Got performance impacts of interfacing COBOL with other languages? Reduce them!

Judith M. Myerson (jmyerson@bellatlantic.net)
System engineer and architect

25 June 2012

Often businesses and government agencies need to interface COBOL with other languages as a more efficient way of updating their enterprise systems than replacing COBOL programs with those in other languages. In reality, COBOL interfaces come with some limitations. In this article, the author explains the performance impacts of interfacing COBOL with other languages and gives some tips to avoid becoming reactive to adverse impacts.

Introduction

Decades ago, when I first started to use mainframe COBOL, I discovered it could not be interfaced with non-COBOL languages. I discussed this with a professor as a possible dissertation topic on performance impacts of interfacing COBOL with non-COBOL languages.

To find out what the performance impacts might be, I experimented with mini COBOL/Fortran interfaces based on "A Fortran Interface to the CODASYL database task group specifications" (see Resources). Fortran was a popular language in those days.

I noted that some COBOL data types do not have Fortran equivalents. Data or objects that were no longer needed, remained on the disk. I saw when memory was allocated more than was needed, there were indications of stack overflows. I fixed the problems with the correct Fortran/COBOL data types. I got around the memory limitations by calling subprograms when needed and releasing them when not needed. The maximum size of processors in those days was very small compared to the enormous processor sizes that we see today.

In this article, I give you tips on avoiding becoming reactive to performance impacts of COBOL interfaces with Java, C/C++, DB2 and Oracle.

1. Prevent timeouts
2. Use build scripts
3. Follow data conversion rules
4. Prevent memory leaks
5. Delete local references
Interface performance impacts: Java and C/C++

When I first used Fortran, it was widespread in the computing community at large. Today, Java® is the most popular language followed by C/C++ as interfaces with COBOL. Fortran currently a favorite with scientists has scaled down the ladder of popularity with general population.

Java as the interface

You can get a COBOL program to call a Java program that would call a different COBOL program. You might encounter memory problems that could result in performance impacts if you are not careful in designing Java programs. These issues include memory leaks, high memory usage, inefficient object creation and inefficient garbage collector behavior. Some other performance problems include issues with tracking object local references and using local references in threads.

Memory leaks in Java are created by referencing objects that are no longer needed. One source of high memory usage are the inefficiently configured caches and a high number of active users who are concurrently accessing the system.

Inefficient object creation becomes obvious when user load increases as the garbage collector must constantly clean up the heap. This might lead to high processor consumption by the garbage collector. Inefficient garbage collector behavior occurs when the garbage collector is not properly configured.

Local references to objects are valid only while the method that you invoke runs. Automatic freeing of the local reference does not always occur after the native method returns. Global references remain valid until you explicitly delete them. Local references become not valid when you attempt to pass local references from one thread to another. To ensure that the objects are not prematurely released during garbage collection, the Java virtual machine (JVM) tracks local and global references.

If you do not explicitly free the local reference, the system will run out of memory as in the following two cases.

- **Case 1: You access a large object in a method to create a local reference to the object for extensive computations.** The local reference prevents this object from being released during garbage collection.
- **Case 2: You do not use all local references created in a method at the same time.** In a COBOL method, you loop through a large array of objects, retrieve the elements as local references, and operate on elements at each iteration. The local reference to the array element after each iteration is not automatically freed.

C/C++ as the interface

In a C/C++ program, when too much memory is used on the call stack, stack overflow can occur. More memory is allocated on the stack than it will fit. The stack size in the call back depends on the system architecture, and amount of available memory.
Stack overflows are made worse by reducing the effective stack size of a given C/C++ program. If running the program in multithreading mode, the programs with threads have less stack space per thread than a program with no threading support or assigned to a single thread.

In calling between COBOL and C/C++ programs, performance issues arise when you attempt to invoke functions in one language that collapse program stack frames of another language. For the languages that do not initiate the collapsing and the termination of active stack frames, adverse effects could result in stack overflow.

For example, normal cleanup or exit functions of the language might not be performed, such as the closing of files by COBOL during run-unit termination, or the cleanup of dynamically acquired resources by the involuntarily terminated language.

**Interface performance impacts: IBM DB2 and Oracle**

IBM COBOL for AIX® is used with IBM DB2® software and is certified with Oracle clients and Oracle clients running Tuxedo. COBOL programs executing embedded SQL statements communicate with DB2 and Oracle through a working storage area called the SQL Communications Area (SQLCA). Before running the SQL statements, the system checks rights and roles, checks against the primary key and looks up foreign key. It checks constraints and indexes, and performs the intricacies of the optimizer and transaction control. The system also locks, logs, commits and rolls back.

Not changing change default values or incorrectly configuring for any of these can cause the SQL execution to fail. All results of the operation are returned into the SQLCODE and SQLSTATE fields in the SQLCA. SQLCODE provides key information on whether SQL statement execution has succeeded or failed.

**Long waits on locks**

One cause of the timeout is the object is locked by another user or by you in another session, but the lock has not been detected by the application. The application waits to get access until it is interrupted by the DB2 timeout.

If you do not change the default value of -1 for `LOCK TIMEOUT` when you are trying to access or edit an object, there is no lock timeouts. If you use `SET CURRENT LOCK TIMEOUT` to set the time to more than 10 or 15 seconds in an SQL embedded statement, there might be a long delay in timeouts.

Waiting on locks for a long time can have an adverse impact on locks. You will get impatient. If the current unit of work roll backs, you will get annoyed. The results of this failed operation are returned into the SQLCODE.

For example, `SQLCODE = -901` indicates an error to lock timeout and what value has been set for `LOCKTIMEOUT`. `SQLCODE = -911` means there was a deadlock/timeout and roll back has been done. `SQLCODE = -913` is interpreted that a deadlock/timeout has occurred but no roll backs.

The `SQLCODE` is part of the `SQLCA` record in the `WORKING-STORAGE SECTION` or `LINKAGE SECTION` of the program. It’s easier to add `SQLCODES` to `sqlca.cbl`. This sample program is part of the `LINKAGE`
SECTION of the checkerr.cbl, an error checking utility program in DB2's building COBOL application on AIX script (bldapp).

**High memory demands**

In embedded COBOL statements, stored procedures offer many advantages over dynamic SQL, including improved performance and scalability. Stored procedures are optimized and saved in native machine language. When a stored procedure is called, it is all ready to go. When you send a SQL statement, the query processor has to parse it, analyze it, and create a query plan for execution.

There is one drawback however. Calling stored procedures might require more memory demands. If these demands are too high, the SQL execution will not succeed.

**Performance tips**

Here are five tips to avoid becoming reactive to performance impacts of interfacing COBOL with Java, C/C++, DB2 and Oracle.

- Prevent timeouts
- Use build scripts
- Prevent memory leaks
- Delete local references

**Tip 1: Prevent timeout**

If any SQLCODE shows a timeout error, check if you have changed the default value of LOCKTIMEOUT to a short period of time, such as 10 or 15 seconds. If you are not sure what the value of LOCKTIMEOUT is, do this:

```
db2 "get db cfg for DBNAME"
```

Then find the text containing:

```
Lock timeout (sec) (LOCKTIMEOUT) = 30
```

The value in this example is too high, consider changing it to 10 seconds by using the following command:

```
db2 "update db cfg for DBNAME using LOCKTIMEOUT 10"
```

Assuming that there is no problems with lock timeouts, look for the following lines:
Listing 1. Lock time out results

Locks held currently = 0
Lock waits = 0
Time database waited on locks (ms) = 0
Lock list memory in use (Bytes) = 576
Deadlocks detected = 0
Lock escalations = 0
Exclusive lock escalations = 0
Agents currently waiting on locks = 0
Lock timeouts = 0

The output result shows that the size of lock list memory in use is 576 bytes. If this size is more than 50 percent of the LOCKLIST size you’ve defined, increase the number of 4 K pages in the LOCKLIST database configuration.

After fixing the value of LOCKTIMEOUT, check if the object in question is locked, take the appropriate action to unlock it, working with the user who locked it. Review the DB2 master log to find process holding DB2 locks.

If you have a firewall between DB2 Connect™ and your system, or between DB2 Connect and DB2, you might need to check whether the firewall setup results in a timeout.

Tip 2: Use build scripts

DB2 provides build scripts that includes an error-checking utility to use when compiling and linking COBOL embedded SQL and DB2 application programming interface (API) programs. You can find the scripts in the sqllib/samples/cobol directory, along with sample programs that can be built with these files. The build file, bldapp, contains the commands to build a DB2 application program as shown below:

Listing 2. Build application script sample

```bash
#!/bin/sh
# SCRIPT: bldapp
#
# PART 1: Check if an embedded SQL program exists.
#
# Usage: bldapp <prog_name> [ <db_name> [ <userid> <password> ]]
# Set DB2PATH to where DB2 will be accessed.
# The default is the standard instance path.
DB2PATH=$HOME/sqllib
# If an embedded SQL program, precompile and bind it.
# if [ -f $1".sqb" ]
# then
# /embprep $1 $2 $3 $4
# fi
#
# PART 2: Compile checkerr.cbl error checking utility.
#
cob2 -qpgmname(mixed) -qlib -I$DB2PATH/include/cobol_a \
 -c checkerr.cbl
#
# PART 3: Compile the program.
#
cob2 -qpgmname(mixed) -qlib -I$DB2PATH/include/cobol_a \
 -c $1.cbl
```

Got performance impacts of interfacing COBOL with other languages? Reduce them!
In the first part, the first parameter, $1$, specifies the name of your source file. Building embedded SQL programs (with file extension of `.sqb`) requires a connection to the database so three optional parameters are also provided: the second parameter, $2$, specifies the name of the database to which you want to connect; the third parameter, $3$, specifies the user ID for the database, and $4$ specifies the password.

For an embedded SQL program, bldapp passes the parameters to the precompile and bind script, embprep. If no database name is supplied, the default sample database is used. The user ID and password parameters are only needed if the instance where the program that is built is different from the instance where the database is located.

The second and third parts show same compiler options on the compiling error-checking utility and the embedded SQL program. The only difference is the name of the COBOL programs: checkerr.cbl and $1$. The parameter $1$ was passed as the name of the source program you specified.

I explain each compiler option as follows:

Table 1. Compiler options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cob2</code></td>
<td>The name for the IBM COBOL Set Compiler.</td>
</tr>
<tr>
<td><code>-qpgmname(mixed)</code></td>
<td>Instructs the compiler to permit CALLs to library entry points with mixed-case names.</td>
</tr>
<tr>
<td><code>-qlib</code></td>
<td>Instructs the compiler to process COPY statements.</td>
</tr>
<tr>
<td><code>-c</code></td>
<td>Performs compile only.</td>
</tr>
<tr>
<td><code>-L$DB2PATH/include/cobol_a</code></td>
<td>Specifies the location of the DB2 include files.</td>
</tr>
</tbody>
</table>

The fourth part shows options for linking the program. I explain each option as follows:

Table 2. Linking options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt;o $1</code></td>
<td>Specifies the output as a shared library file.</td>
</tr>
<tr>
<td><code>$1.o</code></td>
<td>Specifies the stored procedure object file.</td>
</tr>
<tr>
<td><code>checkerr.o</code></td>
<td>Includes the utility object file for error-checking.</td>
</tr>
<tr>
<td><code>-L$DB2PATH/lib</code></td>
<td>Specifies the location of the DB2 runtime shared libraries. If you do not specify the <code>-L</code> option, the compiler assumes the following path: <code>/usr/lib:/lib</code>.</td>
</tr>
<tr>
<td><code>-ldb2</code></td>
<td>Links with the database manager library.</td>
</tr>
</tbody>
</table>

Tip 3: Prevent memory leaks

To prevent memory leaks, you should cancel any COBOL programs you have loaded from Java before the owning Java object is garbage-collected. Use the following call from the "finalize" method of the Java object:
runtime.cobcancel("program")

where program is the name of a COBOL program loaded using a runtime.cobload() call. The following code is an example of a "finalize" method in a Java program:

Listing 3. "finalize" method example

```java
protected void finalize()
{
    try
    {
        runtime.cobcancel("demoFinalizers");
        System.out.println("demoFinalizers - finalize'ed");
    }
    catch(Exception e)
    {
        System.out.println("Error during finalize : "+e.getMessage());
    }
}
```

Tip 4: Delete local references

You explicitly free local references no longer needed. Not freeing them can cause the system to run out of memory due to lack of space that the JVM needs to track local (and global) references.

You use a large object for extensive computations that require high processor consumption. The local reference was created when you accessed the object in a method. It was not automatically released during garbage collection when you no longer need the object for additional computations. You can use the Java Native Interface (JNI) service DeleteLocalRef to explicitly free a local reference within a method.

You create a large number of local references in a method, but do not use all of them at the same time. Because the JVM requires space to keep track of each local reference, you should free those that you no longer need.

For example, in a COBOL method, you loop through a large array of objects, retrieve the elements as local references, and operate on one element at each iteration. You can free the local reference to the array element after each iteration.

In all cases, local references are valid only in the thread in which you create them. Do not pass them from one thread to another. If you pass them, you will not be able to free local references when no longer needed. You should always convert local references to global references whenever there is a possibility that multiple threads might use the same reference.

Conclusion

Avoid becoming reactive with adverse performance impacts of interfacing COBOL with other languages requires planning ahead of time to get proactive with performance tips to reduce or prevent adverse impacts. Developers should communicate with programming teams on preventing timeouts and memory leaks, and deleting local references. They should update sample build scripts to make their tasks easier.
Resources

Learn

• Learn how to build MicroFocus COBOL applications on AIX in *IBM DB2 Universal Database Application Development Guide: Building and Running Applications*.
• Read the author's article, *Get proactive with limitations in COBOL multithreading* to get more information on handling threads.
• The *Work with Web services in enterprise-wide SOA series* by Judith M. Myerson offers information on how to work with Web services in enterprise-wide SOAs.
• Browse the Judith M. Myerson's series, *Use SLAs in a Web services context*, has details on service-level agreement.
• Bring your organization into the future with *RFID in the Supply Chain*, which explains business processes, operational and implementation problems, risks, vulnerabilities, and security and privacy. Find out how RFID is to interact real objects to virtual ones.
• Read Judith M. Myerson's *The Complete Book of Middleware*, which focuses on the essential principles and priorities of system design and emphasizes the new requirements brought forward by the rise of e-commerce and distributed integrated systems.
• Get the business insight and the technical know-how to ensure successful systems integration by reading *Enterprise Systems Integration, Second Edition*.
• Visit the *technology bookstore* for books on these and other technical topics.
• Stay current with *developerWorks technical events and webcasts*.

Get products and technologies

• IBM trial products for download: Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

• Participate in the discussion forum for this content.
• Participate in the discussion forum.
• *developerWorks blogs*: Get involved in the developerWorks community.
About the author

Judith M. Myerson

Judith M. Myerson is a systems architect and engineer. Her areas of interest include enterprise-wide systems, middleware technologies, database technologies, cloud computing, threshold policies, service level agreements, industries, network management, security, RFID technologies, presentation management, and project management. She was one of the first COBOL programmers to write and run a resource security program, and programs interfacing COBOL with Fortran and other languages.

© Copyright IBM Corporation 2012
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)