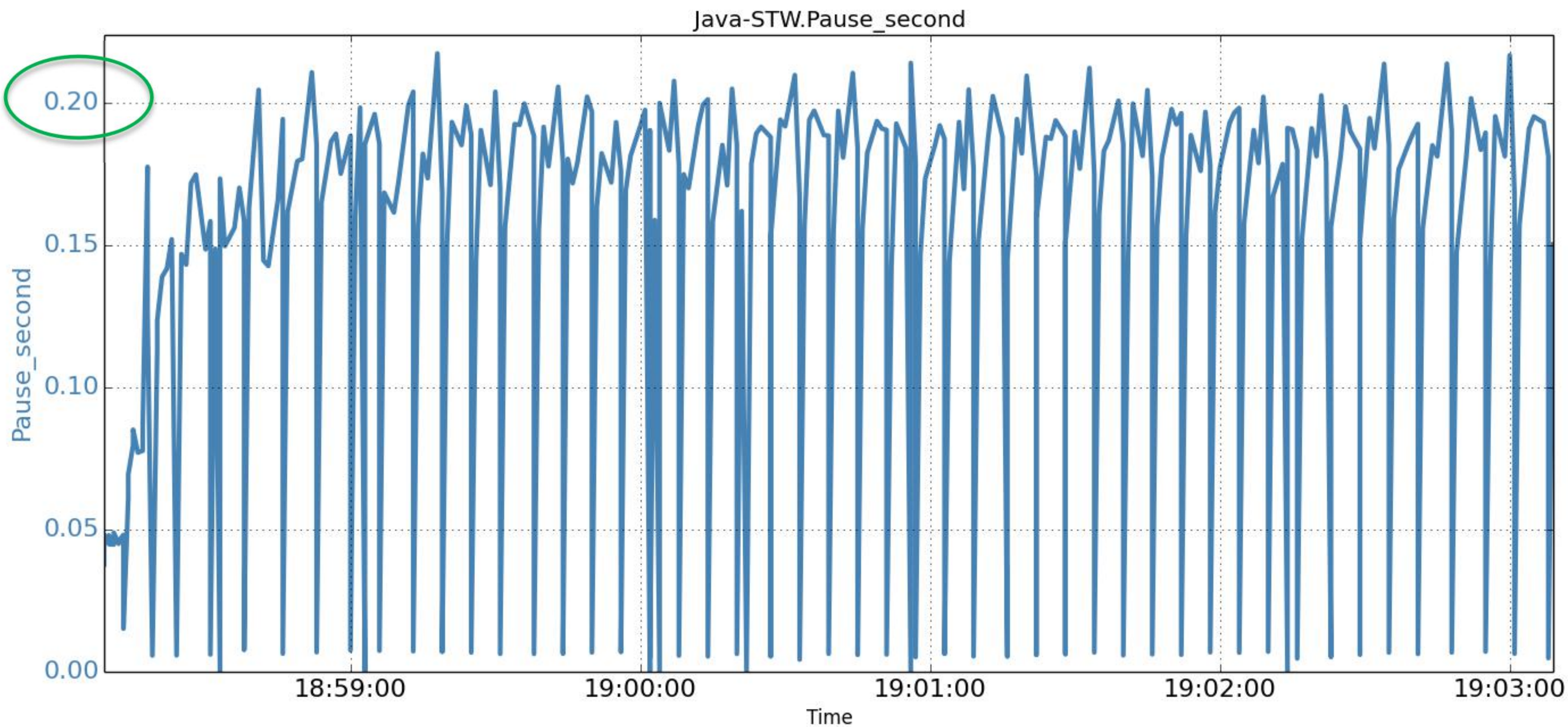
- OS activity such as swapping
  - Data writing to underlying disks
- Administration and housekeeping software
  - System-level software such as CFEngine also perform disk IO
- Other co-located applications
  - Co-located applications that share the disk drives, then other applications contend on IO
- IO of the same JVM instance
  - The particular JVM instance may use disk IO in ways other than GC logging

Solutions

- Enhancing JVM
  - Another thread
  - Exposing JVM flags
- Reducing IO activities
  - OS, other apps, same app
- Latency sensitive applications
  - Separate disk
  - High performing disks such as SSD
  - Tmpfs

Evaluation

- ❑ SSD as the disk



The good, the bad, and the ugly

```
2016-01-14T22:08:28.028+0000: 312052.604: [GC (Allocation Failure) 312064.042: [ParNew
Desired survivor size 1998848 bytes, new threshold 15 (max 15)
- age   1:     1678056 bytes,     1678056 total
: 508096K->3782K(508096K), 0.0142796 secs] 1336653K->835675K(4190400K), 11.4521443 secs]
[Times: user=0.18 sys=0.01, real=11.45 secs]
2016-01-14T22:08:39.481+0000: 312064.058: Total time for which application threads were stopped:
11.4566012 seconds
```

- **The good**: low real time
  - Low user time and low sys time
  - [user=0.18 sys=0.01, real=0.04 secs]
- The bad: non-low (but not high) real time
  - High user time and low sys time
  - [user=8.00 sys=0.02, real=0.50 secs]
- The ugly: high real time
  - High sys time [user=0.02 sys=1.20, real=1.20 secs]
  - Low sys time, low user time [Example?]

Lessons Learned (I)

- Be cautious about Linux's (and other OS) new features
  - Constantly incorporating new features to optimize performance
  - Some features incur performance tradeoff
  - They may backfire in certain scenarios

Lessons Learned (II)

- Root causes can come from seemingly insignificant information
  - Linux emits significant amount of performance information
  - Most of us most of the time mostly only examine a small subset of them
  - Don’t ignore others – understand the interactions of sub-components

Lessons Learned (III)

- Pay attention to multi-layer interaction
  - Application protocol, JVM, OS, storage/networking
  - Most people are familiar with a few layers
  - Optimizations done at one layer may adversely affect other layers
  - Many performance problems are caused by the cross-layer interactions

THANKS!

Zhenyun@gmail.com