Optimizing Applications with SPT

By Stephen F. Macsisak
Hewlett-Packard Company
19447 Pruneridge Avenue.MS 47UA
Cupertino, CA 95014
408 447 5851
Fax 408 447 4278
Paper ID#-454

Optimizing Applications with SPT

By Stephen F. Macsisak
Hewlett-Packard Company
Paper ID#-454

Software Performance Tuner/XL (HP SPT/XL) is an application tuning performance tool. SPT allows a user to monitor and report on resources used by application programs. The tool lets the user identify the "HOT SPOTS" in a native mode application. Both batch and online programs can be easily monitored and information about what part of the program consumes the most CPU (similar to profiling on UNIX systems) and what system services are being used (tracing) can be obtained. Applications on MPE/iX - especially OLTP programs - spend most of their time in system code, but through use of SPT the programmer can identify unneeded calls and quickly highlight the most expensive database and file system intrinsics used in the application.

HP SPT/XL reports the following performance information:

. CPU utilization (i.e., which procedures consume the most CPU)
. Wait times (i.e., disk, memory, software locks)
. File system and HP TurboIMAGE activity
. Intrinsic use
. Compiler library usage
. Transaction response times and transaction counts

No changes have to be made to the program to use SPT. You can get more information (like line numbers instead of procedure address offsets) if the application is compiled with certain compiler options, but no recompiling is necessary. The software tool creates a log file - which can be analyzed later or on another machine - with all the information needed for analysis. HP SPT/XL is an add-on product with a simple install process. SPT only works on native mode applications and can only monitor one instance of a program on a system. Calls to the MPE/iX intrinsics STARTTRAN/ENDTRAN may be embedded in your application code to delimit logical transactions for the purpose of analysis using HP SPT/XL. You can also define logical transaction using any convient intrinsic call (like VREADFIELDS), which more or less delineates a transaction.

Any development shop that is actively writing, modifying or troubleshooting applications should have a copy of the SPT and periodically use it on the application that get executed the most.

How does SPT work?

SPT uses two data collection techniques: tracing and sampling.

Tracing

The data collection technique HP SPT/XL uses provides information about system code or about the intrinsics called by the application. Data is collected by intercepting the calls made by the program to an intrinsic and logging the necessary data; allowing the real intrinsic to be called,

and again, logging the required data after the intrinsic returns. This data provides information on the amount of CPU and elapsed time spent in intrinsics, as well as how much time the application waited for disk I/O, locks, etc. In some cases, certain parameters passed to the intrinsic are also logged (e.g., the database name for the various database intrinsics). Logically, this tracing method can be thought of as inserting an additional library before each system library (XL.PUB.SYS and NL.PUB.SYS). For each monitored intrinsic, these additional libraries have a procedure "stub" which logs the necessary data, calls the real intrinsic, and logs additional data using the measurement interface after the intrinsic returns. Tracing is most valuable in OLTP programs because the sampling technique requires a program to use large amounts of CPU and OLTP programs, often spending most of their time waiting not executing.

Sampling

This data collection technique provides information about where most of the time is being spent within application code. To accomplish this requires frequent logging (sampling) of the location of the instruction the application is currently executing. By sampling the application frequently, a statistical representation of where the application is executing can be obtained. For example, if an application is sampled every few milliseconds for a few minutes, the data can be analyzed to determine exactly what procedures were executing most often and therefore, where most of the time was spent.

SPT also collects all needed symbolic information from system/user libraries and program files and saves the data in the log file. It's a good idea to always set "SYSPROCNAMES" on when collecting data, to ensure you get symbolic information from system libraries.

Application Optimization

You need to identify what needs to be changed to improve performance on a system. Before you drill down to the application, it is a good idea to determine if you have a CPU or DISK or MEMORY bottleneck. Then you should identify what application workload puts the most stress on the system. If five users are in the accounts payable application system and one hundred users are running the order management application, then look at order management instead of anything else. The goal of any performance work should be to identify what will have the highest impact if changed and is the easiest to implement. If a function or program is used infrequently or puts little load on a system, it doesn't matter how inefficient the function is. It's easy to look at code and say that it is inefficient, but it's very hard to determine if a change can have any measurable impact. After you identify the high overhead workload, determine what program is being used the most and use SPT to study the application logic flow.

The way to optimize an application in an OLTP environment is to eliminate waits and serial threading. Usually, the goal is to make the CPU the bottleneck then faster CPUs can be used to improve response time or allow the application to scale. To improve response time or allow more users on the same system, you basically identify the inner loop of an application, remove unnecessary intrinsic calls and code.

The typical OLTP application flow on MPE/iX is:

- Startup
  - Open databases
  - Open files
  - Initialize variables
  - Build edit data tables
  - ….
- Loop
  - Display data on screen
    - Using View/Plus or field mode functions
  - At read completion
    - Edit data; write error messages if necessary and loop to get correct input
  - Read/write/modify databases or files
  - Display results of database access
  - Go to Loop and do it again
- Shutdown application
  - Close databases
  - Close files
  - Write summary data
- End

The following is a list of functions that should not be done in the inner loop of the application because they increase response time. This is not an all-inclusive list but most of the following have been found (using SPT) in the inner loop of a program and removed without changing the application logic. The changes had a major impact on response time or throughput.

Open/Close databases
Opening a TurboIMAGE database dynamically during a transaction can take more then 500 milli-seconds and is system-wide serial threaded. By calling these functions during a transaction, you may have to wait on other programs that are in the startup state; on many systems, a new process gets created every one to ten seconds.

Redundant calls to intrinsic
Programs sometimes call the same routine multiple times even when the data cannot change. It's amazing how often programs call the "GETTIME" intrinsic or the "WHO" intrinsic multiple times in the inner loop of an application. Of course, the session/logon names aren't going to change after the program gets started, so why call the routine?

Open/Close/Creating/Purging/Locking files

Opening files utilizes less CPU and is quicker than opening databases but can still be system-wide serial threaded on directory semaphores. You also increase the number of checkpoints because the access/modified dates/times change on files. Creating a new file in the beginning (alphabetically) of a large directory can put a heavy load on the transaction manger.

Locking/Unlocking files during a transaction could make you wait behind other programs locking the same file. Unlocking a file causes all dirty pages of the file to be flushed to disk. Sometimes programs modify the first and last record of a file (a kind of logging); the system will search the complete virtual address of the file looking for dirty pages even if only one or two pages have been changed. The application will run slower and slower as the file grows longer because the system will have to search more of the address space of the file for dirty pages. One solution would be to use two files, one as a "guard file" to protect the other file which is being modified.

Accessing run-time variables

Within a single session, a run-time variable is not likely to change during a transaction. Yet, some programs read the same variables repeatedly in the inner loop of the program. This doesn't have much overall impact on response time, but does consume CPU: the larger the run-time variable list, the more overhead required to access them.

Putting/Deleting Data in a TurboIMAGE database

A DBPUT or DBDELETE causes at least a set level lock of a database and the lock could be held while a disk I/O completes. Application logic may require these operations, but they will cause an application not to scale very well because you are running at single disk speeds. One solution is to enable the PREFETCH option in TurboIMAGE using DBUTIL which causes the disk I/O to be done before the lock is issued. Another solution is to enable the DSEM option which allows parallel Putting/Deleting to different database sets, if possible; otherwise, TurboIMAGE locks at the database level during a PUT/DELETE. One last consequence of database putting/deleting is increased load on the transaction manager, which causes heavy XM logging and frequent checkpoints.

Disk I/O

Most OLTP programs spend most of their time waiting on disk or other logical bottlenecks. You can't eliminate all disk I/O in a transaction, but the I/O can be minimized by keeping detail sets packed so detail chains are together on the disk not scattered all through the dataset. Packing data sets requires some kind of database tool, but the tool should be run on a regular schedule. The HIGHWATER mark PUT function of TurboIMAGE may help, but causes a dataset to fill up more quickly. There are many other TurboIMAGE optimizations but SPT can help you understand what slows down the application during production.

File Control Functions

Issuing some file control functions can cause a noticeable delay. For example, setting Control Y when using Virtual Terminals causes the server computer to send a request to the PC over the network and wait for a response.

Writing debug message
Often times users put in debug messages and use compiler functions like DISPLAY to have output written to a file. Typically, the compiler library functions are interruptive at run time and incur more overhead than calling FWRITE or PRINT. If the program is in production, it probably should not spew output during the inner loop of the transaction.

None of the above items are difficult to change in an application and most programmers don't deliberately write code to make the application run slower. What frequently happens is that someone builds library functions, then someone else calls the functions without understanding what they do. Or a programmer is doing maintaince and doesn't know the complete flow of a program. Or a large team of programmers works on an application and builds many layers using the best design/programming techniques, but never looks at the actual program flow during a production transaction (it is hard to do). On the other hand, the costs of some system functions are not well understood.

How do you solve these problems? Use SPT to analyze the program during development or study the program when it is in production to determine the "real" logic flow and cost of system functions on a busy system.

An Approach to Analyzing a Production Program with SPT to Identify the Inner Transaction Loop

Collection of data
Use SPT to collect data on one instance of the program in production (make sure the user is active). Don't run the program underneath SPT, but attach SPT to a real production user. Only the user who is being monitored will be affected by the collection process. Collect data on ten to fifty transactions and do the collection during a busy time of the day. Look at the program after the STARTUP phase is over. Ideally, the online user would have already executed several transactions.

Analysis of data
End the collection function and start the analysis part of SPT. When asked by SPT if you want to define transactions, answer no. SPT summarizes data by logical transactions, but we don't know what a logical transaction is yet. Look at the file usage and database usage displays for anything that looks unusual. If the program uses large amounts of CPU or is a batch program, then look at the code sampling data (but in typical OLTP programs very few samples are collected and the data is not relevant). Now progress through the SPT menus looking for the individual transaction softkey and select it. Select the intrinsic trace softkey and examine the trace. If you have already defined transactions, selecting the individual transactions softkey leads you to a Transaction CPU Time Histogram display. You cannot look at the intrinsic trace until you are only looking at one transaction, so keep drilling down until you see the intrinsic trace softkey.

The screen output from the intrinsic trace looks something like the following:

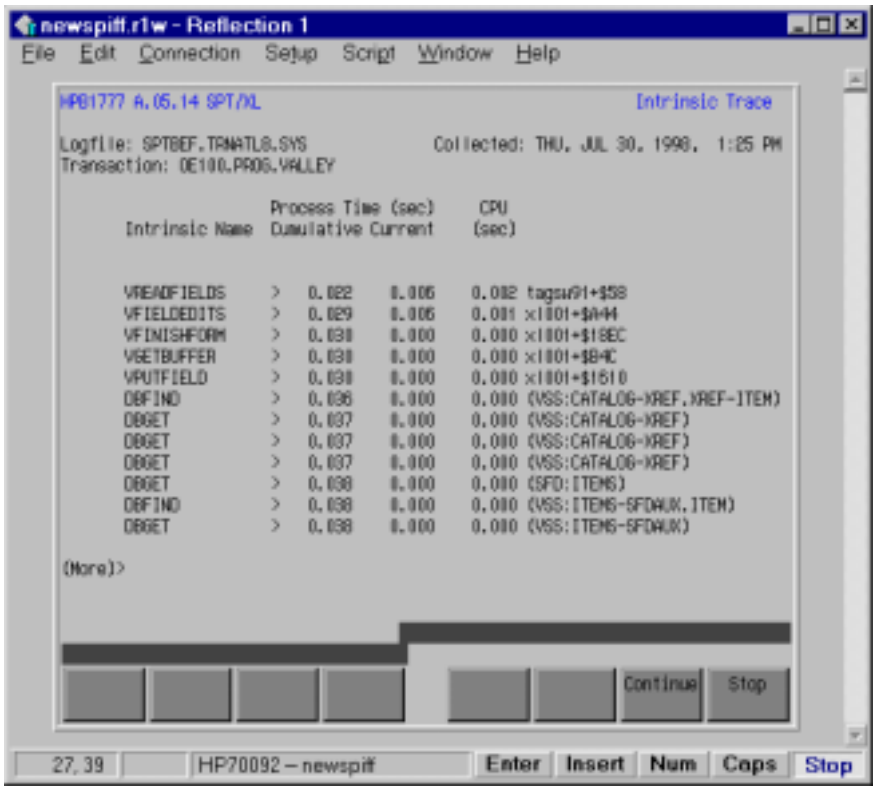


Figure 1. Intrinsic Trace

Screen Definitions

Intrinsic Name Intrinsics encountered for this instance of this transaction.

Process Time Cumulative Cumulative processing time of all intrinsics for this transaction.

Process Time Current Individual processing time for this intrinsic.

CPU (sec) CPU time for this intrinsic.

(Database name) Selected parameter information or calling location address

Our goal is to identify the logical transaction loop of the program. Do this by looking for calls like VREADFIELDS or the FREAD of some file or some number of terminal reads followed by no terminal read calls. After we identify the intrinsic call to bracket transactions, go back and define transactions. Then look at the short transactions vs. the long transactions in the trace and determine why they are different. Also look at total application flow using the intrinsic trace. If an intrinsic call took little or no CPU time but took process time, then the program is waiting on

something. Now determine: is the call necessary? Could it be moved out of the inner loop? Could it be done in a more efficient manner? A DBGET would likely be waiting on a disk I/O if the process time were greater than ten milliseconds. If within one transaction, we do one DBGET and within other transactions, the program does many DBGETSs on the same dataset, then program could be reading a long detail chain and repacking could help response time.

At this point, the performance analyst can continue identifying patterns and expensive intrinsic calls, but someone who is knowledgeable about the application should be involved. Usually the application programmers find it easy to fix the program when the logic flow is clear. You also need a current copy of the source.

Now look at a real data from a production system. The data was collected over about thirty minutes of time. The complaint was erratic response time. The system CPU was not overloaded; the disk I/O were evenly spread across all the volumes and were not excessive on any one disk. The system had adequate memory. What was going on?

Figure 1 depicts the start of the trace. The application is executing a VREADFIELDS intrinsic call. The call takes a few milliseconds of CPU time, but everything is okay. If a call takes less than a millisecond, the current time is zero for that call, but you can see it takes about three DBGETS to increase the cumulative process time by one millisecond. The database name and dataset name is displayed for the database calls. Other intrinsic calls like VREADFIELDS only show the procedure name and address offset from where the application the call was made.

On the next few screens we continue our analysis of a logical transaction using the intrinsic trace.

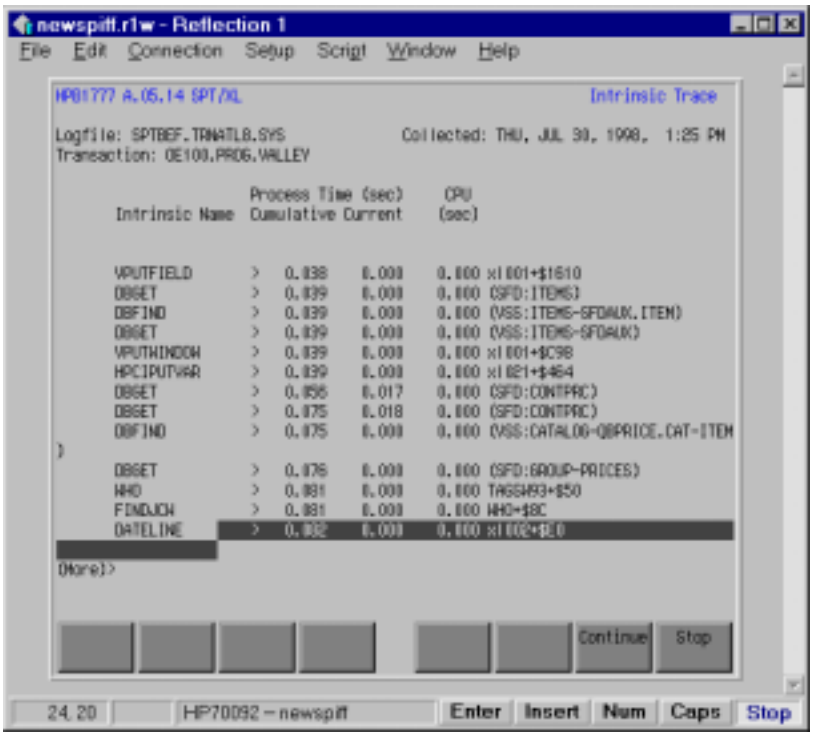


Figure 2

In Figure 2, the intrinsic trace continues for the same program. We can see that the two DBGETS took 17 and 18 milliseconds of process time. Each of the DBGET calls used little or no CPU so the program probably waited on a disk I/O at that point. In addition, the application code called the WHO intrinsic, FINDJCW intrinsic and the DATELINE intrinsic. The results of the WHO intrinsic could not have changed since the last call.

Figure 3

In Figure 3, we skipped forward in the intrinsic trace. Multiple calls to HPCIGETVAR from something in the C run-time library can be seen. It did not take long, but why are we doing it? Of

special interest is the DBCLOSE of LIBDB. If this is the inner loop of the transaction, what are we doing closing a database?

Figure 4

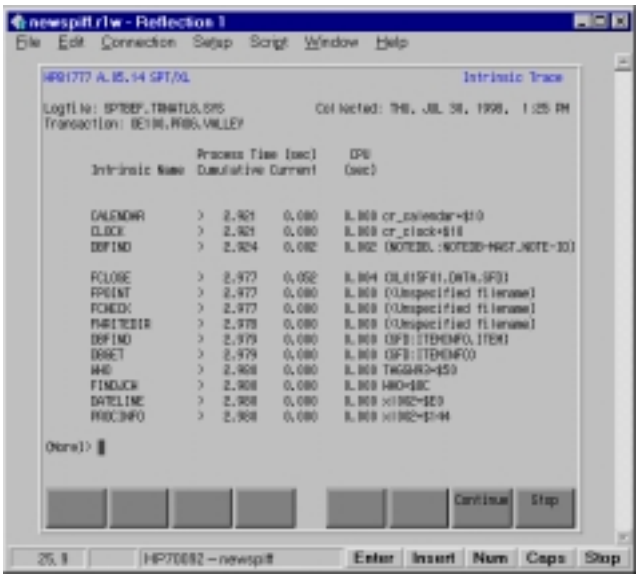
In Figure 4, things get really interesting. We see the application do a FIND/GET/UPDATE/UNLOCK of the VSS database. Also some part of the code calls the WHO intrinsic again and the code does the same thing it did before even though nothing has changed in the run-time environment. Remember, look for patterns and expense calls to understand how the program works. In Figure 3, we closed the LIBDB database. Now we reopen the LIBDB database and it takes a long time - about 1.6 seconds - most of which is wait time not CPU consumption. Of course, the online user is waiting in front of their PC for some kind of response.

Figure 5

Figure 6

In Figure 5, a HPFOPEN call appears when opening STDIN for some reason. The program read something out of the LIBDB database, then the database is closed and reopened. To this point, what intrinsic has taken the most process time? The DBOPEN of LIBDB!

In Figure 6, more access to the LIBDB database is visible, then it is closed! The program repeats itself with the three calls to HPCIGETVAR. We also FOPEN a file. You might go back and look at Figures 3 and 4 and compare them with Figures 6 and 7: a pattern is emerging and the application logic flow is now becoming clearer.



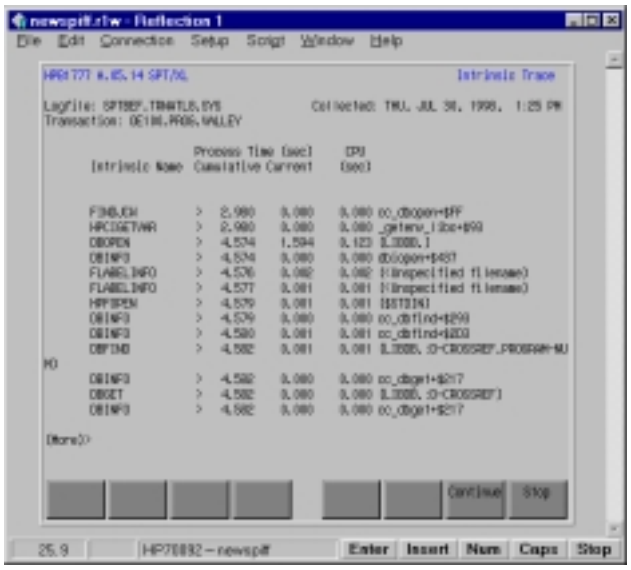


Figure 7

Figure 8

In Figure 7, we close the file opened in Figure 6 and call the “WHO” intrinsic again, etc. In Figure 8, what line has the longest process time? The call to DBOPEN has taken the most process time so far. Now do it again, you are still within the inner loop of the transaction. The user is still out there waiting at the PC and it takes 1.6 seconds of process time again.

End of the Transaction

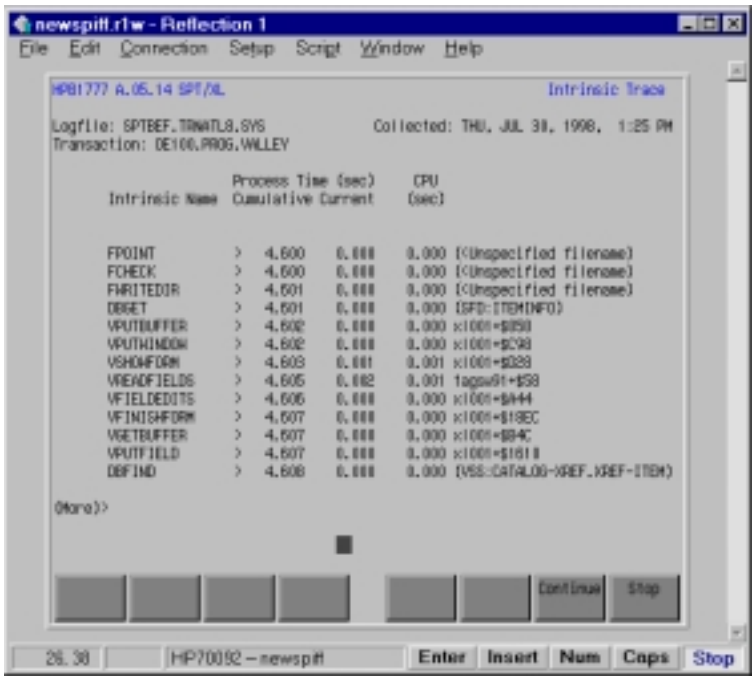


Figure 9

In Figure 9, you are at the end of the logical transaction and the start of another logical transaction. In the log file, this data came from the program, did the same thing repeatedly for about ten transactions. You can see the VREADFIEDS that started the previous transaction. Don't count the time spent waiting on the terminal as process time because its "THINK" time. The logical transaction took about 4.6 seconds during which the same database was open/closed twice, taking about 3.2 seconds. The program also made redundant calls to HPCIGETVAR, the WHO intrinsic and HPFOPEN/CLOSED $STDIN twice and FOPENED/CLOSEED a file. Even though the file opens didn't take too much time, they can be system-wide serial threaded on the directory semaphores and should not be inside the logical transaction. The trace doesn't show it here, but since all the online programs were doing the same thing as all the programs in the startup state, online users were affected by other online users as well as new programs starting up. The erratic response time came from the DBOPENS that were hidden away in a set of library routines and the code really didn't need to do these functions in the inner loop.

The example is a worst-case scenario, but similar types of code sequences inside the inner loop of a program are not uncommon. It takes a tool like SPT to discover the logic flow and help identify expensive intrinsic calls.

The manual for the product is available at HTTP://DOCS.HP.COM and includes a walk though on collecting and analyzing a program. There are two manuals: *HP SPT/XL User's Manual: Analysis Software* and *HP SPT/XL User's Manual: Collection Software*. The product number is B1776A for the SPT Collector and B1777A for the SPT Analyzer.