A PL/SQL Inlining Primer

Charles Wetherell
Oracle Corporation

14 September 2015

Abstract

The PL/SQL language offers the inlining optimization. This primer describes inlining, tells how it works, and guides its use in practical programs.

1 Introduction

The PL/SQL language offers inlining as a performance optimization. Inlining is not a routine part of the system; a PL/SQL programmer must ask specifically for inlining to be applied. If it is such an important optimization, why isn't inlining automatic? When should inlining be used and when is it inappropriate? What is inlining in the first place? This primer defines inlining, describes inlining's effects, tells how to invoke inlining, discusses the pros and cons of inlining, touches on the background theory, and suggests guidelines for the use of inlining. An attentive reader should become an expert on PL/SQL inlining.

A word on examples. First, examples will not necessarily be complete PL/SQL units; where appropriate, they will just be fragments that illustrate a particular point. Second, they will not be decorated with the SQL (for example, `create or replace`) or SQL*Plus (for example, `set serveroutput on`) commonly found in many PL/SQL text books and working programs. Third, the capitalization and layout will be appropriate to make the points clear and not necessarily that commonly used in other PL/SQL writings so that important ideas stand out in the examples and are not lost in a morass of programming detail. A PL/SQL programmer will find it easy to convert these examples to executable code.

2 Basic Principles

Before inlining can be defined, a few basic ideas need explanation. An example will help. Consider the code in Listing 2.1; line 1 begins the definition of a top level PL/SQL unit, [1] procedure `TheWorld`, that supplies a little universe for the explanations. Inside this world a variable `a` is declared and so is an inner (and very simple) function `AddMe`. On line 12, the function is called and finally the value of the function is printed. This is not an elaborate program.

Now some useful (if informal) definitions. In PL/SQL, both procedures and func-

[1] Forget for the moment that standalone procedures are rare in PL/SQL.

Listing 2.1: The First Inlining Example

```
1  procedure TheWorld is
2
3    a pls_integer;
4
5    function AddMe(x pls_integer, y pls_integer)
6      return pls_integer is
7    begin
8      return x + y;  -- Pretty simple!
9    end;
10
11 begin
12   a := AddMe(1, 2);
13   dbms_output.put_line(a);
14 end;
```

tions are known as *subprograms*. Secondly, every subprogram has a *definition* where it is completely described and its body is provided. The most interesting subprogram definition is that of function `AddMe` beginning on line 5. There is also another definition, that of procedure `TheWorld`, beginning on line 1.[2] In general, a subprogram is declared with *formal parameters*. Function `AddMe` has the formal parameters `x` and `y`. Procedure `TheWorld` has none.

Subprograms are not very useful if they are never called. The PL/SQL compiler issues warnings for uncalled subprograms because they are likely the result of programming errors. A *call site* or *invocation* is a location where a request is made for a subprogram to be executed. The call site on line 12 invokes the function `AddMe`. At the call site, an *actual argument* must be supplied for each formal parameter; the technical term is that the actual argument is *associated* with the parameter. At line 12, the actual argument `1` is associated with the formal parameter `x` and the actual argument 2 is associated with the formal parameter `y` of function `AddMe`.[3] This language of actual argument and formal parameter may seem finicky, but in the long run the clarity created outweighs any localized logorrhea.

When a subprogram is called, several steps are taken.

1. Each actual argument is evaluated and the value remembered. Sometimes evaluation may also require a type conversion of the computed value.
2. The value for each actual argument is associated with the appropriate formal parameter.
3. A memorandum of the current execution position is noted.
4. Control is transferred to the called subprogram.

[2]For the PL/SQL purists, a subprogram may also have a *declaration* that does not provide the implementation. Such declarations occur in package specifications, for instance. But a declaration is, in general, optional depending on the use of the subprogram while a definition is always required. If only the definition appears (as here for `AddMe`), then it doubles as a declaration.

[3]PL/SQL allows named and default associations. Nonetheless, when all the rules for these mechanisms are applied, every formal parameter must have an actual argument associated with it. It is not too surprising that the PL/SQL compiler actually makes an explicit association list internally for each call to ensure that the rules are followed.

Listing 2.2: First Example With AddMe Call Inlined

```
1  procedure TheWorld is
2
3    a pls_integer;
4    ActualArg1            pls_integer;
5    ActualArg2            pls_integer;
6    ReturnValueFromAddMe  pls_integer;
7
8    function AddMe(x pls_integer, y pls_integer)
9      return pls_integer is
10   begin
11     return x + y; -- Pretty simple!
12   end;
13
14 begin
15   -- a := AddMe(1, 2);
16   ActualArg1 := 1;
17   ActualArg2 := 2;
18   ReturnValueFromAddMe := ActualArg1 + ActualArg2;
19   a := ReturnValueFromAddMe;
20   dbms_output.put_line(a);
21 end;
```

5. A new storage structure (commonly called a *frame*) is created for the subprogram. The frame provides storage for the local variables and other data needed to execute the subprogram.
6. The frame is initialized with any necessary control data.
7. Execution of the subprogram proper begins. The work starts just after the word `is` in the subprogram's definition.
8. When it encounters a *return*, the subprogram's execution ends. The return may be implicit in a procedure.[4]
9. For a function, the return value is copied back to storage provided by the caller.
10. The subprogram's frame is destroyed.
11. Control returns to the location remembered back in step 3.

The exact work to be done and how hard it is to do depend somewhat on details of the source code, of the subprogram to be called, of the actual arguments, and of the data types involved. For example, evaluation of the actual arguments `1` and `2` is not very hard so that they don't create much work. Sometimes, though, a copy of large data structure might be required. Regardless of the details, it is clear that subprogram invocation is potentially an expensive operation even if many of the steps may be simplified for particular invocations.

And this leads directly to *inlining*. To avoid some of the work, a *copy* of the subprogram (not including the declarative prolog) replaces the call and the modified program is executed instead. Some plumbing is added to make the copy run properly. Listing 2.2 displays the result of inlining the call to function `AddMe`. This also explains the name *inlining*: the code for the called function is now *in line* with the rest of the source code around the call site.

First, notice that some new variables have been added at lines 4 through 6. These variables are used to pass values into and out of the copy of function `AddMe` because

[4] PL/SQL subprogram execution may also end because an exception was raised, but that point only complicates this discussion to no end. The topic will be considered again later.

the copy drops the association between formal parameters and actual arguments. The function definition on line 8 is unchanged. The function call on line 15 is dropped because the call is being replaced; it has been left in the source as a comment as an aid to explanation. Instead of actual argument associations with the formal parameters, the actual arguments are now evaluated and stored in the new variables starting on line 16. The body of the copied function appears on line 18. Notice that the formal parameters `x` and `y` have been replaced with the new variables `ActualArg1` and `ActualArg2` as part of the copying process. Also notice that the copy line assigns the value of the function to the new return value variable `ReturnValueFromAddMe`. Finally, on line 19, that return value is assigned to the variable `a` just as the return value from function `AddMe` was assigned in the original program.

What are the costs and benefits of this inlining operation? First, the original function `AddMe` contained essentially four operations:

- an enter operation to set up the function's frame.
- an add operation to compute `x + y`.
- a store operation to remember the result.
- a return operation to remove the frame and pass execution back to the caller.

In addition, there was a call operation at the call site. To execute the call, a total of five operations were necessary, not counting those needed to evaluate and store the actual arguments and to move the return value into the variable `a`.[5] In the copy, only two of these operations remain: the addition and the storage of the result. In other words, of the five operations for the call, inlining has removed three and left only two. The inlined version of this call is roughly 2.5 times faster than the call itself.

What are the costs? The primary cost is that both the storage for procedure `TheWorld` and the amount of source code has been increased. The procedure has four local variables where it used to have only one and there are four source code lines replacing the one line with the call. And this is not just an artifact of doing the inlining in source; these are real additions to the size of the procedure. If enough inlining is done, the size of the source code can grow very large; the technical term is that the program suffers *code bloat*. It is possible, if highly unlikely, that a bloated PL/SQL unit might consume so many resources that it could not be reliably executed or, possibly, might not even be compilable. In practice, this does not happen, but code bloat will be discussed again later.

Because inlining adds operations to which other optimizations apply, these problems are ameliorated. Basically, operations in the function `AddMe` are not available for cross optimization from the original call site because the function might be called from many places and so what would be a good optimization for one site might be bad or dangerous for another. But once the copy is made, the copied code is *right at the call site* and is available for optimizations that apply only to the copy and that may arise from interactions with other code surrounding the copy.

Listing 2.3 shows the optimizations. Because the original actual arguments are constants, they can be *propagated* forward into the addition operation on line 18. But

[5] In practice, the PL/SQL compiler can sometimes simplify or eliminate some of these operations, but the basic pattern is correctly described.

Listing 2.3: First Example After Optimization

```
1  procedure TheWorld is
2
3    a pls_integer;
4    -- ActualArg1            pls_integer;
5    -- ActualArg2            pls_integer;
6    -- ReturnValueFromAddMe  pls_integer;
7
8    -- function AddMe(x pls_integer, y pls_integer)
9    --   return pls_integer is
10   -- begin
11   --   return x + y; -- Pretty simple!
12   -- end;
13
14 begin
15   -- a := AddMe(1, 2);
16   -- ActualArg1 := 1;
17   -- ActualArg2 := 2;
18   -- ReturnValueFromAddMe := 1 + 2;
19   a := 3;
20   dbms_output.put_line(a);
21 end;
```

now that addition just produces the constant `3` and so the constant can be assigned directly to variable `a` on line 19.[6] Because the computation has all been eliminated, the new variables needed for the inlining copy may also be dropped; the lines that used them are now commented out. And because the function `AddMe` no longer has any calls, it also may be dropped.[7]

The net effect of inlining is to replace about seven operations (argument evaluations, call operations, and return value copy) with one (assignment of a constant to a variable). The execution will be much faster. In addition, the procedure `TheWorld` has become smaller because the object code space needed for function `AddMe` is no longer necessary. In effect, optimization has not only compensated for the bloat introduced by the inlining copy, it has slimmed the entire procedure down even further. The increased visiblity of operations in the copy affords new opportunities for optimization.

To summarize, inlining replaces a call to a subprogram with a copy of its body while making accommodation for the connections of actual arguments, formal parameters, and return values. By itself, inlining saves some of the subprogram call cost, notably the costs of creating, initializing, and then destroying a stack frame. This benefit may be offset by the additional storage needed for variables from the copy and by the increased size of the object code at the (now replaced) call site. But because the copy is now exposed to other PL/SQL optimizations, the final result may be *both* faster

[6] An astute reader might notice that the assignment of `3` to `a` is also unnecessary because the value `3` could be passed directly to `dbms_output.put_line` and variable `a` could be eliminated. This final optimization is not done because the PL/SQL compiler has a policy of keeping assignments to user defined variables *even* when they are useless. The policy is in place to help programming tools make better sense of user source code. The policy is regularly reviewed and may change in the future.

[7] `AddMe` is a local subprogram and so it is possible to determine absolutely that it has no remaining calls. If this example had been coded as a package body and if `AddMe` been a function visible through the package specification, then it could not have been removed because there might still be calls from other units to it.

and smaller than the original call and may reduce the size of the entire surrounding PL/SQL unit. Inlining almost always provides a modest execution speed benefit, but it often provides much larger benefits in both speed and size.

The example shows inlining done at the PL/SQL source code level. The PL/SQL compiler actually operates on an internal representation of the program, not on source. But the principles are exactly the same. The compiler does not do any special magic beyond the ideas discussed so far.[8]

When the copy was made, some new variables were created to hold values. These variables had to be in the same declaration scope as the call site and so Listing 2.2 showed them as `ActualArg1`, `ActualArg2`, and `ReturnValueFromCallMe`. Why not just use the old formal parameters `x` and `y`? The reason is that such a use might *capture* existing names in the environment of the call site; what if there were already a variable `x` in the main declaration list of procedure `TheWorld`?[9] The same mistake could be made with local variables declared inside the copied subprogram. The PL/SQL compiler is very careful to invent new variables for the copy and to ensure that these variables do not capture any existing ones.

Earlier, it was noted that the called subprogram might end with an exception. Even more, the called subprogram might include both exception handlers and inner subprogram definitions. Suffice it to say that the copy is very careful to preserve these items so that they work exactly as they would have worked if the called procedure was not copied to the call site. The details are not enlightening, but correct PL/SQL behavior is maintained. Later, the question of choosing which call sites are candidates for inlining is considered. Both the presence of inner exception handlers and of inner subprograms lower the likelihood of a particular call site's selection for replacement precisely because of the complications these two PL/SQL features entail during the copy.

3 Managing Inlining

Inlining is a PL/SQL compiler optimization and its use is managed in two ways.

- Inlining can be applied to an entire PL/SQL unit by setting the PL/SQL optimization level.
- Inlining can be applied to a particular call site with a pragma.

[8]Programmers familiar with other languages probably will notice that inlining is much like the use of *macros*. There are some differences in theory, but the effects are similar.

[9]For those who enjoy language theory, sloppy inlining might convert a *free* variable into a *bound* variable when that would be an error. This is a common error with macros, one of the reasons they are sometimes regarded as dangerous.

PL/SQL optimization is controlled by the initialization parameter `plsql_optimize_level` which has four legal values.

0 Optimization is almost entirely turned off to retain compatibility with some very old Oracle releases. This level should only be used in extraordinary circumstances.

1 Only local optimizations are applied. This level should be used only for code that needs to be debugged using the PL/SQL debugger.

2 Local and global optimizations are applied. This level is the default for the good reason that it typically speeds PL/SQL code up by a factor of 2 to 3 times.

3 Inlining is automatically applied on top of the optimizations provided by level 2. It is likely that most applications should use this optimization level because it probably will improve performance noticeably.

The optimization level is set in all the normal ways (including `ALTER COMPILE`) common in the Oracle database. For example,

```
alter session set plsql_optimize_level = 3
```

turns on inlining for units compiled later in the same session.

Inlining can also be controlled on a call by call basis with the pragma `INLINE`. The pragma occurs before the statement that contains a call site and and requests or blocks inlining of a particular subprogram in that statement. The `INLINE` pragma has two parameters.

Subprogram identifier The identifier for a subprogram that is visible at this point in the program.

Control A `varchar2` value that is `YES` or `NO`. Case is irrelevant.

If the subprogram identifier appears in the statement immediately following the pragma, then the control value is applied to *all* calls of the subprogram in that statement. If the control value is `YES`, the calls to the subprogram that appear in the statement are inlined. If the control value is `NO`, no calls to the subprogram that appear in the statement are inlined. The pragma is effective only when the optimization level is at least 2. A request for inlining (that is, when the control is `YES`) may be ignored for a variety of technical reasons; if it is, a warning will be issued. Other problems with the pragma will also be diagnosed with warnings.

Pragma `INLINE` may also appear before a declaration.[10] There are two cases. If the declaration contains initializers and the pragma names a subprogram invoked in the initializations, then the subprogram will be inlined in the execution of the initializers.[11] If the pragma's subprogram identifier specifies the subprogram being declared, it controls *all* calls to that subprogram (and any overloads) throughout the declaration scope of the subprogram. This allows inlining to be turned on or off for all invocations of a particular subprogram.

Listing 3.1 provides some examples of pragma `INLINE`. Line 9 requests the inlining of the function call of `MyFunc` in the following assignment statement. On line 12, the pragma stops any inlining of the procedure `MyProc` on the following line. Finally,

[10] Remember that a subprogram definition is also a declaration of the subprogram

[11] This does not apply to subprograms that appear in the default initializers of records.

Listing 3.1: INLINE Pragma Examples

```
1
2  pragma inline(MakeIt, 'YES');
3  x pls_integer := MakeIt(27)/3;
4
5  pragma inline(NotAChance, 'YES');
6  function NotAChance(s varchar2, f number);
7  function NotAChance(i pls_integer);
8
9  pragma inline(MyFunc, 'YES');
10 a := b + MyFunc(1, c*d);
11
12 pragma inline(MyProc, 'NO');
13 MyProc(1, 'abc', x != y);
14
15 pragma inline(AnotherFunc, 'YES');
16 pragma inline(AnotherProc, 'YES');
17 pragma inline(NotAChance,  'NO');
18 AnotherProc(AnotherFunc(x, 2), NotAChance('def', 1.23))
```

the three pragmas beginning on line 15 request the inlining of the calls of function `AnotherFunc` and procedure `AnotherProc` but stop the inlining of function `NotAChance`. Notice that there are three pragmas controlling one statement; pragmas themselves are not statements and so the control passes through them until a statement is found. Remember, though, that requests to inline a subprogram are just that: requests; they may not be honored. A pragma that blocks inlining will always be honored.

On line 2, pragma `INLINE` applies to the declaration of a variable `x` and causes the call to function `MakeIt` to be inlined into the initialization expression computation. The `INLINE` pragma on line 5 requests that all calls to function `NotAChance` be inlined. There are two things to note here. First, the request applies to both overloaded versions of `NotAChance`. Second, the `INLINE` pragma on line 17 cancels the general request to inline function `NotAChance` so that it does not apply to the next statement.

4 How Inlining Works

When the PL/SQL optimization level is at least 2, the inline mechanism is activated. Each subprogram call site is a candidate for inlining. What determines whether a particular call is replaced?

The first difference is between optimization levels 2 and 3.

Level 2 Only call sites or subprograms with pragma `INLINE` are candidates for inlining.

Level 3 All subprogram call sites are candidates for inlining.

Once the initial candidate list is generated, inlining operates the same way at each

optimization level.

Once the candidates have been listed, an inlining *budget* is created. Every replacement (potentially) enlarges the source code where the replacement occurs. The *cost* of the replacement is some measure of the increase in size. One might, for example, charge a cost measured in source lines of the replaced subprogram.[12] The budget is decremented each time a replacement is made and once the budget is exhausted, no more replacements are done. The PL/SQL compiler currently has a budget that is equal to the size of the unit being compiled; this allows inlining to double the unit's size before inlining is stopped. It should be said that it is possible to exceed the budget in contrived examples, but that real world units seldom come close to doing so.

Once the candidate list is built, a particular candidate must be selected. The choice is made by *scoring* the candidates. Each call site is given a score that may grow arbitrarily large; the site with the greatest score will be chosen for inlining. The score is made up from several factors and the various factors may be weighted differently. Here are some of the factors.[13]

Size of subprogram The smaller the subprogram to be inlined, the better.

Constant arguments If some actual arguments are constants, the better because they generally provide more possibilities for other optimizations.

Number of arguments The more actual arguments, the better.

Inner calls The fewer inner calls, the better.

Definition depth The more deeply nested the subprogram definition, the better.

Nesting depth The more deeply nested in loops the call site, the better.

Only call If this is the only call site for the subprogram, it is preferred because the subprogram can be removed after inlining is complete.

Pragma If the `INLINE` pragma requests inlining, a huge bonus applies. If the pragma stops inlining, the score is adjusted so that this call site can never be inlined.

Once all the call sites have been scored, the site with the highest score is inlined. This has two effects. First, the budget is decreased by the cost of the inlining; if the budget ever drops to zero or below, inlining is stopped. Second, the copied code may introduce some new call sites because the body that was copied had subprogram calls within it. These new call sites are added to the list of candidates for the next round of scoring.

Inlining always stops when the budget is exhausted. It may stop before if there

[12] In fact, the cost is measured in terms of *operations* needed to execute the subprogram. This is an internal PL/SQL compiler measure, but it is almost exactly proportional to the number of source code elements (identifiers, operators, reserved words, and the like) used to write the subprogram.

[13] The factors used in the score and their weights may be changed from time to time as experience with inlining suggests that modifications to the PL/SQL compiler would improve its ability to make intelligent inlining choices.

are no suitable call sites left to inline.

5 Limitations

Not all requests for inlining are honored and not all call sites are eligible for inlining. One limit on inlining has already been described: if the inlining budget is exceeded, no further inlining will be done. It is unlikely that this is an issue in real world programs, but it can occur. At optimization level 3, once the budget is exceeded, the compiler silently continues with its other work. But if an inline request is made by the pragma, the failure to inline the call site is noted with a warning.

Sometimes a particular subprogram cannot be inlined or a particular call site is not eligible for inlining because of the structure of the subprogram or the context of the call site. These limits exists because the PL/SQL compiler uses mechanical rules to ensure that each inlining operation is safe. This is a positive check; safety must be assured before inlining can take place. The rules simply may not cover all possible situations perhaps because the rules would become too complicated and costly or perhaps because some new feature has been added and the rules have not yet caught up. Such "holes" in the analysis for inlining are unlikely to have much effect in real world programming and may change from time to time as PL/SQL changes and as the compiler improves.

The most important limit is that only subprograms defined in the current unit can be inlined in the unit. These are known as *local* subprograms. Other subprograms cannot be inlined. Listing 5.1 provides some examples. On line 3, procedure `TallDarkAndHandsome` is declared in package `Foreign`. Package `Myself` defines public procedures `BlandBlond` and `InTheMirror` beginning on lines 7 and 8. In the body of package `Myself`, there is a local procedure `WhatAHunk` defined beginning on line 13. And, of course, there are several call sites for the subprograms. Which of these call sites could be inlined?

The call to procedure `BlandBlond` on line 15 may be inlined because `BlandBlond` is defined in this unit on line 18 – and, of course, it was declared in the package specification on line 7. Similarly, the calls to `BlandBlond` on lines 25 and 31 may also be inlined for the same reason. The call to `InTheMirror` on line 33 follows the same pattern and so it may be inlined. The call to `WhatAHunk` on line 27 may be inlined because that local procedure is defined (and, in passing, declared) on line 13. But the calls to `TallDarkAndHandsome` on lines 26 and 32 may not be inlined because that procedure is declared in another package specification on line 3 and must be defined in another package body. Because of its familiarity and because it feels like part of the PL/SQL language (like many other supplied subprograms), the call to `dbms_output.put_line` on line 20 may seem to be an inlining candidate. It is

Listing 5.1: Inlining Limitations

```
1
2  package Foreign is
3    procedure TallDarkAndHandsome;
4  end Foreign;
5
6  package Myself is
7    procedure BlandBlond;
8    procedure InTheMirror;
9  end Myself;
10
11 package body Myself is
12
13   procedure WhatAHunk is
14   begin
15     BlandBlond;
16   end;
17
18   procedure BlandBlond is
19   begin
20     dbms_output.put_line('I like cottage cheese');
21   end;
22
23   procedure InTheMirror is
24   begin
25     BlandBlond;
26     Foreign.TallDarkAndHandsome;
27     WhatAHunk;
28   end;
29
30 begin
31   BlandBlond;
32   Foreign.TallDarkAndHandsome;
33   InTheMirror;
34 end Myself;
```

a subprogram call just like that to `TallDarkAndHandsome`. But it is also barred by the same rule against foreign subprograms.

Why is there a restriction against inlining foreign subprograms? Because PL/SQL has an automatic validation mechanism for all programs. Whenever one unit references an item in another unit, there is a dependency created between the units. Whenever a unit is known to be invalid, the PL/SQL system attempts to recompile it so that it becomes valid again. For example, in Listing 5.1, the call to `TallDarkAndHandsome` on line 26 creates a dependency from package body `Myself` to package specification `Foreign`. If, for example, the anonymous block

```
begin Myself.InTheMirror; end;
```

were to be executed, the first step would be to ensure that the package specification and the package body of `Myself` were both valid. But that check cannot be completed until it is certain that the package specification for package `Foreign` is also known to be valid. Thus the use of items from one unit in another creates a chain of dependencies that all must be checked – and, if necessary, revalidated – before the starting point can be executed.

Notice that the dependency link is from a use of an item (a subprogram call site,

for example) to the declaration of that item as either a top-level subprogram definition or as an element of a package specification.[14] So long as the dependency link is on the package specification and there is no change in that specification, the corresponding package body may be modified and recompiled repeatedly *without causing the invalidation* of the unit containing the call to the subprogram.

If inlining could be done across unit bodies[15] (that is, globally), then there would be an additional dependency between the call site and the package body where the implementation of the called routine resided. Whenever that package body changed, the caller would also have to be recompiled because the caller would be invalid. While this does not make the notions of dependency and validation any less useful nor does it break the theory behind them, it would make the frequency of unit compilation much higher than it is today. Every bug fix in a package body could easily trigger a giant cascade of recompilations whereas, under the current regime, only changes in visible specifications trigger recompilations.[16]

In summary, the benefits of global inlining do not seem to balance the costs of increased compilation caused by the greater dependency between units.

- Inlining of calls across PL/SQL units (global inlining) could be done. There is no theoretical bar to doing so.
- Global inlining would add many new dependency links to the PL/SQL validation mechanism. The effect would be to cause many more compilations of units because of changes (particularly in package bodies) that currently do not trigger revalidations.

Any demonstration of a real world program that had significant performance improvements from global inlining would be a reason to reconsider the cost/benefit analysis.

6 Compilation Information

Whenever inlining operations are requested or occur, the PL/SQL compiler provides information about what happened. The reports are warnings and informational messages from the compiler and they appear if PL/SQL warnings have been enabled.

Listing 6.1 gives some examples of inlining informational messages. Line 1 is the message that reports an instance of inlining. Here it reports that a call to `Add_Numbers` has been replaced. The message on line 2 says that `Add_Numbers` has been removed

[14] In practice, there are very few top-level subprograms defined or used in PL/SQL.

[15] Again, the only interesting units are package bodies.

[16] And with the use of fine-grained dependency analysis, recompilations are even rarer because only a change that could cause a problem actually triggers the recompilation; an irrelevant change (say, to a comment) is ignored.

Listing 6.1: Inlining Information

```
1  err 6005, sev 3, (13,3) inlining of call of procedure 'ADD_NUMBERS' was done
2  err 6006, sev 3, (6,3) uncalled procedure "ADD_NUMBERS" is removed.
3  err 6002, sev 3, (19,26) Unreachable code
4  err 6004, sev 3, (13,3) inlining of call of procedure 'ADD_NUMBERS' requested
5  err 5012, sev 3, (13,3) pragma INLINE for procedure 'ADD_NUMBERS'
6                           could not be applied
7  err 6008, sev 3, (25,3) call of procedure 'ADD_NUMBERS' will not be inlined
8  err 5011, sev 3, (24,3) pragma INLINE for procedure 'ADDNUMBERS'
9                           does not apply to any calls
10 err 5008, sev 3, (24,10) illegal number of arguments for pragma INLINE
11 err 127, sev 1, (24,10) Pragma INLIN is not a supported pragma
```

entirely from the source of the program. This happened because all the calls were inlined and so no calls remained. While this is not strictly an inlining message (it can happen to any uncalled subprogram), it happens quite commonly when inlining is turned on. Line 3 reports that some code is unreachable. Again, this message can occur whether inlining is turned on or not, but it is much more common when inlining is active. In this case, inlining exposed some constants to the PL/SQL optimizer and the compiler was able to determine that the Boolean expression controlling an `IF` statement was always `FALSE`. This allowed the code in the `THEN` branch to be deleted as unreachable. One of the major virtues of inlining is that such optimizations become much more likely.

The message on line 4 appears when pragma `INLINE` appears in the source code. Line 5 reports that a call site that has had a pragma `INLINE` request cannot, in fact, be inlined. There may be several reasons for this, but the most likely is that the inlining budget has already been exhausted and so no more inlining can be done. When pragma `INLINE` turns off inlining for a particular call site, the message on line 7 may appear, depending on the optimization level. If pragma `INLINE` names a subprogram that does not appear in the following declaration or statement, the message on line 8 appears. When pragma `INLINE` has the wrong number of arguments, the message on line 10 appears. Finally, a typing error causes the error message shown on line 11. Syntactic errors (dropping the trailing semicolon, for example) cause the same kind of error messages as other PL/SQL mistakes would cause.

7 Inlining Guidance

Inlining is a powerful optimization. It can speed particular programs up dramatically. In real programs, many inlining opportunities exist. Most programs have small "helper" subprograms that are called often to provide some simple service. These are excellent candidates for inlining because the call overhead can be avoided. Also, many

programs include large subprograms that are called only once. These are also good candidates for inlining. Quite commonly, the reason for writing a subprogram that is called at just one place is to improve software clarity and reliability. If the subprogram is inlined, the advantages of "in place" execution become available without any need for the programmer to damage the neat logical structure of the program.

The advantages of inlining are apparent. There are two known disadvantages.

- Done to excess, inlining can cause object code bloat.
- Compilation times are generally somewhat longer because each copy of an inlined subprogram requires its own compilation work above that needed for the original subprogram.

Testing suggests that PL/SQL packages to which inlining is applied often have *smaller* object code than their original versions. In other words, optimization level 3 can reduce total storage space. And, because of the budget mechanism explained earlier, the size increase can never become too large. Compilation time for a single unit on modern PL/SQL systems is so fast that the small additional time taken for inlining is almost undetectable.

When the PL/SQL compiler first offered inlining, a policy choice was made to allow users complete control over the operation. Thus, pragma `INLINE` and PL/SQL optimization level 3 make customization of inlining easy and convenient down to the individual call level. Every application can apply the detailed inlining appropriate to its own situation. Combined with the information messages about inlining, users can be very comfortable with their use of inlining and their understanding of its effects.

With all this in mind, how should inlining be used for production applications? The basic recommendation is that all units should be compiled at optimization level 3 – that is, with inlining on – and that pragma `INLINE` should be used sparingly to turn inlining off for those few call sites where it can be proven to be detrimental. Obviously, particular units might also be compiled at optimization level 2 rather than 3 if there is some specific and demonstrable reason that the entire unit should not have inlining applied.

Inlining is a safe and effective optimization that almost always improves application performance and has no important disadvantages in its PL/SQL implementation. Inlining should be routinely applied in every development and production environment. Finally, as with other aspects of PL/SQL program development, keeping warnings enabled at all times will improve understanding of the effects and benefits of inlining.