Using Native Compilation in PL/SQL and Java

© Geoff Ingram, Proton Technology Ltd 2002. No part of this document may be reproduced for any purpose without the express written permission of Proton Technology Limited.

If you're a developer that needs to undertake Oracle database server programming, then you have a choice of two languages: PL/SQL or Java. You can mix also and match the two. PL/SQL is Oracle's own flagship server side application development language. It's without peer for applications that perform mostly database data processing. Java, on the other hand, is a DBMS independent language which includes JDBC as its standard data access interface. Oracle provides a Java Virtual Machine (JVM) that runs within the database server itself, and you need to make sure you install this at database creation time. If you choose Java, you also make available extensive additional functionality that exists in the Java class libraries. For example, if you want to perform data compression or checksums, Java has built in classes to do that. Both Java and PL/SQL sacrifice performance to some extent because they are interpreted languages, whose instructions are converted into executable code at runtime. Java is interpreted by the JVM.

To address the potential performance overhead from code interpreting, Oracle9i provides the ability to generate native machine code for routines written in both languages. This paper shows you how to convert Java and PL/SQL routines into native machine code. It's assumed that you have basic familiarity with concepts like code compiling and linking, and the use of Makefiles.

Using Native Java with JServer Accelerator

Oracle provides a component called JServer Accelerator to generate native executable code for Java programs. Before considering the benefits of native Java versus Java executing within the JVM it's useful to try and present a real-world scenario in which to perform a comparison of the two. Consider a situation where the requirement is to compare table data between two databases separated by a wide area
network (WAN) to ensure that the table contents are identical. This is the type of operation you might perform if you are running data replication (including Oracle's own multimaster replication) between sites. In such a scenario, transporting data between sites isn't a good idea because of the speed of the link. A better approach is to use a checksum to create a numeric signature of the data at each end of the link and compare those instead. The checksum is sometimes referred to as a digest or cyclic redundancy check (CRC).

Checksums crop up quite often in the IT world, as they provide a technique for efficiently checking that two sets of data are identical without a byte-for-byte comparison. For example, if you have two large files on separate machines, the easiest way to check if they are the same is to generate a checksum for both and compare those. Oracle itself stores a checksum in each archived redo log, which the DBMS checks against the file contents before applying the log during a recovery scenario. Solaris, Linux, and most popular flavors of UNIX, have a built-in cksum command to generate the checksum on a file as follows:

```
$ cksum sqlnet.log
1433214650   13378 sqlnet.log
```

If you need to generate a checksum on the contents of a database column, then Java looks like a natural solution because it includes a built-in checksum class that you can call in your Java source code. The following code shows a simple Java class that returns a 32-bit CRC based on the input string passed as a parameter, based on the built-in Java CRC32 class:

```java
import java.util.zip.CRC32;
public class CRC {

    public static long CRC( String s ){
        java.util.zip.CRC32 thisCRC32 = new java.util.zip.CRC32();
        thisCRC32.update(s.getBytes()); // sets CRC32 of input string
        return thisCRC32.getValue(); // returns the checksum
    }
}
```

Before the class can be run within the JVM in the DBMS, it needs to be compiled and loaded into the database. This example uses a Java compiler installed in /usr/bin and assumes that your UNIX environment is set to the ORACLE_SID of the database into which you wish to load the Java class:

```
$ /usr/bin/javac -O -d ./classes -classpath ./classes:$ORACLE_HOME/jdbc/lib/classes111.zip:/usr/java/lib/classes.zip CRC.java

$ loadjava -u system/pwd -verbose -resolve CRC.class
initialization complete
loading : CRC
creating : CRC
resolver :
resolving: CRC
```

Keep in mind that the Apache Web server bundled with the Oracle DBMS contains another Java compiler that you can use instead. Next, a mapping needs to be created to enable the Java procedure to be called from SQL via a PL/SQL function. This example creates a function named JAVA_CHECKSUM:

```sql
CREATE OR REPLACE FUNCTION JAVA_CHECKSUM(str in varchar2)
RETURN NUMBER AS LANGUAGE JAVA
```
The final step is to make sure that JAVA_CHECKSUM produces the expected value for a specified input. The traditional CRC32 test is to use an input string '123456789' and verify that the checksum value is 3421780262:

```sql
select java_checksum('123456789') from dual;
```

<table>
<thead>
<tr>
<th>JAVA_CHECKSUM('123456789')</th>
</tr>
</thead>
<tbody>
<tr>
<td>3421780262</td>
</tr>
</tbody>
</table>

In the early days, one of the drawbacks of the Oracle JVM was performance. This perhaps is not surprising, as a major goal of Java was to deliver application portability through the capability to run the same Java application unchanged in any environment with a JVM. This portability requires the development of a JVM for each platform, which enables the same binary-identical application to run unchanged. The Java application byte-codes generated by the Java compiler are interpreted at run time and executed by the JVM. This approach of compiling Java code on demand at runtime is called Just-in-Time (JIT) compiling. JIT byte code execution isn't as fast as running code that's natively compiled for the processor running on the server where the code executes.

Oracle acknowledged the performance problem of the JIT approach by providing the JServer Accelerator starting with Oracle8i Release 3. The JServer Accelerator and associated tools enable Java classes to be compiled into native executable shared libraries for performance. Although this concept horrifies Java purists, it's worth considering if you need to make your Java run faster. At run time, JServer Accelerator checks for native versions of classes and uses them in preference to the standard Java byte-code versions.

Oracle refers to this approach as Ahead-of-Time compiling. That's a fancy way of saying the executable code exists already in advance. The C language versions of the classes are created using the ncomp utility. These natively compiled classes can only be used with the JServer JVM within the same version of the server in which they were compiled. It's essential that all testing and debugging be performed on the regular Java version of a class before deploying the accelerated C version. In order to create the C version, you need to run the ncomp utility in the directory where the class resides. The ncomp utility requires a file called CRC.classes (for a Java class called CRC.class) that contains the following line:

```java
import CRC.class;
```

When the ncomp utility runs, it generates a C source code file, CRC.c, and compiles it into a shared library, which is installed into $ORACLE_HOME/javavm/admin. The following example shows that the shared library generated by ncomp is libjox8_eb1f599c47_systemUnnamedPackage.so:

```
$ ncomp -user system/pwd /u01/app/oracle/j/classes/CRC.classes
```

```
Tue Jan 01 10:01:41 GMT 2002 installed
/libjox8_eb1f599c47_systemUnnamedPackage.so
```

On Sun Solaris, to check whether the accelerated version of the package is being used, you can run the pmap command on the Oracle shadow process for the session and check whether the shared
library has been loaded. This shows an example for an Oracle 8.1.7 DBMS where the shadow process for
the session is PID 16502:

```
$ /usr/proc/bin/pmap 16502
16502: oracleORAD1 (DESCRIPTION=(LOCAL=no)(ADDRESS=(PROTOCOL=BEQ)))
EF4B0000 16K read/exec
/u01/app/oracle/product/8.1.7/javavm/admin/libjox8_eb1f7d03e6_system
_UnnamedPackage.so
```

The following SQL can be used to show all the accelerated classes that are installed:

```
select DLL_NAME,STATUS from jaccelerator$dlls;
```

<table>
<thead>
<tr>
<th>DLL_NAME</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>/libjox8_eb1f599c47_system_UnnamedPackage.so</td>
<td>installed</td>
</tr>
</tbody>
</table>

In this example, when the accelerated version of the class was used, there was a reduction of
approximately 10 percent in the query elapsed time for the JAVA_CHECKSUM function.

### Using Native PL/SQL

By default, PL/SQL (like Java) is an interpreted language. PL/SQL routines are compiled into an
intermediate format known as P-code, which is suitable for interpretation by the DBMS PL/SQL engine
at run time. Like the Java Accelerator, Oracle9i provides a facility to convert PL/SQL code into native
machine code, rather than P-code, in order to improve performance. PL/SQL implemented in machine
code executes faster than interpreted code, so you should consider native PL/SQL for performance-
critical code. Oracle's implementation of native PL/SQL is very elegant because it's almost transparent to
the developer and therefore easy to use. Before native code can be generated, several Oracle system
parameters must be configured to provide values for the following settings that enable the PL/SQL code
to be converted into a C routine that is then compiled and linked to create a shared library:

- Path of a native C compiler
- Path of a link loader
- Path of the make utility
- Path of a Makefile
- A directory to store the shared libraries created from the PL/SQL code

The paths of the utilities are operating system specific. Also, on some platforms (for example,
Sun Solaris), a C compiler may not be readily available as its an extra cost option. However, if you're
using Linux, the good news is that you should have no problem building native PL/SQL because Linux
comes bundled with the excellent gcc compiler. The following SQL statements demonstrate how to set
the Oracle system parameters to enable the creation of native PL/SQL on Linux:

```
alter system set plsql_native_c_compiler='/usr/bin/cc';
alter system set plsql_native_linker='/usr/bin/ld';
alter system set plsql_native_library_dir=
```
The choice of the library directory - where native code is generated - is up to the DBA, and the Makefile used in the example (spnc_makefile.mk) is the default version shipped by Oracle. You can view the values for all Oracle settings at any time by querying the V$PARAMETER view. Once the required environment is in place, native PL/SQL routines are generated at PL/SQL compile time whenever the following session setting is enabled:

```
alter session set plsql_compiler_flags='NATIVE';
```

**NOTE**

If the Oracle system settings are incorrect for generating native PL/SQL, then no error is reported. Instead, a standard P-code-interpreted version is silently generated.

A quick benchmark can be used to demonstrate performance improvements resulting from running the same PL/SQL routine in native mode compared to interpreted mode. The following example is based on the CHECKSUM function in the OWA_OPT_LOCK package, which Oracle ships as part of the Oracle Web Agent (OWA). This function returns a 32-bit checksum value of an input string based on the Internet 1 protocol. The following commands create a native PL/SQL version of the function in SQL*Plus:

```
alter session set plsql_compiler_flags='NATIVE';
create or replace
function native_checksum( p_buff in varchar2 ) return number is
  l_sum number default 0;
  l_n number;
begin
  for i in 1 .. trunc(length(p_buff||'x')/2) loop
    l_n := ascii(substr(p_buff||'x', 1+(i-1)*2, 1))*256 +
            ascii(substr(p_buff||'x', 2+(i-1)*2, 1));
    l_sum := mod(l_sum+l_n,4294967296);
  end loop;
  while ( l_sum > 65536 ) loop
    l_sum := bitand( l_sum, 65535 ) + trunc(l_sum/65536);
  end loop;
  return l_sum;
end native_checksum;
/
```

If the native compilation was successful, a shared library with the name of the PL/SQL routine and Oracle owner should exist in the directory specified by plsql_native_library_dir setting:

```
$ cd $ORACLE_HOME/plsql/native_libs
$ ls -1 NAT*.so*
NATIVE_CHECKSUM__SYSTEM__1.so
```

The USER_STORED_SETTINGS view in the database can also be used to show PL/SQL objects that were natively compiled as follows:

```
REM see compile state of the routine
select object_name,param_name,param_value
from user_stored_settings
```
where object_name='NATIVE_CHECKSUM';

<table>
<thead>
<tr>
<th>OBJECT_NAME</th>
<th>PARAM_NAME</th>
<th>PARAM_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NATIVE_CHECKSUM</td>
<td>plsql_compiler_flags</td>
<td>NATIVE,NON_DEBUG</td>
</tr>
</tbody>
</table>

After having built the native version successfully, a simple test based on a newly created Oracle database showed the native version to be significantly faster than the interpreted version (10.6 seconds versus 13 seconds) based on the following SQL that also confirms that each version produces the same result:

REM test interpreted vs native version

```
select sum(checksum(owner||'.'||object_name)) checksum
from all_objects;

CHECKSUM
---------
1002777501
```

```
select sum(native_checksum(owner||'.'||object_name)) native_checksum
from all_objects;

NATIVE_CHECKSUM
-----------------
1002791235
```

Summary

It's long been the case that if you required absolutely top performance from your Oracle client applications you had to code in low-level programming languages such as Pro*C or Oracle Call Interface (OCI). On the server-side, PL/SQL and Java didn't provide native code implementations until Oracle9i. You should consider creating native code versions of your PL/SQL and Java stored procedures for applications that require those last few percentage points of performance tweaking.