IBM Software Group

IBM® Java™ Health Center – Overview and Usage

Kevin Grigorenko (kevin.grigorenko@us.ibm.com)
IBM WAS SWAT Team
1 May 2012

WebSphere® Support Technical Exchange

Agenda

- What is IBM Java Health Center
- How to install it
- How to gather data
- How to analyze data
- Screenshots
- Tips & Tricks
- Questions & Answers

IBM Java Health Center

- The IBM Java Health Center is a very low overhead tool that runs in the IBM JVM and provides information on method profiling, garbage collection, I/O, lock analysis, threads, native memory, and more.
- Fully supported by the IBM Java Tools team through PMRs.
- **Important Links:**
  - Product Page: http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/
  - Documentation: http://publib.boulder.ibm.com/infocenter/hctool/v1r0/topic/com.ibm.java.diagnostics.healthcenter.doc/topics/introduction.html
- Similar to HotSpot/Oracle's VisualVM and JRockit Mission Control
- It runs with the IBM JVM (32- or 64-bit) on: AIX, Linux®, Windows®, and z/OS.

When to Use Health Center

- Health Center is particularly good for deep dives into performance issues, high CPU and monitor contention.
- It does not have the overall capabilities of monitoring products such as ITCAM.
- The sampling profiler currently has lower overhead than monitoring products due to very tight coupling with IBM JVM internals and bypassing of JVMTI interfaces.
    - Overhead of sampling profiler usually as low as < 1% or 2%

Agent & Client

- The Health Center Agent is a JVMTI native library that must be enabled (off by default)
  - Most often enabled using a generic JVM argument -Xhealthcenter or -Xhealthcenter:level=headless and restarting the JVM
  - Recent versions can be enabled dynamically using late attach
  - Two modes: 1) Connected to a client through a socket, 2) headless
- The Health Center Client is an Eclipse-based GUI perspective which is used to analyze agent data
  - Two modes: 1) Connected to the agent through a socket, 2) load HCD file produced by the agent or exported from another client

Agent: Getting Started

- The IBM JVM ships with Health Center:
  - …/java/jre/lib/$PLATFORM/libhealthcenter.so
  - …/java/jre/lib/ext/healthcenter.jar
  - …/java/jre/lib/healthcenter.properties
  - …/java/jre/bin/libhealthcenter.so
- Health Center performance and functionality is affected by two things: 1) the version & service release of the JVM, and 2) the version of the agent
  - If practical, it's always best to update the agent because the agent is independent of the JVM and the latest will have bug fixes, additional collection, better command line options, etc.

Minimum JVM Version for Use in Production

- Platform requirements:
  - http://publib.boulder.ibm.com/infocenter/hctool/v1r0/topic/com.ibm.java.diagnostics.healthcenter.doc/topics/platforms.html

IBM Java Version	Minimum Service Release for use in Production	Minimum WAS Release for use in production	Owned Monitor Information	Native Memory Breakdown	Enable verbosegc at runtime	Allocation Sampling
Java 5	SR10	**WAS 6.1.0.27**	✗	✗	✗	✗
Java 6	SR5	**WAS 7.0.0.5**	✓	✗	✗	✗
Java 626	Any	**WAS 8.0**	✓	✓	✓	✓
Java 7	Any	**WAS 8.5**	✓	✓	✓	✓

Checking the Installed Agent Version

- Two ways to figure out the installed version:
  - Run java -version -Xhealthcenter
    - `$ …/java/bin/java -version -Xhealthcenter`

      -
        ```
        …
        Feb 14, 2012 11:06:56 AM
        com.ibm.java.diagnostics.healthcenter.agent.mbean.HCLaunc
        hMBean startAgent
        INFO: Agent version "1.3.0.20101014"
        ```
  - Extract version.properties from …/java/jre/lib/ext/healthcenter.jar
    - `$ jar xvf …/java/jre/lib/ext/healthcenter.jar version.properties`
    - `$ cat version.properties`
      - `jar.version=1.3.0.20101014`

Getting the Latest Agent

- Three ways to get the latest agent:
  - Health Center Documentation Page: http://publib.boulder.ibm.com/infocenter/hctool/v1r0/topic/com.ibm.java.diagnostics.healthcenter.doc/topics/installingagent.html
  - From within IBM Support Assistant, click Help > Help Contents > … Health Center > … Installing the Health Center Agent
- The agent bitness (32- or 64-bit) should match the JVM bitness, not the operating system bitness.
  - For example, if it is a 32-bit JVM running on a 64-bit operating system, download the 32-bit agent.

Updating the Agent

- Some customers are weary about updating the WAS installation files; however, the files being updated do not relate to non-Health Center functionality.
- Consider backing up the existing agent files in case there are problems.
- Procedure (for z/OS, see documentation)
  - Upload the ZIP file to <WAS>/java/$FILE.zip
  - Change directory to <WAS>/java/ and run the command:
    - ./bin/jar xvf $FILE.zip
  - Make sure to chown the updated files properly.
  - Error "in use" because you just tried running health center (e.g. checking the version) and on some operating systems, may persist unless all related Java processes are stopped.
    - If it still doesn't work, and the JVM is run by a non-root user, then try extracting the zip using the root user, and then chown to the non-root user.

Enabling the Agent

- There are three modes:
  - Socket Communication (-Xhealthcenter)
    - By default opens port 1972 or the first available increment up to 2072
    - Communication either in IIOP (default) or JRMP
    - Security available: MBean authentication and/or SSL
    - Use -Xhealthcenter:level=off to only start collection when the first client connects (pseudo late attach). Preferences > Subsystem Enablement to turn off.
  - Headless (-Xhealthcenter:level=headless)
  - Late Attach (into either of the above two modes)
- Logging
  - Log file created in the temp directory named healthcenter.$PID.log
- The agent does use ~50MB native memory in the JVM

Dynamically Starting the Agent

- The Java Late Attach API allows the injection of a native or Java library into a running JVM without restarting it.
- Late attach is available on these IBM JVM levels:
  - Java 5 >= SR10 (disabled by default) [WAS >= 6.1.0.27]
  - Java 6 >= SR6 (enabled by default on non-z/OS platforms) [WAS >= 7.0.0.7]
  - Java 6 R26 [WAS 8] (enabled by default)
  - Java 7 (enabled by default)
- Late attached can be controlled with the generic JVM argument -Dcom.ibm.tools.attach.enable=[yes|no]

Dynamically Starting the Agent

- If the JVM supports late attach and it is enabled:
  - $ cd <WAS>/java/jre/lib/ext
  - List available late attach JVMs:
    - $ ../../bin/java -jar healthcenter.jar
  - Attach to a particular JVM:
    - $ ../../bin/java -jar healthcenter.jar ID=$PID -Dcom.ibm.java.diagnostics.healthcenter.data.collection.level=headless
  - It's currently not possible to disable headless mode with late attach.

Socket Mode

- Generally problematic in production environments because of firewalls; however, here are a few things to note about socket mode:
  - If client X connects, they get all data up to that point that fit in memory buffers. If X disconnects and a new client Y connects, the data that X saw will not be available, only new data since then.
    - Methods classloaded before client Y may not show for Y.
  - In memory buffers have a limited size, so some data may be lost.
  - Export collected data into an HCD file by clicking File > Save Data
  - Client refreshes every 10 seconds
  - Supports requesting java dump, heap dump, and system dump
  - The agent uses a random port for ORB which needs firewall hole: -Dcom.ibm.java.diagnostics.healthcenter.agent.iiop.port=N

Headless Mode

- Usually the best way to use health center in production environments, but also usually requires updating the shipped agent.
- Does not open a socket but instead writes agent data to the local file system:
  - Directory controlled with -Dcom.ibm.java.diagnostics.healthcenter.headless.output.directory=DIR
  - Defaults to the WAS profile directory

Headless Mode

- Files the agent writes to while the JVM is running:
  - EnvironmentSource$PID*
  - JLASource$PID*
  - MemoryCountersSource$PID*
  - MemorySource$PID*
  - MethodDictionarySource$PID*
  - TRACESubscriberSource$PID*
- When the JVM stops, the agent ZIPs these files into a file in the output directory called healthcenter$PID.hcd. The files (other than the HCD) are then deleted.
  - These files are compressed well into the HCD (up to 75%).
- If the JVM crashes, manually ZIP the files into an HCD

Headless Mode

- I've run HC in headless mode in massive production systems with little overhead; however, the agent files can be quite large – the largest I've seen are a few GB per hour.
- Each output file has a maximum size of 2GB
- Recent versions of the agent have options to roll over the files:
  - -Dcom.ibm.java.diagnostics.healthcenter.headless.files.max.size =BYTES
  - -Dcom.ibm.java.diagnostics.healthcenter.headless.files.to.keep=N
    - Use 0 to keep all of them

Headless Mode

- To do rollover, the agent appends _N to each file where N is the iteration number. When one of the files reaches 2GB or the value of max.size (usually the TRACESubscriberSource which has the profiling data), the agent will create an healthcenter$PID_N.hcd file and then start _N+1 files.
- When max.size files is hit, the agent will delete the oldest ones.
- If headless mode is not supported by agent version, error:
  - SEVERE: Health Center agent failed to start. java.lang.IllegalArgumentException: No enum const class com.ibm.java.diagnostics.healthcenter.agent.dataproviders.DataCollectionLevel.HEADLESS
  - Check both healthcenter.log and native_std*.log

Headless Mode

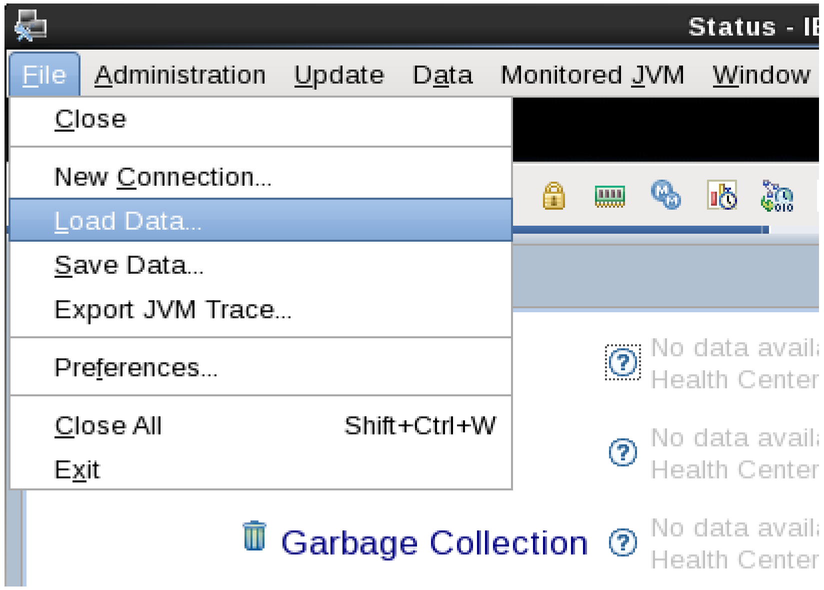
- The volume of data will be a function of many variables:
  - The length of time of the data collection
  - The average number of active threads
  - The average stack depth of said threads
  - The available number of processors
  - The number of method calls per request
  - Etc.
- The best approach is to run a stress test in a stress test environment to gauge how much space will be needed.

Client: Installing

- Four ways to install the client:
  - IBM Support Assistant > Tools Addon > JVM-based Tools
    - IBM Monitoring and Diagnostic Tools for Java - Health Center
  - Extend an existing Eclipse installation
    - Add an update site: http://download.boulder.ibm.com/ibmdl/pub/software/isa/isa410/production/
    - Select the same tool as above

Loading an HCD in the Client

- The file must have a .zip or .hcd extension
- Start the client
- Click Cancel on the dialog that pops up asking to connect to an existing JVM
- Click File > Load Data



Client Timeline

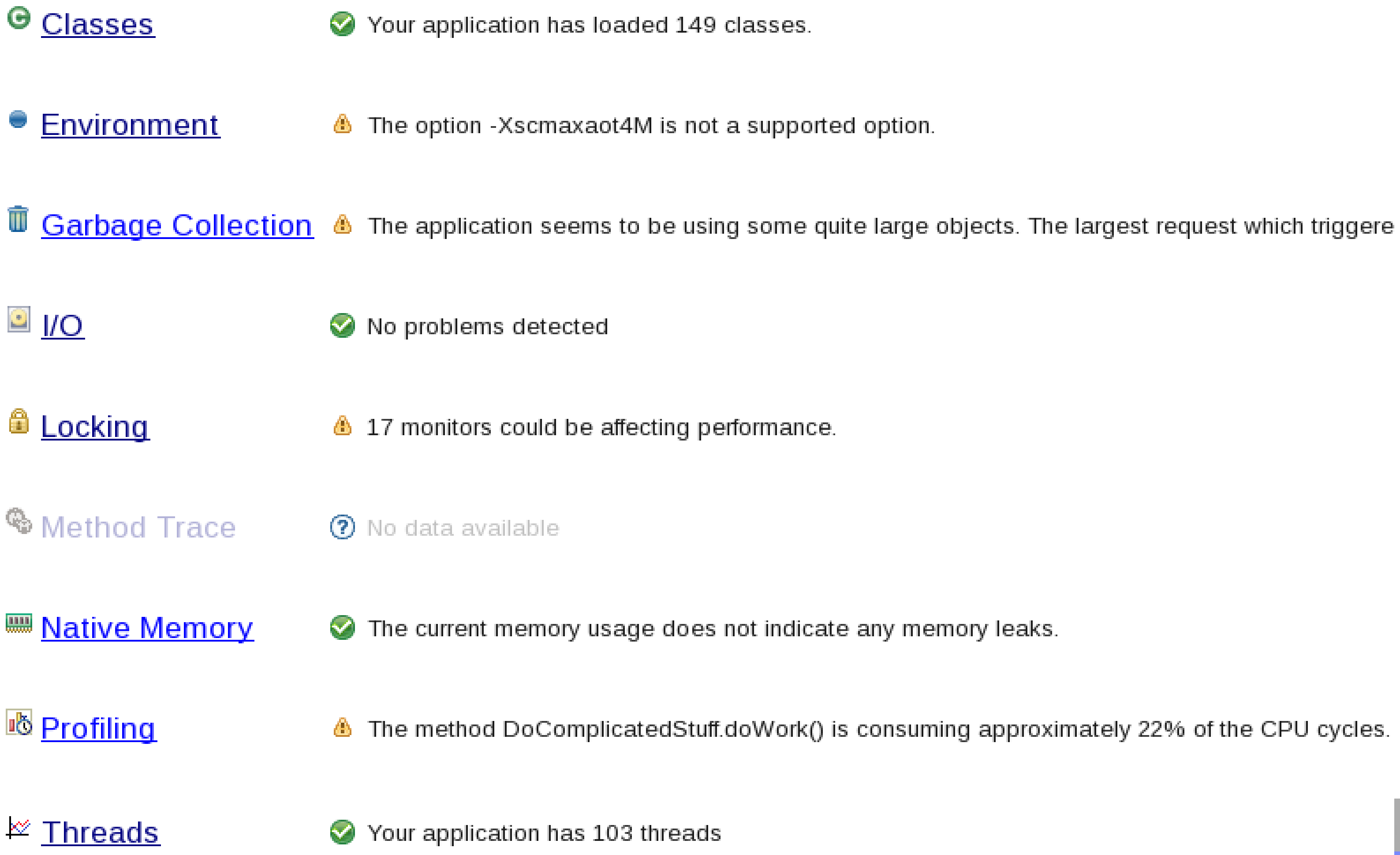
- When zooming in on any of the timelines, all of the data views update to just that time range
- This is great for comparing two time ranges, or focusing in on important time ranges (for example, removing the startup time so that it does not skew the statistics)
- However, the time ranges are specified in time from the start of the JVM instead of in absolute terms.
- The healthcenter.log can be used to approximate when the JVM started (when the agent reports it started).

Client Timeline

- To get an exact start time:
  - Load the HCD file
  - File > Export JVM Trace...
  - java com.ibm.jvm.format.TraceFormat hcd.trc
  - In the resulting .fmt file, search for:
    - First tracepoint: 03:43:38.131447000
  - This is in the GMT/UTC time zone.
  - The end of the HCD:
    - Last tracepoint : 03:59:37.406350000
  - Normally, it is important to use the same JVM version and get the J9TraceFormat.dat file, but in this case we just want the timestamp which does not depend on that info, so use any IBM JVM.

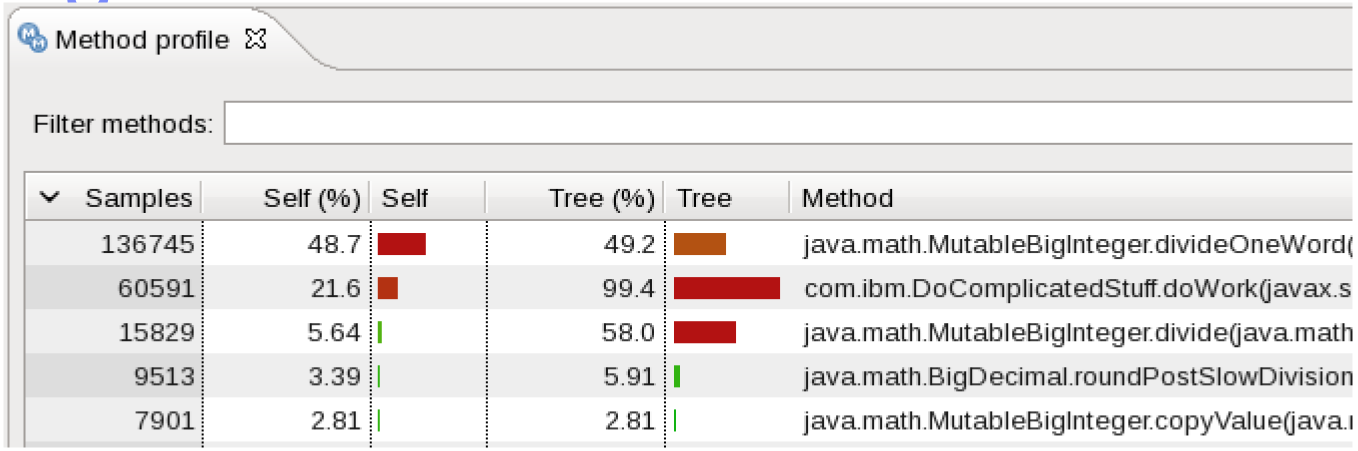
Client Summary

- Summary view highlights warnings



Classes	✓ Your application has loaded 149 classes.
Environment	⚠ The option -Xscmaxaot4M is not a supported option.
Garbage Collection	⚠ The application seems to be using some quite large objects. The largest request which triggere
I/O	✓ No problems detected
Locking	⚠ 17 monitors could be affecting performance.
Method Trace	? No data available
Native Memory	✓ The current memory usage does not indicate any memory leaks.
Profiling	⚠ The method DoComplicatedStuff.doWork() is consuming approximately 22% of the CPU cycles.
Threads	✓ Your application has 103 threads

Profiling



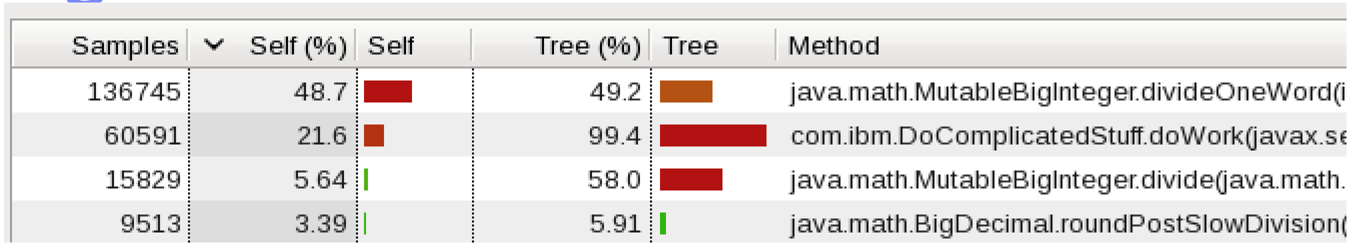
Method profile

Filter methods:

Samples	Self (%)	Self	Tree (%)	Tree	Method
136745	48.7	🟥	49.2	🟧	java.math.MutableBigInteger.divideOneWord(
60591	21.6	🟥	99.4	🟥	com.ibm.DoComplicatedStuff.doWork(javax.s
15829	5.64	🟩	58.0	🟥	java.math.MutableBigInteger.divide(java.math
9513	3.39	🟩	5.91	🟩	java.math.BigDecimal.roundPostSlowDivision
7901	2.81	🟩	2.81	🟩	java.math.MutableBigInteger.copyValue(java.ı

- Self (%): The percentage of samples taken while a particular method was being run at the top of the stack. This value is a good indicator of how expensive a method is...
  - This is roughly the CPU % usage of the Java CPU % usage. So, gather OS CPU stats with HC.
  - For example, if Java CPU% was 50% (let's say of all CPUs), and a method is 50% in Self, then that method used roughly 25% of all CPUs.
- Tree (%): The percentage of samples taken while a particular method was anywhere in the call stack. This value shows the percentage of time that this method, and methods it called (descendants), were being processed. This value gives a good guide to the areas of your application where most processing time is spent.
- Samples: Number of samples while a particular method was being run at the top of the stack.

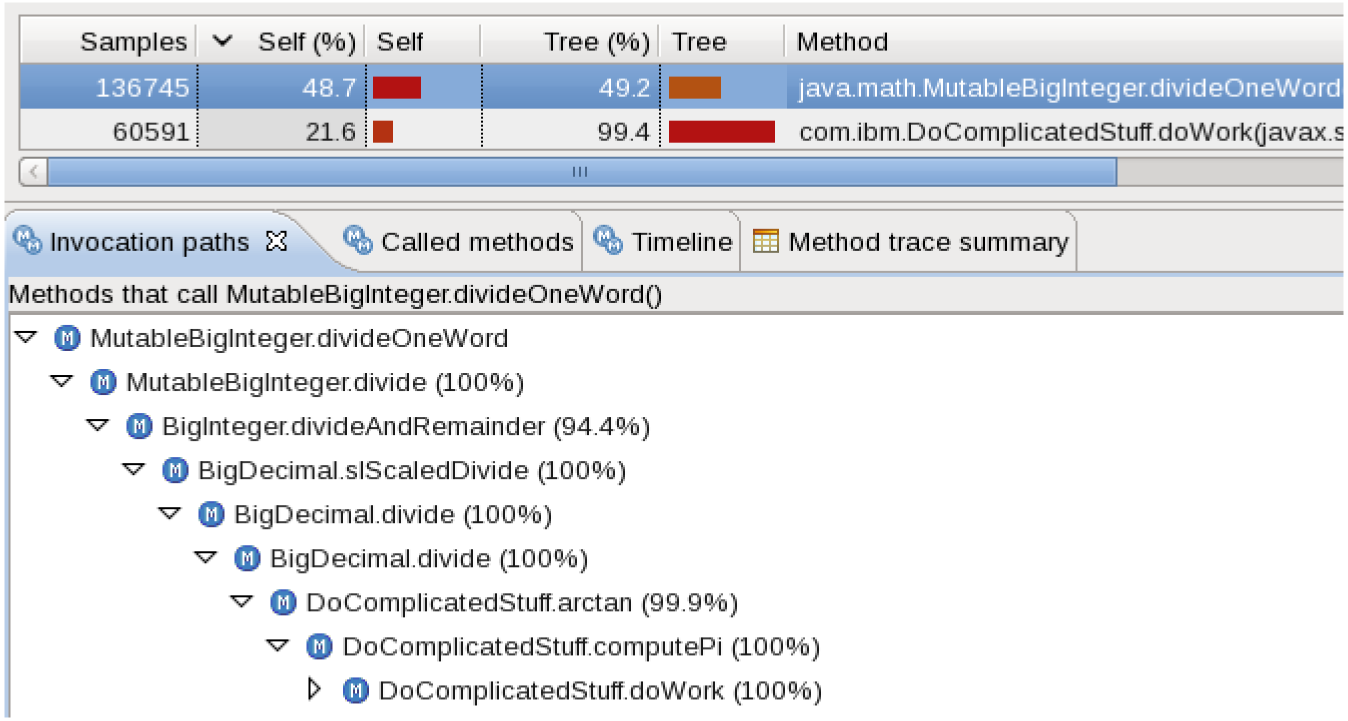
Profiling



Samples	Self (%)	Self	Tree (%)	Tree	Method
136745	48.7	🟥	49.2	🟧	java.math.MutableBigInteger.divideOneWord(i
60591	21.6	🟥	99.4	🟥	com.ibm.DoComplicatedStuff.doWork(javax.se
15829	5.64	🟩	58.0	🟥	java.math.MutableBigInteger.divide(java.math.
9513	3.39	🟩	5.91	🟩	java.math.BigDecimal.roundPostSlowDivision(

- This is a simple case where, if we sort by Self (%), MutableBigInteger.divideOneWord is at the top of sampled stacks almost half the time.
- Sorting by Tree % is sometimes useful, but WAS tends to have big stacks, so a lot of methods will be in a lot of the stacks. There are some heuristics here such as skipping the “common, do-nothing” methods, but this is an art more than a science.

Profiling



Samples	Self (%)	Self	Tree (%)	Tree	Method
136745	48.7	🟥	49.2	🟧	java.math.MutableBigInteger.divideOneWord
60591	21.6	🟥	99.4	🟥	com.ibm.DoComplicatedStuff.doWork(javax.s

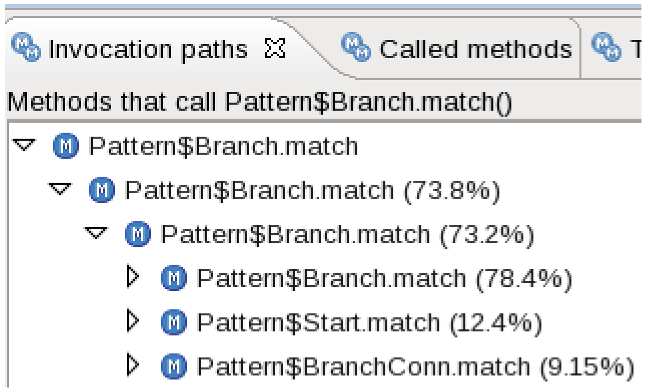
Invocation paths | Called methods | Timeline | Method trace summary

Methods that call MutableBigInteger.divideOneWord()

```
▽ MutableBigInteger.divideOneWord
  ▽ MutableBigInteger.divide (100%)
    ▽ BigInteger.divideAndRemainder (94.4%)
      ▽ BigDecimal.slScaledDivide (100%)
        ▽ BigDecimal.divide (100%)
          ▽ BigDecimal.divide (100%)
            ▽ DoComplicatedStuff.arctan (99.9%)
              ▽ DoComplicatedStuff.computePi (100%)
                ▷ DoComplicatedStuff.doWork (100%)
```

- Select the first row and the Invocation paths view will show who is calling this method.
- Each row contains a percentage of how many times that method called the above method out of all callers of that method.
- DoComplicatedWork.doWork is the primary caller, and this shows itself in the samples too.

Profiling

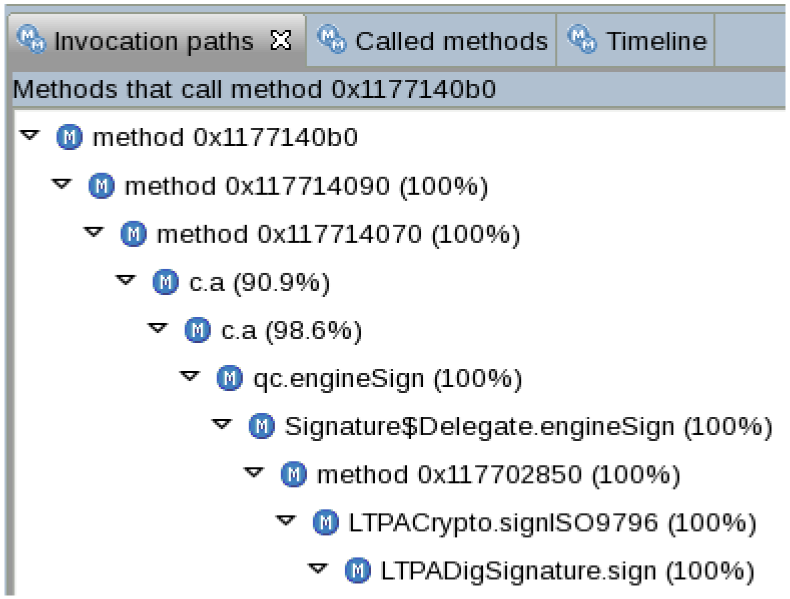


Methods that call Pattern$Branch.match()

- Pattern$Branch.match
  - Pattern$Branch.match (73.8%)
    - Pattern$Branch.match (73.2%)
      - Pattern$Branch.match (78.4%)
      - Pattern$Start.match (12.4%)
      - Pattern$BranchConn.match (9.15%)

- The percentages are not cumulative. For example, in the above, of all calls to the first row, the second row (Branch.match) was 73.8% of them. The third row called the second row 73.2% of the time. So the third row was the indirect caller of the first row .738*.732=54% of the time.

Profiling



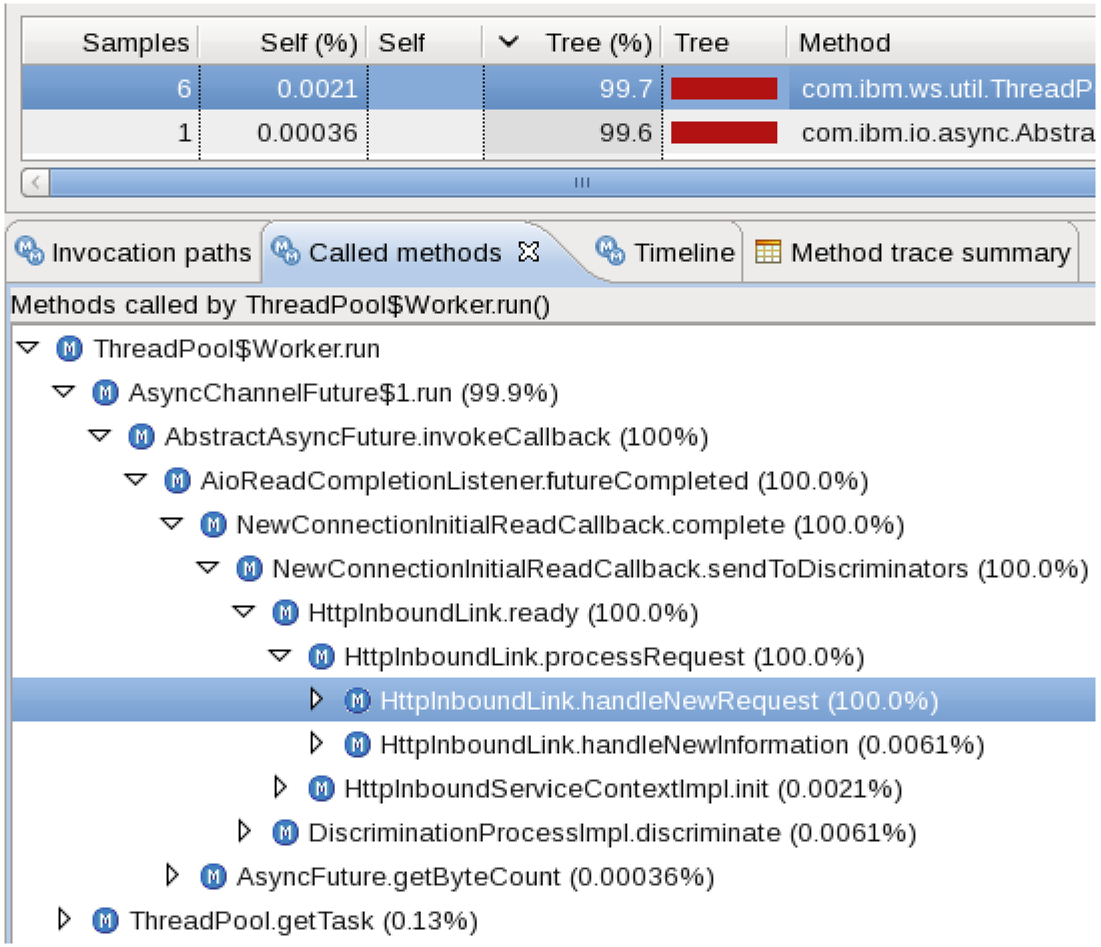
Methods that call method 0x1177140b0

```
method 0x1177140b0
  method 0x117714090 (100%)
    method 0x117714070 (100%)
      c.a (90.9%)
        c.a (98.6%)
          qc.engineSign (100%)
            Signature$Delegate.engineSign (100%)
              method 0x117702850 (100%)
                LTPACrypto.signISO9796 (100%)
                  LTPADigSignature.sign (100%)
```

- Some JVM versions and service releases, and some agent versions, may not be able to determine the method name and you'll see a hexadecimal address. This may also occur with methods that were used before the agent started (and the JVM version didn't have the capability to tell the agent after the fact). Two methods (pun!) to find the method:
  - Take a system dump and use DTFJ to find the method at that address.
  - Infer the method (or its general area) by looking at the invocation paths and called methods. Above, we can infer that method is in LTPA/security.

Profiling

- Breaking down "overall" profiles
  - Sort by Tree (%)
  - Select the first row which is usually ThreadPool$Worker.run()
  - Go to the Called Methods view
  - Follow down the highest percentages (may split)



Samples	Self (%)	Self	Tree (%)	Tree	Method
6	0.0021		99.7	🟥	com.ibm.ws.util.ThreadP
1	0.00036		99.6	🟥	com.ibm.io.async.Abstra

Methods called by ThreadPool$Worker.run()

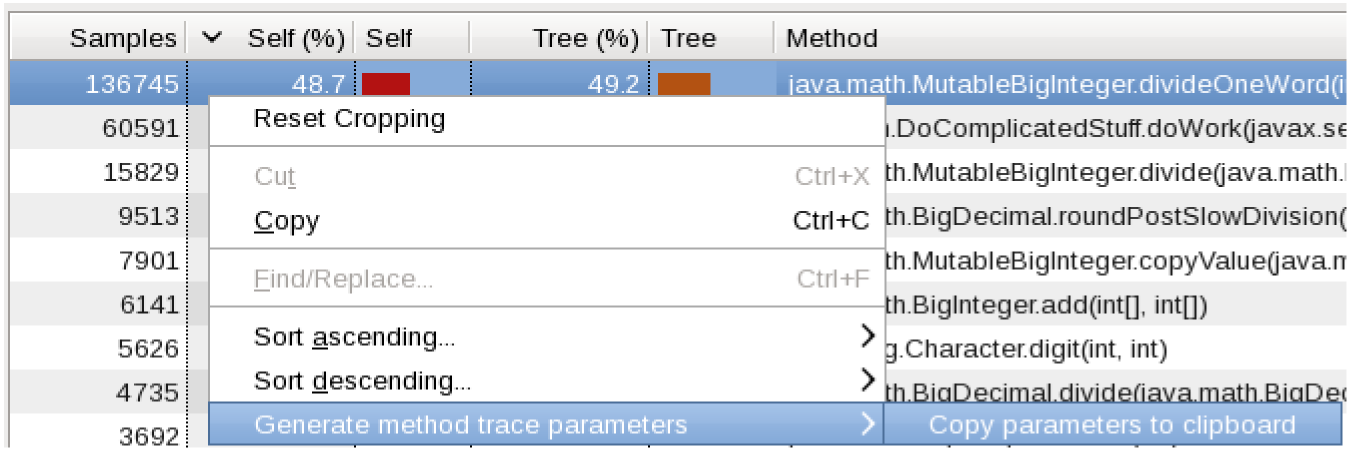
```
ThreadPool$Worker.run
  AsyncChannelFuture$1.run (99.9%)
    AbstractAsyncFuture.invokeCallback (100%)
      AioReadCompletionListener.futureCompleted (100.0%)
        NewConnectionInitialReadCallback.complete (100.0%)
          NewConnectionInitialReadCallback.sendToDiscriminators (100.0%)
            HttpInboundLink.ready (100.0%)
              HttpInboundLink.processRequest (100.0%)
                HttpInboundLink.handleNewRequest (100.0%)
                HttpInboundLink.handleNewInformation (0.0061%)
              HttpInboundServiceContextImpl.init (0.0021%)
            DiscriminationProcessImpl.discriminate (0.0061%)
        AsyncFuture.getByteCount (0.00036%)
  ThreadPool.getTask (0.13%)
```

Profiling

- If garbage collection analysis highlights System.gc calls, profiling view may have caught some of these and will show who called System.gc under Invocation Paths (Filter methods to System.gc)
- This is a statistical profiler, sampling the call stacks periodically rather than recording every method that is run. Methods that do not run often, or methods that run quickly, might not show in the profile list. Methods compiled by the Just-In-Time (JIT) compiler are profiled, but methods that have been inlined are not.
- Methods may be inlined at runtime! This will cause them to “drop down” in the profiling view and the calling method picks up the samples.

Profiling

- To get more details on particular methods:

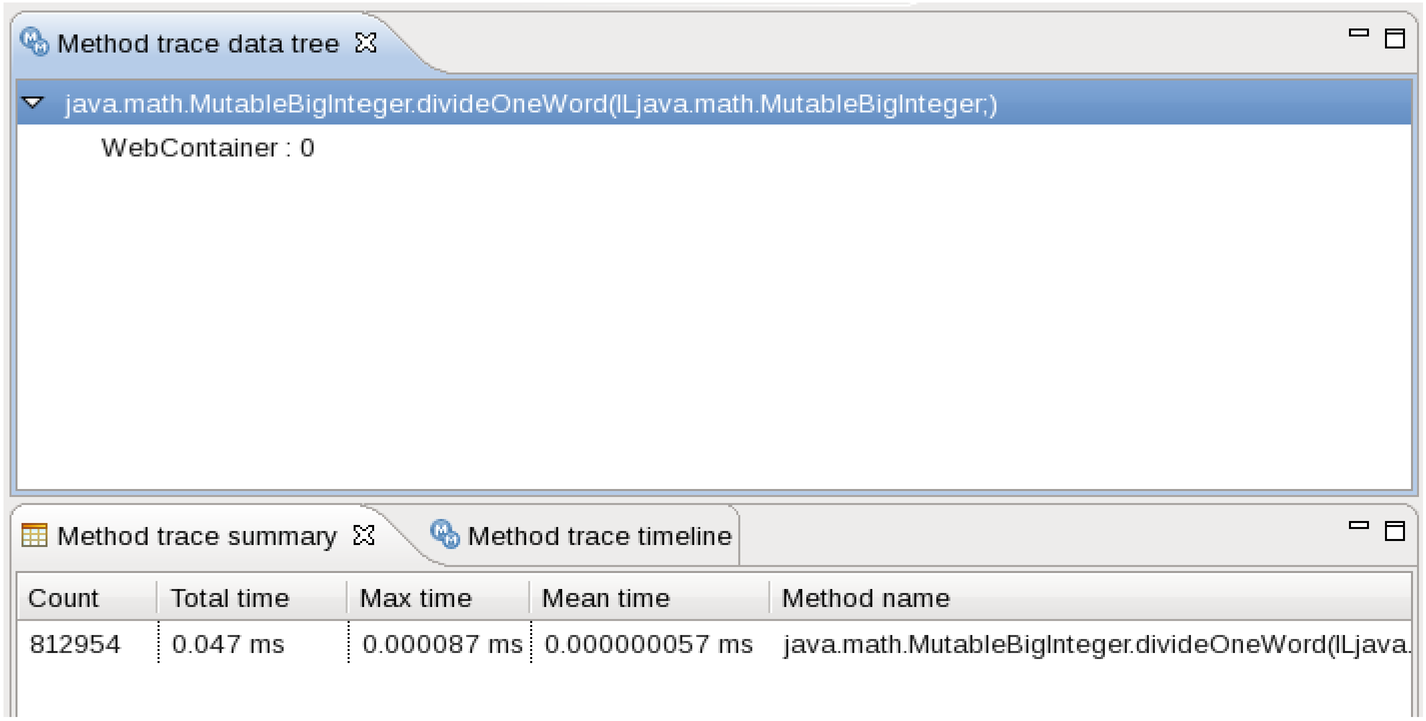


Samples	Self (%)	Self	Tree (%)	Tree	Method
136745	48.7	🟥	49.2	🟧	java.math.MutableBigInteger.divideOneWord(i
60591					ı.DoComplicatedStuff.doWork(javax.se
15829					th.MutableBigInteger.divide(java.math.l
9513					th.BigDecimal.roundPostSlowDivision(
7901					th.MutableBigInteger.copyValue(java.m
6141					th.BigInteger.add(int[], int[])
5626					g.Character.digit(int, int)
4735					th.BigDecimal.divide(java.math.BigDec
3692					

- Then restart the JVM with those generic JVM arguments
  - -Xtrace:maximal=mt,methods={"java/math/MutableBigInteger.divideOneWord"}

Profiling

- With method trace enabled



java.math.MutableBigInteger.divideOneWord(ILjava.math.MutableBigInteger;)
WebContainer : 0

Count	Total time	Max time	Mean time	Method name
812954	0.047 ms	0.000087 ms	0.000000057 ms	java.math.MutableBigInteger.divideOneWord(ILjava.

Low Mode

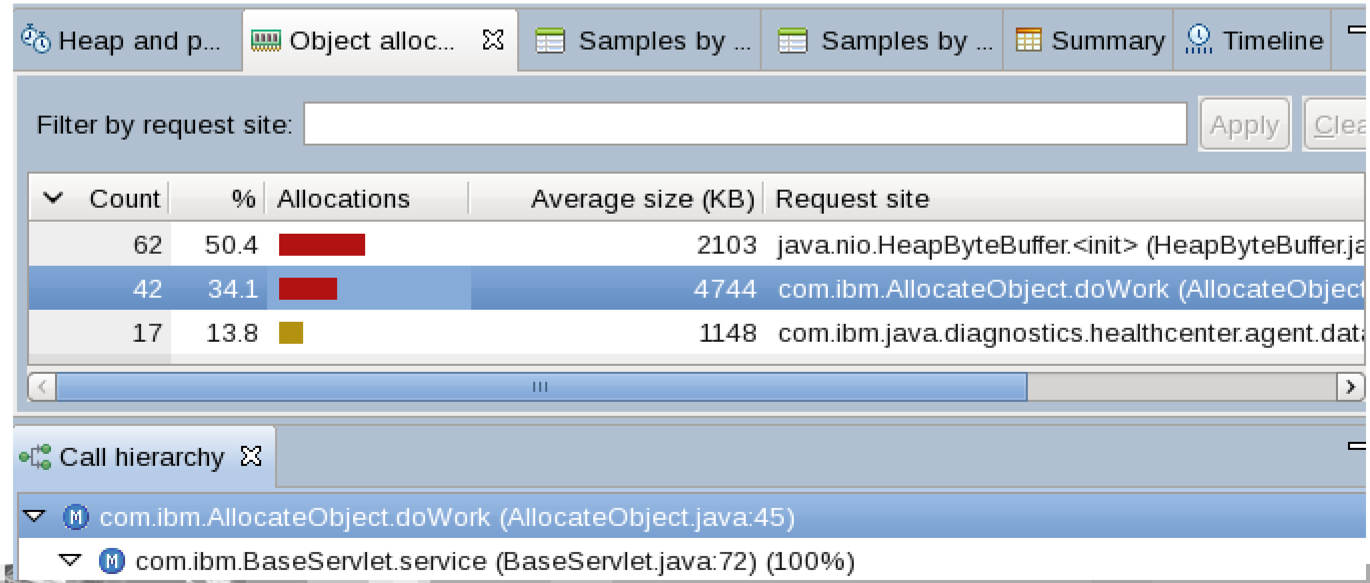
- -Xhealthcenter:level=low disables method profiling since this has the highest overhead and creates the most data. This would be useful if you wanted something else from health center (e.g. garbage collection, native memory, etc.) with less overhead.
- Low cannot be combined with headless (e.g. -Xhealthcenter:level=low,level=headless), so the way to do it is to use headless mode and then:
  - In jre/lib/ext there is a file called healthcenter.jar. If you unpack that you will find a file called TRACESourceConfiguration.properties and this is what defines which data is switched on by Trace. When we run in low mode, we turn off one of the profiling trace points. You can do this manually by editing this file and finding the entry "j9jit.16=on" and then changing it to "j9jit.16=off". If you repackage the jar up you should find that the amount of trace generated is a lot less (but you won't get method profiling).

Profiling Theory

- The Health Center profiler has the same limitations as other sampling profilers; it can't distinguish between a method which is invoked once but takes a really long time to run, and a method which is very quick but invoked frequently. It won't report methods which take a long time where the majority of that time is spent waiting, because it only reports methods which are using CPU. For a class which loops with sleep statements in each loop, Health Center won't report the looping method because that method isn't actually using any CPU, even though it has a long elapsed time.
- These limitations are the same for all sampling profilers. The alternative is to use a tracing profiler, which captures method entry and exit. It will report elapsed time in methods, but it won't report CPU utilisation. This has its own set of disadvantages, since it could suggest optimising a method where most time is actually spent waiting on an external input. Tracing profilers in general also have far higher overhead than sampling profilers. In order to keep the overhead manageable, they tend to focus on just some sections of the codebase, and will only report elapsed time for certain methods or classes. This is risky, since it requires the performance analyst to guess which areas are causing performance problems before doing the performance analysis - serious bottlenecks could be missed entirely.

Large Object Allocations

- Properties:
  - -Dcom.ibm.java.diagnostics.healthcenter.allocation.threshold.low=BYTES
  - -Dcom.ibm.java.diagnostics.healthcenter.allocation.threshold.high=BYTES
- Example:
  - -Dcom.ibm.java.diagnostics.healthcenter.allocation.threshold.low=1048576
- Under Garbage Collection > Object Allocations

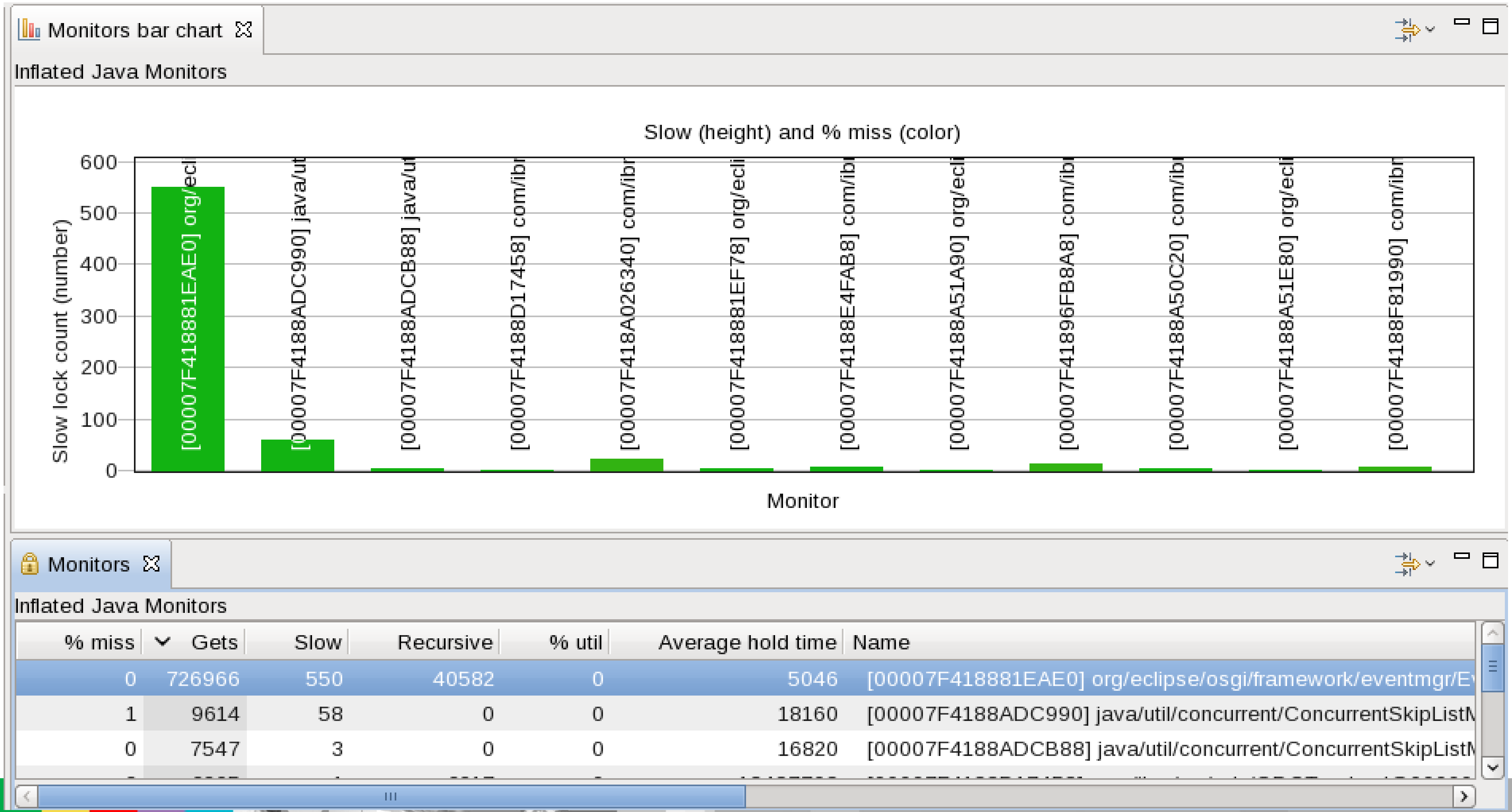


Filter by request site:

Count	%	Allocations	Average size (KB)	Request site
62	50.4	🟥	2103	java.nio.HeapByteBuffer.<init> (HeapByteBuffer.ja
42	34.1	🟥	4744	com.ibm.AllocateObject.doWork (AllocateObject
17	13.8	🟨	1148	com.ibm.java.diagnostics.healthcenter.agent.dat

```
com.ibm.AllocateObject.doWork (AllocateObject.java:45)
  com.ibm.BaseServlet.service (BaseServlet.java:72) (100%)
```

Locking


Monitors bar chart
Inflated Java Monitors
Slow (height) and % miss (color)
Slow lock count (number)
600
500
400
300
200
100
0
[00007F418881EAE0] org/ecl
[00007F4188ADC990] java/ut
[00007F4188ADCB88] java/ut
[00007F4188D17458] com/ibr
[00007F418A026340] com/ibr
[00007F418881EF78] org/ecli
[00007F4188E4FAB8] com/ibr
[00007F4188A51A90] org/ecli
[00007F41896FB8A8] com/ibr
[00007F4188A50C20] com/ibr
[00007F4188A51E80] org/ecli
[00007F4188F81990] com/ibr
Monitor
Monitors

Inflated Java Monitors

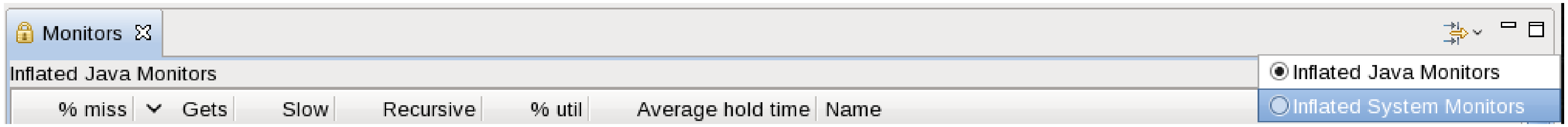
% miss	Gets	Slow	Recursive	% util	Average hold time	Name
0	726966	550	40582	0	5046	[00007F418881EAE0] org/eclipse/osgi/framework/eventmgr/E
1	9614	58	0	0	18160	[00007F4188ADC990] java/util/concurrent/ConcurrentSkipList
0	7547	3	0	0	16820	[00007F4188ADCB88] java/util/concurrent/ConcurrentSkipList

Locking

- Gets: The total number of times the lock has been taken while it was inflated.
- Slow: The total number of non-recursive lock acquires for which the requesting thread had to wait for the lock because it was already owned by another thread.
- % miss: The percentage of the total Gets, or acquires, for which the thread trying to enter the lock on the synchronized code had to block until it could take the lock. % miss = (Slow / Gets) * 100
  - A high % miss shows that frequent contention occurs on the synchronized resource protected by the lock. This contention might be preventing the Java application from scaling further.
  - If a lock has a high % miss value, look at the average hold time and % util. If % util and average hold time are both high, you might need to reduce the amount of work done while the lock is held. If % util is high but the average hold time is low, you might need to make the resource protected by the lock more granular to separate the lock into multiple locks.
- Recursive: The total number of recursive acquires. A recursive acquire occurs when the requesting thread already owns the monitor.
- % util: The amount of time the lock was held, divided by the amount of time the output was taken over.
- Average hold time: The average amount of time the lock was held, or owned, by a thread. For example, the amount of time spent in the synchronized block, measured in processor clock ticks.

Locking

- The height of the bars represents the slow lock count and is relative to all the columns in the graph. A slow count occurs when the requested monitor is already owned by another thread and the requesting thread is blocked.
- The color of each bar is based on the value of the % miss column in the table. The gradient moves from red (100%), through yellow (50%), to green (0%). A red bar indicates that the thread blocks every time that the monitor is requested. A green bar indicates a thread that never blocks.
- Show internal JVM monitor lock information:


Monitors

Inflated Java Monitors

% miss	Gets	Slow	Recursive	% util	Average hold time	Name

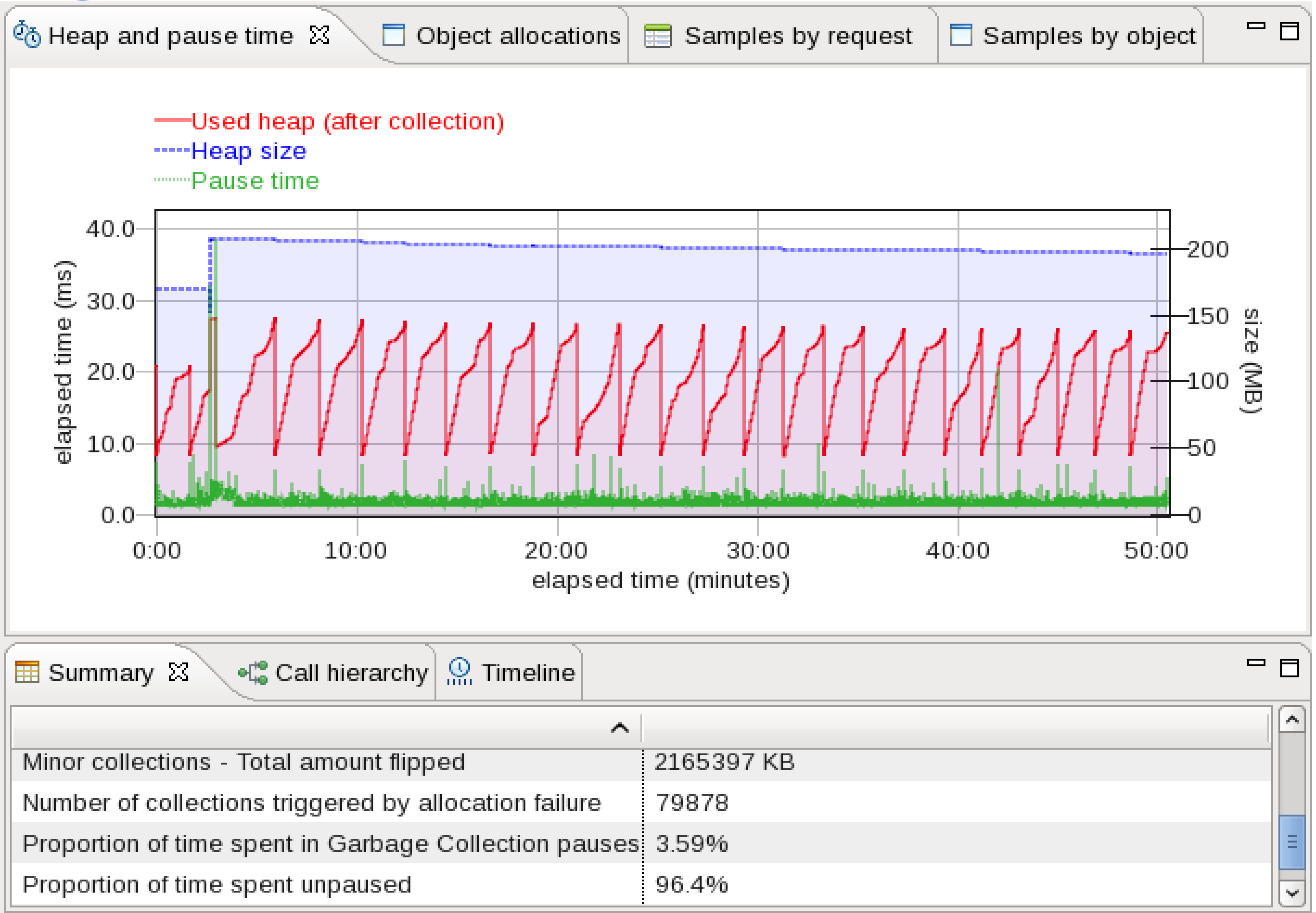
[✓ Inflated Java Monitors] Inflated System Monitors

- If a lock is held while a garbage collection runs, this time is removed from the statistics of that lock.

Lock Name

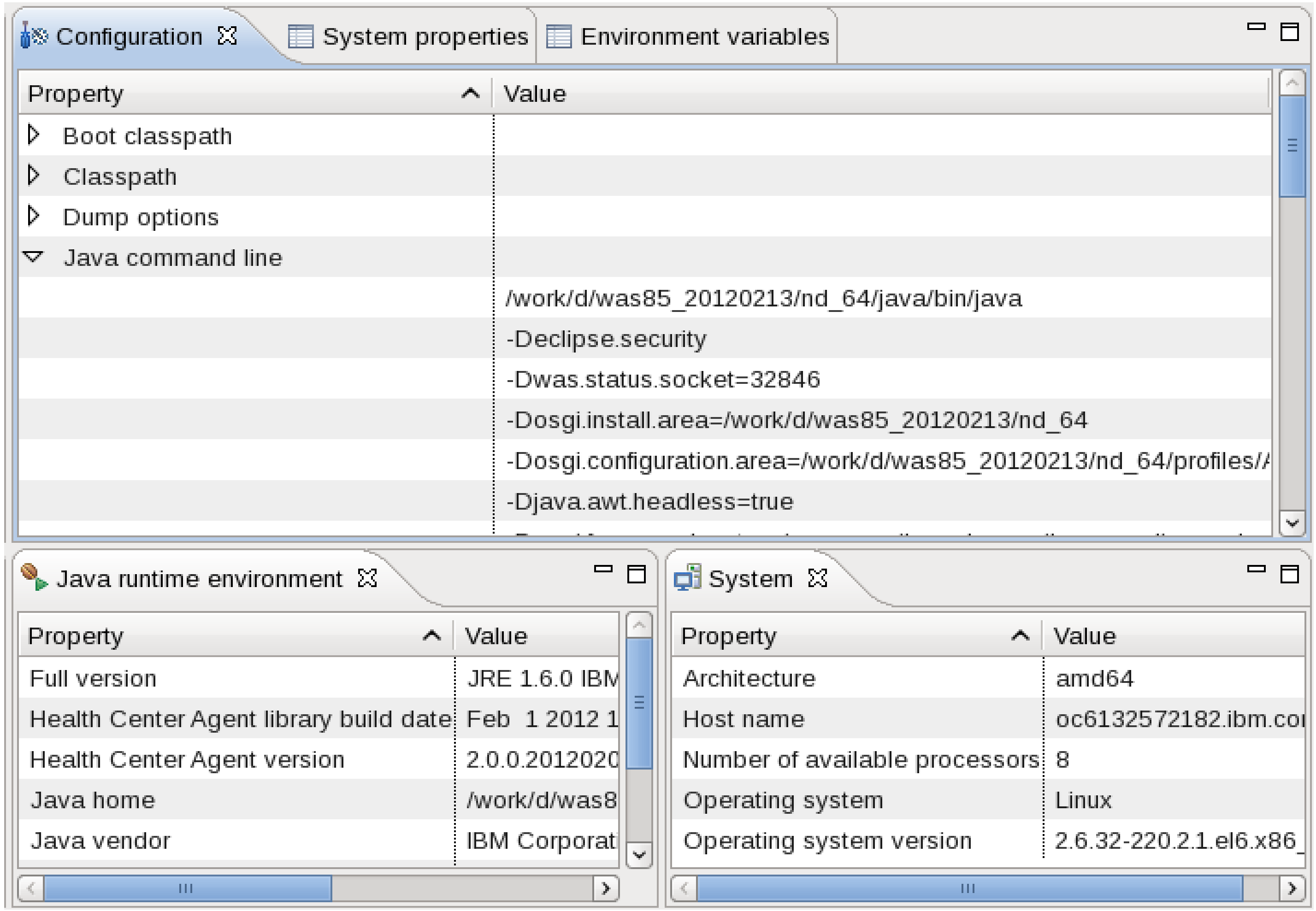
- Lock name of the form:
  - [00007F418A0265E0] java/lang/Object@0000000004A2E408 (Object)
- The number after the @ is the object address that can be looked up in a system dump or heapdump.

Garbage Collection



Minor collections - Total amount flipped	2165397 KB
Number of collections triggered by allocation failure	79878
Proportion of time spent in Garbage Collection pauses	3.59%
Proportion of time spent unpaused	96.4%

Environment

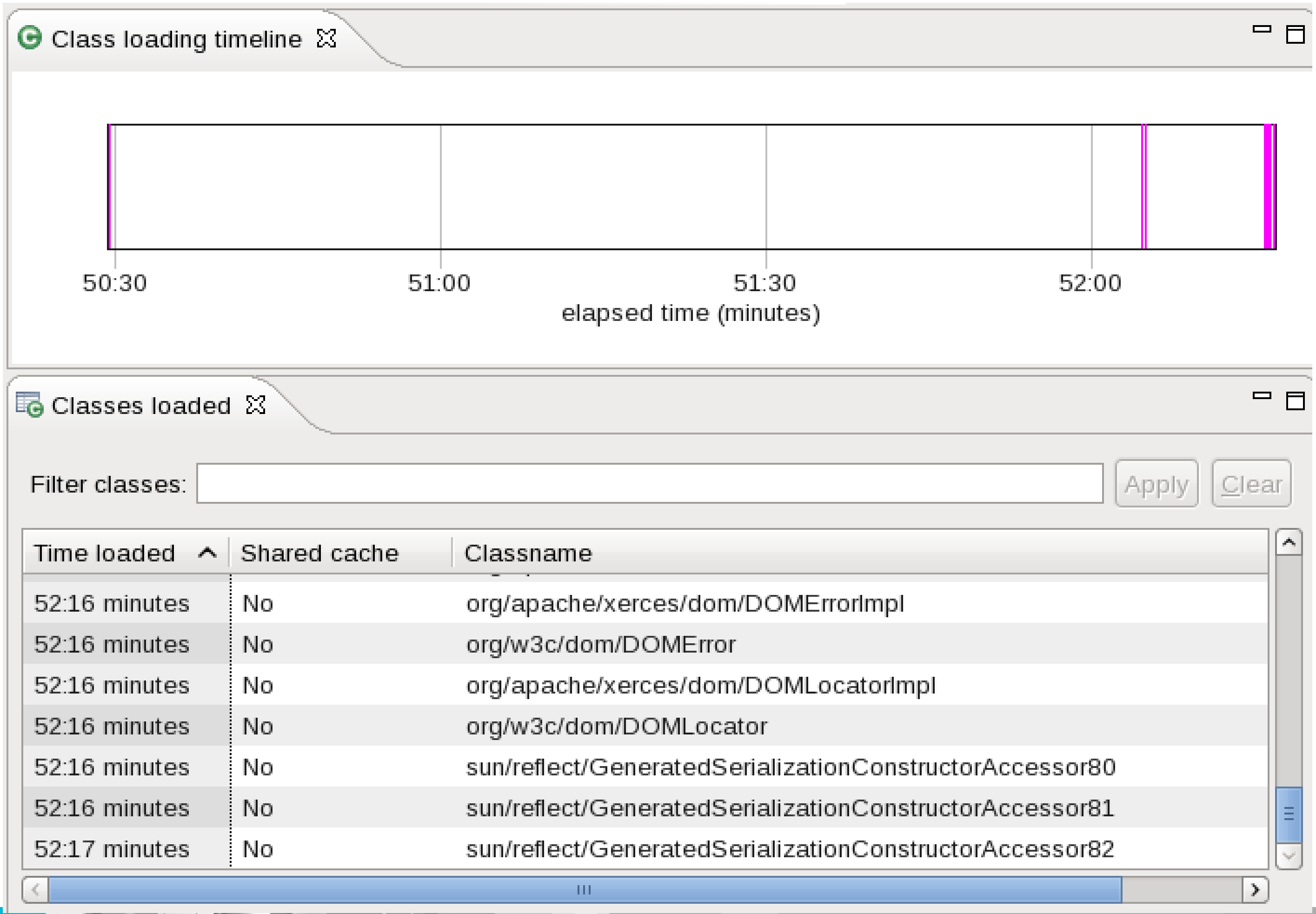


Property	Value
Boot classpath	
Classpath	
Dump options	
Java command line	
	/work/d/was85_20120213/nd_64/java/bin/java
	-Declipse.security
	-Dwas.status.socket=32846
	-Dosgi.install.area=/work/d/was85_20120213/nd_64
	-Dosgi.configuration.area=/work/d/was85_20120213/nd_64/profiles/
	-Djava.awt.headless=true

Property	Value
Full version	JRE 1.6.0 IBM
Health Center Agent library build date	Feb 1 2012 1
Health Center Agent version	2.0.0.2012020
Java home	/work/d/was8
Java vendor	IBM Corporat

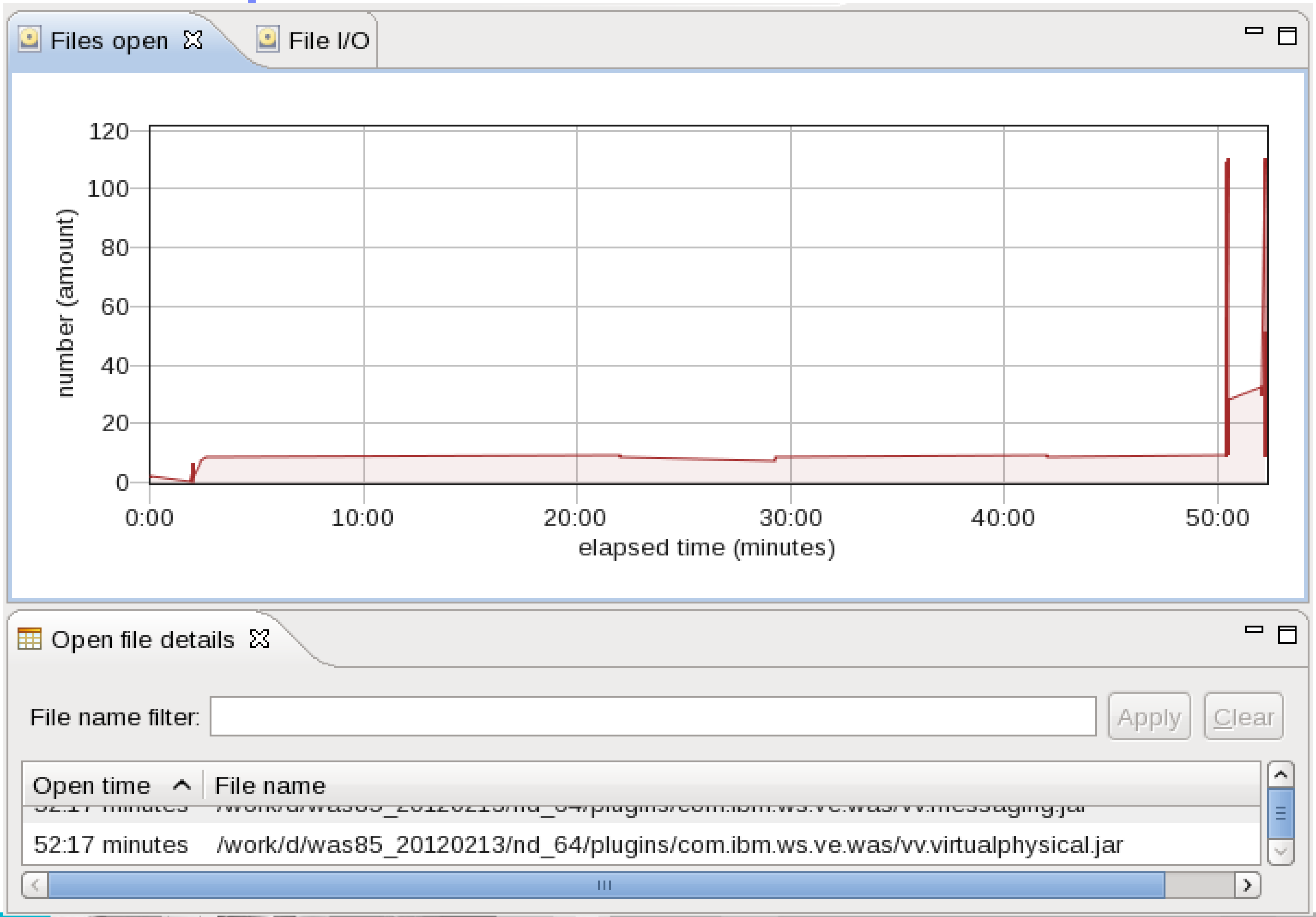
Property	Value
Architecture	amd64
Host name	oc6132572182.ibm.co
Number of available processors	8
Operating system	Linux
Operating system version	2.6.32-220.2.1.el6.x86_

Classes



Time loaded	Shared cache	Classname
52:16 minutes	No	org/apache/xerces/dom/DOMErrorImpl
52:16 minutes	No	org/w3c/dom/DOMError
52:16 minutes	No	org/apache/xerces/dom/DOMLocatorImpl
52:16 minutes	No	org/w3c/dom/DOMLocator
52:16 minutes	No	sun/reflect/GeneratedSerializationConstructorAccessor80
52:16 minutes	No	sun/reflect/GeneratedSerializationConstructorAccessor81
52:17 minutes	No	sun/reflect/GeneratedSerializationConstructorAccessor82

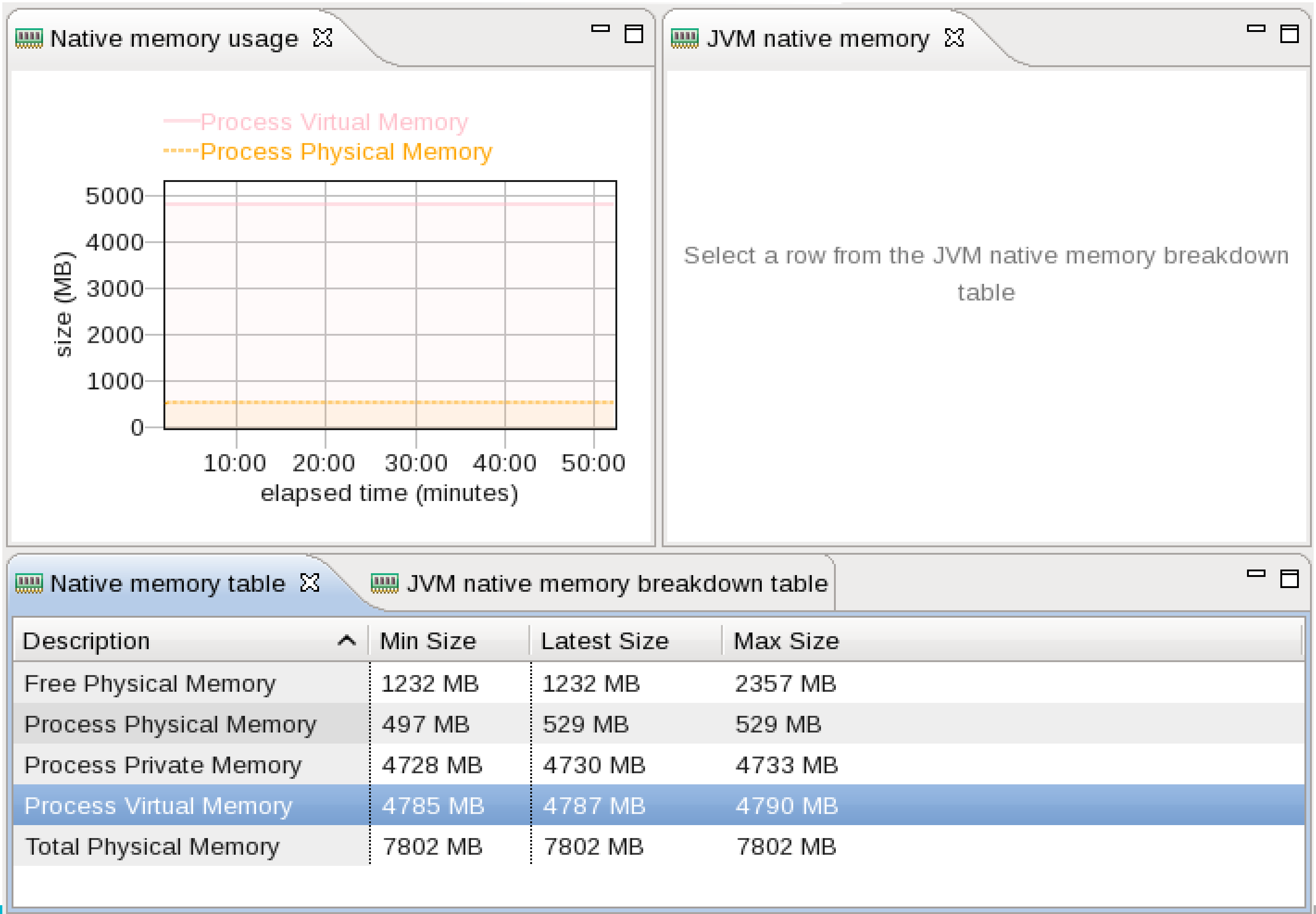
I/O Files Open



File name filter:

Open time	File name
52:17 minutes	/work/d/was85_20120213/nd_64/plugins/com.ibm.ws.ve.was/vv.messaging.jar
52:17 minutes	/work/d/was85_20120213/nd_64/plugins/com.ibm.ws.ve.was/vv.virtualphysical.jar

Native Memory

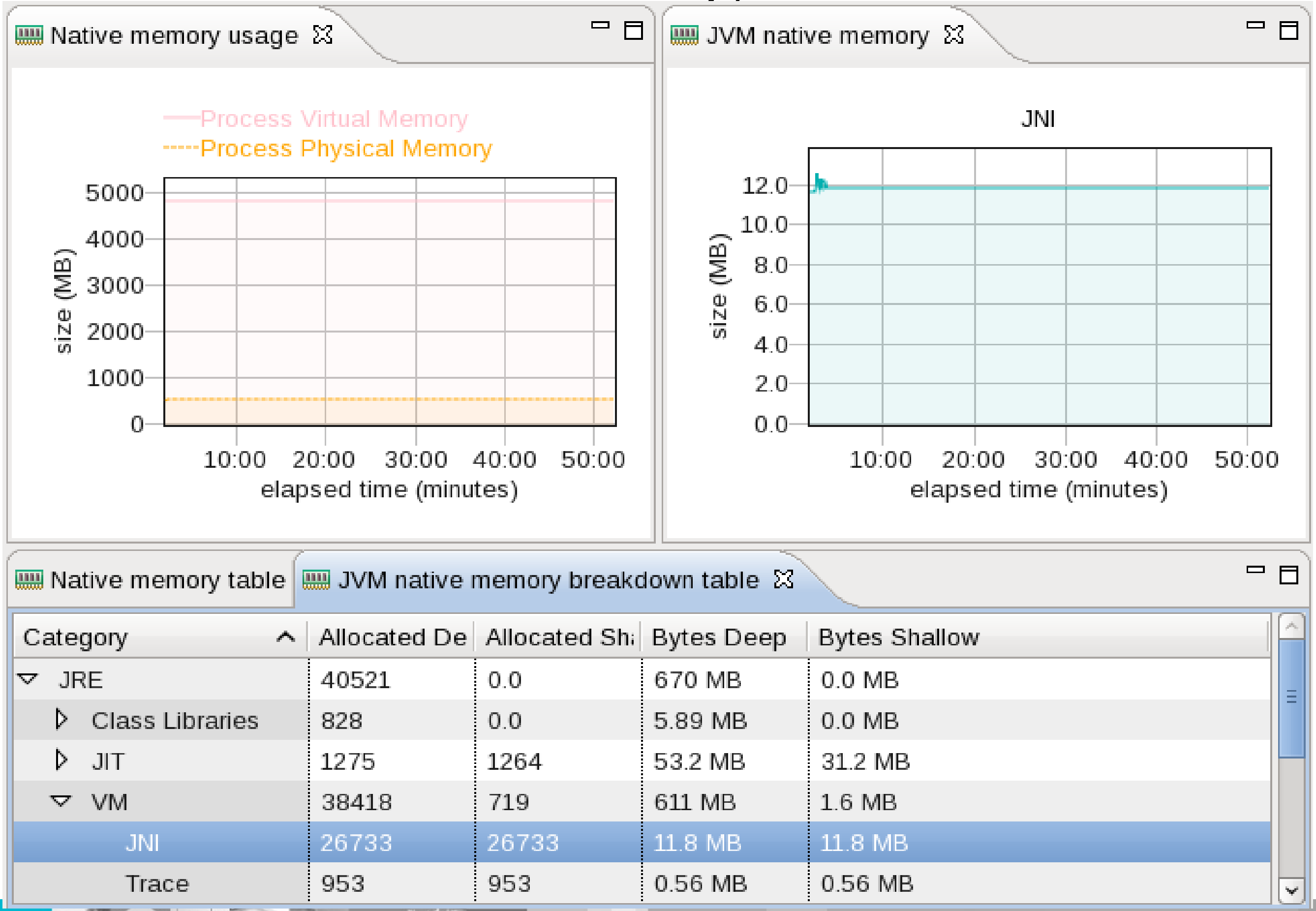


Select a row from the JVM native memory breakdown table

Description	Min Size	Latest Size	Max Size
Free Physical Memory	1232 MB	1232 MB	2357 MB
Process Physical Memory	497 MB	529 MB	529 MB
Process Private Memory	4728 MB	4730 MB	4733 MB
Process Virtual Memory	4785 MB	4787 MB	4790 MB
Total Physical Memory	7802 MB	7802 MB	7802 MB

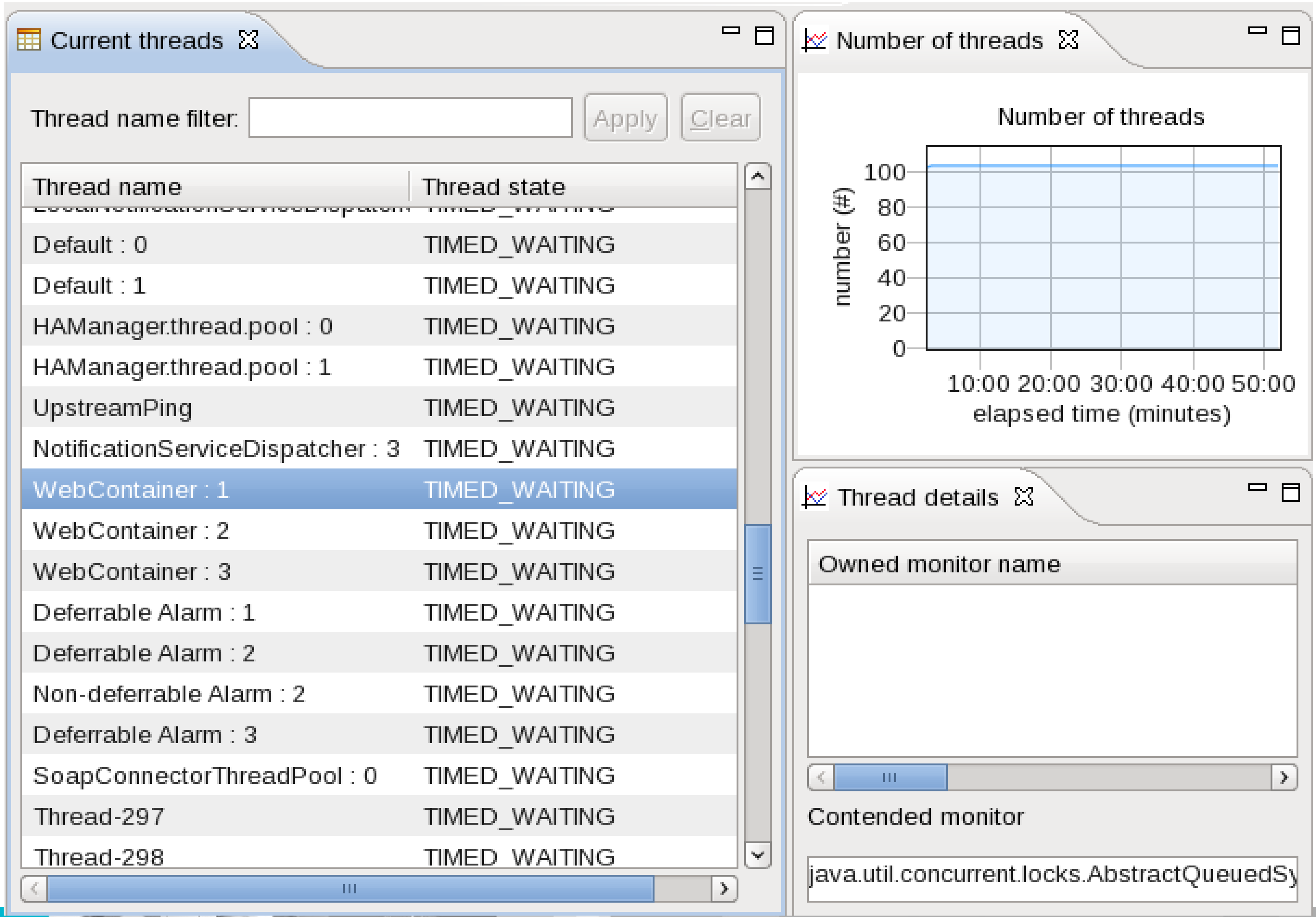
Native Memory - Breakdown

- Select Process Virtual Memory, then click breakdown



Category	Allocated De	Allocated Sh	Bytes Deep	Bytes Shallow
JRE	40521	0.0	670 MB	0.0 MB
Class Libraries	828	0.0	5.89 MB	0.0 MB
JIT	1275	1264	53.2 MB	31.2 MB
VM	38418	719	611 MB	1.6 MB
JNI	26733	26733	11.8 MB	11.8 MB
Trace	953	953	0.56 MB	0.56 MB

Threads


Current threads

Thread name filter: Apply Clear

Thread name	Thread state
Default : 0	TIMED_WAITING
Default : 1	TIMED_WAITING
HAManager.thread.pool : 0	TIMED_WAITING
HAManager.thread.pool : 1	TIMED_WAITING
UpstreamPing	TIMED_WAITING
NotificationServiceDispatcher : 3	TIMED_WAITING
WebContainer : 1	TIMED_WAITING
WebContainer : 2	TIMED_WAITING
WebContainer : 3	TIMED_WAITING
Deferrable Alarm : 1	TIMED_WAITING
Deferrable Alarm : 2	TIMED_WAITING
Non-deferrable Alarm : 2	TIMED_WAITING
Deferrable Alarm : 3	TIMED_WAITING
SoapConnectorThreadPool : 0	TIMED_WAITING
Thread-297	TIMED_WAITING
Thread-298	TIMED_WAITING

Number of threads

Thread details

Owned monitor name

Contended monitor

java.util.concurrent.locks.AbstractQueuedSy

Getting Help

- Java Tools Email: javatool@uk.ibm.com
- Public Forum: http://www.ibm.com/developerworks/forums/forum.jspa?forumID=1461

MustGather

- If agent < version 2, upgrade the Health Center agent and stop/start the JVM:
  - Download: http://publib.boulder.ibm.com/infocenter/hctool/v1r0/topic/com.ibm.java.diagnostics.healthcenter.doc/topics/installingagent.html
  - cd <WAS>/java/
  - ./bin/jar xvf $FILE.zip
- If running WAS >= 7.0.0.7 on a non/z-OS platform, you may enable health center with late attach:
  - cd <WAS>/java/bin
  - ./java -jar ../jre/lib/ext/healthcenter.jar ID=$PID -Dcom.ibm.java.diagnostics.healthcenter.data.collection.level=headless -Dcom.ibm.java.diagnostics.healthcenter.headless.files.max.size=268435456 -Dcom.ibm.java.diagnostics.healthcenter.headless.files.to.keep=8
- Otherwise, restart the JVM with the following generic JVM arguments:
  - -Xhealthcenter:level=headless -Dcom.ibm.java.diagnostics.healthcenter.headless.files.max.size=268435456 -Dcom.ibm.java.diagnostics.healthcenter.headless.files.to.keep=8
- Start the WAIT data collector: https://wait.researchlabs.ibm.com/submit/dataCollector.html
- Reproduce the problem, stop the JVM(s) completely, stop the WAIT collector, upload *.hcd, WAS logs, <TEMP>/healthcenter.*.log, and waitData

Conclusion

- In summary, IBM Java Health Center is an extremely powerful tool which has a very low-overhead sampling profiler (among other features) and can be used on recent versions of WAS (>= 6.1.0.27 and >= 7.0.0.5) to determine the root cause of performance issues.
  - In general, recommend the headless mode to customers, which does involve ensuring the latest agent binaries are installed.

Reference

- Summary of Links:
  - Product Page: http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/
  - Documentation: http://publib.boulder.ibm.com/infocenter/hctool/v1r0/topic/com.ibm.java.diagnostics.healthcenter.doc/topics/introduction.html
  - Download the latest agent: http://publib.boulder.ibm.com/infocenter/hctool/v1r0/topic/com.ibm.java.diagnostics.healthcenter.doc/topics/installingagent.html
  - Download the visualization client:
    - Download IBM Support Assistant: http://www-01.ibm.com/software/support/isa/
    - Then install the Health Center Tool Add-on

Additional WebSphere Product Resources

- Learn about upcoming WebSphere Support Technical Exchange webcasts, and access previously recorded presentations at: http://www.ibm.com/software/websphere/support/supp_tech.html
- Discover the latest trends in WebSphere Technology and implementation, participate in technically-focused briefings, webcasts and podcasts at: http://www.ibm.com/developerworks/websphere/community/
- Join the Global WebSphere Community: http://www.websphereusergroup.org
- Access key product show-me demos and tutorials by visiting IBM Education Assistant: http://www.ibm.com/software/info/education/assistant
- View a webcast replay with step-by-step instructions for using the Service Request (SR) tool for submitting problems electronically: http://www.ibm.com/software/websphere/support/d2w.html
- Sign up to receive weekly technical My Notifications emails: http://www.ibm.com/software/support/einfo.html

Connect with us!

1. **Get notified on upcoming webcasts**
   Send an e-mail to wsehelp@us.ibm.com with subject line "wste subscribe" to get a list of mailing lists and to subscribe
2. **Tell us what you want to learn**
   Send us suggestions for future topics or improvements about our webcasts to wsehelp@us.ibm.com
3. **Be connected!**
   Connect with us on Facebook
   Connect with us on Twitter

Questions and Answers