Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

SOGETI

IGATE
Speed.Agility.Imagination

JavaOne™

ORACLE® Diamond Partner

Oracle OpenWorld | San Francisco | 28.10.2015 |
Miroslaw Bartecki, Capgemini

HOL1997 – How to become a winner in the JVM performance tuning battle

Agenda

1 **Performance tuning state of art**

2 **Key steps to made tuning done successfully**

3 **Why milliseconds matter**

4 **Performance problems inside overloaded system**

- Key performance killers areas
- Big system, big load, big challenge

5 **Web transactional system tuning advices**

- How to catch problem evidence
- How to made G1 GC tuning
- When do threads adjustment

6 **Lab tasks uncovered**

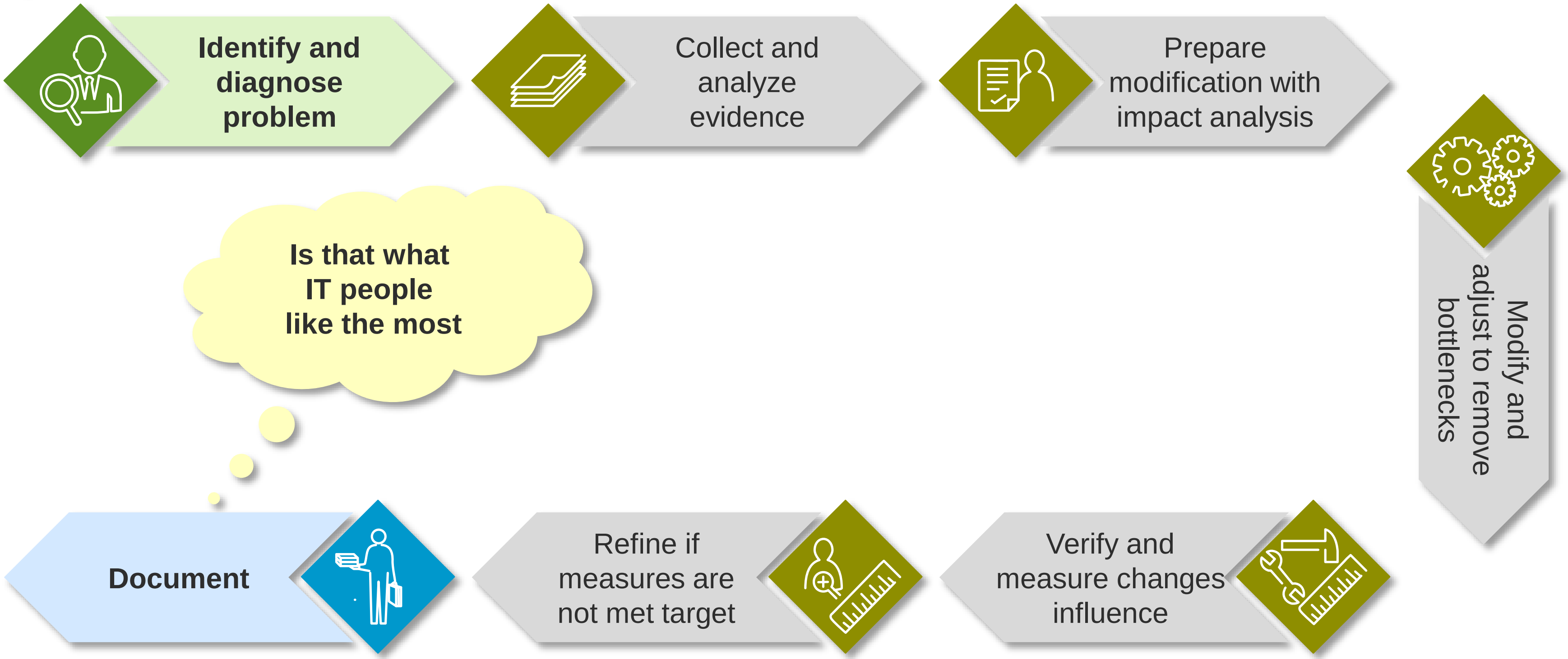
State of art in performance tuning

Performance tuning what is that about

Performance tuning is a response to increased load and related with that decreased performance.

Modification of system to handle a higher load is an performance tuning.

Technical view on performance tuning steps



Assumption – Presentation case

- Java Enterprise Application with web front end
- High end user load – thousands of concurrent users and more
- No serious stability issues
- Small end user load = Fast processing time
- For small load on a system, performance problems are not revealed
- Big load makes milliseconds latencies a big problem
- Small problem for low loaded system is a big slowness or killer in heavy loaded system

Why performance problem happen

Usually it looks like that ☺

Those problems are not uncovered on test phase

Performance tests are not cover 100% end user behaviors

End user nature of work with a system depends on many factors
e.g. cultural differences, habits, different approach to use functionalities etc.

Calculation why milliseconds matter

Short calculation

- One page application with high end user load
- 100 000 pages load per hour x 6 ajax calls to server
- = 600 000 end user interactions with server
- One http style interaction with server is about 3 db trasactions and 10 business method calls
- = 6 000 000 method calls

6 mln x 100 ms processing time = 600 000 s = 166,6 hours of processing time

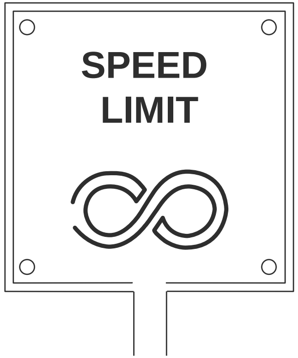
6 mln x 10 ms processing time = 60 000 s = 16, 6 hours of processing time

Is this matter for end user ?

- 1 page load = 6 ajax calls * 10 business method calls * 100 ms = 6 seconds ☹
- 1 page load = 6 ajax calls * 10 business method calls * 10 ms = 0,6 seconds ☹

Key performance problems areas in Java Enterprise world

- Slow web components processing (JSF lifecycle etc.)
- Slow business processing inside web application
- Too much method calls
- Slow or too much db queries
- Memory leaks
- Concurrency problems
- Bad JVM garbage collector configuration
- Bad threads parameters adjustment
- To much data processed in one transaction
- To much rely on technology when system is build without considering performance impact
  - Data serialization, conversion, mapping



Remark: On the lab we will focus on the marked on blue

Problem evidence

How to catch production problem evidence

- Measure performance by using
  - Application performance monitoring tools like open source Nagios, Zabbix or commercial one
  - Java profiler kind tools with code instrumentation features like Zorka, Jensor etc.
- Review application server and OS system configs
- Make java JVM heap dump
- Gather java virtual machine garbage collector logs
- Perform java virtual machine thread dumps

Catching production problem evidence – Technical details heap dump

- Making a heap dump is possible by:
  - JVM command line parameter: `-XX:+HeapDumpOnOutOfMemoryError`
  - OS command line tool: `jmap -dump:file=<<path_to_file>> <<java_process_id>>`
  - JVM command line parameter and sent OS process signal to JVM
    - `-XX:+HeapDumpOnCtrlBreak`
    - Send a SIGQUIT signal (-3 kill for Unix and Ctrl-Break for Windows) to the running Java process
    - Signal will create a heap dump without aborting the JVM.
  - Via JMX Mbean
    - Run *jconsole* and connect it to the corresponding JVM
    - Invoke MBean *com.sun.management.HotSpotDiagnostic* -> method: *dumpHeap (String, boolean)*

Catching production problem evidence – Thread dump technical details

- Making a thread dump is possible by:
  - OS command line tool:
    - `jstack <pid> >> threaddumps.log`
    - `jcmd <pid> Thread.print >> threaddumps.log`
  - jconsole tool plugin – for example
    - `jconsole -pluginpath/path/to/file/tda.jar`
  - JVM command line parameter and sent OS process signal to JVM
    - Send a SIGQUIT signal (-3 kill for Unix and Ctrl-Break for Windows) to the running Java process
    - Thread dump will be written to JVM stdout
  - To redirect JVM thread dump output on break signal to separate file
    - `-XX:+UnlockDiagnosticVMOptions -XX:+LogVMOutput -XX:LogFile=jvm.log`
    - Send a SIGQUIT signal (-3 kill for Unix and Ctrl-Break for Windows) to the running Java process

JVM garbage collector

Let's start tuning – Garbage collector (1 of 2)

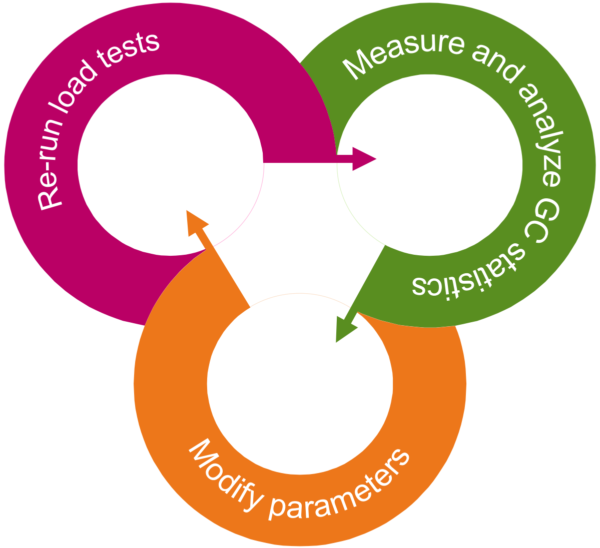
- Garbage collector (GC) is a JVM subsystem responsible for object allocation and reclamation
- It can consume big CPU resources if full cleaning cycle will come
- To tune we have to choose what our goals are:
  - Pause time
  - Throughput
  - Footprint

Let’s start tuning – Garbage collector (2 of 2)

Good idea to start GC tuning is to:

- Define tuning goal
- Clean all GC parameters from JVM
- Run application load tests

And



What GC algorithms we can pick up

- Serial
- Parallel
- CMS
- G1
- C4 from Azul
- Shenandoah – similar to G1, useful for large heap, JDK 8,9

G1 garbage collector

Let’s focus on G1 Garbage collector

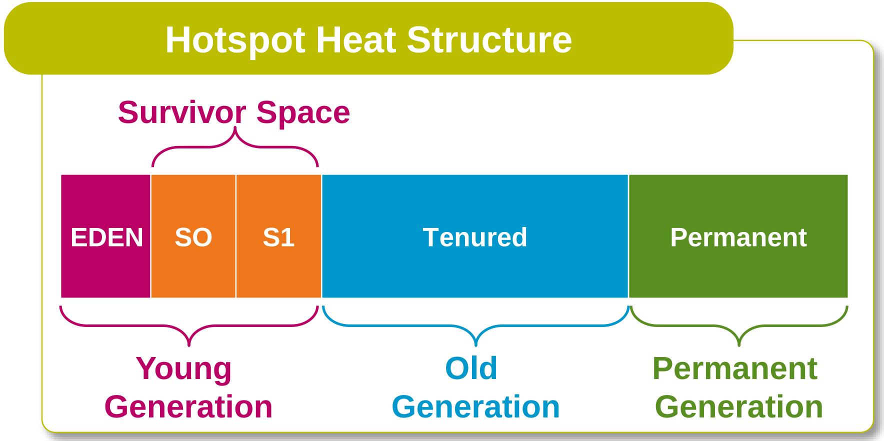
What is G1?

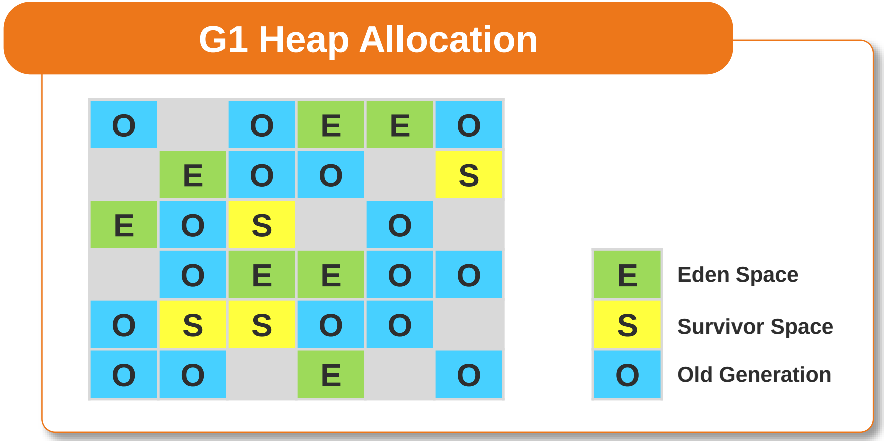
- G1 is the HotSpot (concurrent and parallel) low-pause collector
- Work on it initiated in 2004, first supported release JDK 7u4 (April 2012)
- For multi-processor (cores…) machines and large memories
- With new memory structure

Like official doc states is good for applications that:

- Compact free space without lengthy GC induced pause times
- Need more predictable GC pause durations
- Do not want to sacrifice a lot of throughput performance
- Do not require a much larger Java heap

Hot sport GC heap structure (serial, parallel, CMS) vs G1





Why take care about G1 GC?

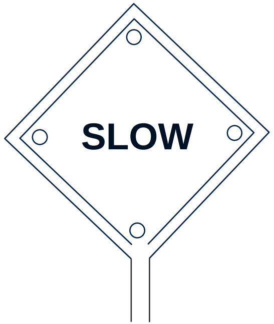
- Oracle is planning to make G1 as default in JDK 9 JEP-248
- G1 is good candidate to be used with concurrent no lock memory cleaning
- G1 takes what was best had JRockit JVM like pause targets
- G1 is not a source of JVM crashes like CMS in some conditions had
- Most of web online java apps have target to use low pauses GC throughput – G1 likes that
- G 1 is easier to tune than CMS or parallel
- With every new JVM release G1 is performing better that’s why is good idea to use latest release

G1 common myths

- Is self tuning one
- G1 pause target it's all you need
- G1 will perform on defaults effectively with any object sizes
- There is not too much parameters to setup

G1 GC most common problems

- Degradation of pause targets which are never met
- To fast or too slow promotion from young to old generation
- Reference processing takes to much time
- Insufficient space to deal with objects by GC
  - Evacuation failure or to-space exhausted mean no more free regions to promote to the old generation
- To much humongous allocations
  - [G1Ergonomics (Concurrent Cycles) reason: occupancy higher than threshold, occupancy: x bytes, allocation request: x bytes, threshold: x bytes (45.00 %), source: concurrent humongous allocation]
- GC CPU utilization is 100%
- Stop the World influence end user application servicing freeze



What are humongous objects in G1 GC

For G1 GC, any object that is more than half a region size is considered a **“Humongous object”.**

Such an object is allocated directly in the old generation into **“Humongous regions”**

To have such objects in old gen is → BAD ! IDEA

G1 calculation example for **humongous allocations**

- Xmx size → 16 GB
- By default divided by 200 regions
- Region size → 8 MB
- Humongous size limit 50% of region size → 4 MB
- You need to deal with objects bigger than 6 MB
- You need to have bigger region size to get bigger humongous limit
- Set –`XX:G1HeapRegionSize`=16 MB
- Your object will fit under 50% limit 8MB and will be no longer humongous

Best practices in G1 setup

- Set occupancy threshold that triggers a marking cycle according to your application needs
  - `-XX:InitiatingHeapOccupancyPercent=45`
- Adjust pause target limits with respect to served application business needs
  - Entry rule to adjust maximum pause time limits is 90% on application processing and 10% on GC
  - `-XX:MaxGCPauseMillis=400`
- Avoid low pause targets like 10 ms because they will be not met for typical JEE application
- Speed up marking phase if needed by -XX:`ConcGCThreads` (default ParallelGCThreads/4)
- Start marking cycle earlier by decrease -XX:`InitiatingHeapOccupancyPercent`

Best practices in G1 setup – Logging

- Thumb rule – always enable G1 logging
  - You will know what job is made by GC
  - Setup below flags to have detailed G1 logging
    - -verbose:gc – to enable GC logging
    - -Xloggc:/opt/gclogs/gcg1log – to define GC log place
    - -XX:+PrintGCDateStamps – to have date and uptime details information
    - -XX:+PrintGCDetails – to have GC phases information
    - -XX:+PrintAdaptiveSizePolicy – to have ergonomics information
    - -XX:+PrintTenuringDistribution – to have survivor and age information

Best practices in G1 setup – Utilization

- Reduce number of G1 Full GCs
  - It is single threaded
  - G1 + big heap + full GC cycle = super slowness
  - Put in runtime prams `-XX:+PrintAdaptiveSizePolicy` to analyze GC logs also on “Full” cycles presence
- Prevent situation when heap space is over-utilized
  - G1 need a free space to operate on objects
  - If there is not enough space to operate on objects increase heap size

Best practices in G1 setup – Humongous objects

- Prevent to humongous allocations
  - G1 has policy to split heap into parts (regions)
  - Default is to divide overall size by 2048 (e.g. 2048 M/2048 = 1M region size)
  - Region size delimits size humongous object -50% of region size (above example objects bigger than 500 k are humongous)
  - Adaptive size policy parameter will let you know when this will happen
  - Adjust max heap and region size -XX:`G1HeapRegionSize`

Best practices in G1 setup – Latencies

- Enable -XX:+ParallelRefProcEnabled to process week references in parallel
  - Some application are using weak references (for example session caching)
  - GC spends a lot of time to try to figure which references can be cleaned up
  - Remark phase is single-threaded by default unless this option
  - `-XX:+PrintReferenceGC` to check details
  - Typical root cause of having week references are ThreadLocal, RMI, external libraries
- Do not user G1 for very low latencies because it still based on “stop the world” idea
- Use the latest JDK release to get best/fastest G1 implementation

Best practices in G1 setup – Space

- Setup proper segment size to impact eden/survivor space <=32 MB
  - You shouldn't setup NewSpace size because it's ignored by G1 (except experimental flags)
  - You can use only ratio based options
  - Good segment size can give better throughput
- Use high waste % of heap to allow effective GC operations like `-XX:G1HeapWastePercent=10`
- Increase memory for "to-space" if you will see message like below `-XX:G1ReservePercent`
  - GC pause (G1 Evacuation Pause) (mixed) (to-space overflow)
- Use large pages for bigger than 8GB heaps `-XX:+UseLargePages`

Threads

Java application servers and threads

- All current application servers are using concurrent threads to host deployed applications
- Thread configuration model differs per application server
  - Some of them have generic pools used by all embedded subsystems
  - Some of them have specific pools specialized to some operations like http pool, asynchronous pool etc.
  - Some of the are using mixed model with generic pools and couple of specialized
- One common thing is that if thread locking will happen it can stop entire processing
- Threads health is base knowledge about application server processing condition

Why to take care about threads

- Application server performance depends on threads “health”
- Thread pool performance have direct influence on throughput
  - To much waits on condition by thread means longer response time
  - To much time taken by thread on finish processing means lower throughput
- More threads doesn’t always means good
  - To much threads to review state means a lot of context switching on OS/CPU level
  - In worse example system can spent time only on threads review with no time to process things

This man is not multithreaded

Why thread locks happening

- Shared resource used by application is not responding like
  - Database transactions stale and all jdbc threads are taken by waiting to close transaction
  - File system access has to wait to OS stack response
- Synchronized code block are in frequently used part of source
- Other threads are depending on locked one
  - HTTP threads are depended on business processing threads outcome
  - Business processing threads are depended on source data from database

How to tune threads

- Measure threads behavior by
  - Making threads dump
  - Compare couple of thread dumps to see difference/progress
- Analyze thread dump on locks and waits presence
- Review suspicious code processed by problematic threads
- Modify threads settings on application server side if needed like too small pool
  - Like `maxThreads="150" minSpareThreads="25" maxSpareThreads="75"` for Tomcat http connector settings
  - Like `core-threads count="10" per-cpu="20"` for JBoss 7 http subsystem
  - Like `ThreadCount` in config.xml for Weblogic HTTP and RMI subsystem
- Suggest architecture or code change if required (if it is a source of problems)

Hands on Labs

Lab purpose

1. Give overall view what are typical tuning tasks
2. Train skills to gather evidence of performance problems
3. Analyze example application response times, garbage collector logs and thread health
4. Emphasize common tuning patterns for G1 garbage collector

Lab flow tasks

- Trace sample application response time problem using zorka tool
- Enable G1 garbage collector logging
- Analyze collected G1 logs
- Tune G1 parameters accordingly to identified bottlenecks
- Collect and analyze G1 logs after tuning
- Analyze GC throughput
- Make heap dump
- Analyze heap content
- Make thread dump and analyze issues

Tools used during labs

- Oracle Linux
- Java 8
- WebLogic Application Server + MariaDB
- Zorka
- Oracle jvisualVM with plugins
- GCViewer
- IBM Heap Analyzer
- Oracle jmap, jstack, jcmd
- IBM Thread and Monitor Dump Analyzer for Java

Resources and materials

- Pictures used inside presentation are on Creative Commons licence
- Source of data are taken from
  - Official Oracle Java documentation
  - Live production experience
  - Community knowledge

Contact information

Miroslaw Bartecki
Solution Architect
miroslaw.bartecki@capgemini.com

Capgemini Poland
Uniwersytecka 13
40-007 Katowice
Poland

About Capgemini and Sogeti

Now with 180,000 people in over 40 countries, Capgemini is one of the world's foremost providers of consulting, technology and outsourcing services. The Group reported 2014 global revenues of EUR 10.573 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

Sogeti is a leading provider of technology and software testing, specializing in Application, Infrastructure and Engineering Services. Sogeti offers cutting-edge solutions around Testing, Business Intelligence & Analytics, Mobile, Cloud and Cyber Security. Sogeti brings together more than 20,000 professionals in 15 countries and has a strong local presence in over 100 locations in Europe, USA and India. Sogeti is a wholly-owned subsidiary of Cap Gemini S.A., listed on the Paris Stock Exchange.

www.capgemini.com

www.sogeti.com