WOPR18 Experience Report
Author: Neil Davies, Equinox Limited, Auckland New Zealand
http://www.equinox.co.nz

Chasing Tornadoes

Data Visualisation and Analysis during Performance Troubleshooting

The Assignment

A couple of years ago I was called to a site to help solve urgent performance issues at a company implementing a new logistics package. The incumbent testing consultancy had been testing the application for 12 months, with on-going performance issues unable to be understood, explained or resolved by the testers, vendors or technical teams. It was a hostile environment, with legal action pending between the client and vendors, and restricted communication and access to the vendors and technical teams.

This assignment, while not complex, is an example of how availability of the right data, at the right level, and analysed in the right way, can significantly improve the ability to find and resolve performance issues.

In this assignment, even with the bureaucracy and communication challenges described, the key performance problems were found and resolved in less than 2 weeks.

Assessment

Some background information:

- Technologies of interest: Websphere App Server, Sun One LDAP Server, IBM HTTP Server, DB2, AIX.
- *Neoload* was being used as test tool by an external testing company.
- Two performance issues had been identified, with the main one described as “occasional poor performance” or “performance spikes”.

From the available information, I found:

- Significant challenges with the test tool:
    a. The testing tool reports only averages per time period.
    b. No ability to get at the raw transaction data.
    c. Limited visualisations only designed for “averages”
    d. Functionality not suitable for identifying and qualifying the main type of performance issue.
- Inappropriate use of statistical metrics meant that wrong or no conclusions were being made about performance:
    a. Pseudo-percentiles were the main statistic used to quantify response (ie the metric is represented as a “95th percentile response”, but is actually “percentile of average”) This is bad for two reasons:
        i. The results are *positively misleading* by producing better metrics than actual percentiles.

ii. The nature of the performance issue and relative low quantity of samples means that percentiles would always be inconsistent between tests.

b. *Percent Improvement* was being used to measure success of a test. Because of the inconsistency of results, these comparisons were meaningless.

- Attempted resolutions were not evidence based.

Combining all of the above, the whole exercise had little chance of success.

Step 1: Issue Qualification

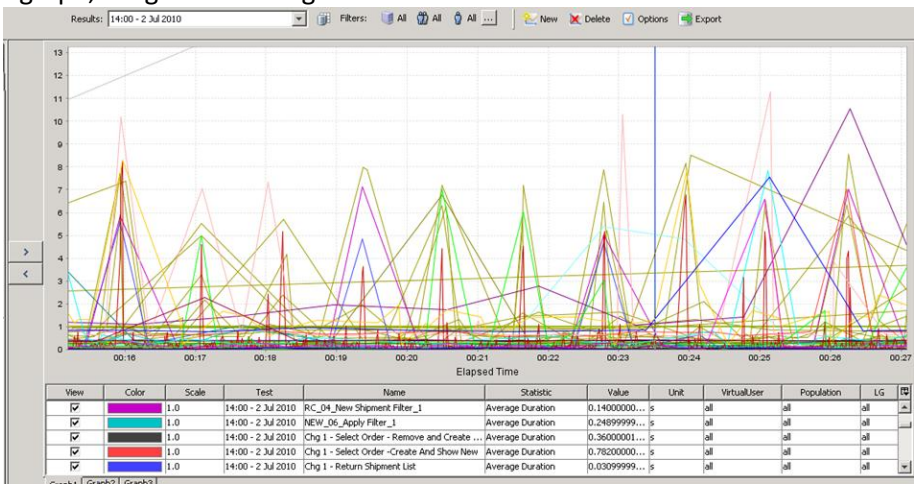
The first goal is to gather data allowing you to understanding the nature, frequency, impact, scope etc of any issues. (Just knowing this can start to point to possible causes.)

To do this, I first want to see response times of *all* requests made. Without that, we may be missing something – either an issue or valuable information.

What can we get from within the tool?

We can not get raw data from the test tool, but we can try to get as close as possible: The sampling interval can be set way down to 1 second interval (enough to *see* intermittent performance issues with magnitudes greater than that). If transaction rates are appropriately low (ie normally less than one per second) this will be "close to" individual points.

Re-running a test, then painstakingly drilling down and selecting each and every transaction and adding it to a graph, we get something that looks like this:



This is not very convincing and clear. But we now some information on the issue: The approximate interval and scale (ie about 1 "spike" per minute, of about 5-8 seconds). But we can do better.

Gathering data externally to the Test Tool:
This is a web application. Luckily, the *equivalent* to raw timing data for individual HTTP requests can be extracted from any apache based Web Server quite easily as follows:

1. Add %D to the access log format.
2. Response times are then recorded with each request in microseconds. Timestamps in the log are recorded to 1 second time resolution. Finer grained would be preferably, but we have to live with what we can get.
3. Parse and load into your favourite graphing / visualisation tool…

Here is a graph using *Tableau*, a data visualisation tool normally used in the Business Intelligence space. The graph is a variant of a "scatter" graph; but rather than a single point per HTTP request, I use a *horizontal line* in time from the start-time to end-time of the request. The response time is on the vertical axis.

This type of graph is excellent for showing points of freezing in a system, more so than standard scatter graphs. What we can see in the graph:

- Groups of requests are stacked up -- looking like ***tornadoes***. The start times (left hand side) are staggered, and the completion time (right hand side) are aligned. Each stack indicates that the requests are pausing while being held up on the *same internal resource*. Once that resource is released, all the requests complete quickly.
- Two types of pauses can be seen: one regularly about 7 seconds long, and a larger one.
- The pauses are independent (the first larger pause overlaps a smaller one).
- The smaller pause looks to be exactly 60 seconds *between* occurrences? (A scheduled process perhaps? this is an important clue!)
- Neither of them are system-wide pauses (other requests can occur during both pauses)

Other points:

- Adding drop-lines can help visualising of this sort of issue, by more clearly identify the aligned end points.
- You can do something like this with Microsoft Excel using horizontal error-bars with custom values, however it is much harder for a number of reasons:
    a. Getting them on there in the first place is more difficult
    b. Dreadful graphing performance with any larger number of data points
    c. Very limited analysis ability (eg filtering, colouring by transaction etc)

Step 2: Drilling-down with Statistical Sampling

The large spike was not seen often, so I concentrated on the smaller repeatable one.
Application experts could not provide any commonality between the URLs involved in the issue and those not, so it was up to us to find out more information of what was happening.

Statistical Sampling of the Application Server thread state was performed using regular stack dumps, to find out what the threads were doing during pauses.

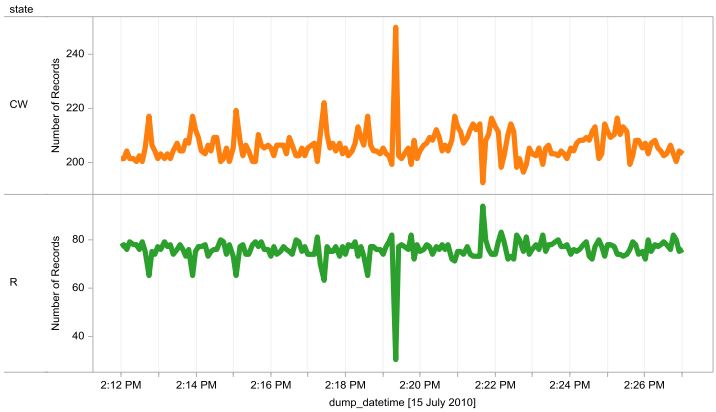
- Data gathering was hindered by IBM bug: console log stack traces were blank, so had to use javacore files. (Much higher overhead, and about 3.5MB dumped for each one!)
- With freezes of what looks like 7 seconds, a snapshot every 5 seconds should just overlap every instance… and hopefully not kill the system.
- A script then extracts some "possibly useful" data from the thread stack traces, into a usable CSV file.
- This data is then loading into the visualisation tool.

Information Extracted from the stack traces was:

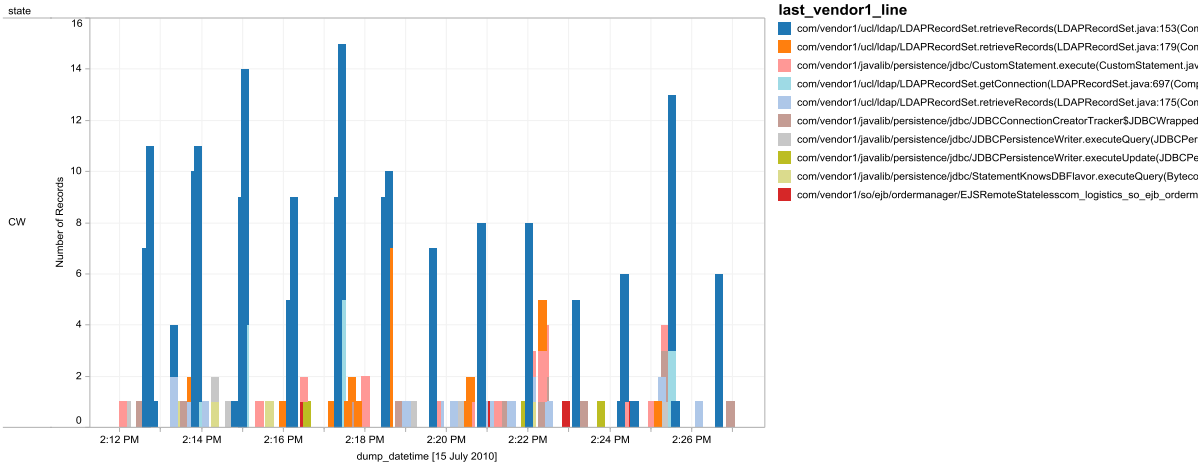
- Key data for each thread: eg thread ID, name and type, state (blocked, waiting, running)
- The "last line executed" for some higher-level class paths: eg com/ibm, com/oracle com/*vendor* etc. This was set up by browsing a few dumps manually first to figure out where in the package path the main code was.

The process of discovery:

1. Viewing a simple graph of the number of threads in each state over time shows up blips at the correct intervals: this is a good sign that we'll find something in the data.



2. Plotting instead the count of threads by "Last Vendor line executed" over time quickly shows what piece of code was doing during pauses.



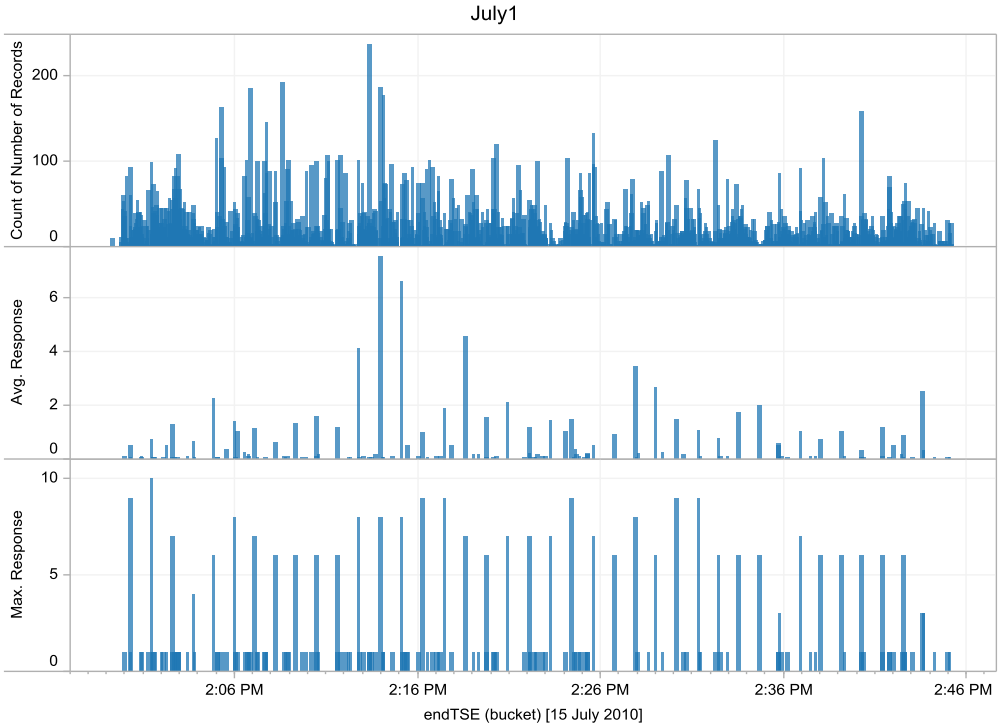
We can see that during the pauses, the threads are all running LDAP code. *Note that until this point, LDAP was discounted by both the client and vendor, with no access and information given. ("Don't worry about that - it is only used on login")*

Step 3: Further Drill-down

The immediate response in seeing the above graph is “it is the LDAP server”, and that is almost certainly going to be the case. But being paranoid it could *theoretically* be in the LDAP code on the app server itself (extremely unlikely) or in the network between app server and LDAP (most unlikely). I mention this because people hate to have fingers pointed at them, and giving options (even highly unlikely ones) is a good strategy to show that you are impartial and are not just “laying blame”.

LDAP request log files are available which should quickly verify this. These give per-second resolution stats only, which are completely useless for finding out about “good” responses (expected to be in milliseconds) but suitable to confirm if the 7-second pauses are within the LDAP request.

The LDAP log files were extracted and processed to give *response-time per call* and other information. Plotting response (and number of records) over time, we instantly see what we were looking for: (LDAP response times the same magnitude and timing as the end-user pauses)



Points of note:

- Changing the timestamp bucket size affects Average, but not Maximum. This is one of the few situations where I have used “maximum response” as a useful metric. In this case, it gives the duration of a pause during that time period.
- This is *proof* that the pauses are within the LDAP call itself.
- A very large number of calls are being made – alerting us to the fact that the LDAP information is probably not being cached in the App Server.

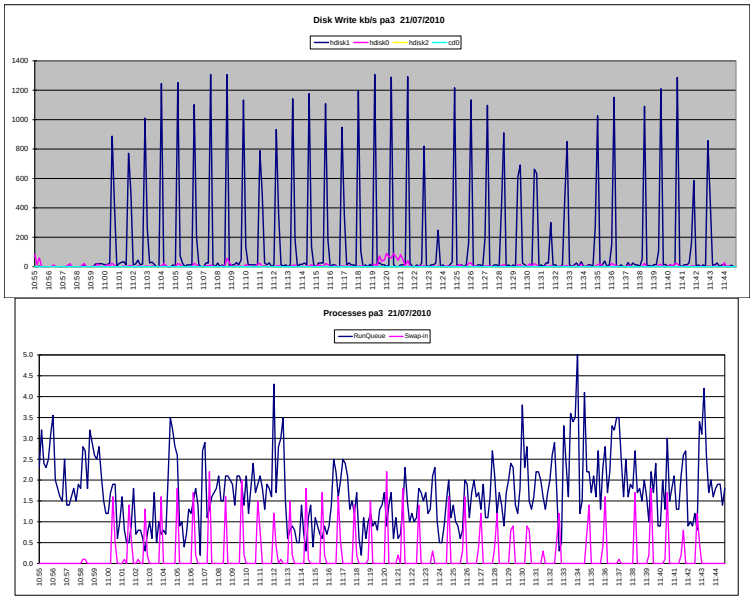
Step 4: Root Cause and Solution

About this time, the technical teams are now really starting to take notice and actually be involved in helping solve the problem. It is now time for the admin teams to look at details of the LDAP configuration and server behaviour.

General system stats from the LDAP AIX server were recorded using *nmon* in an attempt to find other unusual behaviour that might correlate to the slowness. The time resolution had to be in the seconds to ensure the issue is noticeable (data at 10 second intervals was available.) Analysis was just using the standard "out-of-the-box" nmon analyser spreadsheet.

What I could see from the results

- High bursts of disk write I/O to the application file system.
- Some process *swap-in* activity is seen (this is bad)
- About 60% of memory is used for file system cache
- The swapper process (responsible for writing pages to disk) is showing activity – and this has a 60-second interval!



I had no prior experience with AIX- performance at all at this point, so I was relying on the other teams to understand specifics of AIX and any other the systems we can't touch. But my non expert theories and questions to answer at this stage were:

- The root cause is somewhere within memory allocation and management.
- A theory: high *file system* write bursts is causing swapping of LDAP related *processes* and triggers the freezes.
- Based on this, I asked the SMEs to check application and O/S memory allocations and settings (…and jumped on to google myself in the interim)

Solutions:

- The first change made was to set **lru_file_repage = 0**. This means the page stealing algorithm will only use file system cache (if more than minperm% is in use, which is normal)

Impact of Change:

- Performance was dramatically better, but not completely solved – see below graph.

- nmon shows swapping reduced by about 75%, but was still present.
- *More work to be done*

Further Changes and Final Words:

- The LDAP server cache settings were finally checked and found to be at defaults. After allocating a couple of GBs of memory to LDAP, the problem completely disappeared.
- Cutting down *%maxperm* may also have solved the problem.
- Caching of LDAP data on the App Server was also finally turned on. (This might have even resolved the problem by itself.)

There are no graphs to show the final results: After the first improvements were made, the tech teams became more confident and we were told our services were no longer needed. Performance troubleshooting and tuning can be a thankless job!

In Summary

- The assignment shows the importance of having detailed data:
  - If you use metrics, understand their weaknesses.
  - Issues smaller than your sampling interval will not be seen or understood.
  - Visualise the RAW transactional data, or else risk missing something!
- An appropriate visualisation can be very convincing. The “tornado” graphs in this case were suitable to show to non-technical management. Even if they did not truly understand the details, it looked good!
- Statistical sampling can be valuable and done with low-tech methods (ie no need to install or enable complex tracing / profiling software)
- The tool you use to analyse and find issues is less important than the data itself.