Customising DBMS_APPLICATION_INFO

Gerry Miller
Oracle DBA
gerry@millerandbowman.com

November 2, 2005

1 Introduction

DBMS_APPLICATION_INFO is one of the most useful yet least used of the Oracle supplied PL/SQL packages. It is designed to allow application modules to be registered with the database in order for developers and DBAs to monitor execution and performance. When used in its original state the package can be quite valuable, but it is when its intended functionality is exploited by creating a custom version that it becomes a powerful debugging, tracing and auditing tool. The solutions offered in this paper describe a straightforward method of implementation that is reliable, robust and dynamic. In addition to outlining the basic functions of the DBMS_APPLICATION_INFO package, the creation of a custom version is discussed, highlighting possible problems and solving them. As a result, the package is demonstrated to be a valuable tool that can be utilized at the most basic or complex level, depending on the particular need at the time. The use of code in the custom version allows not only the conventional functionality to be enhanced but also new functionality to be introduced. Performance implications are discussed and comparisons made between the supplied package and increasingly customised versions.

2 The DBMS_APPLICATION_INFO Package

2.1 Package Description

The DBMS_APPLICATION_INFO package has six procedures:

```
SET_MODULE
Parameters:    Module_name IN VARCHAR2
               Action_name IN VARCHAR2
```

This module is probably the most used in the package and is the most powerful when creating a custom version. It accepts two parameters, MODULE_NAME and ACTION_NAME. The SYS-owned package uses these values to update the MODULE and ACTION columns in the V$SESSION and V$SQLAREA views.

```
SET_ACTION
Parameter:     Action_name IN VARCHAR2
```

This module accepts only the ACTION_NAME parameter, which is then passed on to the V$SESSION and V$SQLAREA packages. It is commonly used in the middle of a module when the MODULE_NAME has already been registered.

```
READ_MODULE
Parameters:    Module_name OUT VARCHAR2
               Action_name OUT VARCHAR2
```

The compliment of SET_MODULE, this procedure populates two OUT parameters with the current MODULE_NAME and ACTION_NAME for the session

```
SET_CLIENT_INFO
Parameter:     Client_info IN VARCHAR2
```

This procedure is similar to SET_MODULE and allows additional information about the client application to be registered, although only the V$SESSION view is updated with its value.

```
READ_CLIENT_INFO
Parameter:     Client_info OUT VARCHAR2
```

The compliment of SET_CLIENT_INFO, this procedure populates an OUT parameter with the current CLIENT_INFO value.

```
SET_SESSION_LONGOPS
Parameters:    Rindex      IN/OUT   BINARY_INTEGER
               Slno        IN/OUT   BINARY_INTEGER
               Op_Name     IN       VARCHAR2 ,
               Target      IN       BINARY_INTEGER ,
               Context     IN       BINARY_INTEGER,
               Sofar       IN       NUMBER,
               Totalwork   IN       NUMBER,
               Target_desc IN       VARCHAR2,
               Units       IN       VARCHAR2)
```

This procedure populates and updates the V$SESSION_LONGOPS view.

It also has a constant,set_session_longops_nohint, which has a value of -1 (minus one) and is used in the initial call to SET_SESSION_LONGOPS.

This paper focuses only on the SET_MODULE and SET_ACTION modules.

2.2 Intended Usage

When database performance problems arise it is common practice to use the Oracle dynamic performance views, trace files or both, to identify the offending sessions and to then drill down to the source of the problem. At this point it is useful to know from which application or system program module the SQL is being run, and this information can be provided by DBMS_APPLICATION_INFO. The intended usage of the package, as documented in the Oracle PL/SQL Packages and Types Reference [1], is to instrument application module usage and operation. This is effected by inserting the values supplied by the parameters of the SET_MODULE, SET_ACTION into the V$SESSION and V$SQLAREA views and, if tracing is enabled, writing them to the trace file, and is referred to in the documentation as "registering" the application with the database. Calls to the SET_CLIENT_INFO Procedure Are also recorded in the V$SESSION view, but not in the trace file. However, an additional use is intended

```
"If you want to gather your own statistics based on module, you can
 implement a wrapper around this package by writing a version of
 this package in another schema that first gathers statistics and
 then calls the SYS version of the package. The public synonym for
 DBMS_APPLICATION_INFO can then be changed to point to the DBA's
 version of the package."
```

This is an important paragraph in the Oracle documentation as it allays any fears that Oracle may not support such customisation.

2.3 Custom Package

As previously stated, the intended use of DBMS_APPLICATION_INFO is for developers/DBAs to create their own version of it in order to enhance its capabilities, the custom version calling the original. By creating a custom version of DBMS_APPLICATION_INFO code can be added to provide functionality that would not necessarily be considered application-specific. The first consideration when creating a custom version of DBMS_APPLICATION_INFO is that it must have at least the same structure as the Oracle supplied version, that is, six procedures with the same parameters as the original, as well as the constant, set_session_longops_nohint. Each procedure should at some point call its counterpart in the original in order to update the V$ views and trace files. The reason for this is that, even if it is not the intention to add extra functionality to a procedure in the custom package, the application must be able to call that procedure without problems. The basic package specification is therefore constructed thus:

```
CREATE OR REPLACE PACKAGE dbms_application_info
AS
PROCEDURE READ_CLIENT_INFO(CLIENT_INFO OUT VARCHAR2);
PROCEDURE READ_MODULE(MODULE_NAME OUT VARCHAR2,
ACTION_NAME OUT VARCHAR2);
PROCEDURE SET_ACTION(ACTION_NAME IN VARCHAR2);
PROCEDURE SET_CLIENT_INFO(CLIENT_INFO IN VARCHAR2);
PROCEDURE SET_MODULE(MODULE_NAME IN VARCHAR2,
ACTION_NAME IN VARCHAR2);
PROCEDURE SET_SESSION_LONGOPS(
   RINDEX IN OUT BINARY_INTEGER,
   SLNO   IN OUT BINARY_INTEGER,
   OP_NAME IN VARCHAR2,
   TARGET IN   BINARY_INTEGER,
   CONTEXT  IN BINARY_INTEGER,
   SOFAR  IN   NUMBER,
   TOTALWORK  IN NUMBER,
   TARGET_DESC  IN VARCHAR2,
   UNITS  IN VARCHAR2);
   set_session_longops_nohint constant pls_integer := -1;
END DBMS_APPLICATION_INFO;
/
```

with each module having a similar structure to:

```
PROCEDURE SET_MODULE(MODULE_NAME IN VARCHAR2,
ACTION_NAME IN VARCHAR2)
IS
BEGIN
    SYS.DBMS_APPLICATION_INFO.SET_MODULE(MODULE_NAME,ACTION_NAME);
END;
```

This Basic custom package can then be enhanced to introduce additional functionality.

3 Enhancing Existing Functionality

3.1 Session Monitoring

In its most elementary form, as each call to the SET_MODULE procedure is executed, the MODULE and ACTION values are updated in the dynamic performance views, which is useful information when monitoring session activity. This module can be called at various levels, and is commonly called when a program first begins execution; examples of this can be seen with programs like TOAD, SQL*Plus and Oracle Forms. However, calls can be made to SET_MODULE at any stage of a program, and if made in each program module - for example, each procedure or function in a PL/SQL package - then program flow can be monitored. The Oracle documentation suggests that:

```
"Your applications should set the name of the module and name of the action
 automatically each time a user enters that module".
```

Unfortunately, this can result in misinformation being provided. For example, consider a package, called DEMO_PKG, which has two procedures, A and B, and one function, C. The process flow of the package is as follows:

```
Procedure A calls Function C
    Function C completes and returns to Procedure A
Procedure A calls Procedure B
```

```
        Procedure B calls Function C
            Function C completes and returns to Procedure B
        Procedure B completes and returns to Procedure A
```

A call to DBMS_APPLICATION_INFO.SET_MODULE is inserted at the beginning of each procedure, for example:

```
    DBMS_APPLICATION_INFO.SET_MODULE('Procedure A',NULL).
```

Consider further a scenario wherein, on return from Function C, Procedure B executes a horrific query, taking forever to run and tying up resources. The session identifier (SID) of the offending session is identified and the following query is executed:

```
SELECT module, action FROM V$SESSION WHERE sid=<sid>;
```

Resulting in:

```
MODULE            ACTION
-------------    ----------
Function C
```

This, of course, is incorrect. The call at the beginning of each PL/SQL module registers with the database only when the module is entered, but not when control is returned to it from a called module. As Function C was the last module registered with the database, the DBA will look for the cause of the problem there, completely on the wrong track. It is therefore evident that it is necessary not only to register each module at its entry point, but also at its exit point. For example:

```
BEGIN
DBMS_APPLICATION_INFO.SET_MODULE('Function C','Enter')
.........
DBMS_APPLICATION_INFO.SET_MODULE('Function C','Exit')
END;
```

The query on V$SESSION now returns:

```
MODULE          ACTION
------------    ----------
Function C      Exit
```

It is now known that control has returned from Function C, but which procedure it has returned to, A or B, is not evident. In order to be able to keep track of previous modules their names must be recorded in a stack and code added to modify the calls to the Supplied package. Rather than include such code in the application module, it can be added via a custom version of DBMS_APPLICATION_INFO. In this example, variables are added to the package definition:

```
    TYPE v_module IS table of VARCHAR2(48);
    g_module  v_module; --stack
    g_level INTEGER;  --stack pointer
    g_action VARCHAR2(32);
```

The g_module variable is a stack used to store the names of the previous modules and is initialised in the main body of the DBMS_APPLICATION_INFO with the name of the current module.

```
BEGIN
    g_level := 1;
    SYS.DBMS_APPLICATION_INFO.READ_MODULE
         (g_module(g_level),g_action);
END dbms_application_info;
```

The SET_MODULE procedure is then modified thus:

```
PROCEDURE SET_MODULE(
    MODULE_NAME in VARCHAR2,
    ACTION_NAME IN VARCHAR2)
IS
BEGIN
    IF action_name = 'Enter' THEN
         g_level := g_level+1;
        IF g_level > g_module.LAST THEN
            g_module.extend;
        END IF;
        g_module(g_level) := module_name;
        SYS.DBMS_APPLICATION_INFO.SET_MODULE
            (MODULE_NAME,'Called by '||g_module(g_level-1));
    ELSIF action_name = 'Exit' THEN
         g_level := g_level-1;
        SYS.DBMS_APPLICATION_INFO.SET_MODULE
            (g_module(g_level),'Returned from '||module_name);
     ELSE
         SYS.DBMS_APPLICATION_INFO.SET_MODULE
            (MODULE_NAME,ACTION_NAME);
    END IF;
END;
```

In any subsequent calls to SET_MODULE or SET_ACTION, if the value of action name is neither 'Enter' nor 'Exit' then the call is simply passed on to the Supplied package. However, if the value of action name is 'Enter' then the module name is pushed onto the stack. When the call is passed on to the Supplied package, the origin of the call is substituted for the original action, e.g. 'Called by Procedure B'. If the value of action name is 'Exit' then the module name is popped off the stack and the parameter values that are passed on are modified to be more informative: the module name is changed to the name of the module that is being resumed, while the action name shows from where control is being passed.

Now the query from V$SESSION returns:

```
MODULE           ACTION
-------------    ----------
Procedure B      Returned from Function C
```

The information returned is now accurate and program flow easier to follow.

There are still some shortcomings in this version. For example, if Function C is called from a particular procedure more than once then it is not obvious from which module the call has been made. However, notwithstanding the potential overheads involved, this could also be handled in the custom SET_MODULE procedure.

3.2 Session Tracing

If tracing is enabled then as each call to SET_MODULE or SET_ACTION is made, the module and action values are written to the trace file. Pre-10g this takes the format:

```
APPNAME mod='<module name>' mh=0 act=<action name>'' ah=0
```

while the introduction of 10g saw a change to the output format:

```
*** ACTION NAME:(<action name>) 2005-08-06 06:56:53.718
*** MODULE NAME:(<module name>) 2005-08-06 06:56:53.718
```

Although this change in format seems rather verbose for a trace file, the addition of the date and time stamp is undoubtedly an advantage.

As the values are written to the trace file in chronological order, the extra code needed to keep track of the program flow is now redundant and consequently so is the previously described customised package. However, in addition to the default output to tracefiles, it is possible to print custom messages to not only the trace file but also to the alert log, if desired, using the KSDWRT procedure in the DBMS_SYSTEM package. This Procedure accepts the parameters:

```
DEST: The destination of the message;
      1= trace
      2=  alert log
      3=  both
TST:  The text of the message
```

This has many advantages over the default messages provided by DBMS_APPLICATION_INFO, not the least of which being that even if tracing is not enabled, a trace file is created and the message written to it. This can be advantageous in a situation where it is necessary to monitor a long-running job to identify how long each module is taking without having to worry about filling up the trace filesystem or about the overhead of tracing that some perceive to be an issue. By adding calls to KSDWRT that include the module name and a timestamp in the custom package, a detailed account of the program flow can be obtained. If the updates to the V$ views are not wanted then the call to the original package can be omitted. An example of such a case would be an overnight batch job that would not be monitored during runtime. If other users call the custom package then a condition can be added to restrict the calls to KSDWRT, in order to prevent a trace file being created every time a call is made to DBMS_APPLICATION_INFO:

```
PROCEDURE set_module(module_name in VARCHAR2,action_name IN VARCHAR2)
IS
BEGIN
```

```
        IF  v_trace_flag = 'Y' THEN
            IF action_name = 'Enter' THEN
                g_level := g_level+1;
                g_module(g_level) := module_name;
                SYS.DBMS_SYSTEM.KSDWRT
                      (1,Module_name||' - '||
                    to_char(sysdate,'DD-MON-YYYY HH24:MI:SS'));
            ELSIF action_name = 'Exit' THEN
                g_level := g_level-1;
                SYS.DBMS_SYSTEM.KSDWRT
                      (1,g_module(g_level)||' - '||
                    to_char(sysdate,'DD-MON-YYYY HH24:MI:SS'));
            ELSE
                SYS.DBMS_SYSTEM.KSDWRT
                    (1,module_name||' - '||action_name);
            END IF;
        ELSE
            SYS.DBMS_APPLICATION_INFO.SET_MODULE(module_name,action_name);
         END IF;
    END;
```

If the v_trace_flag value were set then this module would result in trace file output similar to:

```
    Procedure A - 02-OCT-2005 08:12:05
    Function  C - 02-OCT-2005 08:13:15
    Procedure A - 02-OCT-2005 08:13:45
    Procedure B - 02-OCT-2005 08:14:09
    Function  C - 02-OCT-2005 08:15:19
    Procedure B - 02-OCT-2005 08:17:12
    Procedure A - 02-OCT-2005 08:19:01
```

Depending on how much overhead can be tolerated, code can be added to provide whatever output is desired. For example, simply adding "rpad(' ',g_level*3)" to the module name when writing to the trace file indents each line according to the level of the nested procedures, resulting in the much more readable:

```
    Procedure A - 02-OCT-2005 08:12:05
        Function C - 02-OCT-2005 08:13:15
    Procedure A - 02-OCT-2005 08:13:45
        Procedure B - 02-OCT-2005 08:14:09
            Function C - 02-OCT-2005 08:15:19
        Procedure B - 02-OCT-2005 08:17:12
    Procedure A - 02-OCT-2005 08:19:01
```

Alternatively, if the number of calls to DBMS_APPLICATION_INFO is expected to be significant then formatting the output in comma-separated format will facilitate its loading into a database table, making it easier to collate:

```
    Procedure A,Enter,02-OCT-2005 08:12:05
    Function  C,Called from Procedure A,02-OCT-2005 08:13:15
```

```
Procedure A,Returned from Function C,02-OCT-2005 08:13:45
Procedure B,Called from Procedure A,02-OCT-2005 08:14:09
Function  C,Called from Procedure B,02-OCT-2005 08:15:19
Procedure B,Returned from Function C,02-OCT-2005 08:17:12
Procedure A,Returned from Procedure B,02-OCT-2005 08:19:01
```

This output can be loaded into a table with the format:

Column_name	Datatype
Module	VARCHAR2(48)
Action	VARCHAR2(32)
Start_Time	DATE

From where queries can quickly be executed to quickly identify any long-running modules.

If execution time is not an issue then code can be added to process as well as format the trace output. For example, instead of merely writing a timestamp to the tracefile, the elapsed time of each module could be calculated and written out. Loaded into a database table, this data would then be easily queried to identify long running modules.

4 Additional Functionality

As well as improving the the conventional functionality of the DBMS_APPLICATION_INFO package, customisation can be used for a number of other tasks.

4.1 Executing DML

INSERT, UPDATE and DELETE statements can be incorporated in the custom package if desired, for example, to log application information in the database. Apart from being more resource intensive than writing information to trace files, there are other issues that must be considered. If there are no commit statements issued in the application that is calling the custom package and it crashes or performs a rollback operation, then all DML commands issued during that transaction are lost. In addition, any query that includes a call to a function that is calling the custom package will result in an error. For example:

```
SELECT demo_pkg.c FROM dual;

ORA-14551: cannot perform a DML operation inside a query
```

In Oracle Server release 9.2 and above both of these problems can be solved by adding a PRAGMA AUTONOMOUS_TRANSACTION in the declaration section and a COMMIT statement at the end of the SET_MODULE procedure:

```
PROCEDURE set_module(module_name in VARCHAR2,action_name IN VARCHAR2)
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
```

```
        INSERT INTO some_table values (1);
        commit;
        SYS.DBMS_APPLICATION_INFO.SET_MODULE(module_name,action_name);
    END;
```

In release prior to 9.2 the solution is more convoluted. If the session invoking the new DBMS_APPLICATION_INFO package is running a distributed transaction it will fail with the error:

```
ORA-00164 - autonomous transaction disallowed within distributed transactions
```

A distributed transaction includes one or more statements that, individually or as a group, update data on two or more distinct nodes of a distributed database. There are a number of solutions to this problem, including using the DBMS_PIPE package or the Oracle Advanced Queueing option to send the session module and action values to another session.

4.2 Conditional Tracing

Customisation can be used to enable tracing when a particular application module is called. The most obvious way to do this would be simply:

```
PROCEDURE set_module(module_name in VARCHAR2,action_name IN VARCHAR2)
IS
BEGIN
    IF module_name = 'Procedure B' THEN
         execute immediate 'alter session set events
            ''10046 trace name context forever, level 12''';
END IF;
    SYS.DBMS_APPLICATION_INFO.SET_MODULE(module_name,action_name);
END;
```

Oracle version 10g saw the introduction of the DBMS_MONITOR package, which with one command can enable tracing based not only on module but on a combination of other criteria including action, client_id and service_name. However, DBMS_APPLICATION_INFO can provide a level of granularity that DBMS_MONITOR cannot. For example, tracing can be enabled on every module except a specific one:

```
IF module_name != 'Procedure B' THEN....
```

The author did exactly this when activating tracing for all modules during a project, and encountering a bug that was purported to only occur when a specific application module had tracing enabled. The offending module was excluded from tracing and the project continued. Furthermore, retrieving information from, say, the V$SESSION view, can allow even more selective operations. For example, it is quite common for a number of database users to have the same database user ID, but be identifiable by their operating system user ID, which is recorded in V$SESSION.OSUSER. This information can be used to restrict tracing - or whatever else one may want to do - to a particular user. The operating system user id can be retrieved from V$SESSION in the main body of the DBMS_APPLICATION_INFO package, and then used to determine a course of action:

```
PROCEDURE set_module(module_name in VARCHAR2,action_name IN VARCHAR2)
IS
BEGIN
    IF module_name = 'Procedure B' THEN
        IF g_osuser = 'fred' THEN
            execute immediate 'alter session set events
                ''10046 trace name context forever, level 12''';
        END IF;
    END IF;
    SYS.DBMS_APPLICATION_INFO.SET_MODULE(module_name,action_name);
END;
```

Selective operations such as this can easily be performed using information retrieved from both system and application tables.

Another feature of the DBMS_MONITOR package is that if tracing is enabled for a particular module, the module is identified by a call to DBMS_APPLICATION_INFO, and if that module calls another, then tracing is turned of during the execution of the subordinate module. This may be the desired behaviour, but if the requirement is to trace all activity from the start until the end of a module, then the code would have to be first examined to identify which modules will be called - and which modules they in turn will call - and tracing enabled for these modules also. Using DBMS_APPLICATION_INFO, one can enable tracing at the start of a module and disable it at the end, ensuring that all activity is traced, including that in sub-modules. Alternatively, the logic in the custom package can be modified to provide similar behaviour to DBMS_MONITOR.

4.3 Restricting Execution

It is sometimes necessary to restrict the execution of a particular application module to one session in order to avoid contention. A method of doing this is demonstrated by Padhi [2], in which the author outlines how to enforce this restriction by checking V$SESSION.MODULE for versions of the current module. If any exist then a message is displayed to that effect and the program terminates, otherwise SET_MODULE is used to register the module and the program continues. The code used to enforce this is contained in the application module, which must be considered the correct approach if the module is designed to always be restricted in this way. However, if such a restriction is required on a temporary basis, for example, if it is necessary to provide a proof of concept, then the logic can be included in the custom DBMS_APPLICATION_INFO package as follows.

```
PROCEDURE set_module(module_name IN VARCHAR2,action_name IN VARCHAR2)
IS
l_lockhandle VARCHAR2(128);
l_lockstatus NUMBER;
BEGIN
   IF module_name = 'Procedure B' THEN
      dbms_lock.allocate_unique(module_name,l_lockhandle);
      IF action_name = 'Enter' THEN
         l_lockstatus := dbms_lock.request(l_lockhandle,dbms_lock.x_mode,1);
```

```
                IF l_lockstatus = 0 THEN
                   SYS.DBMS_APPLICATION_INFO.READ_MODULE(g_module_name,g_action_name);
    dbms_output.put_line('g_module='||g_module_name||','||g_action_name);
                   SYS.DBMS_APPLICATION_INFO.SET_MODULE(module_name,action_name);
         ELSE
    RAISE_APPLICATION_ERROR(-20000,module_name||' is already running.');
                END IF;
             ELSIF action_name = 'Exit' THEN
                l_lockstatus := dbms_lock.release(l_lockhandle);
                SYS.DBMS_APPLICATION_INFO.SET_MODULE(g_module_name,g_action_name);
             ELSE
         SYS.DBMS_APPLICATION_INFO.SET_MODULE(module_name,action_name);
      END IF;
           END IF;
        END;
```

In this code, if the MODULE and or ACTION match specific criteria, a user lock is requested and, if successfully obtained, the program continues. When the module is has completed, the lock is released. During this time, if any other session tries to execute the same module then it will exit with the error message: "Module is already running.". Alternatively, the second program could request the lock in WAIT mode, which would see it wait until the first module was complete before starting execution.

This principle can also be used to restrict trace enabling to one instance of a program. For example, if a batch job runs a number of streams simultaneously and it is decided to enable tracing during the run, it may be expedient to only trace one of the streams if restricted by filesystem space:

```
PROCEDURE set_module(module_name in VARCHAR2,action_name IN VARCHAR2)
IS
l_count NUMBER;
BEGIN
    IF module_name = 'Procedure B'  AND action_name = 'Enter'  THEN
        SELECT count(*)
        INTO l_count
        FROM v$session
        WHERE module = module_name
         ;
        IF l_count = 0 THEN
             execute immediate 'alter session set events
                ''10046 trace name context forever, level 12''';
        END IF;
        SYS.DBMS_APPLICATION_INFO.SET_MODULE(module_name,action_name);
    END IF;
END;
```

In this example, the first stream to start checks the V$SESSION view for existing instances of this module. If none are found then tracing is activated and the module registered with the database. When subsequent instances start they find that there is already one running and they continue without tracing enabled.

4.4 Bug#4641478

The author encountered a problem when using DBMS_APPLICATION_INFO on a 10g database, and this has now been logged as bug#4641478. The behaviour of the package in previous versions when running DEMO_PKG with tracing enabled would be to write the following lines to the trace file:

```
APPNAME mod='Procedure A' mh=0 act='Enter ' ah=0
APPNAME mod='Function C' mh=0 act='Called from Procedure A' ah=0
APPNAME mod='Procedure A' mh=0 act='Returned from Function C' ah=0
APPNAME mod='Procedure B' mh=0 act='Called from Procedure A' ah=0
APPNAME mod='Function C' mh=0 act='Called from Procedure B' ah=0
APPNAME mod='Procedure B' mh=0 act='Returned from Function C' ah=0
APPNAME mod='Procedure A' mh=0 act='Returned from Procedure B' ah=0
```

while in 10g the output would be:

```
*** ACTION NAME:(Returned from Procedure B) 2005-08-06 06:56:53.718
*** MODULE NAME:(Procedure A) 2005-08-06 06:56:53.718
```

which is the last call made by the program.

The bug occurs only in PL/SQL programs, and the V$ views are updated as expected. The module and action information can be forced to be written to the trace file by preceding the call with a SQL statement, a commit, or a call to one of the SYS.DBMS_SYSTEM modules that writes to the trace file. A workaround is to create a custom package, which includes a call to DBMS_SYSTEM.KSDWRT that writes a null string to the trace file, after having ascertained that tracing is enabled.

```
PROCEDURE SET_MODULE(MODULE_NAME IN VARCHAR2,
                     ACTION_NAME IN VARCHAR2)
IS
l_trace_level NUMBER;
BEGIN
    sys.dbms_system.read_ev(10046,l_trace_level);
    IF l_trace_level > 0 THEN
        sys.dbms_system.ksdwrt(1,'');
    END IF;
    SYS.DBMS_APPLICATION_INFO.SET_MODULE(MODULE_NAME,ACTION_NAME);
END
;
```

This module checks the current trace level; a value greater than zero signifies that tracing is enabled, and the call to SET_MODULE is forced to the trace file.

5 Implementation

5.1 Access Methods

Although the Oracle documentation lists only one method of implementing the custom package, namely redirecting the public synonym DBMS_APPLICATION_INFO, there are a number of ways to do so depending on the needs of each application:

- Redirecting the public synonym:

This method ensures that every call to DBMS_APPLICATION_INFO is made to the custom version.

- Creating a private synonym:

If only selected users need access to the custom version then a private synonym in those schemas will ensure that all other users call the original package.

- Creating package with different names:

A library of custom versions can be accumulated with different purposes, with each one named appropriately, for example:

```
DBMS_APPLICATION_INFO_TRACE
DBMS_APPLICATION_INFO_RETURN
DBMS_APPLICATION_INFO_RESTRICT
```

Public or private synonyms can then be directed to the package desired:

```
CREATE OR REPLACE SYNONYM dbms_application_info
FOR dbms_application_info_trace;
```

- Creating the package in another schema

If only one user needs access to the custom version then creating it in that user's schema and having no synonyms directed to it will ensure that no other users will call it.

The variety of options facilitates the implementation of multiple custom packages across one application or database.

5.2 Caveats

As with all stored procedures and packages, care should be taken when recompiling the custom package or redirecting its synonyms in an environment where it is in use. Any of the above methods will result in library cache waits until any transaction currently using the package releases it.

It should be noted that any calls or references to objects in another schema, for example SYS.DBMS_SYSTEM, would result in a compilation error unless the owner of the custom package has explicit permissions on those objects.

5.3 Code Modification

In order to implement the comprehensive functionality enabled by customised versions of DBMS_APPLICATION_INFO, a call to SET_MODULE must be made at the beginning and end of each program module. This task can be off-putting for developers, particularly in an application where there are a significant number of modules. However, in programming languages that have text-based source code, the modifications can easily be made programmatically, using a scripting language like perl, awk or even PL/SQL.

6 Performance Implications

The effect on the performance of the application and database must be taken into consideration. As each level of complexity is added to the custom package, resulting performance degradation may not be acceptable and trade-offs may have to be made. The following tests were performed on an Oracle 8.1.7.4 database running SunOS 5.8 on an 8 CPU machine. DBMS_APPLICATION_INFO.SET_MODULE was called 1048576 times, the elapsed time being recorded in centiseconds (*c*secs) using the DBMS_UTILITY.GET_TIME procedure before and after each 1048576 calls, which were run 20 times serially. This procedure was performed on:

- The Supplied package
- The Basic custom package
- The Return package (Custom package with code to identify calling modules)
- The Supplied package with tracing enabled

With the following results:

Package	Avg per 1048576 calls (*c*secs)	Avg per 1 call (*c*secs)	Calls per second
Supplied	215	0.000205	487709
Basic	276	0.000263	379918
Return	709	0.000676	147895
Supplied + trace	1382	0.001317	76335

It can be seen that calls to the Supplied package are insignificant at around 2 microseconds (μsecs) per call, which means that a program making half a million calls would have just over one second added to its processing time. The implementation of the Basic custom package adds an average 0.58 μsecs per call. The Return package introduces extra code to the programming logic, albeit a different amount according to the conditions in the IF statement. It is interesting to note that the addition of this few lines of code - none of which accesses the database - has more impact than calling the original or basic modules. Once tracing is enabled, a comparatively large increase is encountered, although each call is still only taking 13 μsecs. As an example of the performance impact: if a call is made to DBMS_APPLICATION_INFO one hundred times per second then a one-hour program will be extended by 0.738 of one second using the Supplied package and 4.741 seconds using the Supplied package + trace.

The extremely low overhead of the Supplied package in particular demonstrates that calls to DBMS_APPLICATION_INFO included in a program during development and testing can be retained in the code when it is deployed in production without fear of adversely affecting performance. Not only will this assist the DBA in monitoring the program at all times, but will allow a custom version to be implemented at any time if necessary.

7 Summary

Customising the DBMS_APPLICATION_INFO package makes it a dynamic tool for developers and DBAs. It can be used to provide meaningful information in real-time, produce formatted output of custom content to tracefiles (even when Oracle tracing is not enabled), and conditionally restrict or allow particular operations. The multiple ways in which the custom package can be implemented allows different functionality to be exploited in different schemas or combined in the same package. Instrumentation is kept separate from the application code in the development environment and the negligible performance impact of the Supplied package means that the program can be promoted to production with virtually no negative effect. Once there, it provides a platform from where troubleshooting measures can be introduced at any time without modifying the application code.

References

[1] Oracle Corporation. PL/SQL Packages and Types Reference 10g Release 1 (10.1), Oracle Database Documentation Library. Part Number B10802-01.

[2] Amar Kumar Padhi. Use oracle's dbms_application_info to prevent routines from running simultaneously, March 10 2004. `http://www.databasejournal.com/features/oracle/article.php/3321961`.