PL/SQL JUST GOT FASTER

Bryn Llewellyn, PL/SQL Product Manager, Oracle Corporation
Charles Wetherell, Consulting Member of Technical Staff, Oracle Corporation
Håkan Arpfors, Senior Software Architect, IFS AB

Delivered as paper # 40168 at Oracle OpenWorld, San Francisco, September 2003

Note: This copy accompanies the slides that were used at the OracleWorld presentation, San Francisco, Thursday 11-Sep-2003. The paper will be revised in the light of feedback from that session, and later to reflect performance data from the production version of Oracle Database 10g. Please check out this page on OTN:

`otn.oracle.com/tech/pl_sql/htdocs/New_In_10gR1.htm`

We will post the latest version there.

0. INTRODUCTION

Substantial changes have been made in Oracle Database 10*g* to the system for compiling and running PL/SQL programs in the database, resulting in dramatic performance improvements. This paper describes the general nature of these changes, and presents some performance results which come from four distinct sources:

- Tests using the benchmark suite developed and owned by the PL/SQL Language Development Team at Oracle HQ.
- Customer tests conducted during the 10*g* Beta Program by IFS.
- Tests using the benchmark suite developed and owned by the Applications Development Team at Oracle HQ.
- Tests using a test suite which is downloadable for customers to use from the OTN site.

They all tell the same story.

PL/SQL is a programming language with two purposes: first, to access SQL and second, to write procedural programs ranging in size from small fragments all the way up to complete applications. When PL/SQL is a carrier for SQL statements, the performance of the PL/SQL portion of the combination is generally not very important; execution of even the simplest interesting SQL operation takes much more time than does the typical PL/SQL statement. When PL/SQL is used as a conventional programming language, then its performance is important, just as is that of any other programming language. This paper addresses this second use case only.

We'll use the terms 8*i*, 9*i*R1, 9*i*R2 and 10*g* for brevity in the following with these meanings:

- 8*i* – Oracle8*i* Database version 8.1.x
- 9*i*R1 – Oracle9*i* Database Release 1 version 9.0.x
- 9*i*R2 – Oracle9*i* Database Release 2 version 9.2.x
- 10*g* – Oracle Database 10*g* version 10.1.x

1. IMPROVEMENTS TO THE PL/SQL SYSTEM

We'll review here how PL/SQL is compiled and executed in the database and against that background explain how changes have been made in both the front-end and the back-end of the compiler and in the architecture and the implementation of the PL/SQL virtual machine.

This will provide a framework to point out some other 10*g* enhancements in PL/SQL, but these are not the focus of

the present paper.

1.1. PL/SQL COMPILATION AND EXECUTION

At a high level, the following is applicable to the compilation and execution of a wide class of programming languages (especially interpreted ones like for example Java).

1.1.1. THE FRONT-END

This analyzes the source code for a single PL/SQL compilation unit, checking it for syntactical and semantic correctness. (A classic example of a semantic error is a syntactically correct reference to an element in a non-existent package.) The output of the front-end is either an internal representation which exactly captures the source code's semantics, or an error report. You sometimes hear the name *Diana* (for *Descriptive Intermediate Attributed Notation for Ada*) for this internal representation, reflecting PL/SQL's historical debt to the Ada language.

The front-end guarantees that when it doesn't report an error, the Diana it outputs is correct and needs no further downstream correctness checking.

The front-end always needs to be enhanced when new language features (like the *CASE* statement) are introduced. Depending on the new feature, it may be that the changes needed to support it are confined to the front-end.

1.1.2. THE BACK-END (A.K.A. CODE-GENERATOR)

This consumes the *Diana* and generates an executable representation of the program in the machine code of the target machine. Pre-9*i*R1, the output was always code for the PL/SQL Virtual Machine (*a.k.a. PVM*). 9*i*R1 introduced the option (via *native* compilation) to output code for the hardware of the underlying machine. You sometimes hear the term *MCode* used for the output of the code-generator that runs on the PVM.

The code-generator is critically responsible for producing efficient run-time code.

The terms *Diana* and *MCode* are both used occasionally in the Oracle Documentation. Neither is usually in the vocabulary of ordinary mortal PL/SQL programmers (unless size limits are encountered) but it will be convenient to use them in this paper.

1.1.3. THE PL/SQL VIRTUAL MACHINE

The instruction set of the PVM is exactly analogous to that of a computer chip. The difference is of course that the PVM is implemented in software (due to code written in C and linked into the ORACLE executable) while the instruction set of a chip is implemented directly in hardware. The term *virtual machine* often goes hand in hand with the term *interpreted language*. Both PL/SQL and Java are compiled to run each on its specific virtual machine. And this is for the same reason: to simplify the task of implementing the language on a new hardware platform. Note though that neither for PL/SQL nor for Java is the actual source code interpreted at run-time. Both execute compiled machine code on a virtual machine implemented in software.

Just as is the case for the instruction set for any chip, the PVM's set has been designed to support the efficient execution of the kinds of program it typically runs. And just as with chips, successive revisions of the Oracle database bring changes to the PVM and for the same reasons: greater efficiency, both generically and specifically for common source-code idioms.

It's helpful to distinguish between two kinds of PVM change: *architecture* changes, *i.e.* additions of new or removal of old instructions together with changes in how these are consumed; and *implementation* changes, *i.e.* changes to how the instructions themselves are implemented (mapped ultimately to actual machine instructions).

1.1.4 NATIVE COMPILATION OF PL/SQL

The back-end can generate its output code in two different representations. In both cases it consumes the Diana for the current PL/SQL unit and computes the MCode. In the *interpreted* mode, it simply stores the MCode in system-managed structures in the *SYS* schema. In the *native* mode, it translates the MCode into *C* source code with the same semantics. (The MCode is not retained in this mode.) Broadly speaking, the housekeeping tasks that the PVM does on the fly (setting up and destroying temporary variables, managing stack frames for subprogram invocation, and so on) are now expressed at compile time in *C*. And the real work that the PVM does on the fly (by branching to call the appropriate routine that implements the current instruction) is expressed, again now at compile time, by calling that

same routine from the generated *C*. The generated C is translated for execution as a dynamically linkable shared library for the ORACLE executable by automatically calling the appropriate platform-specific tools. We'll call this a *DLL* for brevity. (The term varies from platform to platform.) New in 10*g*, this DLL is stored, just as the MCode is, in system managed structures in the *SYS* schema.

Thus a given PL/SQL unit will execute faster when compiled native than when compiled interpreted, precisely and only because of the increased efficiency of the housekeeping tasks. The real work remains the same.

1.1.5. Run time considerations

The object code for a PL/SQL unit (whether MCode or a native DLL) is fetched on demand according to execution flow. The run-time system discovers just in time if the unit was compiled interpreted or native, and sets it up for execution accordingly. The two flavors of object code are fully interoperable. Following the explanation above, the greatest benefits from native compilation will be seen when both the *caller* and the *callee* PL/SQL unit are compiled native, even when the real work each or both does is trivial.

1.2. How to make PL/SQL programs run faster

Obviously the performance characteristics of a PL/SQL unit are influenced by how the author chooses and expresses the algorithm in PL/SQL. Without a psychic subsystem in the front-end, nothing can be done to determine the programmer's ultimate high-level intent! These two programs to calculate the *n*th term in the Fibonacci series illustrate the point:

```
function Fib_N ( n in pls_integer ) return pls_integer -- Recursive solution
is
begin
  return
    case n
      when 1 then 1
      when 2 then 1
      else        Fib_N(n - 1) + Fib_N(n - 2)
    end;
end Fib_N;

function Fib_N ( n in pls_integer ) return pls_integer -- Iterative solution
is
  current       pls_integer;
  last_but_one  pls_integer;
  last          pls_integer;
begin
  case n
    when 1 then
      return 1;
    when 2 then
      return 1;
    else
      last_but_one := 1;
      last         := 1;
      for j in 3..n
      loop
        current      := last_but_one + last;
        last_but_one := last;
        last         := current;
      end loop;
      return current;
  end case;
end Fib_N;
```

The reader can see that both programs produce the same result for all values of *n*, and that the intent is therefore identical. The iterative solution is of orders of magnitude faster, but the transformation from the recursive solution to the iteratative solution is way beyond the limits of current compiler technology.

However, offline human analysis of extant PL/SQL programs combined with a general understanding of the intent of

a wide range of typical tasks can lead to the invention of new language features. This typically happens for every major Oracle Database release. (The introduction in 9*i*R2 of the *index-by-varchar2* table provides a compelling example.) We'll list the changes for 10*g* below.

The user who takes advantage of new language features will see a performance improvement for appropriate use-cases.

The efficiency of the code that that back-end generates (roughly speaking the number of instructions that it produces to represent a particular Diana) critically determines the performance of the program. This is the concern of the PL/SQL Development Team, and is completely outside of the user's control (except in so far as the code-generator exposes some knobs). There are huge opportunities here for radical improvement from release to release.

The efficiency of the routines that implement the PVM (relevant therefore for both the interpreted and the native modes) also determines the performance of the program.

The relative potential for improvement in program performance due to changes in the code-generator or in the PVM can be understood by analogy with what determines the speed of a car journey. The choice of route (as the crow flies or hugely circuitous) is analogous to the code generation phase. And the speed at which you drive is analogous to the expressiveness and the efficiency of the PVM. Famously, choosing the optimal route, even when you drive conservatively in a modest car, gets you there much quicker than choosing a poor route and driving flat out in a sports car. Of course, driving a sports car flat out on the optimal route is quickest of all.

1.3. WHAT CHANGES WERE MADE IN 10*g*?

1.3.1. FRONT-END SUPPORT FOR NEW LANGUAGE FEATURES

These changes are necessarily *evolutionary*. 10*g* introduces support for these new language features:

- the *binary_float* and *binary_double* datatypes (*a.k.a.* the IEEE datatypes)
- the *regexp_like, regexp_instr, regexp_substr* and *regexp_replace* builtins to support regular expression manipulation via standard POSIX syntax
- multiset operations on nested table instances, supporting operations like *equals, union, intersect, except, member,…*
- the user-defined quote character
- *INDICES OF* and *VALUES OF* syntax for *FORALL*

Of course these features have an efficient implementation. For example, the IEEE datatypes enjoy the benefit of machine arithmetic for mathematical real numbers. Thus it's immediately obvious that if an appropriate algorithmic task is expressed in PL/SQL using any of these new features, then it will run very much faster than one which avoided them.

Note: all but the *FORALL* features in the above list are formally speaking new features in SQL. PL/SQL has a moral obligation to support all such SQL features in a hermetic PL/SQL context (for example in PL/SQL assignment statements). This is a unique strength of PL/SQL and is closely related to the fact that it shares the same datatype system as SQL. Of course the PL/SQL Language Development Team has to do work to make this happen, but this is transparent to the user.

Finally in this section, we need to explain the change in the regime for integer arithmetic using the PL/SQL datatypes *binary_integer* and *pls_integer*. That we have two datatypes for nominally the same purpose is due to historical reasons. The high order purpose is to represent mathematical integers compactly (*i.e.* in 32 bits) and to implement their arithmetic operations optimally by using machine arithmetic. The SQL datatype *integer* is in fact a constrained subtype of *number*, and thus even in the context of a pure PL/SQL program its representation is rather large (roughly speaking a binary coded decimal in several bytes) and its arithmetic operations are implemented by *C* routines (*a.k.a. library arithmetic*) rather than in hardware. Both these features of *integer* are required to support the semantics defined in the SQL standard, which semantics are very often not needed in a typical PL/SQL program! Unfortunately, pre 10*g*, *binary_integer* and *pls_integer* meet the stated purpose to different extents. The younger *pls_integer* (younger in the history of PL/SQL) is represented in 32 bits and its arithmetic operations are implemented in hardware. The older *binary_integer* is also represented in 32 bits but its arithmetic operations are implemented using the library software

routines for *number*. And again unfortunately, the PL/SQL programmer is not always free to choose the ideal *pls_integer*. For example, he may need to call extant subprograms whose formals are declared as *binary_integer*. Or he may want to use the constrained subtypes *naturaln* and *positiven* precisely to enjoy the benefits of the compiler generated constraint checking. It turns out that *naturaln* and *positiven* are subtypes of *binary_integer*!

10*g* removes this confusion by redefining *binary_integer* to be identical to *pls_integer*. This has both a usability benefit and an immediate performance benefit for extant programs that accidentally or unavoidably use *binary_integer*. The effect is to speed up integer arithmetic in PL/SQL by a factor of 2.5 to 3. Further, the rules for integer arithmetic are now clear and understandable.

1.3.2. BRAND NEW CODE-GENERATOR USING STATE-OF-THE-ART OPTIMIZING TECHNOLOGY

This change is *revolutionary*. Pre-10*g*, the code-generator was relatively mechanical. Roughly speaking, what you wrote in PL/SQL is what ultimately the PVM executed for you. If you gave instructions which amounted to going around the houses to get from A to B, then your program followed that circuitous route at run time. This old code-generator has been surgically removed and replaced by a brand new one that has the intelligence to spot that there's a direct route from A to B, and to produce the correspondingly more efficient MCode.

In this analogy, it is the result – getting to B from A – that counts. How you get there doesn't matter. Thus what exceptions you might raise if you followed an indirect route, and could not raise by following the direct route, are within the terms of reference of an optimizing compiler *defined* not to matter. It follows that specially contrived programs can be written so that their 10*g* behavior is detectably different from their pre-10*g* behavior. Having said that, no such cases have been detected in extensive regression testing – and most would agree that such programs are badly written. This will be the subject of a detailed technical white paper that we'll publish on OTN.

The essential characteristic of an optimizing compiler is that it takes as broad a view as possible of what the source code (in our case represented as Diana) expresses and removes any computations which it can prove are not necessary to guarantee the final correct result. This is what gives the potential for huge performance benefits.

The new code-generator has a knob to control the effort it applies to eliminating superfluous computations. This is exposed as a new instance parameter *plsql_optimize_level*, and has the allowed values 1 and 2. The latter means apply more effort and it does therefore imply slightly longer compilation time. Nevertheless it is the default, and Oracle Corporation recommends using level 1 only in situations where this becomes critical (for example the run-time compilation of dynamically generated trivial PL/SQL).

By the way, the new code-generator has existed side-by-side with the old one in the ORACLE executable for quite some time, and a switch has allowed choosing between the one or the other. This is possible because the input and the output for this subsystem are precisely defined. The switch has not been exposed for customers, but the huge suites of regression tests used by various development groups at Oracle HQ have routinely been run under both regimes during this period of co-existence. This allows us to be completely confident about the semantic correctness of the new code-generator.

1.3.3. SUBSTANTIAL PVM UPGRADE

The PVM has been significantly reworked in 10*g* in all of the ways outlined above.

- Obsolete instructions have been removed, and new ones have been added. For example pre 10*g*, an assignment like this

  ```
  s := 'a' || 'b' || 'c' || ... || 'z';
  ```

  was implemented as a series of pairwise concatenations with corresponding verbose housekeeping of generated temporaries. Such concatenation is an extremely common PL/SQL idiom (for example in connection with populating actual parameters for *Htp.Print*). 10*g* introduces new PVM instructions specifically to support one-shot concatenation of many terms. Another example, essential to support the front-end enhancement to support *binary_float* and *binary_double*, is the new instructions that support the IEEE conformant arithmetic operations of these directly via hardware arithmetic.

- The system for consuming the PVM instructions has been streamlined (analogous to a more efficient scheme for handling registers in a new revision of a chip).
- The C routines that implement the instructions have been tuned (analogous to basing a new revision of a chip on smaller and therefore faster integrated circuitry).

Note that there's a reciprocal relationship between code-generator enhancements and machine enhancements, both in general and also for PL/SQL execution. Design work for the code-generator can call out the requirement for new PVM instructions. And changes to the PVM motivated by design work within that domain can call out changes that could be made with advantage to the code-generator. In fact the new code-generator has been under design and construction since the release of Oracle 8.0. Though it has been introduced by revolutionary change now in 10*g*, PVM changes due to the same project have been "leaked" into successive Oracle releases from 8*i* onwards.

1.3.4. CHANGES IN THE REGIME FOR NATIVE COMPILATION

Pre 10*g*, the DLL generated by the back-end was stored canonically as a file on an operating system directory. This meant that special (manual) consideration had to be given to online backup, and that PL/SQL native compilation and RAC were interoperable only when the platform supported a genuine shared file system. In 10*g*, the DLL is stored canonically in the database and is cached as a file on an operating system directory only on demand. Of course all the necessary cache invalidation support is in place so that all instances participating in a RAC configuration will see the new copy of a DLL when a particular PL/SQL unit is recompiled native, exactly as they see the corresponding new copy of the MCode when the unit is recompiled interpreted.

The configuration steps that the DBA follows to set up for native PL/SQL compilation have been radically simplified. The dependency on the platform's make utility has been removed. In 10*g*, the corresponding logic is implemented directly in the ORACLE executable, and the requirement for configuration data has thus been correspondingly simplified. The only surviving degrees of freedom for DLL generation are the locations of the C compiler and linker. These are given in a configuration file which must be located at a non-negotiable location in the $ORACLE_HOME tree. The only supported tools are those supplied by the platform vendor, and these are typically in a standard locations, so the DBA is very unlikely to need to access the configuration file.

Thus the instance parameters *plsql_native_make_utility, plsql_native_make_file_name, plsql_native_c_compiler, plsql_native_linker* are now redundant.

And as a usability enhancement, the choice of *native* or *interpreted* mode is now governed by the new instance parameter *plsql_code_type*. (The old *plsql_compiler_flags* is retained for backwards compatibility.)

The subsystem that derives the *C* code from the MCode has been reworked to generate more efficient *C*, and of course to acknowledge the new *C* routines that implement the new PVM instructions.

10*g* also provides supported utilities to convert all PL/SQL units in a database to *native* or to *interpreted* to make it simple to gain the benefits of native execution.

2. PERFORMANCE RESULTS

The goal of all the above changes is to speed up computational PL/SQL programs. The goal was to achieve a factor of roughly two with respect to Oracle 8.0. (Recall that the new PL/SQL code-generator project was conceived when this version was current.) The only way to know if that goal is met is by measuring PL/SQL execution performance.

2.1. THE PL/SQL LANGUAGE DEVELOPMENT TEAM'S BENCHMARK SUITE

This consists of a suite of twelve programs designed to cover a wide range of common PL/SQL programming tasks. Consistent with the aim of improving purely computational PL/SQL programs, these are independent of a connection to a database, except in as much as this is necessary in a production Oracle system to provide the operating environment. Various instrumentation techniques have been implemented in addition to the obvious measurement of completion time for a program. For example, both the number of PVM instructions and the number of actual machine instructions that are executed when the test program runs are counted. An algorithm has been developed to combine all these metrics to a single figure of merit.

This benchmark suite is known internally as *PLSMarks*. Clearly the type of testing it implements is possible only by the

by the PL/SQL Language Development Team.

Improvement from release to release is characterized by the ratio of the figure of merit at the later release to that at the earlier release. This will of course always be greater than one, since the benchmark suite is used for routine regression testing!

The following table shows the improvement ratio for 10*g* (Beta2 version) relative to Oracle 8.0.6 and to Oracle9*i* Database Release 2 version 9.2.0 at the two available values for *plsql_optimize_level*. In all cases, the measurements were made using the interpreted mode. The ranges represent the lower and upper limits of the statistical confidence band.

	Level 1	*Level 2*
8.0.6	2.0 – 2.4	2.2 – 2.6
9iR2	1.5 – 1.6	1.5 –1.7

We can deduce from these numbers that 9*i*R2 is already better then 8.0.6 by a factor of about 1.4 by virtue of the incremental release of PVM improvements as explained above.

2.2. Oracle Database 10*g* in Action: Performance Data from IFS AB Measured During the Beta Program

IFS (*www.ifsworld.com*) develops and supplies component-based business applications for medium and large enterprises.

IFS Applications, which is based on web and portal technology, offers 60+ enterprise application components used in manufacturing, supply chain management, customer relationship management, service provision, financials, product development, maintenance and human resource administration. IFS offers customers an easier, more open alternative that can be implemented step by step.

A leading global business applications supplier, IFS has 3,000 employees, with sales in 45 countries, and more than 350,000 users worldwide. The company is listed on the Stockholm Stock Exchange (XSSE: IFS).

IFS Applications use Oracle extensively and most of the business logic is written in PL/SQL. The application has about 3000 tables, about 5000 views and about 5000 packages with about 4.2 million lines of PL/SQL code. All PL/SQL code is executed in the database.

At the time of writing, IFS is actively participating in the 10*g* Beta Program. They have conducted extensive PL/SQL performance tests. The results are presented in *Appendix A*.

They conducted two types of tests…

- They timed two specially written pure PL/SQL test programs which emulate the kinds of computation (especially string manipulation) that are typical in their applications.
- And they timed a mechanical scenario which repeatedly executes a large number of distinct business transactions to emulate the profile of a typical workload. These transactions of course spend appreciable time in SQL.

They confirmed that the results from these programs were identical in 10*g* and in 9*i*R2.

The timings from the pure PL/SQL tests showed an overall speed improvement, comparing 9*i*R1 *interpreted* with 10*g* *native level 2* ranging between *a ratio of 1.76* and *a ratio of 2.59*. They reported…

- *"We are very excited about the figures we saw in our initial tests."*

The timings from the mixed SQL – PL/SQL scenario showed an overall average speed improvement of about 18% (*i.e.* a *ratio of 1.18*). They reported…

- *"We are excited over the figures. This overall performance increase is really good for our customers."*

2.3. The Oracle Corp Applications Development Team's benchmark suite

The Applications Development Team at Oracle HQ has several benchmark suits that model a range of typical

scenarios, covering for example simulated high concurrency manual data entry and end-of-month payroll batch processing. The benchmarks yield appropriate figures of merit, typically throughput in transactions per second. (The batch processing is inherently parallelizable.)

They also have instrumented programs in a test environment to isolate the time spent in PL/SQL for compute-intensive processing.

They have informally provided us with some preliminary results, measured for 10*g* Beta2 *versus* 9*i*R2.

The improvement factor for throughput for the simulated data entry processing is in the 1.10x to 1.15x range. And the improvement factor for pure PL/SQL processing is on the order of 2x.

2.4. RESULTS FROM THE DOWNLOADABLE PERFORMANCE TEST HARNESS

The kit will be downloadable from here:

```
otn.oracle.com/tech/pl_sql/htdocs/New_In_10gR1.htm
```

It provides: a suite of test programs; a harness to time them in various environments and to record the timings in machine-readable form; and a utility to analyze the timings and to produce a nicely formatted report. It's an informal equivalent to the *PLSMarks* suite. It differs in ambition level (some of the test programs were chosen because they dramatically illustrate an improvement that's directly attributable to a specific enhancement in PL/SQL) and in technical sophistication (the metrics are restricted to timings). However, it does have the advantage that it's transparent and can be used by a customer on their equipment after a fully mechanical install. Moreover, it's straightforward to add one's own programs (and of course to remove the shipped ones).

The downloadable kit includes the performance report generated at Oracle HQ for the set of supplied test programs.

The mechanical testing explores these degrees of freedom:

- *Oracle version:*
  - Oracle8*i* Database version 8.1.7.4
  - Oracle9*i* Database Release 2 version 9.2.0.3
  - Oracle Database 10*g*, version 10.1.0.1 – Beta2
- *plsql_code_type:*
  - *interpreted*
  - *native*
- *plsql_optimize_level:*
  - *1*
  - *2*

The kit is described fully in a separate paper, included with the kit itself. That paper presents and interprets the results. We direct the reader to that paper for all detail, and limit ourselves here to presenting the maximum and minimum values for the improvement factors across the set of test programs.
Note that not all test programs could be run at 8*i* (they had been written originally to showcase new 9*i*R1 and 9*i*R2 features) and so for simplicity all improvement factors are calculated for 10*g* Beta2 compared to 9*i*R2.

The improvement factors are broken down thus:

- *Oracle version:* this represents the difference due to moving from 9*i*R2 to 10*g* with *plsql_optimize_level=2*, holding the *plsql_code_type* constant. Thus it is measured twice (for *plsql_code_type=interpreted* and for *plsql_code_type=native*) and the average is calculated.
- *plsql_code_type:* this is measured three times for 9iR2, for 10*g* with *plsql_optimize_level=2* and for 10*g* with *plsql_optimize_level=2* and the average is calculated.

- *plsql_optimize_level:* this is measured twice (of course only at 10*g*) for *plsql_code_type=interpreted* and for *plsql_code_type=native* and the average is calculated.
- *Best:* this is measured just once (so it's conveniently presented in the table below as its own average) and represents the difference going from 9*i*R2 *interpreted* to 10*g native level 2*.

	Max	Min
Code type (9i, 10g-L1, 10g-L2):	3.10	1.20
Optimize level (10g interpreted, 10g native):	1.10	0.97
Oracle version (interpreted, native):	5.93	1.09
Best:	10.63	1.31

The maximum values are dramatically large! They are due to test programs that were specially contrived to illustrate a specific effect (like for example the speed up of operations on *positiven* and *naturaln* datatypes).

3. CONCLUSION

PL/SQL has indeed just gotten faster, and dramatically so. Improvement factors must always be qualified. Implicitly, we've been considering the improvements experienced by pure PL/SQL programs, and we've seen that the measured improvement factor varies, of course depending on what it's measured relative to, but also on what determining environment factors are changed. The biggest improvement factor would be seen by comparing a program executing at 8.0.6 and at 10*g* with native compilation and level 2 optimization. As explained (because some of the improvements that are due to a single long term development project have been introduced incrementally in successive releases from 8.0.6 onwards) a smaller improvement factor is seen if the baseline is 9*i*R2 with interpreted compilation. And of course smaller still if the baseline is 9*i*R2 with native compilation. Moreover, measuring between any two specified compilation conditions, different programs enjoy different improvement factors.

The explanations above allow all these effects to be readily understood.

Notwithstanding these caveats, we claim it's reasonable to say:

"PL/SQL just got faster – by a factor of 2x"

By virtue of this, we believe that new classes of application that until now have been implemented in a 3GL Oracle client (especially for example written in Pro*C) will enjoy a net speed up if re-implemented in database PL/SQL. Such applications extract large quantities of data from a database, carry out a computationally intensive transformation, and insert large volumes of derived data back into the database.

Though the computations themselves are still slower in PL/SQL than in say C, the speed up in PL/SQL that 10*g* brings will shift the break-even point when movement of data, and in particular its transformation from the Oracle datatype system to that of the 3GL, is also responsible for a large component of the execution time.

APPENDIX A:
PERFORMANCE DATA FROM IFS AB MEASURED DURING THE 10*g* BETA PROGRAM

At the time of writing, customers on the 10*g* Beta program are using the so-called Beta1 drop of the code. In-house tests have shown a small overall improvement (on the order of a few percent) in the Beta2 code compared to the Beta1 code.

A.1. SPECIALLY WRITTEN TEST PROGRAMS

IFS created two pure PL/SQL test programs which emulate the kinds of computation (especially string manipulation) that are typical in their applications. For each of these they measured the elapsed time to completion for each of ten repeat runs under each of the two compilation regimes for 9*i*R1 and the four compilation regimes for 10*g*.

In all other respects, their experiment (of course) followed the same approach as described in the downloadable PL/SQL performance test harness referred to above, and the results are presented here in the same way, identified by the test program names used by IFS.

A.1.1. CLIENT_SYS

			Elapsed time *centiseconds*
9i	Interpreted	n/a	24.8
9i	Native	n/a	19.0
10g	Interpreted	opt_level_1	15.9
10g	Interpreted	opt_level_2	12.0
10g	Native	opt_level_1	10.1
10g	Native	opt_level_2	9.6

	9i I	9i N	10g I L1	10g I L2	10g N L1	10g N L2
9i I	1.00	1.30	1.56	2.06	2.46	**2.59**
9i N		1.00	1.19	1.58	1.89	1.99
10g I L1			1.00	1.32	1.58	1.66
10g I L2				1.00	1.19	1.26
10g N L1					1.00	1.05
10g N L2						1.00

				Avg
Code type (9i, 10g-L1, 10g-L2):	1.30	1.58	1.26	1.38
Optimize level (10g interpreted, 10g native):	1.32	1.05		1.19
Oracle version (interpreted, native):	2.06	1.99		2.02
Best:				**2.59**

A.1.2. MESSAGE_SYS

			Elapsed time *centiseconds*
9i	Interpreted	n/a	37.9
9i	Native	n/a	31.5
10g	Interpreted	opt_level_1	29.3
10g	Interpreted	opt_level_2	24.0
10g	Native	opt_level_1	21.3
10g	Native	opt_level_2	21.6

	9i I	9i N	10g I L1	10g I L2	10g N L1	10g N L2
9i I	1.00	1.20	1.29	1.58	1.78	**1.76**
9i N		1.00	1.07	1.31	1.48	1.46
10g I L1			1.00	1.22	1.38	1.36
10g I L2				1.00	1.13	1.11
10g N L1					1.00	0.99
10g N L2						1.00

				Avg
Code type (9i, 10g-L1, 10g-L2):	1.20	1.38	1.11	1.23
Optimize level (10g interpreted, 10g native):	1.22	0.99		1.10
Oracle version (interpreted, native):	1.58	1.46		1.52
Best:				**1.76**

These figures are comparable to those reported for the PL/SQL performance test harness for those test programs where there are no "special effects" (such as passively enjoying the benefit of the new implementation of *binary_integer* using machine arithmetic or changing the code to use the new *binary_double* datatype in place of *number*) involved.

A.2. Real IFS Applications Scenarios

IFS have a harness that automatically runs a *scenario*. The scenario is series of typical business transactions from their productized applications code, run in sequence. It has about 60 distinct transactions, and the scenario runs each one many times. The typical number is about 20, but some are run as many as 200 times to model the distribution in typical real work loads. The overall total number of transaction run in the scenario is about 2000.

The PL/SQL implementation of each business transaction invokes SQL, and the tables accessed by these during the test held typically large quantities of real data.

Thus this test emulated real world conditions, where an appreciable proportion of the time to complete a transaction is spent in SQL.

The time to complete the scenario was measured three times in each of the six compilation regimes.

The results are thus amenable to the same visualization as. The times are in seconds.

			Elapsed time *seconds*
9i	Interpreted	n/a	99.2

9i	Native	n/a	87.8
10g	Interpreted	opt_level_1	88.2
10g	Interpreted	opt_level_2	86.7
10g	Native	opt_level_1	85.5
10g	Native	opt_level_2	84.0

	9i I	9i N	10g I L1	10g I L2	10g N L1	10g N L2
9i I	1.00	1.13	1.12	1.14	1.16	**1.18**
9i N		1.00	0.99	1.01	1.03	1.05
10g I L1			1.00	1.02	1.03	1.05
10g I L2				1.00	1.01	1.03
10g N L1					1.00	1.02
10g N L2						1.00

				Avg
Code type (9i, 10g-L1, 10g-L2):	1.13	1.03	1.03	1.06
Optimize level (10g interpreted, 10g native):	1.02	1.02		1.02
Oracle version (interpreted, native):	1.14	1.05		1.10
Best:				**1.18**