Performance Analysis White Paper

VERSION 2018

Contents

Performance Analysis

When faced with a system that isn't performing as well as you would like, how do you go about tracking down what needs to be changed? Where do you start?

This document outlines the steps to take, and it includes a number of examples of performance analysis carried out by the JADE Benchmark Laboratory in addressing real-world performance issues.

Many of the concepts discussed in this document are generic, and apply equally to many other development and deployment platforms, while others address specific JADE constructs and configurations. This is not an exhaustive discussion of the subject, but it should get you started in the right direction.

There is a lot of information in this document. It may pay to come back in a few months' time and read it again. Other things are likely to jump out at you then.

For guidance in designing a JADE system so that it performs well, refer to the *JADE Performance Design Tips* white paper. For further information visit the JADE Web site at https://www.jadeworld.com/developer-center/resource-library/white-papers.

Note This white paper is relevant to JADE 7.0 and later. In earlier releases, a number of mechanisms operated differently, and it is likely that many may also change with future releases. In particular, JADE 7.0 and 7.1 use their own disk cache as the primary cache for database objects, while JADE 6.3 and earlier releases used Windows file-system cache. (For more details, see the *JADE Server Memory Allocation* white paper.) In addition, JADE 7.0 and 7.1 support much-better concurrency between multiple processes within a node. Use the 6.3 version of this document if your database is 6.3 or earlier.

Any description of non-JADE tools is thought to be accurate at the time of publication. Their operation may change at any time. Consult their documentation for up-to-date information.

For more details about performance analysis, see the following subsections.

What's New for JADE 7.0 and 7.1?

The persistent database now implements its own disk cache. This allows for more flexibility and performance optimization than was available with Windows file-system cache.

Much-greater concurrency is possible for processes within a node. The protocols for sharing common resources, including the JADE Object Manager (JOM) persistent object cache, have been improved to provide better performance and to support more processes per node.

There is a new RPC transport (**HPSM**), which allows for greater concurrency than the **JadeLocal** transport.

Overview

Most performance problems are caused by excessive consumption of finite resources. Performance improvement can be seen as a process of improving the management of scarce resources.

As a resource becomes saturated, a bottleneck occurs and additional requests for the resource are queued. There are two basic types of bottleneck: physical and logical.

Physical bottlenecks occur when physical resources are saturated; for example:

- All available CPUs are busy at the time an additional request for CPU time is made.
- All available memory space is being used.
- The database or logs disk is busy at the time an additional request is made.

Logical bottlenecks occur when programming logic restricts access to functionality; for example:

- Locking a JADE object to prevent overlapping updates.

Identifying your most-scarce resources will help you focus your optimizations where they can do the most good. As you optimize the use of one resource, another resource may become your most scarce. Sometimes there are direct trade-offs between resources. Using resources is not a problem; in fact it is necessary, but wasting them or balancing them poorly results in unnecessary performance problems.

Excessive consumption of CPU or IO can stem from inefficient coding, inappropriate initialization file settings, or inadequate hardware. Excessive contention for collection locks can be caused by poor data model design, lack of a good locking strategy, or poor operational procedures.

This is by no means a complete list, but we shall try to address the more-common issues that have been seen by JADE users in the field.

Essential Concepts

This section contains the following topics.

Where Do JADE Methods Execute?

JADE methods execute only in JADE nodes. A JADE node is the fundamental building block of JADE's distributed architecture. Each node contains the JADE Object Manager (JOM), the JADE Interpreter, various caches, and one or more JADE processes.

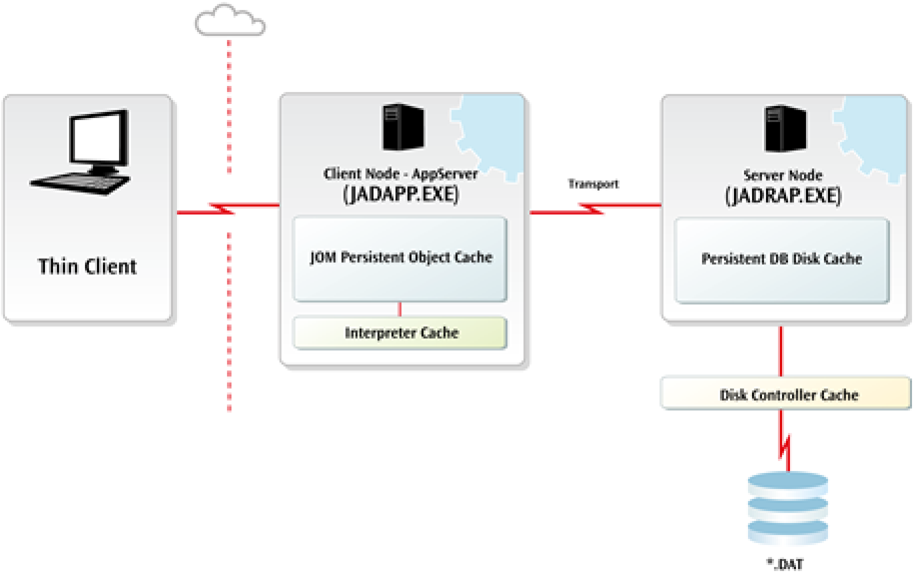
Note The JADE thin client is *not* a JADE node; JADE methods do not execute there, although a great deal of effort has been expended to make it look as though they do.

In most production systems, there is one database server node (**jadrap.exe**), one or more application server nodes (**jadapp.exe**), and one or more fat/standard client nodes (**jade.exe**) for background processing, Web services, or HTML forms.

When **jade.exe** is run in single user mode, there is one node only.

Common Caches

The following diagram shows the common JADE caches, described later in this section.



Interpreter Cache

Every user (JADE process) has an Interpreter method cache dedicated to it, assuming that multiple caches are being used. Within this cache, methods contend for space. If the user performs functions requiring more methods than can fit in the cache, the least-recently used methods are removed until there is room to load the new method.

JOM Persistent Object Cache

Every JADE node has a JADE Object Manager (JOM) persistent object cache. Every JADE process running in the node contends for this cache.

Persistent Database Disk Cache

Starting with JADE 7.0, every JADE database has a Persistent Database (PDB) disk cache in the database server. All JADE nodes and server applications contend for this disk cache. All objects that are read in from the database on disk are read through this cache. All persistent objects that are created or updated are written through disk cache. The PDB module uses direct IO to read and write database blocks.

Note The PDB disk cache is separate from the server node JOM persistent object cache. The disk cache is used only by the PDB itself. The server node JOM persistent object cache is shared by server applications and **serverExecution** methods, similar to the way that the JOM persistent object cache in any node is shared by all processes running in that node.

File-System Cache

JADE 7.0 and 7.1 use file-system cache for some things such as non-database files, but it is not as important as it was in JADE 6.3. Each instance of the Windows operating system maintains a file-system cache. All JADE databases and other programs running on the server machine contend for space in this cache.

In JADE 6.3, file-system cache was the primary cache for persistent database objects. From JADE 7.0, the primary cache is the PDB disk cache.

Disk Hardware Cache

Most disk subsystems have a built-in hardware cache. All programs and databases using files on that subsystem contend for space in this cache. If the disk subsystem is shared by multiple machines, then all programs on multiple machines may be contending for this cache. The different machines may be running different operating systems, but they still contend for the same cache.

JOM Transient Object Caches

There is also a JOM transient object cache in every JADE node. All processes in the node contend for it. To support **serverExecution** methods, there is a remote transient cache in the server node.

Tools

The most-useful tools for analyzing JADE performance are the JADE Monitor, JADE Profiler, and operating system tools such as Windows Task Manager and Performance Monitor. Examples of appropriate use of each are discussed in this document.

The JADE Monitor and JADE Profiler are the most useful for analyzing your JADE system's behavior, as they were designed specifically for JADE.

Less Is More

Generally speaking, you improve performance by reducing the amount of work done. Specifically, you can look into the following areas.

- Ensure that database queries access only the required data.
- Minimize the number of trips to the database server to fetch or lock objects. This includes setting JOM persistent object cache sizes in JADE nodes large enough, to avoid re-fetching the same objects. It also means having a good locking strategy.
- Minimize the number of trips to the disk subsystem. This usually means having enough RAM installed on the server machine, and ensuring that it is available for use as database disk cache.
- Minimize the amount of code that is executed; for example, eliminate unnecessary validations, and redundant and debugging code.
- Minimize the number of messages between thin clients and application servers. This requires care in coding and an awareness of the constructs that cause these messages to be sent.

Analyzing Performance

This section contains the following topics.

- What has Changed?
- Getting Started - Taking Measurements
- Starting the Analysis
- What is Your Biggest Problem?

What has Changed?

This is the first and most-important question you need to ask when starting to analyze a performance issue. This is the most-likely shortcut to finding the culprit. Whilst immediate changes can sometimes be easy to diagnose and resolve, a more-structured approach is likely to be required to resolve gradual performance changes.

Resolutions to performance issues are more likely to be expedient if performance is monitored regularly. At the very least, the following items should be recorded and analyzed on a regular basis.

- Business transaction response time
- Indicators from the system statistics screen in JADE Monitor
- Server resources indicated in the host performance screen in JADE Monitor

The majority of business transactions should be instrumented with something like **cnStartTransaction** and **cnEndTransaction** in JADECareStart (CardSchema). By using these simple constructs around a logical business transaction, there will be hard response time data to compare between releases or business periods.

By monitoring server performance, you can measure the changing use of the limited server resources.

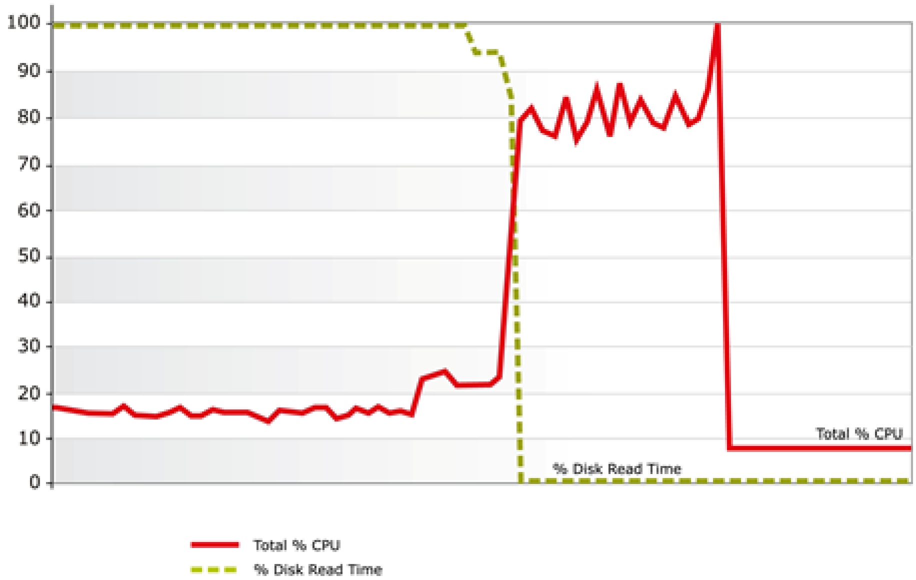
By recording and analyzing this data on a regular basis, you are likely to have the minimum essential data to avert any potential impacts to your business. For example, an increasing quantity of Get Objects from the system statistics screen may indicate growing working sets of data that may be need to be resolved with application logic changes.

Getting Started - Taking Measurements

The first step in analyzing a performance issue is to characterize it as primarily CPU-bound, IO-bound, or contention-bound. This requires a system-wide performance analysis, which measures CPU and IO rates. Your tool of choice could be Windows Performance Monitor, JADE Monitor, JADECare Systems Manager statistics reports, or a similar utility.

Whatever tool you decide upon, use it regularly so that you are familiar with it and the normal performance profile of your system.

Example Sample Period



The above diagram is a graphical representation (similar to the Windows Performance Monitor) of an example benchmark run that transitions through phases of being IO-bound, CPU-bound, and finally contention-bound.

The solid line is **Total % CPU**, with full scale being 100 percent.

The dashed line is **% Disk Read Time** on the database drive, with full scale being 100 percent, meaning that the disk is busy 100 percent of the time servicing read requests to that drive.

See "Appendix", later in this document, for a number of other common counters that may be helpful for performance analysis.

Starting the Analysis

When you have some measurements, you can begin to analyze them.

The main question is: *"What was your code waiting for?"*.

If the **Total % CPU** line is high, it means someone was waiting for the CPU to complete the required functions. If an IO line is high, someone is waiting for IOs to complete.

For more details, see the following subsections.

Analysis of the Sample Period

Initially, the system is IO-bound, with the reads to the database drive off the scale over 100 percent. CPU is low, around 17 percent utilization. During this phase, the system is IO-bound. The system has hit a physical bottleneck in the disk subsystem.

As the database disk cache warms up, the disk reads gradually drop off. As a result, the transaction rate increases, as indicated by the **Total % CPU** line. The system has transitioned to a phase where it is essentially CPU-bound. Again, a physical bottleneck has been encountered.

Towards the end of the example period, the **% Disk Read Time** stays close to zero (**0**) percent, but the CPU peaks slightly then drops to its lowest level of 8 or 9 percent. The system has encountered a logical bottleneck.

Something has changed, and the users are now being held up; they are waiting for something. If they were not waiting, the **Total % CPU** line would be high, as it was before. They could be waiting for application locks, pagefile activity, context switching, or a number of other things.

A contention-bound problem can be harder to diagnose, but a good understanding of the tools and counters can lead you to a resolution.

General Hints

If the **Total % CPU** line is at 100 percent, for example, it means that someone was waiting 100 percent of the time for the CPU to finish what it was doing. In the case of a multi-processor machine like the two-processor machine shown in the previous image, it means there were always two or more processes waiting for the CPU to do its thing. If it is a 16-way machine at 100 percent, then 16 or more processes are always waiting for CPU processing to finish.

If the **% Disk Read Time** is at 100 percent, it means someone was waiting for read IO 100 percent of the time. If the **% Disk Read Time** is 266 percent (a queue depth of 2.66), on average 2.66 processes were waiting for IOs at the time. If the **% Disk Read Time** is at 12 percent, it means someone was waiting for read IO 12 percent of the time.

Note The database server normally does asynchronous reads, so the one Windows process **jadrap.exe** can create a read queue all by itself, if multiple JADE applications are running. If only one JADE application is running, the read queue depth will not be larger than **1**.

If processing is waiting more on CPU than on other things, then it is *CPU-bound*. If it is waiting more on IO than on other things, then it is *IO-bound*.

If there is a mixture of tasks running, it is quite possible that some are CPU-bound and some are IO-bound. In this case, you could use the Windows Task Manager to identify which processes are using the most CPU time.

If there are a number of processes doing reads (for example, a number of thin clients processing transactions) and read IO on the database drive is 46 percent, then 46 percent of the time one or more of those thin clients is waiting for a server thread, which is waiting for the read IO to complete.

If there is not much CPU time being logged and not much IO time either, then you are probably waiting on locks or some other logical bottleneck.

What is Your Biggest Problem?

You may be in the midst of a crisis, in which case you will know what the biggest problem is. If you don't have a particular fire to fight at the moment or maybe the system is just slow overall, just look at the measurements and choose your biggest bottleneck.

Either way, address one problem at a time. This is an iterative process. When your biggest problem has been identified and rectified, then move on to your next-biggest problem, which by then of course will have become your biggest problem! Note for later any anomalous behavior you might see, but focus on your biggest problem first.

Take your biggest problem, and then proceed to the appropriate next step.

The Next Step - Excessive Contention

There will always be some contention, be it for shared physical resources such as CPU or disk, or logical constraints such as locking or queue limits. For example, housekeeping operations such as defragmentation or file archiving may be taking place on the server machine. These consume CPU and disk time resources, and as a result there are fewer of these resources available to your JADE system. If these operations take place at a time when the JADE system requires a lot of the resources (peak times, for example), they can negatively impact performance.

If your JADE system uses a shared Storage Area Network (SAN) and other machines use the SAN as well, any file operations performed by those other machines can potentially slow down the JADE system. For example, a file maintenance process running on another machine, even running under a different operating system, can slow down your database backups or online functions if they share the same SAN.

If the network is shared, someone else can slow down your network accesses by using the network for something else. For example, transferring a number of large files over the network can slow down your thin client transactions.

Locks are a fact of life in any multi-processing environment. Locks are used to prevent simultaneous access when that access would not be valid. As a side effect, they can create logical bottlenecks. For example, if objects were not exclusively locked while in the process of being updated, one updating user could overwrite another updating user's changes. By locking the object during the update, anyone else who wants to update the object has to wait until the first update is finished. In a well-designed system, the wait is usually very short.

One area of potential contention in JADE systems is lock time on collections. If an updater adds an object to a collection, the collection is exclusively locked until the transaction is committed. While the collection is exclusively locked, no other updater or inquirer can lock the collection. The longer an exclusive lock is held, the greater the risk of contention because of queued locks.

There are strategies to deal with this contention, as follows.

- Keep collection scope as local as possible. For example, when storing customer transactions, use a collection on the customer object rather than a global collection keyed by customer. This results in a separate collection instance per customer, which greatly reduces the likelihood of contention.
- Keep transactions short.
- Go into transaction state as late as possible.
- Populate collections late in the transaction.
- Use an iterator or a **foreach** instruction with the **discreteLock** option, to reduce lock time on the collection.
- If a collection is frequently accessed and updated, you may be able to avoid contention by making a transient clone of the collection.
- Delay updates to collections by offloading them to another process. Note that this can introduce recovery and consistency issues, and therefore should be used only with great caution.

For more details, see the following subsections.

Using JADE Monitor for Lock Contention Issues

If you suspect you have a lock contention problem, use JADE Monitor to record locks and queued locks, under **Locks** in the Navigator. Refresh manually (Ctrl+F5) while reproducing the problem situation, or activate the timer. Log the results to a file. Enabling system statistics may also be useful.

When you find evidence of the locking issue, you will need to know what caused it. Enable logging of the **Users** sheet, especially if applications are starting and stopping. JADE Monitor shows who is holding the lock or queued lock in the **User** column (as shown in the image in "Diagnosing the Problem" under "Example 1 - A Long Time in Transaction State", later in this document). This contains the instance id of the **Process** object for the user, which is convenient for cross-reference with the **Users** sheet.

Background applications should log their start and stop times in an application log file, so you can know when they were processing.

To diagnose an issue with a large number of queued locks of very short duration, you could use Lock Contentions or Lock Chronology. To see who is locking objects that are highly contended, use Summary by Classes or Summary by Oid. Use these for short periods only. They can impact system performance while in the monitoring state, and can generate very large work files.

Example 1 - A Long Time in Transaction State

Background

Some batch work is run daily, consisting of four reports that are run concurrently. Normally, all of these reports finish in 43 minutes and use 100 percent CPU during that time. Three of the reports are inquiry-only, and the fourth does a lot of updating.

The Issue

After a small change to one of the reports, the batch work now takes 57 minutes and CPU consumption dropped to just over 50 percent CPU for the first 25 minutes. IO rates never were very high before, and did not change appreciably.

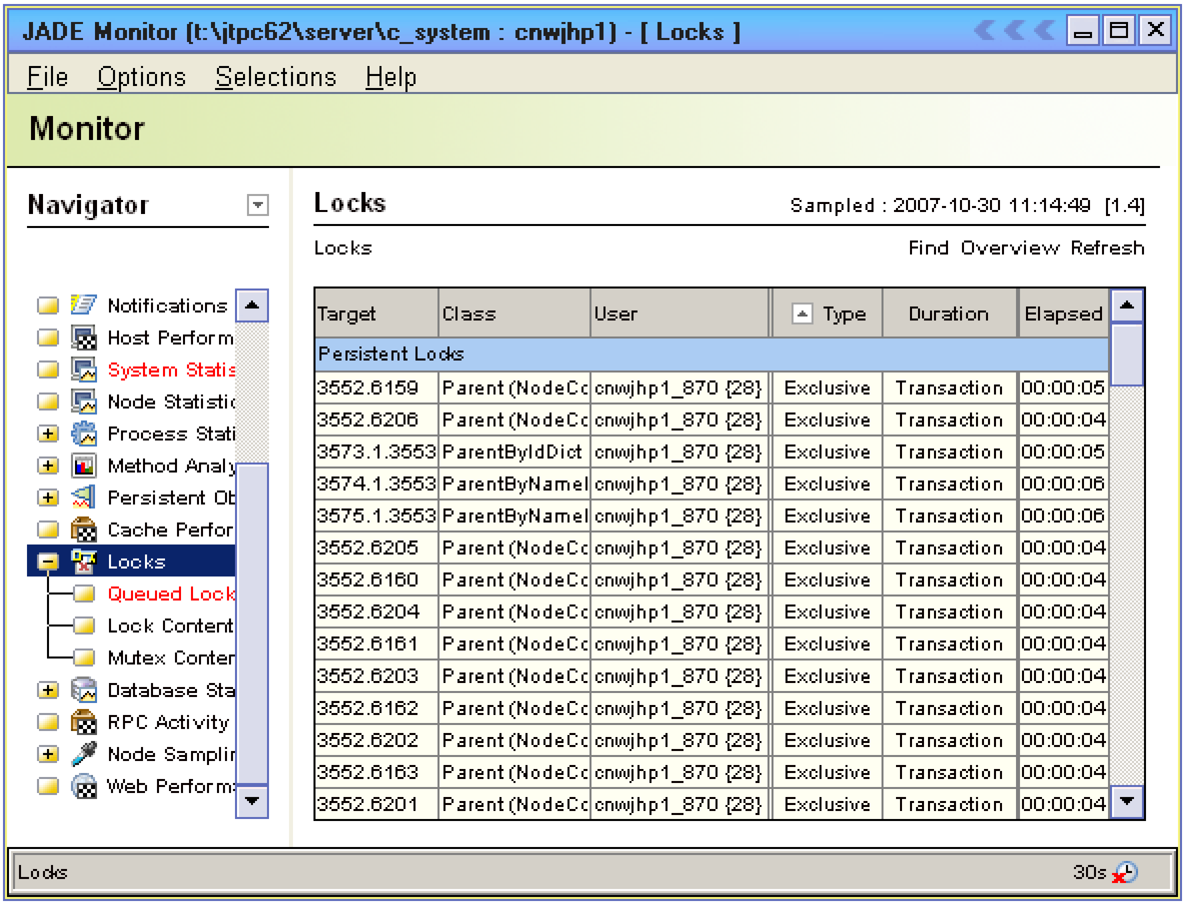
It looks like a logical bottleneck has been introduced.

Diagnosing the Problem

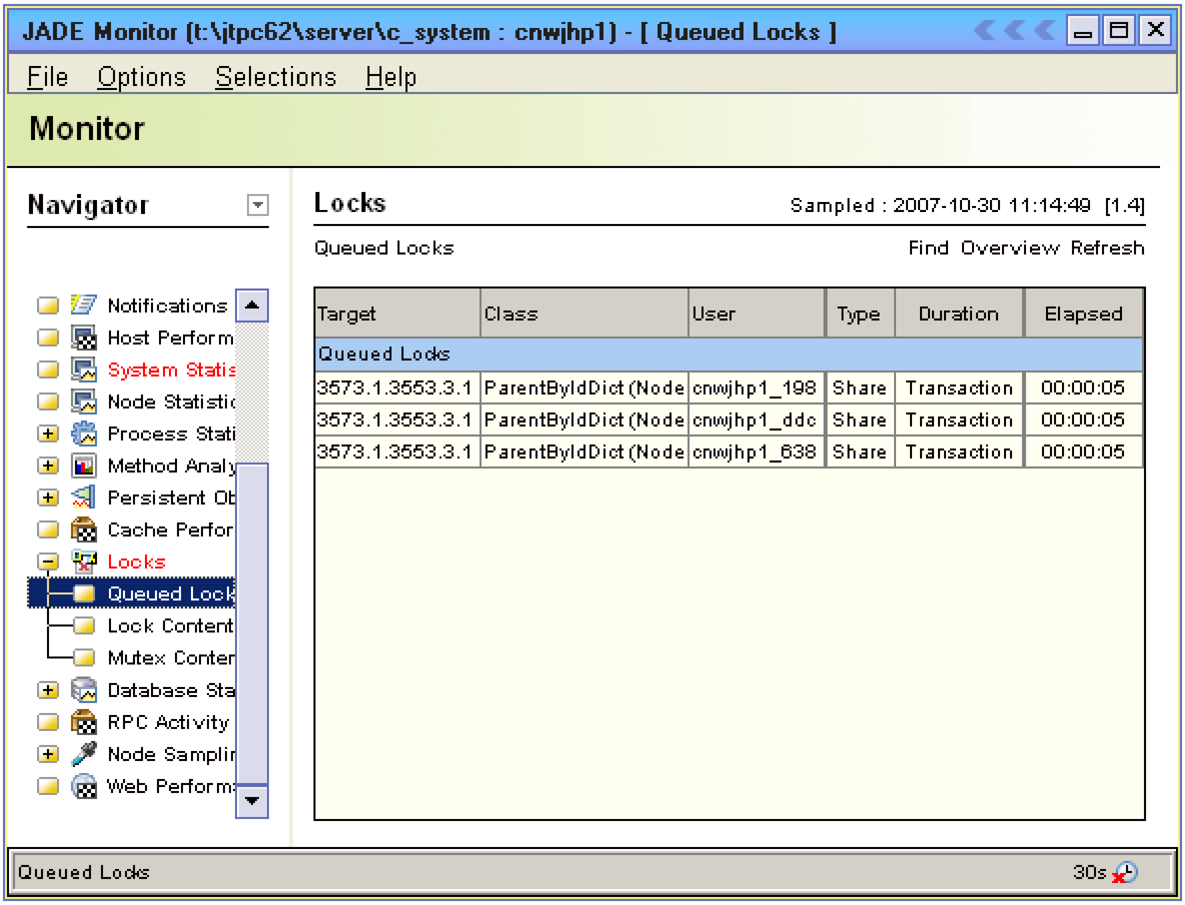
To diagnose this further, we would normally ask "What has changed?", and frequently the answer becomes obvious. As this looks like a logical bottleneck, we turn to JADE Monitor for confirmation.

When the batch work was investigated in the past, no locking issues were observed. There were a lot of exclusive locks and they were held for relatively long durations, but there were no queued locks.

After the change to the application, the Monitor shows there were about the same number of locks and they were held for shorter periods. They were all held by the updating report, as before.



However, there are also now some queued locks.



From this, we can see that one application has three collections exclusively locked and the other three applications are queued up waiting for a share lock on one of those collections.

Looking at the application's code, we find that the three inquiry reports' primary access is an iterator over **myRoot.allParentsById**, which is a collection of type **ParentByIdDict**. The updating report always *did* modify a large number of objects, but no collections were changed.

The small application change had been to add a parent object in some rare circumstances, which resulted in adding the object into the three collections, exclusively locking them. Commits were already being done periodically, based on the number of objects updated, and the locks were being held a lot longer than they needed to be.

The Solution

The application was changed to do a commit after creating each parent object, in addition to commits being done based on updated object count. This resulted in the collections being exclusively locked for a very short period.

After applying the fix, the report set was finishing in about the same time it always had, and returned the process to being CPU-bound with all CPUs busy during the report period.

An alternative solution would be to use update locks. With update locks, the collection would not be exclusively locked until the updating report commits. Up until that time, the inquiry reports can still access the collection even after it was updated. This can affect your locking strategy and exception handling. For more details, see Chapter 6 of the *JADE Developer's Reference*.

Conclusion

If frequently accessed collections are left exclusively locked for long periods, inquiry users can be left waiting for long periods. Transactions can be altered to reduce the length of time locks are held. Using update locks may improve concurrency by delaying the lock upgrade to exclusive.

Example 2 - Updaters Locking Out Updaters

Background

There are a number of updating reports in this database, which online users can run whenever they like.

The Issue

On quiet days, the average report takes 3 minutes, but on busy days, the run times fluctuate wildly; sometimes 3 minutes, sometimes 12 minutes. The variability of these reports has been in place since the original go-live of the system.

Diagnosing the Problem

Once again, the observations provide the following key indicators.

- Variable response times
- Has been the case since implementation

By comparing the JADECare Start instrumentation times, it is noticeable that response times are high when two or three updating reports are being run simultaneously.

By reviewing the CPU and IO load, it is noticeable that sometimes they are both quite high and sometimes they are much lower. It seems likely that an application locking contention issue has been encountered.

Do not rely on theory and application knowledge to predict where the issue is. Get objective raw data instead. The first thing to do is to find out, via the JADE Monitor, where the contention is – probably queued locks on a collection or two.

The situation is reproduced, with two or three updating reports being run simultaneously. JADE Monitor is used to monitor locks and queued locks. It can soon be seen that one of the reports is holding locks for which the others are waiting. Effectively, there is only one report actually executing, even though multiple reports are running.

The Solution

Resolving this type of issue can be a challenging process, but there are several ways of approaching it.

- Delay updates to the contended collections, if possible, by setting the inverses late in the transaction.
- Modify key values late in the transaction. Read all objects you need to read before starting the updates, especially before updating collections.
- Modify the reports to avoid updating the contended collections. Have a clean-up report that runs after-hours to make the changes to the collections.
- If the contentions cannot otherwise be resolved, change the scheduler to run only a single instance of these contending reports at a time.
- Re-factor the database, turning one global collection into a lot of smaller exclusive collections on relevant parent objects.

Clearly, not all of these techniques are going to be practical for every situation, but use them if you can.

Another technique that can be used is called *cache warming.* This technique is different from most, in that you are intentionally using more CPU and IO resource in order to minimize contention points.

Sometimes a business process will require a lot of processing *after* modifying a collection. For example, you may go through a number of objects in a collection, check a number of other objects for each one, and eventually modify a collection. Within the single transaction, you may need to do this multiple times, which locks everyone else out. However, if the reads happen faster, the collection will stay locked for a shorter period. The required reads will happen faster if the objects involved are already in the various caches.

By adding a cache-warming method to an existing report or online transaction prior to starting the updating loop, you are able to pre-read the required objects. In the cache-warming method, you would go through the usual collections, fetch the objects, and so on, but skip all of the updates.

If an object is to be added, the collections to which the object will be added can be warmed. This can be done with **Collection::includes** or **Dictionary::includesKey**. Similarly, if objects are to be deleted, the collections can be warmed with **Collection::includes**.

Even if you skip over some of the accesses or get the keys wrong when using **Dictionary::includesKey**, warming most of the cache will still shorten transaction time.

In one real-world case, there were 95 reports that needed to run within a specific time. They all updated the same collection, and normally just locked each other out the entire time they were in transaction state. Cache warming was used to reduce the length of time spent in transaction state, from 510 seconds to 135 seconds. The existing report logic was not changed.

Note Cache warming is primarily useful for updater versus updater contention, when other techniques are not practical. Cache warming normally increases overall system load. For single-threaded processes or where there is little contention, cache warming will only slow down the application.

Cache warming is normally used when it is worth a bit more system load to reduce lock durations.

Conclusion

Updater versus updater contention is typically more difficult to resolve, but if you understand the nature of the contention, there are a number of potential solutions.

The Next Step - Excessive CPU Consumption

If CPU consumption is your biggest resource utilization issue, you need to establish which processes are consuming it. The quickest way to establish this is normally with Windows Task Manager. Go to the **Processes** tab, as shown in the following image, and then click on the **CPU** heading to sort by that column. Click again to toggle between the ascending and descending sort sequence.

Note that the Windows processes are identified only by the executable name; for example, **jadrap.exe**. The important thing to identify is *which* Windows process identifiers (PIDs) are utilizing the CPU resource.

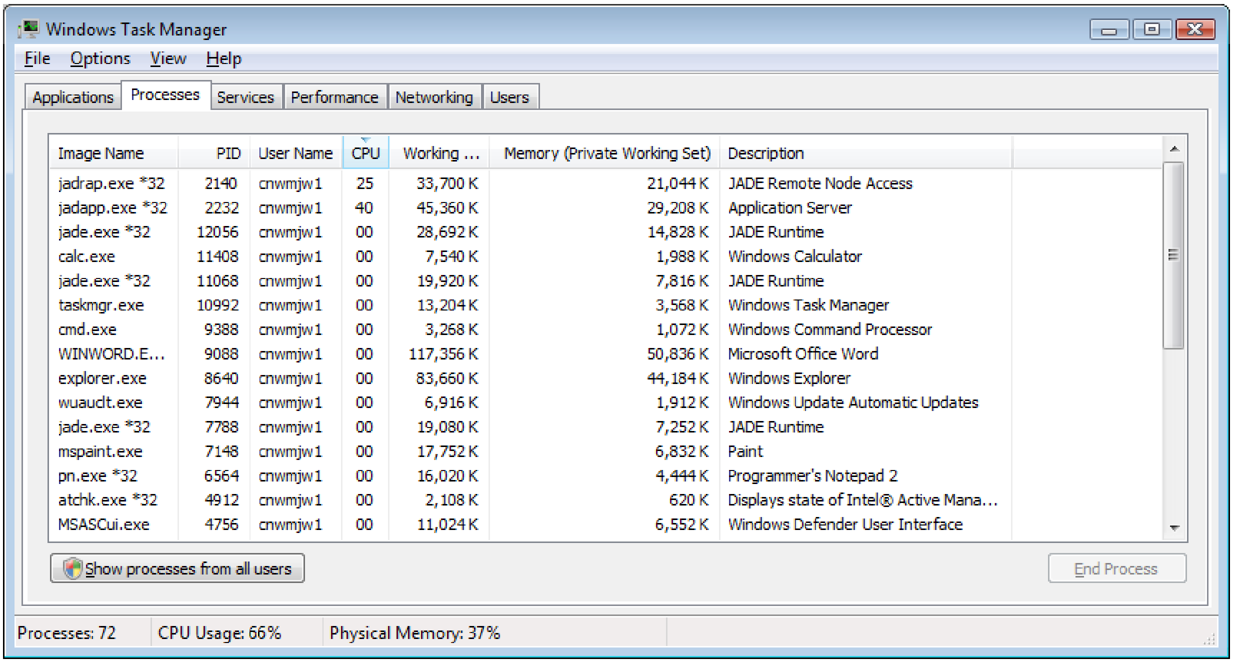


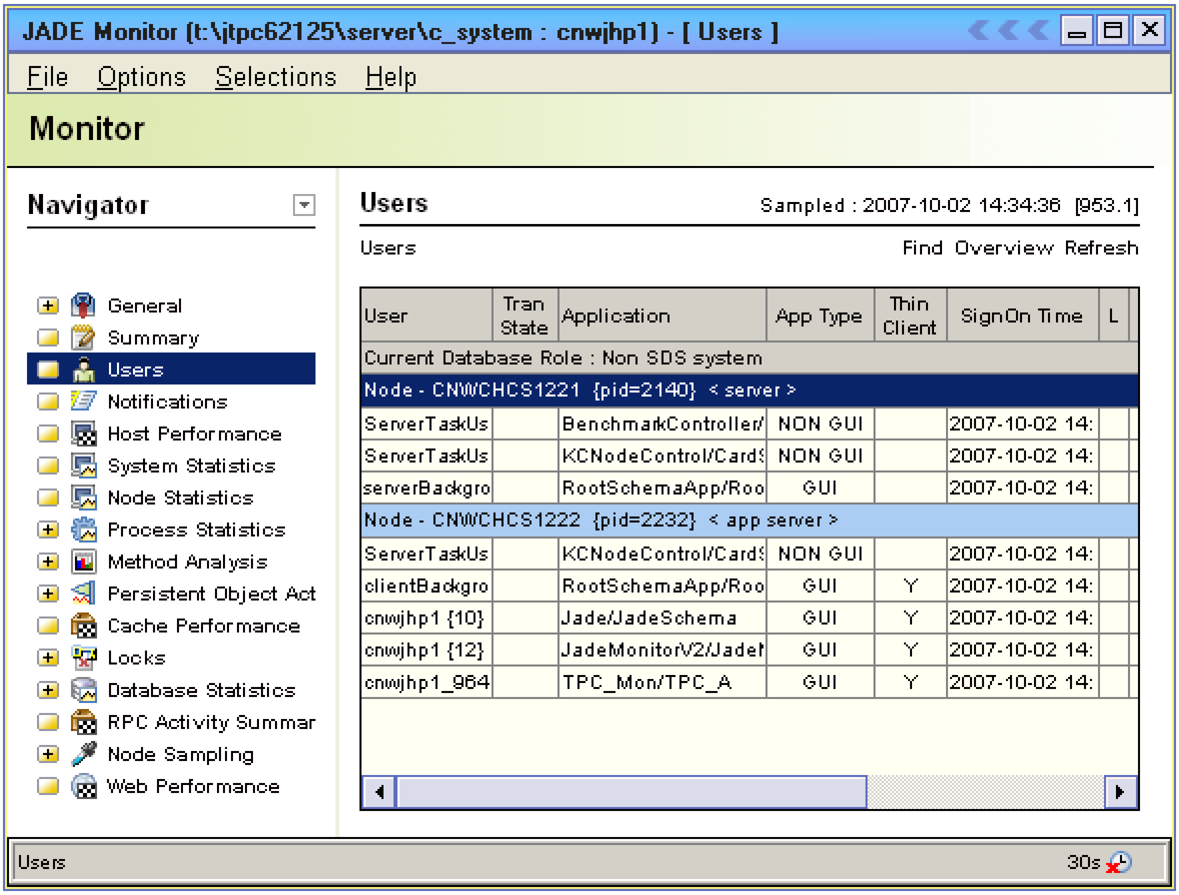
Image Name	PID	User Name	CPU	Working ...	Memory (Private Working Set)	Description
jadrap.exe *32	2140	cnwmjw1	25	33,700 K	21,044 K	JADE Remote Node Access
jadapp.exe *32	2232	cnwmjw1	40	45,360 K	29,208 K	Application Server
jade.exe *32	12056	cnwmjw1	00	28,692 K	14,828 K	JADE Runtime
calc.exe	11408	cnwmjw1	00	7,540 K	1,988 K	Windows Calculator
jade.exe *32	11068	cnwmjw1	00	19,920 K	7,816 K	JADE Runtime
taskmgr.exe	10992	cnwmjw1	00	13,204 K	3,568 K	Windows Task Manager
cmd.exe	9388	cnwmjw1	00	3,268 K	1,072 K	Windows Command Processor
WINWORD.E...	9088	cnwmjw1	00	117,356 K	50,836 K	Microsoft Office Word
explorer.exe	8640	cnwmjw1	00	83,660 K	44,184 K	Windows Explorer
wuauclt.exe	7944	cnwmjw1	00	6,916 K	1,912 K	Windows Update Automatic Updates
jade.exe *32	7788	cnwmjw1	00	19,080 K	7,252 K	JADE Runtime
mspaint.exe	7148	cnwmjw1	00	17,752 K	6,832 K	Paint
pn.exe *32	6564	cnwmjw1	00	16,020 K	4,444 K	Programmer's Notepad 2
atchk.exe *32	4912	cnwmjw1	00	2,108 K	620 K	Displays state of Intel® Active Mana...
MSASCui.exe	4756	cnwmjw1	00	11,024 K	6,552 K	Windows Defender User Interface

In this example, the **jadapp.exe** is using 40 percent CPU, and the **jadrap.exe** is using 25 percent. There are also multiple **jade.exe** programs running on the server. They are using zero (**0**) percent CPU, so they are not the problem.

Any JADE node that initiates database activity will cause the database server node to consume CPU as well. In this example, the application server is busy doing something involving the database.

When you have established the PID of the process or processes utilizing the resource, open the **Users** sheet in JADE Monitor. The PID displayed on the **Users** sheet corresponds to the PID displayed in the Windows Task Manager.

In the following image, the Windows PID of 2140 that was utilizing 25 percent of available CPU corresponds with the server, as shown in the highlighted row.



These are some of the more-common possibilities that you may encounter as you try to track down high CPU users.

- A **jadrap.exe** is showing high CPU consumption. In that case, look for a **serverExecution** method or **ServerApplication**. Server applications can be seen in the JADE Monitor (there are two of them in the above image). **serverExecution** methods may require application knowledge to determine which client initiated them.
- A **jade.exe** is showing the most CPU consumption. If this is on the server machine, the **jade.exe** is likely a background processor of some sort, perhaps doing batch updates or processing files as they arrive. At any rate, you can look up the PID in the JADE Monitor, to see what JADE application it is running. If there is significant **jadrap.exe** CPU as well, it is doing database activity. If not, it may be crunching numbers, or reading and writing files. On the other hand, it may be looping.
- The **jadapp.exe** is showing the most CPU consumption (as in the case shown in the above image). In this case, look for a thin client connected to that application server. If there is significant **jadrap.exe** CPU as well, the thin client process is doing database activity. If not, it may be crunching numbers, reading and writing files, or it may be looping.

  In the above image, the **TPC_Mon** thin client application seems like a good candidate.

For more details, see the following subsections.

Example 1 - Using a Non-Optimized *foreach* Instruction

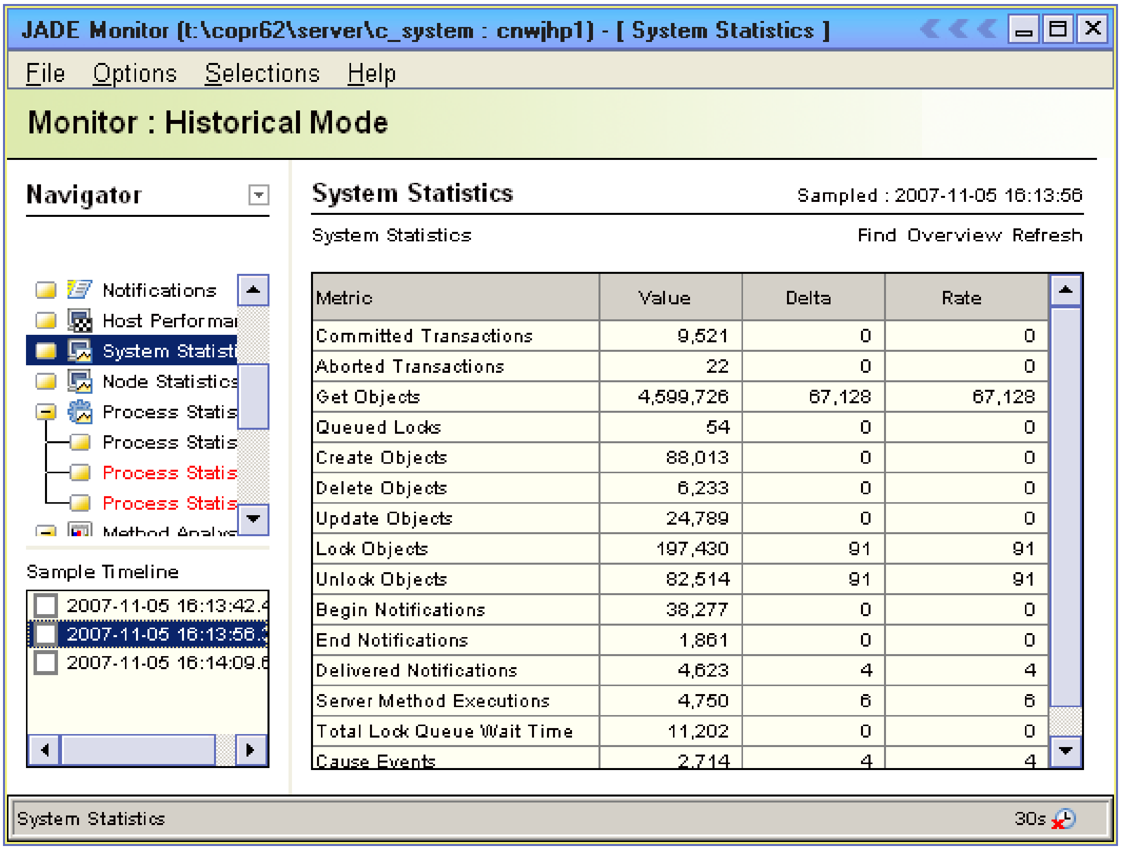
Background

A database suddenly started using much more CPU than it had before, eight days after going into live production. No performance issues had been identified during development and testing.

Diagnosing the Problem

Using Windows Task Manager, it was seen that a **jadrap.exe** was consuming about 20 percent CPU and a **jade.exe** with PID 2400 was consuming about 17 percent.

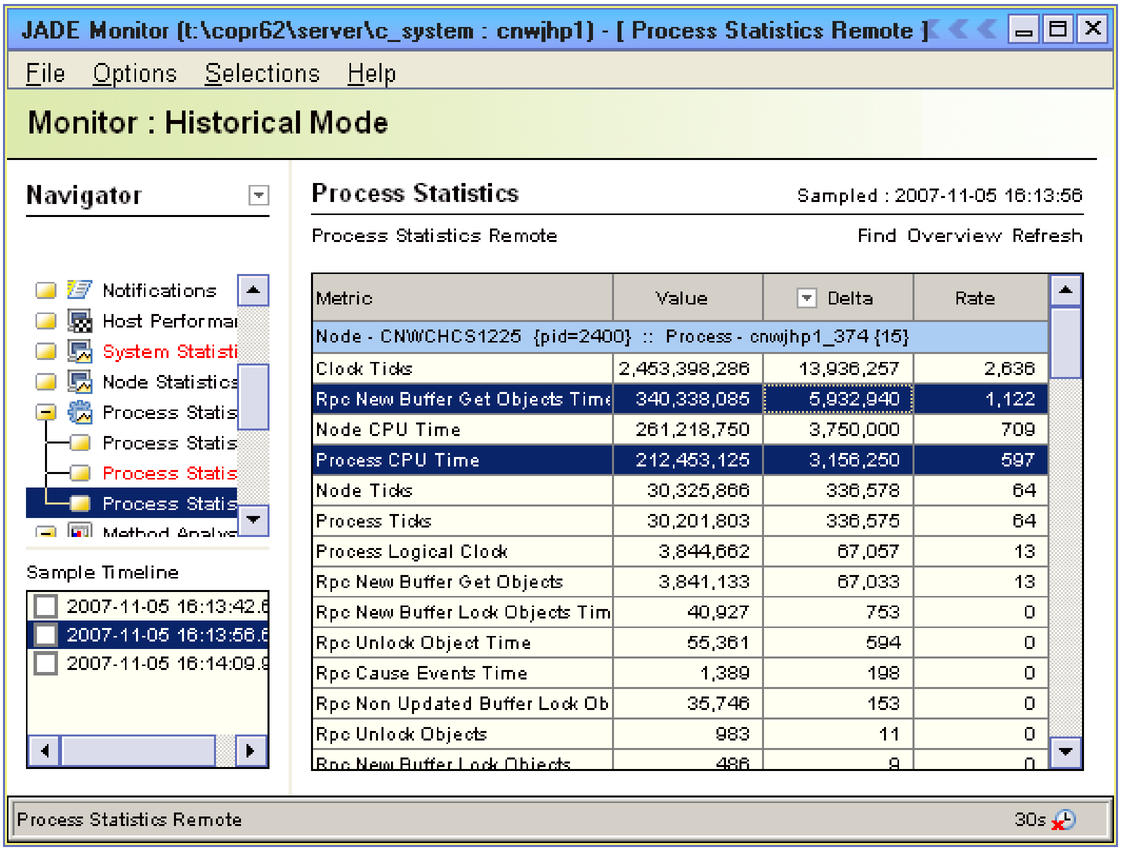
The application running in the **jade.exe** was identified via its PID as a background scheduling application. From the JADE Monitor **System Statistics** sheet, it can be seen that there are a high number of Get Objects, more than 67,000 per second, but nothing else immediately stands out.



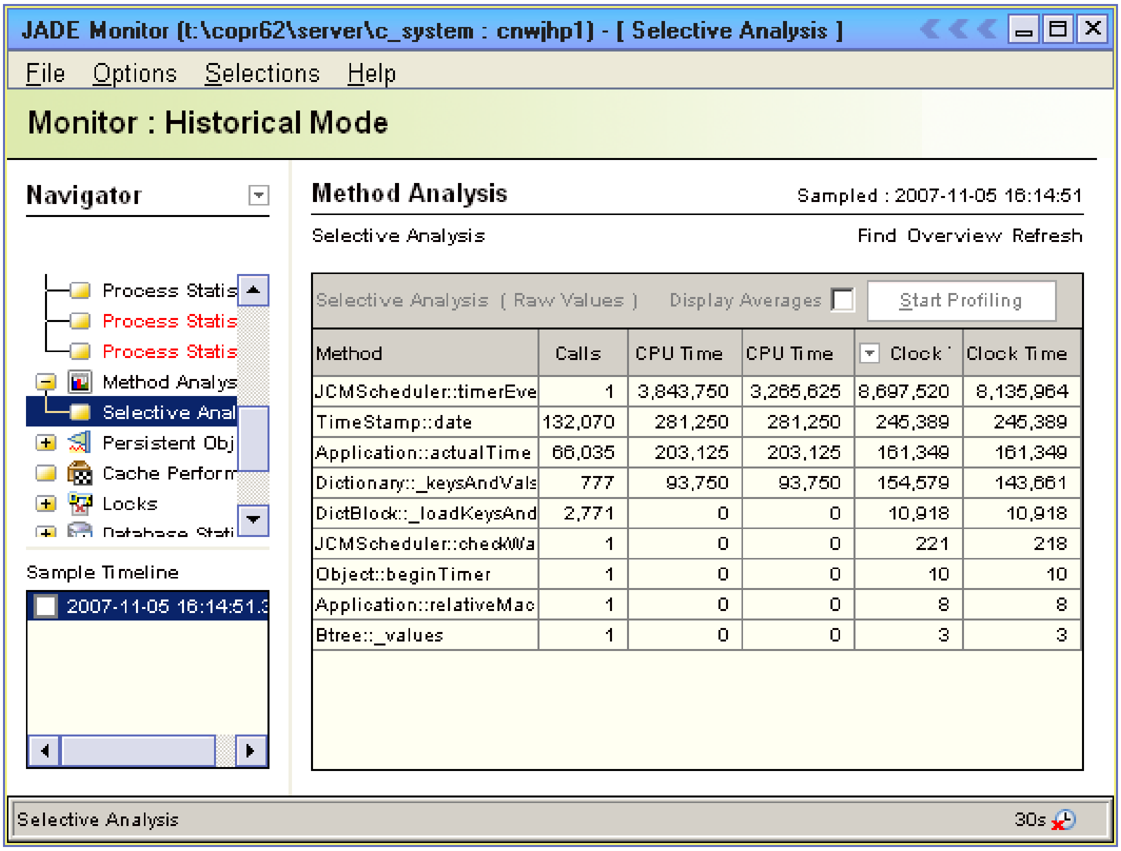
Using the JADE Monitor **Remote Request Statistics** sheet, having identified the node by its PID, it can be seen that this JADE application consumed over three seconds of CPU during the sample period (the **Delta** column of Process CPU Time), and spent nearly six seconds waiting for objects to be retrieved from the database server (the **Delta** column of Rpc New Buffer Get Objects Time).

Note The sample timestamps are the same in the System Statistics and Process Statistics screen examples, because **Refresh All** (Ctrl+F5) was used to get a coherent sample.

The screen example in the following image shows the JADE Monitor in historical mode. In historical mode, you can load a log file of previously captured data into the JADE Monitor. The source of the data can be from your own JADE system or from another JADE system. For details about logging a sampled activity to file for subsequent loading as historical data in another JADE Monitor session, see "Logging a Sample to a File", in the *JADE Monitor User Guide*.



The application was then profiled using the JADE Monitor **Method Analysis** sheet, shown in the following image.



This analysis shows that all of the time is being taken in the **JCMScheduler::timerEvent** method.

The number of calls to external methods is shown in the profiler; in this case 132,070 calls to **TimeStamp::date** and 66,035 to **Application::actualTime**. Looking at the **JCMScheduler::timerEvent** method source with the number of calls in mind, the problem is narrowed down to a single **foreach** instruction, similar to the following.

```
foreach obj in coll where obj.startTime.date >= app.actualTime.date-5 do
    ...
endforeach;
```

It had been thought that this was the optimized version of the **foreach** instruction (as **obj.startTime** is the only key of **coll**), but it is not. JADE has to fetch each object into the node's cache to run the **TimeStamp::date** method on it. Hence all 66,035 objects were being brought into the node each time this **foreach** instruction was executed.

The Solution

Change the **foreach** instruction to be optimized, as follows.

```
foreach obj in coll where obj.startTime >= selectedTimeToStart do
    ...
endforeach;
```

When **obj.startTime** and **selectedTimeToStart** are both **TimeStamps**, the **foreach** instruction is optimized, meaning that JADE can use a keyed lookup to find the starting point in the collection.

Conclusion

Previously, all of the objects required for this function were found in the node's cache, so the un-optimized **foreach** instruction had not been spotted during testing. It was only noticed when more data was being read than would fit in the node's cache.

Tip Test with production data volumes, if at all possible.

The JADE Profiler shows the number of calls to a method. Checking these for reasonableness may lead you to the cause of a performance problem.

Example 2 - Method Cache Too Small

Background

A database suddenly started using much more CPU than it had before, the day after a new release has gone live in production. No performance issues had been identified during development and testing.

Diagnosing the Problem

Interpreter method cache is a frequently overlooked initialization file setting, and setting it too small usually results in excessive CPU consumption. If the "normal" working set of business transactions changes (due to code or business changes), the method cache sizes should be reviewed.

The background JADE Profiler was used on a presentation client performing a representative workload. The following is an extract from the latter parts of the resulting log. The log covers 122 user transactions; basically button or menu clicks. The example is from a production system (although method names have been changed), and using the production initialization file settings.

```
Methods ordered by total method load time:
     Time        Count      Average    Method
      171          210         0.81    MyTransient::change
      170          199         0.85    SysEvent::checkMaintain
      106          148         0.72    NavGroup_Pages::isItVisible
...
Cache limit: 524288
Maximum cache size: 524288
No cache overruns
```

We can see that interpreter method cache was set to 512K bytes (cache limit: 524288). This is a very small setting for a production database.

There were no cache overruns, which means that the interpreter did not have to exceed the specified size to execute a single method *and* all of the methods it called. Note that this does *not* mean the cache was large enough to be efficient. In fact it was swapping hundreds of methods in and out of its cache every second.

We can see from the upper part of this example log that these three methods were reloaded into interpreter method cache about 200 times each – almost twice per user transaction. There were a lot more methods called a similar number of times in this test. This reloading of methods normally manifests as CPU consumption, as various caches are searched until the method is found and can be loaded.

Normally, when the method cache is set too small, the method can be found in the node's object cache or in the operating system's file-system cache. The observed increase in CPU consumption is a result of cache searching, communication between processes, and actually loading the method (which builds the execution tree).

The Solution

In the above test, larger method cache sizes were tried. In the end, 8M bytes was found to be the optimal size. At this size, there were very few method reloads. The reloads that remained were a few logon or other infrequently used methods, and they were reloaded once or twice only.

When set to 8M bytes, the average transaction response time decreased by a third. This means that previously, when the cache size was set to 512K bytes, a third of the elapsed time for each transaction was spent unnecessarily reloading these methods.

Conclusion

For most production systems, multiple interpreter cache settings in the range 2M bytes through 8M bytes are usually suitable. A lot of factors influence the decision, though, especially the size of the methods and the number of methods called for a user's working day. It is best to use the background profiler and check for the size used and the number of reloads. This should be reviewed before every production release and after changes in business practices.

Different nodes or processes can and should use different settings. In the case of a high number of server threads, for example, it may be a better option to use a single or smaller interpreter cache for the server node. If memory is not a constraint, though, multiple caches will perform better.

While setting the method cache too small can result in excessive CPU consumption and poor response time, setting it too large can waste memory. The objective when tuning the interpreter cache is to hold the most-commonly used methods in cache. For overall system performance, memory is often better used in other caches rather than holding infrequently used methods in the interpreter method cache.

The Next Step - Excessive IO

This section contains examples of excessive IO, as follows.

- Example 1 - Not Enough Database Disk Cache
- Example 2 - A Large Number of Reads Flushing Out Cache

Example 1 - Not Enough Database Disk Cache

Background

In preparation for the go-live of a new production database, the data conversion was being looked at, to try to minimize the time taken.

The Problem

For much of the data, the load went quickly. However, there were some specific classes that loaded slowly. The worst one was a single class with over 100 million instances, and there were five collections over this class.

Diagnosing the Problem

Windows Performance Monitor was used to track CPU and IO rates. During the first part of the load of this class, CPU was at 100 percent, disk reads were nil, and disk writes were heavy, but the file system was coping. The only resource pushed to capacity was CPU, which meant that all possible optimizations had been done. CPU optimizations had already been addressed.

As the load of this class progressed, there was an increasing amount of read IO on the database drive. The records-per-second rate slowed gradually from 2,300 to 165. This was a write-only situation; why all of the reads?

Thinking through the process, it became clear that inserts into the collections was the issue.

When a new object is created, a reference is set that adds it into the five collections. JADE then goes to locate the specific collection blocks that must be modified. If they are not in the node's persistent object cache, they are requested from the database server. If they are not in the PDB disk cache, the database server issues a read command to the disk hardware.

Looking at the magnitude of the data involved, the problem was clear. The class itself was 22G bytes. The five collections totaled another 27G bytes. Total RAM on the server was 7G bytes. The amount of PDB disk cache required was probably about 30G bytes – all of the collections plus a little for the class itself. The server didn't have enough RAM to hold the collections for a single class, and the performance suffered.

The Solution

As a test, the test server was upgraded from 7G bytes to 16G bytes of RAM. The total load time for the one class went from 133 hours down to 24 hours, just from the one hardware change.

More RAM was ordered for the production server. Not only did it reduce the time required for the data conversion, but it also improved the overall performance of the production system.

Conclusion

PDB disk cache is important to the running of JADE systems. Although the above example times represent a data load scenario, there is also a significant benefit to normal production run time. The primary benefit is normally that higher-level collection blocks are found in memory, and the extra trips to disk are saved. This is typically reflected in online user response times and report run times.

Note Increasing the amount of PDB disk cache is one of the easiest ways to improve the performance of most JADE systems. All you have to do is add memory, if you are using the default setting of the **DiskCacheMaxSegments** parameter in the [PersistentDb] section of the JADE initialization file. If you have specified a non-default value for the **DiskCacheMaxSegments** parameter, you would need to increase it to a value that takes advantage of the extra memory.

Example 2 - A Large Number of Reads Flushing Out Cache

Background

A background application runs in a standard (fat) client node by itself. It produces a summary report upon request from presentation client users. The report is run hundreds of times a day, and generally takes less than a second.

A second application is added to the node, which provides a similar function but for a larger report. This report is requested only a few times a day.

The Problem

The original report, which used to run within a second, started taking 10 seconds or more from time to time. This delay was not acceptable for some users.

Diagnosing the Problem

From examination of the application logs, it was soon noticed that the slow runs of the original report always happened immediately after the new report was run. This was tried in a test database, and found to be easily reproducible.

Using the **System Statistics** sheet of JADE Monitor in the test environment, it was seen that the original report normally caused only a few dozen Get Objects per run. The report read through a large number of objects, but most of them did not change often so they were usually in the node's persistent object cache.

On each run of the new report, there were more than 254,000 Get Objects. After the new report was run, the original report caused over 116,000 Get Objects – a huge increase over a few dozen.

The objects fetched by the new report were flushing out the node's cache, so the original report had to read them in again.

The Solution

The node's cache could have been increased, or the new report could have been moved to its own node, but it was not run often enough to justify either of those. Instead, the **Process::adjustObjectCachePriority** method was called for each object read in by the new report, reducing its cache priority by one.

When an object is read into cache, its priority is automatically set to **1**. Reducing the priority by one means that if you caused the object to be read into cache, its priority will be set to zero (**0**). This effectively removes the object from cache immediately. The new report therefore caused only one object to be in cache at a time, preventing the cache from being totally filled with the new report's objects.

The original report's objects remained in cache, allowing its runs to be completed within a second again.

Conclusion

If it is unlikely that you will want an object in cache again, consider reducing its cache priority after use, so that it is removed. This will allow other objects, collection blocks, and other buffers more room in the cache, which can improve overall performance.

If you do this for newly created objects such as auditing information, be sure to reduce the cache priority after the commit and not before.

Appendix

This appendix contains the following topics.

- More about Windows Performance Monitor
- Additional Windows Tools

More about Windows Performance Monitor

You can access Windows Performance Monitor (**perfmon**) from Control Panel's **Administrative Tools**.

Right-click the form and then select **Add Counters**. For each counter, use **Show Description** to find out what it is. Some of the explanations assume a degree of knowledge about the internals of Windows.

A good basic set of counters for diagnosing performance issues would include:

- LogicalDisk object, % Disk Read Time, and % Disk Write Time on each disk used. You could use Avg Read and Write Queue Lengths instead.

  % Disk Read Time is more useful than Disk Reads/sec, because % Disk Read Time directly reports the amount of time an application is spending waiting for IO. If you look only at the count of IOs, you don't know how long they were taking.

- Processor object, % Processor Time, and Total.

Some additional counters are:

- System object, ContextSwitches/sec.
- LogicalDisk object, Avg Disk sec/Read, and Avg Disk sec/Write.

The JADE Monitor also provides most of these counters on the **Host Performance** sheet. You can get individual samples averaged over a selected interval, with either manual or timer refreshes. You can also log the information to a file for subsequent viewing and analysis.

Additional Windows Tools

You can obtain a Windows Performance Toolkit at https://docs.microsoft.com/en-us/.