Case Study
Don’t Guess Where Your Performance Problem Is

Xense Limited

The Problem

This case study examines a performance problem that a client had with a custom Documentum job. The particular process in question was a docbasic method that built complex document structures from template Virtual Documents. Custom logic was used to select the correct template and further rules were evaluated to select the relevant nodes from the Virtual Document.

The client had specific performance criteria that were not being met by the existing system configuration. The developer was asked to provide a solution.

The Wrong Solution

The solution proposed by the developer sounded entirely feasible; the code was to be rewritten in C and would incorporate caching of template structures so that if the same template was required during a single run of the batch process it was only loaded once.

The development and testing would take several weeks of effort but would be well worth it if performance improved. Sadly the improvement was only of the order of 5-10%; considerably less than required. Why didn’t performance improve to the extent hoped, and more importantly how could the developer have discovered this prior to beginning coding?

The Right Solution

Nearly 40 years ago Gene Amdahl expressed an idea that many performance analysts now know as Amdahl’s law. Amdahl was concerned with the relative merits of improving the speed of CPUs versus utilising multiple slower CPUs in parallel, however his insights have a more general application.

Figure 1a shows a schematic diagram of the execution profile of a program. The execution is nominally divided into 2 parts – the elapsed time of the execution path that will be affected by our proposed performance improvement (shaded area) and the remainder of the execution path which is unaffected by our changes. Figure 1b shows the result of implementing a performance improvement that speeds up the affected execution path 5 times. Despite the impressive speedup achieved for this section of the runtime the overall execution time is only improved by 20%.

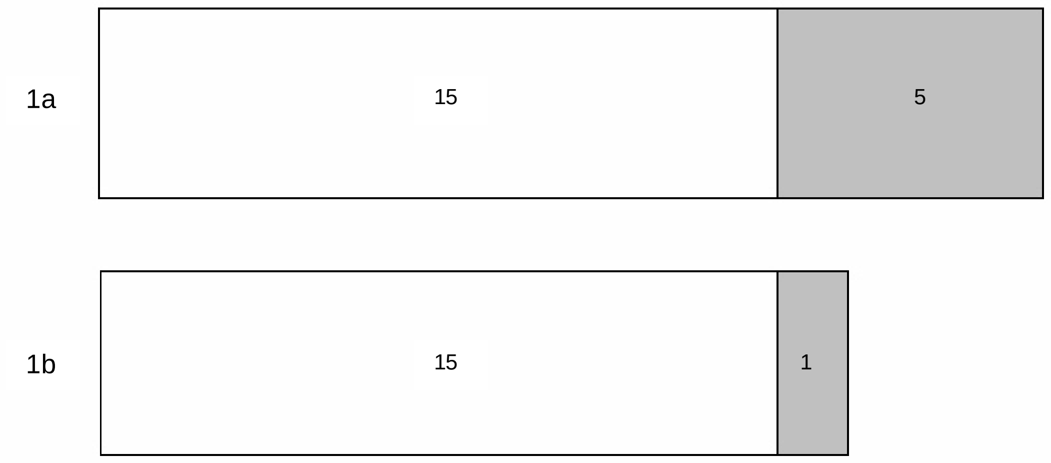

Figure 1 Program Speedup (20% improvement)

In fact even if we reduce the runtime of the targeted section of code to zero, the improvement will be only 25%. Of course if the execution profile resembles that shown in figure 2a our scope for improvement is greater; in this case the improvement shown in figure 2b is 60% with a maximum possible improvement of 75%.

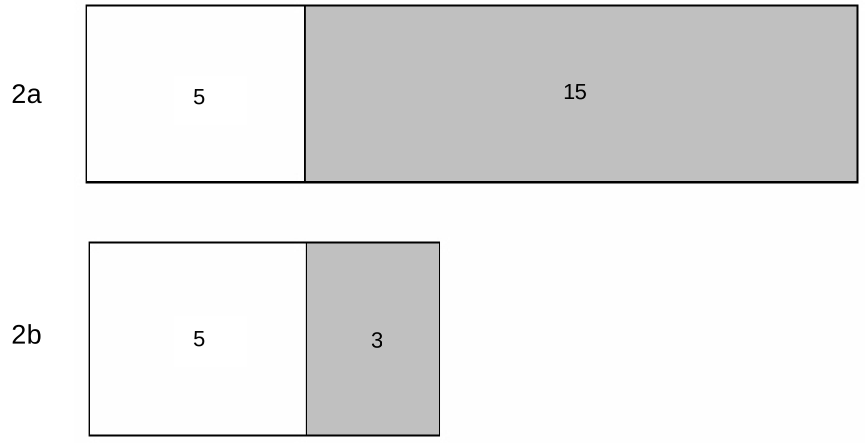

Figure 2 Program Speedup (60% improvement)

Clearly the proportion of the runtime that can be affected by any improvement is an important factor in assessing the benefit to be derived from the optimisation effort. Of course what we need to know is whether our problem and the proposed solution is like figure 1 or figure 2.

In fact, for many Documentum programs this is relatively easy. Figure 3 shows the technology stack for a typical dmbasic program that accesses the Content Server.

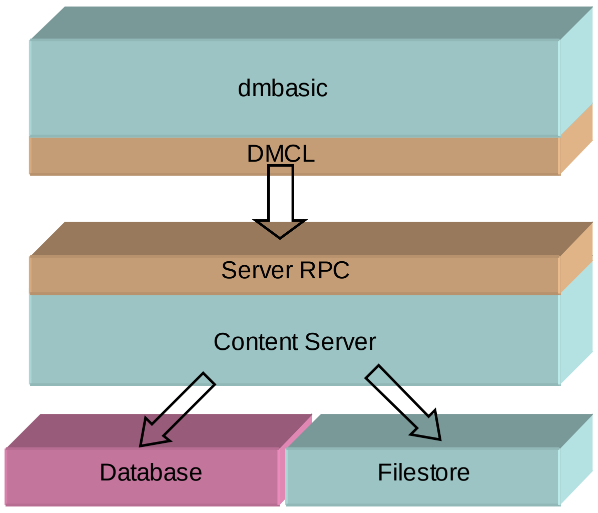

Figure 3 Technology Stack for dmbasic

A dmbasic client will run in a process separate to the Content Server, either local or remote to the Content Server. Whether running locally or remotely the program will typically make a connection to the Content Server and issue API calls. The component that enables the communication between the dmbasic program and the Content Server is called the Documentum Client Library (DMCL). This is implemented as a DLL (dmcl40.dll) on Windows or a shared library (dmcl40.so) on Unix.

The communication protocol between DMCL and the Content Server is a Remote Procedure Call (RPC). Functions called by the DMCL are executed on the Content Server. Each RPC typically results in 1 or more calls by the Content Server to the database and/or operations on the repository filestore(s).

Figure 4 shows a typical execution profile for a dmbasic program. The 5 sections shown represent the elapsed time for the dmbasic, DMCL, Content Server, Filestore and Database components from figure 3 (the Server RPC is not analysed separately from the rest of the Content Server). The diagram does not explicitly show the time spent operating over network connections since this particular system ran over a LAN and network latencies were not significant, however if any of the components were separated by low bandwidth and/or high latency network connections this aspect would need to be considered.

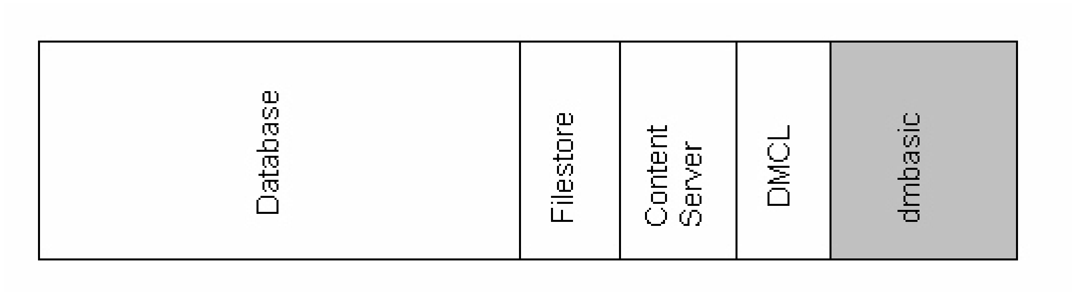

Figure 4 Typical Execution Profile of a dmbasic program

So how much impact would recoding the dmbasic into C have? Only the dmbasic component would be affected by such a change. How can we estimate the size of the dmbasic component relative to the other components? The best way is to use a DMCL trace of the execution of the program. A DMCL trace records each individual request and response made by the client code (in this case the dmbasic program) to the DMCL layer. It also records the elapsed time to service each call.

By measuring the total program execution time and the total time spent servicing DMCL calls it is possible to work out the size of the dmbasic execution box. DMCL Tracing can be enabled using the 'trace' API call – for this purpose tracing at level 9 or 10 is required as it gives the elapsed time to service each call. See the Documentum Content Server API reference for further details on the options for this call.

The resulting trace file will contain lines similar to the following:

```
# [ 740 ] Wed Jul 06 22:57:15 2005 562000 ( 0.000 sec) ( 0 rpc) API> connect,dm3,dmadmin,xxxxxxxx
# [ 740 ] Wed Jul 06 22:57:15 2005 578000 ( 0.016 sec) ( 0 rpc) Connection Pooling: User dmadmin
creates a new connection
# [ 740 ] Wed Jul 06 22:57:15 2005 687000 ( 0.125 sec) ( 1 rpc) INFO: New SessionID (010000038000811e)
# [ 740 ] Wed Jul 06 22:57:15 2005 734000 ( 0.172 sec) ( 5 rpc) Res: 's0'
# [ 740 ] Wed Jul 06 22:57:15 2005 734000 ( 0.000 sec) ( 5 rpc) API> query,c,select count(*) as cnt
from dm_folder
# [ 740 ] Wed Jul 06 22:57:15 2005 828000 ( 0.094 sec) ( 6 rpc) Res: 'q0'
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 6 rpc) API> next,c,q0
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) Res: 'OK'
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) API> get,c,q0,cnt
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) Res: '1358'
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) API> next,c,q0
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) Res: ' '
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) API> close,c,q0
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) Res: 'OK'
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) API> disconnect,c
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) Res: 'OK'
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) Total Time: 0.281 sec
# [ 740 ] Wed Jul 06 22:57:15 2005 843000 ( 0.000 sec) ( 7 rpc) Total Server Requests: 7
```

The trace file records a timestamp on the call from the client to the DMCL (the line containing API>) and a timestamp on the function return from the DMCL (the line containing Res:). In addition the trace file conveniently calculates the elapsed time between these 2 events (the first item in parentheses on the Res: line). By summing all the DMCL trace function elapsed times we can arrive at an estimate of the elapsed time spent in the DMCL, Content Server, Database and Filestore components. Since

Total Elapsed Time = dmbasic + DMCL + Content Server + Database + Filestore

We have :

dmbasic = Total Elapsed Time − (DMCL + Content Server + Database + Filestore)

Or

dmbasic = Total Elapsed Time − DMCL trace time

Using the figures in the example trace:

Total Elapsed Time = 0.281 secs
DMCL Trace time = 0.172 + 0.94 = 0.266 secs
Elapsed time for dmbasic = 0.281 − 0.266 = 0.015 secs

One of the problems with undertaking such an analysis is the sheer volume of trace data that can be produced. Until recently the usual approach taken to dealing with trace data was to parse it into a CSV file and then loading the CSV file into a spreadsheet application such as Excel. Parsing is typically done with awk or perl.

With the advent of Documentum Foundation Classes (DFC) and the web applications based on WDK the size of a typical trace has exploded. Often the number of DMCL calls made exceeds the size allowed for spreadsheets like Excel. A solution to this problem is the Xense DMCLPROF. DMCLPROF has a summary facility that allows a quick overview of the contents of a DMCL Trace. For example here is the summary for the small trace shown earlier:

```
DMCL Trace Summary
==================

Top 10 api calls:

Time       API
--------   --------------------------------
   0.172 - connect,dm3,dmadmin,xxxxxxxx
   0.094 - query,c,select count(*) as cnt from dm_folder
   0.000 - disconnect,c
   0.000 - close,c,q0
   0.000 - next,c,q0
   0.000 - get,c,q0,cnt
   0.000 - next,c,q0


API call summary:

API                      Freq    Duration
------------------- -------- -----------
close                       1       0.000
connect                     1       0.172
disconnect                  1       0.000
get                         1       0.000
next                        2       0.000
query                       1       0.094


Time Split:
Start Time           : Wed Jul 06 22:57:15 2005 562000
Total Trace duration :       0.281
DMCL call duration   :       0.266
Residual             :       0.015
```

The important section is the last 3 lines that automate the calculation discussed above.

Using this technique it is possible to predict the effectiveness of a particular optimization. In fact most dmbasic programs that access the Content Server typically do not spend much of their runtime executing the dmbasic code.

So we can see that the developer could have predicted up-front that the proposed conversion from dmbasic to C would not produce a significant speedup. What about the proposal to cache the templates in memory to avoid reloading them? Again the DMCL trace would allow us to see which operations are consuming the majority of the program runtime.

When the trace for the dmbasic program was examined it became apparent that the operations involved in locating and loading the templates were insignificant. In fact the majority of the elapsed time was consumed by a handful of long running queries unrelated to the templates; several hours of query tuning resulted in substantial performance improvements that met the clients requirements.

Conclusion

If you have, or think you have, a performance problem don′t guess where it is or how to fix it. Simple analysis can usually show where the problem is or at least where it is not! This advice could save you weeks or months or wasted work.