

Instrumented Code is Better Code

Mark Williams

www.cheshamdb.com

mawilliams@cheshamdb.com

INTRODUCTION

Modern corporate systems are complex and generally involve multiple technologies, operating systems, hardware platforms, etc. Many vendors of the components which make up today's systems are including pervasive code instrumentation in the products they ship. Code instrumentation makes it easier to identify exactly where a performance issue lies. For example, think of the Extended SQL Trace capabilities in Oracle Database and how difficult it would be to identify performance issues in its absence.

Yet this practice of code instrumentation is often completely ignored in a corporate environment. In this context "corporate environment" means applications which are developed by in-house corporate IT staff or contractors and then typically deployed to internal users. Of course, this does not preclude an in-house developed application such as a public facing website as well. All of this is in contrast to, for example, an off the shelf application.

There are a variety of reasons why code is not instrumented. Perhaps lack of knowledge about what instrumentation offers and just how to instrument code leads some to not take advantage of it. However, "Too much overhead and code-bloat!" is an oft-heard justification. But, is it really overhead or code bloat? Even if it is not overhead or bloat, is it really necessary? Finally, if it is necessary, how is code instrumented and what can this instrumentation reveal? In this paper I explore the benefits of code instrumentation in corporate applications. While the concepts are generic in that they can be applied to any language or environment, the options explored and the implementations are, generally, specific to the .NET environment and the C# language.

WHAT IS BETTER CODE?

Just what is better code is, at the same time, both subjective and objective. It may seem strange that what I call "better code" can be both subjective and objective at the same time. Let me explain. For example, if a piece of code compiles correctly and executes correctly that would, in virtually all cases, be considered better code than code which did not compile or did not execute correctly. I think few would disagree with that notion.

However, once one moves beyond the utter basics of failure to compile or execute correctly, what constitutes better code is a matter of determining the criteria that are important in your environment and then evaluating code against that criteria. It is the act of determining the criteria that is subjective and the act of measurement against those criteria that is objective.

Here's a real-life example of at least one aspect of this process. If you or someone in your environment develops code in a "curly braces and semi-colons" language such as C, C#, Java or any other such language, then the placement of the curly braces may be a hotly contested item! After weeks of grueling and contentious staff meetings to discuss this issue it was determined that the braces should be positioned in the classic C-style. Here is the ubiquitous "Hello, world" program in a C# console application using the decided upon corporate style:

```
using System;

namespace HelloWorld {
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Hello, world.");
        }
    }
}
```

Listing 1: The "Hello, world" program in C# with classic C-style brace placement

For the sake of comparison, here is that same source code formatted using brace placement more akin to C++-style applications:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, world.");
        }
    }
}
```

Listing 2: The "Hello, world" program in C# with C++-style brace placement

If you have never had the pleasure of participating in such discussions, rest assured that the above is not a far-fetched, fabricated example! In fact, other code formatting issues such as "tabs vs. spaces", the amount of indenting for various items, and placement of keywords such as "else" in an "if" block are often included in these types of discussions.

The discussion to this point has concerned itself with code at design-time and these are important issues. Coding standards, which also includes naming standards, do make code

"better code" by providing consistent formatting, naming practices etc. which affords continuity from program to program in the source code base.

What about code at run-time? Is there anything you can do to make your code "better code" at run-time? Would code that conveys meaningful information about its run-time performance be considered "better code" in your environment? I believe it would and if you also believe that, then one of the best things you can do with your source code at design-time to make it "better code" at run-time is to instrument that code! The remainder of this paper discusses this activity.

WHAT IS CODE INSTRUMENTATION?

Code instrumentation is actually a simple concept and process despite a name which, I've been told, on occasion evokes strange convoluted images such as diving down to assembly language to implement some low-level action. On the contrary to such low-level activities, it is simply the inclusion of code (in the same language as the rest of the code) in an application that permits the application to emit performance information at run-time. That's really it. There's no descent into the lower echelons of assembly and pushing and popping registers, etc. In the world of Oracle the most prominent example of code instrumentation is likely the Extended SQL Trace capabilities (also known as the 10046 event) built into the Oracle kernel.

My definition of code instrumentation is: Code instrumentation is the strategic placement of host-language constructs (function or procedure calls) such that an application may emit performance information at run-time.

It is my hope that this definition will be further rounded-out and made fuller as the paper progresses.

ISN'T INSTRUMENTATION JUST LOGGING OR DEBUG CODE?

One question that may come to you during discussions of code instrumentation is, "Isn't instrumentation just a glorified form of logging or debugging?"

I make a distinction between "instrumentation" and "logging" or "debug" code. To me the key difference between instrumentation and logging or debug code is that instrumentation is targeted specifically at *performance* and, thus, has specific attributes which support that focus, whereas logging or debug code frequently will do things like log interesting events in an application's lifecycle or print the value of a specific variable.

The Windows Event Viewer is a good example of logging vs. instrumentation. The Event Viewer contains events that the designers of an application have deemed interesting enough to publish into the Event Viewer system for possible inspection. For example, the Oracle software contains code to publish logging information such as a service starting or stopping to the Event Viewer system. It would, however, be cumbersome to write and

extract the necessary information for performance reporting with a tool such as the Event Viewer. For example, it is possible for an event log to wrap around upon itself and overwrite older information. This is an undesirable trait when gathering performance information. True, it is possible to take measures such that this event is not likely to happen, but it may happen without special action to prevent it.

I classify debug code into two categories. There is code which has special symbols and information embedded in it by a compiler (or the equivalent tool for the language / environment of your choice) to enable the code to be executed in a debugging tool. An application binary which has such information in it is often declared to be a "debug build" of the application.

There is also code which a developer has written that emits application state during the application's execution. In this case there is generally a mechanism such as passing a command-line parameter to the application that indicates it should produce such information during execution. This type of code is often used to "announce" actions that the code is taking. As a simple example, code may check if the "debug flag" was passed to it upon start up, and, if so, during execution emit information that indicates which procedures are being called and the value of the parameters being passed to those procedures. Of course, code in the procedures may also, in turn, emit information about local variable values.

WHY IS CODE INSTRUMENTATION NEEDED?

Debug code which emits application state during execution is a close-cousin to instrumentation for performance. Notice that I said "for performance" in that sentence. Embedded debug code may do a fantastic job of providing a view into an application's internal state, but what does it do for a performance investigator? The application state is only part of the picture - to complete the picture you need one critical piece of information. That critical piece of information is... time.

Once you add the time component to debug code you are well on the way to creating instrumented code. But why are you creating instrumented code? Why is it needed? The answer is almost laughably simple: because now you can tell how long something takes! Can you imagine trying to troubleshoot a performance issue knowing only some of an application's internal state such as the order in which (perhaps some) events took place and the values of (perhaps only some) of the variables? Yet this is exactly what some people have to do when the instrumentation is missing from an application.

Tom Kyte has a nice summary of why code instrumentation is needed. Tom says, "It [performance] is many times an application issue and when the application is spread over 14 tiers of complexity, tracking down the bottleneck is grievously hard. If you just whip together an application and throw it out there without any thought to monitoring it over time, be prepared to have poor performance and no clue as to why or where."¹

¹ Instrumentation (<http://tkyte.blogspot.com/2005/06/instrumentation.html>)

One thing that some developers try to do is "cheat the system" by embedding instrumentation, but only enabling it in "debug" builds. This is typically done by using pre-processor information or macros that include code when certain symbols are defined (or if they are not defined). Here's a simple example of that using the "Hello, world" program from above:

```
using System;

namespace HelloWorld {
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Hello, world - ");

#if DEBUG
            Console.WriteLine("I'm a debug build!");
#else
            Console.WriteLine("I'm a release build!");
#endif
        }
    }
}
```

Listing 3: The "Hello, world" program with optional inclusion and exclusion of code

In this case if the `DEBUG` symbol is defined, which it would be during a debug build, then the code to emit the "I'm a debug build!" text to the console would be included. The key thing about this sort of activity is that the code to emit the "I'm a release build!" text would *not* be included. Of course in a real application the code would likely be something a bit more meaningful than that used in this trivial sample.

The problem with this activity is that you have no way of changing the behavior at run-time. "So what?" you may hear, "we can always just drop the debug build on the system and re-execute the process in question." But it is not always possible to just replace an executable with another executable. If the process is currently running as it may be on, say, a web server then the entire process would have to be stopped to replace the release build with a debug build. There are other issues such as geography to consider as well. It may not be a simple task to send a debug build of an application to another location and get it properly installed, etc.

Therefore, in order to make it simple to enable the instrumentation code, the code should "just be there" – i.e. it should be included in a release build of the application and *not made optional*. Tom echoes this same notion when he says, "Also, make this instrumentation part of the production code, don't leave it out! Why? Because, funny thing about production - you are not allowed to drop in "debug" code at the drop of a hat, but you are allowed to update a row in a configuration table, or in a configuration file! Your trace code, like Oracle's should always be there, just waiting to be enabled."²

² *Ibid.*

REVIEW OF ORACLE KERNEL INSTRUMENTATION

As mentioned earlier, Oracle's Extended SQL Trace capability is probably the most visible form of instrumentation in the Oracle world. While a complete treatment of this subject is likely not necessary, a review of this capability and the information it provides will provide both a context and a base upon which to build. If you are unfamiliar with Oracle's Extended SQL Trace or if you wish to get further details, the best resource available is *Optimizing Oracle Performance* (O'Reilly, 2003) by Cary Millsap with Jeff Holt.

When examining an Extended SQL Trace file, the characteristic of the trace file which elevates it above "simply debug code" is that there is time information contained in the output. In fact, there are four different kinds of time information in the file. As an example, here is a reformatted snippet of a trace file created on my local PC running Windows Vista 64-bit:

```
...
PARSING IN CURSOR #5 len=87 dep=0 uid=82 oct=3 lid=82 tim=592842132530
hv=3908967438 ad='7ff23a43e60' sqlid='3qu4zf7ngw70f'

select employee_id, last_name, first_name from employees order by
last_name, first_name

END OF STMT

PARSE #5: c=15600, e=100024, p=4, cr=146, cu=0, mis=1, r=0, dep=0, og=1,
plh=3447538987, tim=592842132530

EXEC #5: c=0, e=0, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, plh=3447538987,
tim=592842132530

WAIT #5: nam='SQL*Net message to client' ela= 3 driver id=1111838976
#bytes=1 p3=0 obj#=-1 tim=592842132884

WAIT #5: nam='db file sequential read' ela= 9756 file#=5 block#=83
blocks=1 obj#=70296 tim=592842142861

WAIT #5: nam='db file scattered read' ela= 849 file#=5 block#=84
blocks=5 obj#=70296 tim=592842143803

FETCH #5: c=0, e=0, p=6, cr=7, cu=0, mis=0, r=1, dep=0, og=1, plh=3447538987,
tim=592842132530

WAIT #5: nam='SQL*Net message from client' ela= 175
driver id=1111838976 #bytes=1 p3=0 obj#=70296 tim=592842144145

WAIT #5: nam='SQL*Net message to client' ela= 2 driver id=1111838976
#bytes=1 p3=0 obj#=70296 tim=592842144194
...
```

Listing 4: An extract of an 11.1.0.7 (64-bit) Extended SQL Trace on Windows 64-bit

As you can see in the text in Listing 4, I have added line-breaks and placed the time information in bold and underlined text to make it easier to read. This information comes from a trace file that was created by the HR sample user executing the query shown at the beginning of the listing.

Broadly speaking, the Oracle kernel breaks timing information into two categories: database calls and system calls. When examining a trace file as above, the database calls are denoted by PARSE, EXEC, and FETCH identifiers. The system calls are denoted by a WAIT identifier and a "nam=" token to specify the name of any particular system call (from the Oracle perspective). You can find additional information on the "wait events" in the Appendix of the Oracle Database Reference for your version and platform or by querying the V\$EVENT_NAME view.

As you may suspect, the manner in which Oracle acquires the timing values placed in trace files is operating system specific. In addition, the resolution of each value may vary from version to version (in particular from pre-9i to post-9i). Here's a brief description of the four types of timing information in the file in Listing 4 as they are encountered:

- tim - This value represents a system counter (or timer) expressed in microseconds (µs) or millionths of a second. This value is also known as a timestamp.
- c - This value represents the amount of CPU time used by the current database call in microseconds.
- e - This value represents the amount of elapsed time for a database call expressed in microseconds.
- ela - This value represents the amount of elapsed time expressed in microseconds for, what Oracle determines is, a system call.

Note that there are some unexpected timestamps in the above trace file. For example, the value "592842132530" repeats several times. I have also seen cases where the values appear to go back in time. I believe these issues are due to the situation discussed in Oracle MetaLink Note: 601528.1 (Times Reported For Waits in Trace File Are Too High).

INSTRUMENTATION IMPLEMENTATION

Notwithstanding the fixed interval markers represented by the "tim=" values in a trace file, there are two types of time information present: elapsed time and cpu time. In both cases system calls take place in order to gather the information necessary for these values. For an application which uses the .NET Framework, calls into the .NET Framework take place. These calls may subsequently call into the operating system proper. This discussion takes places in the context of an application that uses the .NET Framework.

In order for an application to determine the duration of an event or call, it must collect timing information immediately preceding the timed action and immediately following it.

Once this is done, it is a simple matter of subtraction of the first value from the second in order to determine the duration. The .NET Framework provides two entities which are often used for this purpose: the `DateTime` structure and the `Stopwatch` class. The `Stopwatch` class is specifically designed for timing purposes whereas the `DateTime` structure is a general purpose type used to work with dates and times.

No matter what item is used for timing, the general pattern is as follows:

```
t0 = get_current_timing_value();  
  
perform_action();  
  
t1 = get_current_timing_value();  
  
duration = t1 - t0;
```

Listing 5: pseudo-code for how a duration is calculated

Of course, if no manipulation occurs, the value of the "duration" variable will be expressed in the native units used by the timing function. In the case of the `DateTime` structure, some manipulation typically occurs. For example, the above pseudo-code implemented using `DateTime` would look like the following when calculating the duration in terms of seconds:

```
DateTime timeStart;  
DateTime timeEnd;  
double    duration;  
  
timeStart = DateTime.Now;  
  
perform_action();  
  
timeEnd = DateTime.Now;  
  
duration = timeEnd.Subtract(timeStart).TotalSeconds;
```

Listing 6: Calculating a duration using `DateTime`

One implementation of this same pattern using the `Stopwatch` class looks as follows:

```
long duration;  
  
System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();  
  
sw.Start();  
  
perform_action();  
  
sw.Stop();  
  
duration = sw.ElapsedMilliseconds;
```

Listing 7: Calculating a duration using `Stopwatch`

As the above two examples show, there are different methods for calculating durations and each has its own characteristics. As you may have noticed, the first sample (Listing 6) calculates the duration in terms of seconds whereas the second sample (Listing 7) calculates the same duration in terms of milliseconds. Understanding the characteristics of a timer implementation is an important aspect to your overall instrumentation scheme.

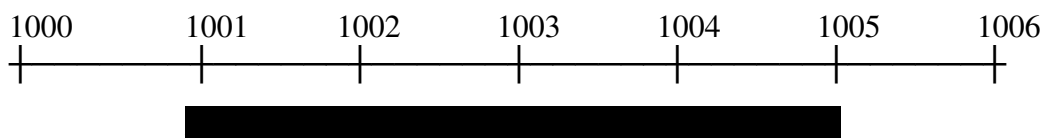
TIMER FREQUENCY, RESOLUTION, AND ACCOUNTING ERRORS

When software measures its own performance it gets a discrete value before an action, performs the action, gets another discrete value after the action, and finally calculates the difference between the after and before values. The source of these discrete values is generically referred to as a timer and there are two inversely related characteristics of a timer which are vital to understand when it comes to instrumentation: frequency and resolution. These two characteristics contribute, in turn, to something known as "accounting errors".

If you are familiar with the world of music, you may have heard of a device known as a metronome. This device makes clicking sounds over a fixed interval and the interval can be adjusted to a specific number of clicks per minute. So, for example, if you set a metronome to a value of 60, it will emit 1 click every second (that is, 60 clicks per minute). In much the same way as a metronome keeps constant time, a computer timer does the same over an interval. These kinds of timers are known as interval timers and the number of clicks (or pulses, or ticks) over a specific time period defines the frequency for that timer.

A timer's resolution expresses the same value as the frequency - it just does so as a reciprocal! What this means is that, taking the metronome sample from above, if a timer emits 60 pulses per minute, the resolution of that timer is 1 second. In other words, the frequency tells you how many ticks there will be over a given interval whereas the resolution of the timer tells you how much time will elapse between ticks.

The fact that a digital timer cannot have infinite resolution (i.e. there must be *some* amount of time between ticks) means that there is also the possibility for accounting errors to show up during measurement. The following diagram illustrates how this happens.



In this diagram a timer has emitted a total of 7 ticks beginning with tick #1000 and ending with tick #1006. During this time an application called a function to get the current tick value just before the timer ticked over to tick #1001; therefore, the value used by the application will be 1000. At the end of the process duration being measured, the application again gets the current tick value. At this time the tick has just changed over to


```
// calculate the resolution in nanoseconds
// 1 billion nanoseconds = 1 second
// so, here we have 1 billion / ticks per second
Int64 nanosecPerTick = (1000000000) / frequency;

// display the resolution
Console.WriteLine("  Timer is accurate within {0} nanoseconds",
                  nanosecPerTick);
}
}
```

Listing 8: C# sample code to determine timer information

Executing this sample on my home PC results in the following output:

```
Operations timed using the system's high-resolution performance
counter.
  Timer frequency in ticks per second = 14318180
  Timer is accurate within 69 nanoseconds
```

As you can see, the timer on my system operates at a frequency of 14,318,180 ticks per second and this equates to a resolution of 69 nanoseconds. Recall from earlier that the resolution indicates how much time passes between ticks. This means the timer on my system generates a tick every 69 nanoseconds. Of course, the more fine the resolution the more frequently a tick or pulse is generated. However, having too high of a resolution can be detrimental to overall system performance. Operating system designers must take this into account when designing the operating system.

THE COST OF SELF-MEASUREMENT

As I mentioned near the beginning of this paper the "extra cost" or "performance penalty" of including instrumentation in code is often used as a reason for not doing so. There is no doubt that instrumented code will have extra instructions to execute when that instrumented section of code is invoked. But, how much does this slow the "real" code down? In a sense you almost cannot answer this question. Why? Well, because if code is not instrumented, you have no way of knowing with any degree of precision how long it takes to execute!

The code in Listing 9 is designed to allow an approximation of how much cost (in terms of time) the inclusion of code instrumentation via the Stopwatch class has. As you can see, it creates a "primary" stopwatch, executes 1,000,000 iterations of starting and stopping a "secondary" stopwatch and then gets the average time in milliseconds.

```
// This sample is designed to approximate the cost of
// invoking a pair of Stopwatch start and stop methods.

using System;
using System.Diagnostics;
```

```
namespace StopwatchCost {
    class Program {
        static void Main(string[] args) {
            // the stop watches for timing
            Stopwatch sw1 = new Stopwatch();
            Stopwatch sw2 = new Stopwatch();

            // start the "primary" stopwatch
            sw1.Start();

            // invoke a pair of start/stop methods
            // for 1,000,000 iterations.
            // of course this includes the for loop
            for (int i = 0; i < 1000001; i++) {
                sw2.Start();
                sw2.Stop();
            }

            // stop the "primary" stopwatch
            sw1.Stop();

            // display the average duration in milliseconds
            Console.WriteLine("Average duration: {0} ms",
                (double)sw1.ElapsedMilliseconds/1000000);
        }
    }
}
```

Listing 9: C# sample code to approximate the "cost" of a stop watch

On my PC several executions of this sample result in an average time of 0.00124 ms. I would encourage you to perform several such experiments on your system to develop a profile of the cost of instrumenting your code. I believe you will find it to be much less than expected and the benefit of having timing information outweighs a slight performance degradation.

By judiciously instrumenting your applications you have the ability to pinpoint performance issues at run-time. As Cary Millsap says:

I'm suspicious about most so-called best practices... There is one "best practice," though, that I believe in deeply: "You should always measure your application's performance and target your optimization efforts at places where your code will benefit from it most."³

If you do not instrument your code, how will you be able to measure your application's performance and target your efforts correctly?

CPU CONSUMPTION AND DIVISION OF LABOR

³ For Developers: Making Friends with the Oracle Database (http://method-r.com/downloads/cat_view/38-papers-and-articles)

As your application is running (that is, it has not exited) it may consume CPU time. When the application (well, really a thread in Windows and a process in other operating systems) is in what is known as "user mode" or "kernel mode" CPU is being consumed. If the thread (or process) is in another mode such as ready-to-run (i.e. on the run queue) or is asleep, it may not be using the CPU.

The Process class in the .NET Framework allows you to determine how much CPU time has been used by a process or thread. You can further divide this CPU consumption into CPU used in "user mode" or CPU used in "kernel mode". In the Process class, the read-only property UserProcessorTime reports the amount of CPU time used in "user mode". Likewise, the read-only PrivilegedProcessorTime property reports the amount of time in "kernel mode". If you do not wish to differentiate between the two, the read-only TotalProcessorTime property may be used.

Fortunately the Process class provides a static method (GetCurrentProcess) which allows you to, naturally, get an instance of the Process class for the current process. You can then access the appropriate processor time properties and calculate the division of labor between "user mode" and "kernel mode" processor time. This is illustrated in Listing 10 which follows. This listing is a bit more complicated and lengthy than previous listings.

```
// this sample illustrates getting the processor
// time values from the Process class

using System;
using System.Diagnostics;
using System.IO;

namespace CPUTime {
    class Program {
        static void Main(string[] args) {
            // used in loop below
            int dummy = 0;
            long square = 0;
            string fname = "test.txt";

            // holds the start processor time values
            TimeSpan tsTotalStart = new TimeSpan();
            TimeSpan tsPrivilegedStart = new TimeSpan();
            TimeSpan tsUserStart = new TimeSpan();

            // holds the end processor time values
            TimeSpan tsTotalEnd = new TimeSpan();
            TimeSpan tsPrivilegedEnd = new TimeSpan();
            TimeSpan tsUserEnd = new TimeSpan();

            // get the current process
            Process p = Process.GetCurrentProcess();

            // get current processor values (start)
            tsTotalStart = p.TotalProcessorTime;
```

```
tsPrivilegedStart = p.PrivilegedProcessorTime;
tsUserStart = p.UserProcessorTime;

// perform cpu and kernel operations in loop
for (int i = 0; i < 1000; i++) {
    if (!File.Exists(fname)) {
        using (StreamWriter sw = File.CreateText(fname)) {
            sw.WriteLine("Hello, world!");
        }
    }

    File.Delete(fname);

    square = dummy * dummy;
}

// get current processor values (end)
tsTotalEnd = p.TotalProcessorTime;
tsPrivilegedEnd = p.PrivilegedProcessorTime;
tsUserEnd = p.UserProcessorTime;

// display results
Console.WriteLine("        Total Processor Time: {0} ms",
    (tsTotalEnd - tsTotalStart).Milliseconds);

Console.WriteLine("Privileged Processor Time: {0} ms",
    (tsPrivilegedEnd - tsPrivilegedStart).Milliseconds);

Console.WriteLine("        User Processor Time: {0} ms",
    (tsUserEnd - tsUserStart).Milliseconds);
}
}
```

Listing 10: C# sample code to get various CPU consumption values

A sample run of the above code on my system shows the following results and, as you can see, the total time is simply the sum of the privileged time and the user time:

```
Total Processor Time: 216 ms
Privileged Processor Time: 76 ms
User Processor Time: 140 ms
```

CONCLUSION

At the beginning of this paper I stated that many developers feel that code instrumentation introduces too much overhead and code-bloat into their applications. However, by using a couple of classes provided by the .NET Framework, it is simple to add constructs to your code that allow it to report on performance at run-time. I believe that this is such a critical functionality of applications that it should be considered *de rigueur* in corporate applications. Without it you have no other way of determining with precision or confidence where in your application a performance issue may be occurring.

One of the phrases I have heard from Cary Millsap on several occasions that I really like is: Why guess when you can know?⁴

If for no other reason than that phrase, instrumented code is better code.

This paper should be considered a starting point for implementing code instrumentation in your environment. It illustrates the basics you need to use the Stopwatch and Process classes to add the same capabilities to your applications as exist in the Oracle Extended SQL Trace capabilities. You can, of course, create your own classes and assemblies using these classes to separate the instrumentation implementation into reusable modules for your environment. I strongly suggest adding the ability of your applications to enable Oracle's Extended SQL Trace via a command-line switch or other suitable method. It is a powerful combination when you can combine Oracle's instrumentation and your own application's instrumentation for troubleshooting performance issues.

⁴ *Ibid* for one example.