Using the Coverage Profiler

By Doug Hennig

Overview

Visual FoxPro (VFP) 6 includes an important tool called the Coverage Profiler that provides both code coverage and profiling. This document examines how to use this tool for effective application testing, and shows how simple add-ins can increase its usefulness.

I'm not going to discuss when or why, but how. In other words, we're not going to look at testing and profiling theories and concepts, but rather how the Coverage Profiler works and how to use it. See the Bibliography for a short list of books and articles on testing.

Introduction

VFP 5 added the ability to perform coverage logging. The result was a log file showing what code executed and how long it took. VFP 6 greatly extends the usefulness of coverage logging by providing a tool called the Coverage Profiler to analyze log files. VFP 6 Service Pack (SP) 3 includes several improvements to both coverage logging and the Coverage Profiler; see the Appendix for a list of the enhancements.

Before we get into the Coverage Profiler, let's discuss a few terms. "Coverage" means identifying which statements in your code were executed in one or more execution runs. The purpose of coverage is to ensure that the code was thoroughly tested by executing every "coverable" statement (we'll discuss the difference between "coverable" and "uncoverable" code later) from every way it can be called. "Profiling" means measuring the performance of your code to determine where the bottlenecks are. Code may take a long time (relatively) to execute because it's slow, executes many times, or both. The purpose of profiling is to improve the performance of the code by eliminating or minimizing the bottlenecks.

Logging Your Code

Before using the Coverage Profiler, you have to log your code. You do this by either using the SET COVERAGE TO <filename> command or clicking on the Toggle Coverage Logging button in the Debugger toolbar to log the code to the specified text file (include the ADDITIVE clause of the command or the Overwrite checkbox in the dialog to add to an existing file). You can also do this programmatically in your code if you want to start logging at a certain point rather than the start of your program. Logging slows your code down considerably, so later when we look at duration times, remember that these times should be considered relative, not absolute. This is yet another example of how the act of observing a system changes the system's behavior, kind of like how kids can be angels when someone else is around but hellions when it's just their parents.

After turning logging on, run your program. If you want to log the entire application, run the APP or main program for the application. Otherwise, run just the code you're interested in (you may have to set paths and other environmental things first). After you're finished running the code, turn logging off and close the log file by either using SET COVERAGE TO without specifying a filename or choosing the Toggle Coverage Logging button in the Debugger toolbar.

The Log File

One thing you'll immediately notice is that the log file can be huge, especially if you're logging object code. Even a simple program that only takes a few seconds to run can generate a log file of hundreds of KB in size. Be sure to have enough disk space free before you start!

The log file is a comma-delimited text file similar to the one shown below. I numbered the lines in this example and added spaces after commas for formatting purposes only. Also, this doesn't show contiguous lines, only certain representative lines, for brevity. Table 1 describes the contents of a log file.

```
1. 0.578851, , dbcx, 4, f:\sessions\coverage\source\dbcx.fxp, 1
2. 0.000313, dbcxmgr, dbcxmgr.init, 39, f:\sessions\coverage\source\dbcx\dbcxmgr.vct, 2
3. 0.000234, dbcxmgr, dbcxmgr.reseterror, 14, f:\sessions\coverage\source\dbcx\dbcxmgr.vct, 3
4. 0.000250, dbcxmgr, dbcxmgr.opendbcxmeta, 19, f:\sessions\coverage\source\dbcx\dbcxmgr.vct, 3
5. 0.105006, coremgr, coremgr.dbcxgetproplist, 26, f:\sessions\coverage\source\dbcx\dbcxmgr.vct, 4
6. 0.000218, basemgr, basemgr.opendbcxmeta, 17, f:\sessions\coverage\source\dbcx\dbcxmgr.vct, 5
7.         , basemgr, basemgr.opendbcxmeta, 20, f:\sessions\coverage\source\dbcx\dbcxmgr.vct, 5
```

Column	Contents
1	The duration of the line of code. This value includes the time it takes to execute any methods, procedures, or functions called by this line. For example, the code profiled in line 1 takes 0.578851 seconds because that includes the time to execute most of the code in the entire log. Note that some lines, such as line 7 above, show no time for certain types of statements; see *What's Not Covered*. In VFP 6 SP3, this now has 6 digits of precision.
2	The class the line of code is in; empty for non-objects. In line 1, this is blank because this code resides in an FXP. In the others, it contains the name of the class.
3	The name of the procedure or method. In the case of an object, it includes the object hierarchy (for example, frmCustomer.pgfForm.Page1.txtTextBox.GotFocus in the case of a textbox on a page of a pageframe in a form). In the case of an APP or EXE, it contains the name of the main program.
4	The line number. For PRG, MPR, or QPR files, the line number is relative to the start of the file. For all others (including stored procedures in a DBC), it's relative to the start of the method or procedure.

5	The fully qualified name of the file the object or procedure is in.
6	The call stack level (new to VFP 6).

Table 1. Contents of coverage log file.

What's Not Covered

Some of the lines in the executed code don't appear in the log file. For example, the second entry in the log above is the first line of code executed in DBCXMgr.Init, yet it's line 39 in this method; what happened to lines 1 through 38? Also, the seventh entry in the log has no duration.

Lines that never appear in the log include:

- all but the last line in a multi-line code statement
- comments
- blank lines
- #DEFINEs
- lines between TEXT/ENDTEXT statements
- DEFINE CLASS/ENDDEFINE and initial property assignment statements in a programmatically defined class
- PROCEDURE/ENDPROC and FUNCTION/ENDFUNCTION lines
- PARAMETERS and LPARAMETERS

Notice that all of these but the first one have one thing in common: they don't actually execute themselves but simply mark the beginning or end of a control structure, function, or class, or otherwise do nothing directly.

Some statements, like OTHERWISE, CASE, ELSE, and ENDIF, appear inconsistently in the log file. If an OTHERWISE or CASE is taken, it appears. If an earlier CASE is taken, sometimes subsequent CASE and OTHERWISE statements appear and sometimes they don't. ELSE and ENDIF statements sometimes appear and sometimes not.

Some lines appear in the log but with no duration:

- DO CASE
- CASE that calls a function or method or isn't actually taken

We'll see why some lines not appearing in the log can be an issue later, especially when it comes to measuring lines of code covered in an execution run.

The Coverage Profiler

The Coverage Profiler is an application included with VFP 6 (it also works in VFP 5 with a little coaxing) that provides analysis of a coverage log file, both from coverage and profiling points of view. The _COVERAGE system variable points to the application (COVERAGE.APP in the VFP home directory by default), which can be run either from the Coverage Profiler item in the Tools menu or with DO (_COVERAGE). If you run it using the latter, you can optionally pass up to three parameters:

- The name of a log file to open.
- .T. to run with no user interface. In this case, the Coverage Profiler will mark all the code both for coverage and profiling and output the results to a table you can later analyze (it also displays this table in a browse window for you).
- The name of an add-in to automatically run (we'll discuss add-ins later).

If coverage logging is enabled when the Coverage Profiler is run, it turns off logging and automatically opens the log file that was in use. When you close the Coverage Profiler, it asks if you'd like to reenable coverage logging (either overwriting or appending to the same log file) or leave it off.

If logging wasn't enabled and you didn't pass the name of a log file, you'll be prompted to select the log file you want to analyze.

Before we get into the functionality of the Coverage Profiler, here are some tips:

- Although you can create the log file on a slower machine if necessary, run the Coverage Profiler on the fastest machine possible. Loading the log file and locating and marking source code is very time-consuming because there are usually lots of log records. Also, you'll probably need lots of disk space; it's not uncommon for the cursors created by the Coverage Profiler to be several MB (or even hundreds of MB for large logs) in size.
- Make your project directory the current one before you start coverage logging so file paths are more easily resolved.
- Avoid renaming objects dynamically. It's more difficult for the Coverage Profiler to find objects in the source code files if you rename them at runtime.
- If possible, avoid using source files with the same name, even with different extensions.
- Before starting coverage logging, build the application with Debug Info turned on, Encrypted turned off, and Recompile All Files turned on. This ensures the latest copy of the source code will be logged and analyzed.

Coverage Profiler Functionality

The Coverage Profiler appears in a modeless formset as shown in Figure 1. There are three main areas in this window: the toolbar, the “module” list (I use the term “module” generically in this document to refer to classes, PRGs, and other source code files), and the code frame. An invisible splitter between the module list and the code frame allows you to adjust the sizes of these areas. The module list shows the modules that were hit during the execution run (although not in the order the code was executed) and the code frame shows the code for the selected module (again, not in the order the code was executed).

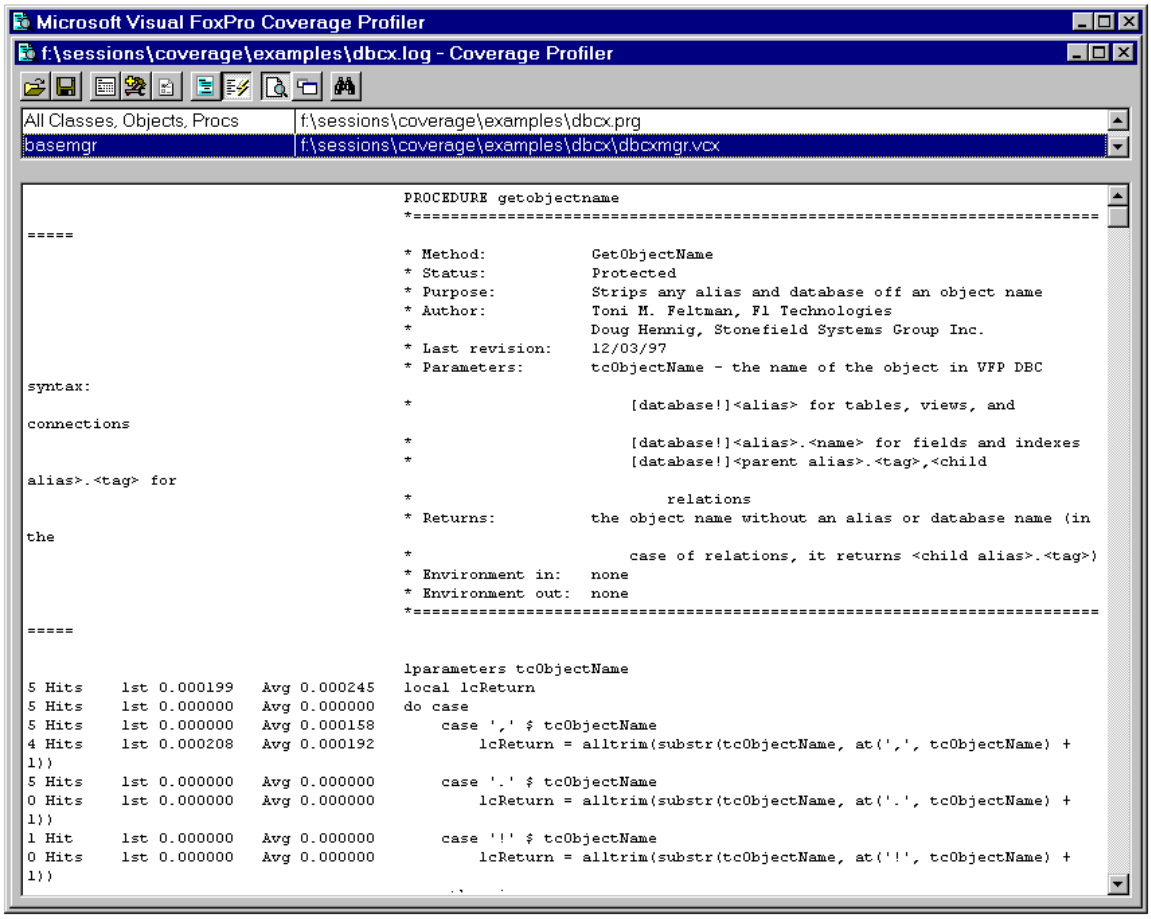


Figure 1. Coverage Profiler user interface.

You can switch between Preview and Zoom modes using the appropriate button in the toolbar. In Preview mode (the default), the toolbar, module list, and code frame appear in the same window. In Zoom mode, the toolbar and module list appear in one window and the code frame in another; these windows can be independently sized and arranged. When you’re in Zoom mode, you can also cascade the windows using an item in the context menu.

Unless you have the “Mark all code while log loads” option turned on (we’ll look at options in a few moments), the Coverage Profiler will “mark” the code for the first module in the list.

"Marking" means opening the source code file, reading in all the code, then comparing on a line-by-line basis with the log file and marking which lines were executed, how many times, and how long it took. The first time you click on a module in the list, the Coverage Profiler will mark that module's code, then display it in the code frame. If you want to avoid marking code automatically like this, ensure you're in Zoom mode, then right-click anywhere in the Coverage Profiler and choose Fast Zoom Mode from the context menu.

You can switch between Coverage and Profile analysis modes using the Coverage and Profile Mode buttons. The different between these is that Coverage mode shows which code executed and which did not (using mark characters; by default, executed code isn't marked and unexecuted code is marked with a vertical bar), while Profile mode (shown in Figure 1) shows how many times each statement was executed, the amount of time the first hit took, and the average time for each hit. As we'll see, you can define which mode you'd like the Coverage Profiler to start in.

The Find button (new in SP3) displays a dialog for locating text. You can indicate if the search is case sensitive and if it should wrap around, plus you can indicate where the Coverage Profiler should search: in classes, files, and source code.

Lies, Damn Lies, and Statistics

The Statistics button displays a dialog of statistics about the selected module and the source code file the module is in (if applicable); this dialog is shown in Figure 2. You can also display this dialog from the context menu.

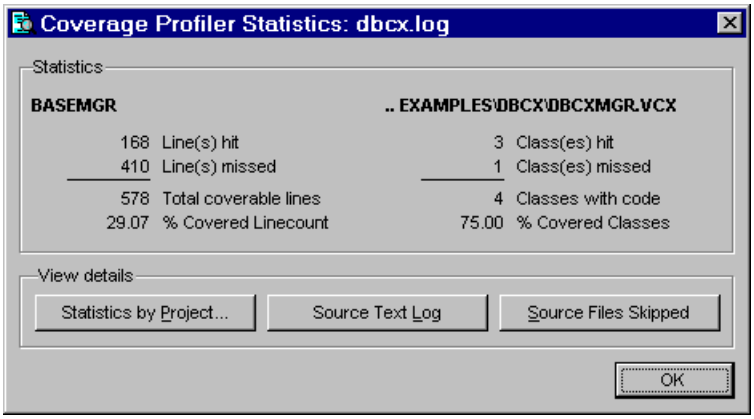


Figure 2. Coverage Profiler Statistics.

In Figure 2, we can see that there were 578 coverable lines in the BaseMgr class, of which 168 or about 29% were executed. Notice that there may be more lines of code in this class; because of omitted or zero duration lines in the log file (discussed earlier), the number of lines displayed in the statistics dialog won't usually balance with the number of actual lines in your code. You'll need to focus on which lines in your code are actually "coverable" when looking at which ones have or have not been executed. The figure also shows that of the four classes in DBCXMGR.VCX, three were hit.

In addition to showing statistics, this dialog has buttons for other functions. The Statistics by Project button prompts you for the project the executed code belongs to and then runs the report shown in Figure 3. This report shows details for each file in the project, which files weren't logged, and coverage statistics for the project as a whole. It's more convenient to print this report and go through it when analyzing the coverage of your application than to go through the modules one at a time in the Coverage Profiler.

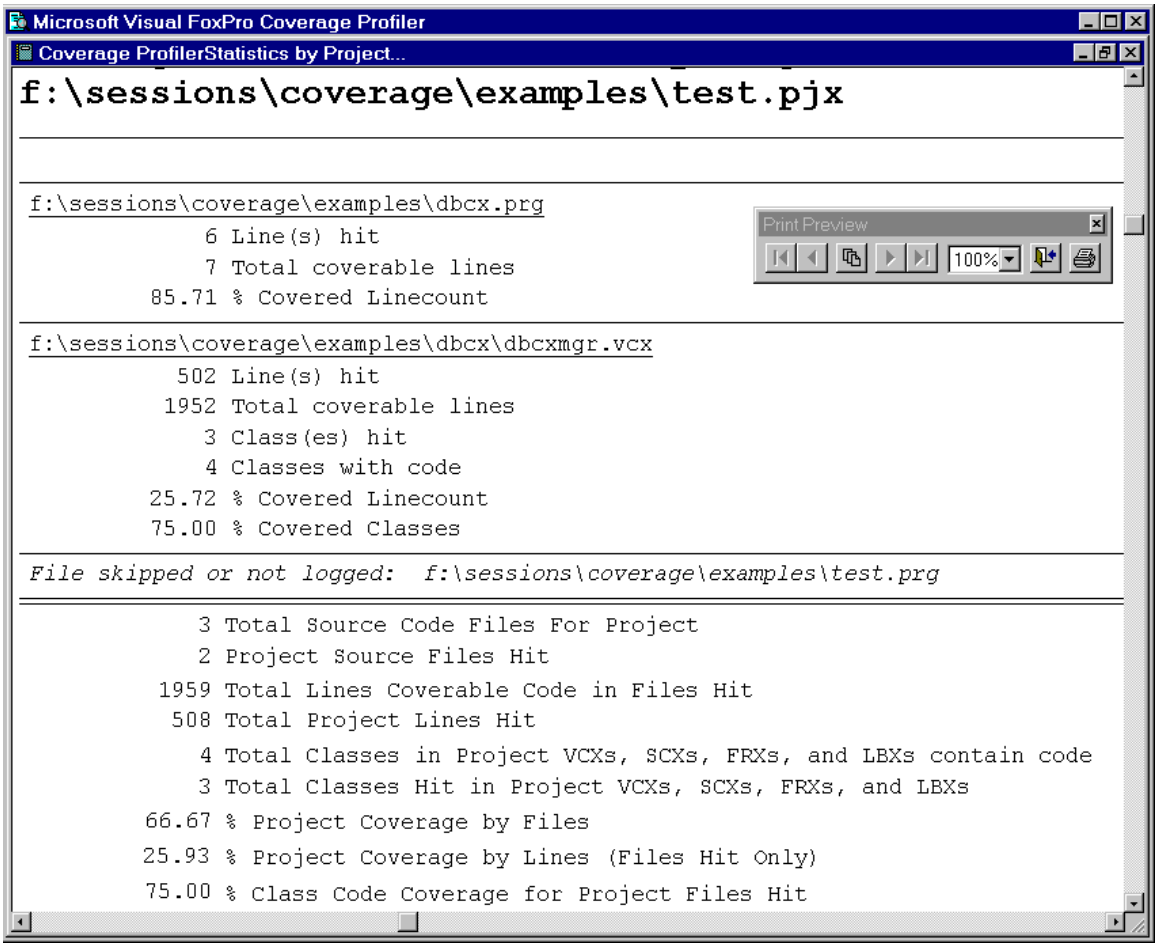


Figure 3. Statistics by Project Report.

The Source Text Log button displays the log file in a read-only window. You can also display this window by choosing View Source Log from the context menu. The Source Files Skipped button displays the list of files that were skipped in a browse window.

Configuration Options

The Options button displays the dialog shown in Figure 4 so you can configure the Coverage Profiler to suit your tastes. The options are described in Table 2. Some of these options can also be chosen from the context menu for the Coverage Profiler; these items are identified in the table.

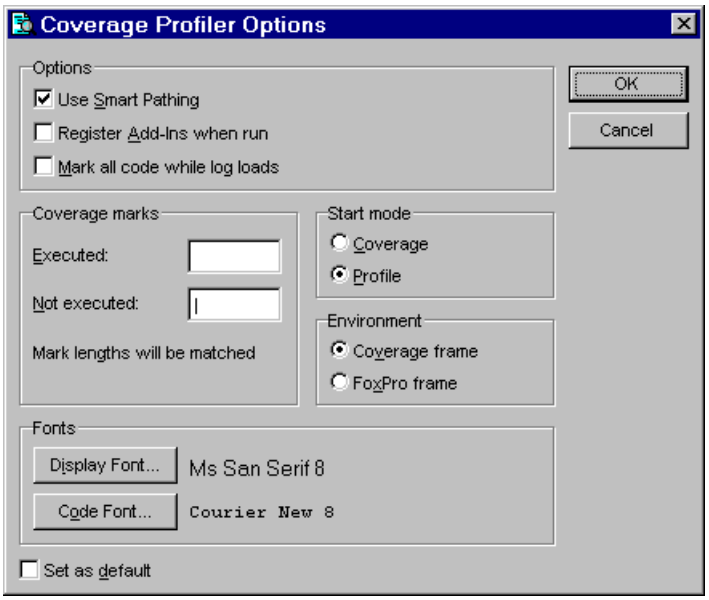


Figure 4. Coverage Profiler Options Dialog.

Option	Description
Use Smart Pathing	If this option is turned on, which it is by default, the Coverage Profiler will, like the Project Manager, automatically look for files in the same locations as previously found files. This option is also available in the context menu.
Register Add-Ins when run	The Coverage Profiler will automatically register add-ins when they're run if this option is checked. You can also register add-ins in the Add-In dialog.
Mark all code while log loads	If this option is checked, the Coverage Profiler will mark all code when it loads a log. It will take considerably longer to display the Coverage Profiler if this is turned on, but then you won't have to wait while it marks code the first time you select a class or file.
Coverage marks	You can define which characters mark code that's executed (no mark by default) and not executed (a vertical bar by default).
Start mode	You can specify whether Coverage Profiler should open in Coverage (the default) or Profile Mode.
Environment	The Coverage Profiler runs in a top-level window ("Coverage frame") by default. Set this option to "FoxPro frame" to use a window inside the main Visual FoxPro window. Note: you have to close and restart the Coverage Profiler for this to take effect.
Fonts	This specifies the fonts the Coverage Profiler uses for dialogs and

	for code. These options can also be set from the context menu.
Set as default	Check this option to save these settings as default options.

Table 2. Coverage Profiler Options.

Other Features

The Open button displays a dialog from which you can pick a log file to open. You can only have one log file open at a time, although you could create a subclass of the coverage engine if you want to support multiple log files. The Save button creates a table showing information about each module, including marked code.

The Add-ins button displays a dialog in which you can choose a Coverage Profiler add-in to run and, optionally, register so it will be available the next time you use the Coverage Profiler. You can also display the add-ins dialog from the context menu. We'll discuss add-ins in more detail later.

Taking 'er Out for a Spin

To try out the Coverage Profiler on a small but fairly complex set of code, let's use DBCX. DBCX is a public domain data dictionary extension manager created by a collaborative effort between several leading FoxPro development companies. The main class, DBCXMgr, is primarily a Facade class: it presents an interface but the majority of the work is actually done by other classes. These other classes, called extension managers, are subclasses of the abstract BaseMgr class. The only manager we'll use in this example is CoreMgr, which manages "core" properties (such as structural information) for data items (tables, fields, indexes, views, etc.). CoreMgr is automatically instantiated by DBCXMgr, so all we have to worry about is getting DBCXMgr up and running.

DBCX.PRG is a simple program that opens the TESTDATA database that comes with VFP, instantiates DBCXMgr, and has DBCXMgr "validate" the database (create records for each data item in the database in a set of meta data tables). SET COVERAGE TO DBCX, then run DBCX.PRG. After it's done, bring up the Coverage Profiler so you can see what happened.

Here's are some of the interesting things I found when I ran DBCX.PRG (your durations times may be different than mine, depending on various factors).

DBCX.LOG, the coverage file generated by this short run, was over 2 MB in size, and had over 23,000 lines of text. Pretty big for a program that took around ten seconds to run!

It took about 0.6 seconds to instantiate DBCXMgr. Why so long? There could be several reasons: there might be a lot of code, there might be some code that's executed a lot of times, there might be some slow code, or some combination of these (lots of slow code that's executed a lot of times should be avoided <g>). Let's see what the cause is here. Choose DBCXMgr in the list of modules, then find the Init method. Rather than looking at every line, we'll just look for those that execute several times and/or take more than, say,

0.001 seconds to execute. Why focus on these? Well, what good does it do to look for ways to optimize code that runs fast and only executes once? Yes, we might get some improvements there, but we'll get a lot more bang for the buck by concentrating on slow code and repetitive code such as loops.

Only a few lines of code met these criteria. One is a SQL SELECT statement. Maybe it's not Rushmore optimized, so that's something we might squeeze some time out of. However, it only executes once and there will never be more than a handful of records it has to select, so the amount of effort spent optimizing this statement may not be worth the effort. Another is a call to the InstantiateManager method, which instantiates one or more extension manager classes (since it was only hit once, only one class was instantiated). Looking at that method, there are two lines of code that meet the same criteria. One is a FILE() statement that ensures a VCX exists; that's a disk-bound activity, so there's not much we can do about it. The other is an AddObject statement that instantiates an extension manager class. We might want to drill down into the Init method of that class to see if there are any performance gains we can make there. Another "slow" line is a call to the CreatePropCursor method, which asks each extension manager to populate a cursor with a list of the meta data it manages. This is another method we might want to spend some time digging into, seeing how we could improve its performance.

The call to DBCXMgr.Validate in DBCX.PRG took almost nine seconds. That's by far the vast majority of the total time spent, so let's take a peek there. One statement, which calls CoreMgr.Validate to validate the database, executes once and takes 0.12 seconds. CoreMgr.Validate is also called another five times (once for each table in the database), each averaging almost a second. Two more calls to CoreMgr.Validate, for views, average close to two seconds. One final set of calls, for four relations, took about 0.08 seconds each. Clearly, looking into CoreMgr.Validate would be a good idea, since it seems to be called frequently *and* takes quite a while (relatively) to execute.

We could dig into these routines and see where the bottlenecks are, but I think you get the idea. Focus on statements that are executed a lot of times (usually in a loop but also called excessively, perhaps needlessly) and those that take longer than a certain amount of time to execute. You have to decide what threshold you want to use for these.

Extending the Coverage Profiler

The Coverage Profiler is actually just an application that presents an interface for the Coverage engine, a set of classes that provide coverage and profiling services. The source code for the application and the engine is included with VFP 6, in XSOURCE.ZIP in the TOOLS\XSOURCE directory of the VFP home directory. The Coverage Profiler can be extended in one of two ways: by subclassing the Coverage engine and/or interface classes and creating a different Coverage Profiler application, or by creating and running add-ins that add functionality to the Coverage Profiler.

The mechanics of both of these are beyond the scope of this document, but an excellent article written by Lisa Slater Nicholls called "Visual FoxPro Coverage Profiler Add-Ins and Subclasses" is available from the MSDN Web site (http://msdn.microsoft.com/vfoxpro/technical/articles/applications.asp) This article explores this topic in great detail and includes

some sample add-ins. Also, the VFP Help topic on the Coverage Profiler (from the help contents, choose Using Visual FoxPro, Programmer's Guide, What's New in Visual FoxPro, Application Development and Developer Productivity) includes information on how to create add-ins or create your own Coverage Profiler application using the coverage engine or subclasses.

Just to whet your appetite, though, here's a couple of add-ins I find useful.

Coverage Snippet Analyzer

Markus Egger has written an incredibly useful add-in for the Coverage Profiler called the Coverage Snippet Analyzer. This add-in, which is available for download from his Web site (www.eps-software.com), provides the following capabilities:

- Procedures and methods headings are displayed in yellow (you can turn this off) so you can quickly see where one method ends and the next one starts.

- Next and previous procedure buttons quickly navigate from one procedure to the next.

- In addition to the number of hits, duration for the first hit, and average duration, the grid displays the total time taken for each line of code (hits multiplied by average duration).

- Code that's slow or called often is color-coded (you can turn these off independently). Slow code (average duration > 0.2 seconds) is shown in red, very slow code (average duration > 1 second) is bolded and shown in red, code that's either slow or executed frequently or both (total time > 0.2 seconds) is shown in blue, and code that's either very slow or executed very frequently or both (total time > 1 second) is bolded and shown in blue. The hit count for code that's executed more than 10 times is shown with a red background.

- You can display all methods or just a single method.

- You can display all code, only executed lines, or only unexecuted lines.

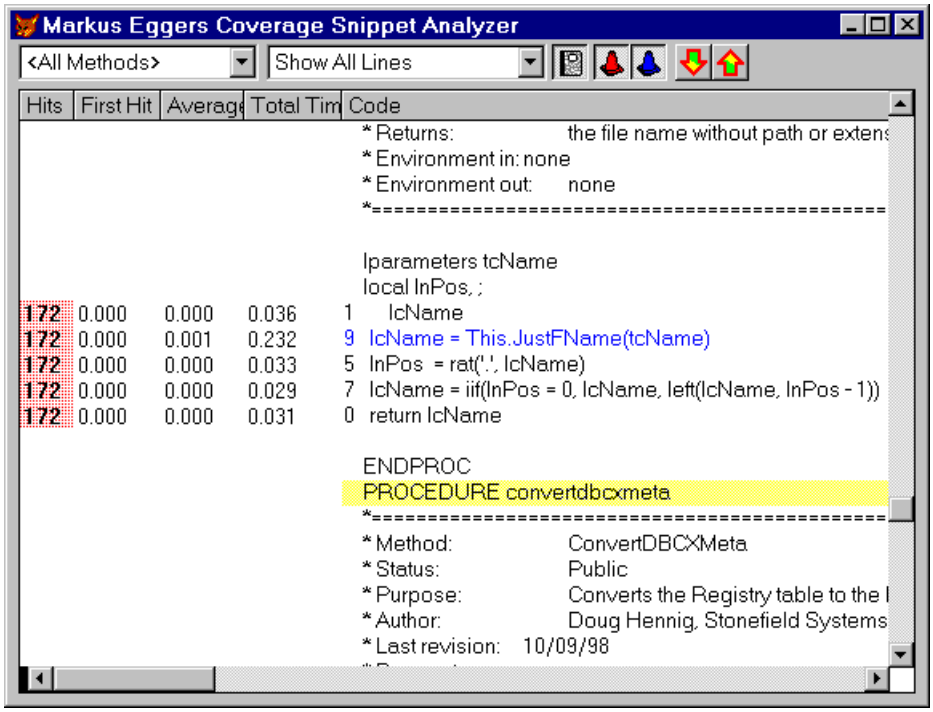


Figure 4. Markus Egger's Coverage Snippet Analyzer.

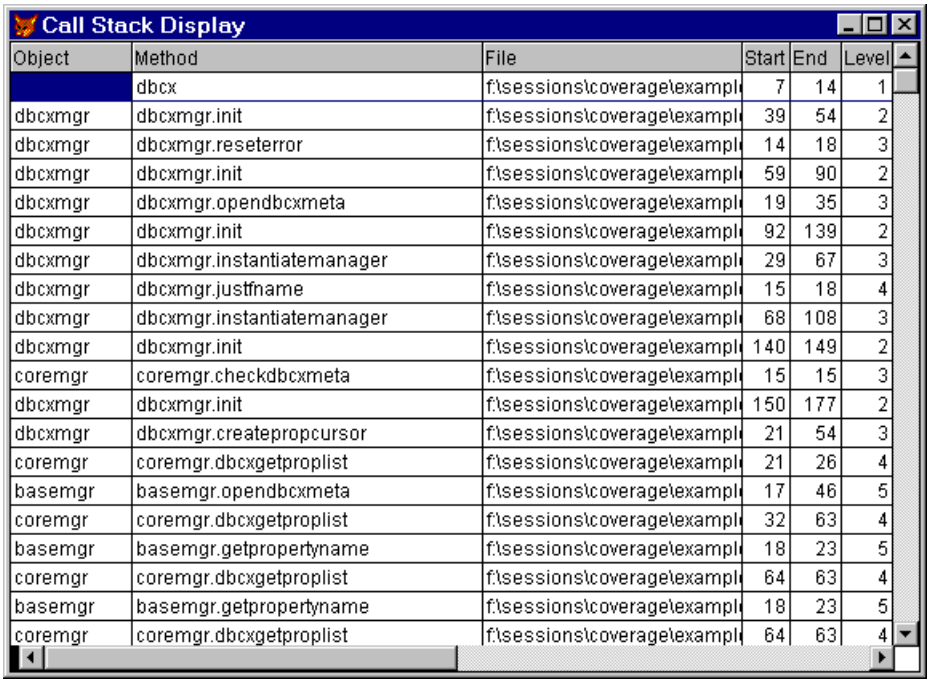
This add-in makes it much easier to filter out all the code in a log so you can focus on only those lines that need to be looked at.

To use this add-in, bring up the Coverage Profiler and switch (if necessary) to Profile mode. Select the class or program you'd like to view in the Coverage Snippet Analyzer, then click the Add-In button and run MAXCOV.APP.

Call Stack Display

In the January 1999 issue of FoxPro Advisor, Malcolm Rubel published a Coverage Profiler add-in that displays call stack information. Although a coverage log file contains call stack information, the Coverage Profiler doesn't do anything with this information. As a result, when you look at the code in the Coverage Profiler, it's hard to tell what code was called from where. Mac's add-in displays the code in the order it was executed, showing the starting and ending line numbers for the range of code in each method. The source code accompanying this session is a modified version of Mac's code.

The call stack display add-in consists of two files: CSTACK.PRG and CALLSTAK.SCX. CSTACK is the actual add-in; it adds a button to the Coverage Profiler that calls CALLSTAK.SCX to display the call stack information.



Object	Method	File	Start	End	Level
	dbcx	f:\sessions\coverage\exampl	7	14	1
dbcxmgr	dbcxmgr.init	f:\sessions\coverage\exampl	39	54	2
dbcxmgr	dbcxmgr.reseterror	f:\sessions\coverage\exampl	14	18	3
dbcxmgr	dbcxmgr.init	f:\sessions\coverage\exampl	59	90	2
dbcxmgr	dbcxmgr.opendbcxmeta	f:\sessions\coverage\exampl	19	35	3
dbcxmgr	dbcxmgr.init	f:\sessions\coverage\exampl	92	139	2
dbcxmgr	dbcxmgr.instantiatemanager	f:\sessions\coverage\exampl	29	67	3
dbcxmgr	dbcxmgr.justfname	f:\sessions\coverage\exampl	15	18	4
dbcxmgr	dbcxmgr.instantiatemanager	f:\sessions\coverage\exampl	68	108	3
dbcxmgr	dbcxmgr.init	f:\sessions\coverage\exampl	140	149	2
coremgr	coremgr.checkdbcxmeta	f:\sessions\coverage\exampl	15	15	3
dbcxmgr	dbcxmgr.init	f:\sessions\coverage\exampl	150	177	2
dbcxmgr	dbcxmgr.createpropcursor	f:\sessions\coverage\exampl	21	54	3
coremgr	coremgr.dbcxgetproplist	f:\sessions\coverage\exampl	21	26	4
basemgr	basemgr.opendbcxmeta	f:\sessions\coverage\exampl	17	46	5
coremgr	coremgr.dbcxgetproplist	f:\sessions\coverage\exampl	32	63	4
basemgr	basemgr.getpropertyname	f:\sessions\coverage\exampl	18	23	5
coremgr	coremgr.dbcxgetproplist	f:\sessions\coverage\exampl	64	63	4
basemgr	basemgr.getpropertyname	f:\sessions\coverage\exampl	18	23	5
coremgr	coremgr.dbcxgetproplist	f:\sessions\coverage\exampl	64	63	4

Figure 5. Call Stack Display.

Summary

Application testing and profiling are two very important yet different tasks, and the Coverage Profiler can help with both of them. Be sure to check out this tool and spend time determining how it can help you in your development efforts. Also, if you create subclasses or add-ins for the Coverage Profiler, share them with the community by posting them on the Universal Thread or Compuserve.

Acknowledgements

I would like to acknowledge the following people who directly or indirectly helped with the information in this document: Lisa Slater Nicholls, Markus Egger, and Malcolm Rubel.

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341
Fax: (306) 586-5080
Email: dhennig@stonefield.com
World Wide Web: www.stonefield.com

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP).

Bibliography

This isn't an exhaustive list, just some books and articles I've read on the topic of application testing.

- Code Complete, Steve McConnell, Microsoft Press.
- Writing Solid Code, Steve Maguire, Microsoft Press.
- "Best Practices: Development Checklist: Testing", Jefferey A. Donnici, FoxTalk, July 1998.
- "Testing Visual FoxPro Applications: Is Your Application Covered", Malcolm Rubel, presented at 2nd Annual Southern California Visual FoxPro Conference, August 1998.

Appendix: Coverage Logging and Coverage Profiler Changes in VFP 6 Service Pack 3

The Coverage Profiler has been enhanced in VFP 6 Service Pack 3 (SP3).

- SET COVERAGE now uses 6-digit precision for durations. This means all those lines of code that used to show 0.000 for the duration will now show something more meaningful, such as 0.000457.
- A new Find dialog searches classes, file names, and source code for text strings, making it much faster and easier to find exactly what you're looking for.
- Window states are now saved and restored. You can turn this off by setting the lSaveFormPositions property of a Cov_Engine subclass to .F. (it's .F. in Cov_Engine but .T. in Cov_Standard, the class actually used by the Coverage Profiler application). If you're interested in using this behavior in your own forms, check out the Cov_SavePosition class.
- Some additional keywords are now considered "uncoverable".

- Other minor enhancements: it handles log files with no extension; Cov_Standard has a cProjectFRX property that contains the name of the report file to use for project statistics, so this can be more easily changed; add-ins included in the Coverage Profiler APP can now be specified without a path.