OpenVMS Technical Journal V14

The Oracle Rdb Run-Time Code Generator for the OpenVMS Itanium Platform

Norman Lastovica, Senior Managing Engineer, Oracle Corporation

Introduction

The Oracle Rdb database engine generates platform-specific executable code subroutines at run time. On VAX systems, VAX executable instructions are generated. On Alpha systems, Alpha executable instructions are generated. When Oracle Rdb was ported to the HP Integrity platform, the ability to execute run-time created subroutines was required as well. This paper discusses background of the original interpretation implementation with a later transaction to native Itanium instruction generation.

Code Generation

When a user's request (such as the SQL statement "SELECT * FROM CUSTOMERS WHERE CITY = 'ESPOO' OR CITY = 'SALIDA' ORDER BY LAST_NAME") is passed to the database engine, a number of executable subroutines are created, at run-time, to perform various request-specific functions. These functions may include copying data fields, performing null-field handling, doing data field comparisons, and so on. This run-time request-specific code is an integral part of Oracle Rdb's database engine and helps to provide high levels of performance.

On VAX systems, such subroutines would contain VAX instructions (such as MOVC3, RET, MOVL and so on). When Oracle Rdb was ported to the OpenVMS and Tru64 environments for the Alpha platform, the code generation capabilities were extended to create Alpha instructions (such as CMOV, LDA, STQ, and so on). The logic of the subroutines, for the most part, is the same between the platforms; just the executable instructions, the register usage, and the system calling standard are different.

The following example contains a sequence of run-time generated instructions on a VAX system. Note the CISC architecture of the VAX computer with use of complex instructions that contain multiple operands along several addressing modes along with relatively high code density.

```
00FB1B18:   .WORD   ^M<R2,R3,R4,R5,R6,R7>
00FB1B1A:   MOVL    B^04(AP), R6
00FB1B1E:   MOVAB   @#00FB1B68, R7
00FB1B25:   MOVC5   S^#00, (SP), (SP), S^#05, B^01(R7)
00FB1B2C:   MOVL    B^69(R6), R0
00FB1B30:   ROTL    S^#08, R0, R0
00FB1B34:   BICL3   #FF00FF00, R0, R1
00FB1B3C:   XORL2   R1, R0
00FB1B3F:   ROTL    S^#10, R0, R0
00FB1B43:   XORB2   #80, R0
00FB1B47:   XORL3   R1, R0, B^02(R7)
00FB1B4C:   CLRB    B^01(R7)
00FB1B4F:   BBC     S^#02, W^00EF(R6), 00FB1B5C
00FB1B55:   XORB2   S^#01, B^01(R7)
00FB1B59:   CLRL    B^02(R7)
00FB1B5C:   RET
```

The next example contains a sample of run-time generated code on an Alpha system. Note that the Alpha is a more traditional "RISC"-style architecture where instructions are simpler, fixed size, and that the memory reference instructions either read or write memory, but do not atomically read and update memory in a single instruction.

```
01A060AC  A00C8138      LDL             R0,#XF8138(R12)
01A060B0  A02C808C      LDL             R1,#XF808C(R12)
01A060B4  44010400      BIS             R0,R1,R0
01A060B8  F4000004      BNE             R0,#X0000004 -> 1A060CC
01A060BC  A00C8064      LDL             R0,#XF8064(R12)
01A060C0  A02C813C      LDL             R1,#XF813C(R12)
01A060C4  400105A0      CMPEQ           R0,R1,R0
01A060C8  F400000D      BNE             R0,#X000000D -> 1A06100
01A060CC  A00C8110      LDL             R0,#XF8110(R12)
01A060D0  A02C8090      LDL             R1,#XF8090(R12)
01A060D4  44010400      BIS             R0,R1,R0
01A060D8  F400000F      BNE             R0,#X000000F -> 1A06118
```

```
01A060DC  A36B0144      LDL            R27,#X00144(R11)
01A060E0  220C8118      LDA            R16,#XF8118(R12)
01A060E4  223F001F      LDA            R17,#X0001F(R31)
01A060E8  224C8068      LDA            R18,#XF8068(R12)
01A060EC  233F0003      LDA            R25,#X00003(R31)
01A060F0  A75B0050      LDQ            R26,#X00050(R27)
01A060F4  A77B0058      LDQ            R27,#X00058(R27)
01A060F8  6B5A4000      JSR            R26,(R26)      "ots$strcmp_eqls"
01A060FC  E4000006      BEQ            R0,#X0000006 -> 1A06118
01A06100  201F0001      LDA            R0,#X00001(R31)
01A06104  A75E0008      LDQ            R26,#X00008(SP)
01A06108  A7BE0010      LDQ            FP,#X00010(SP)
01A0610C  23DE0020      LDA            SP,#X00020(SP)
01A06110  6BFA8001      RET            R31,(R26),1
01A06114  00000000      CALL_PAL       HALT
01A06118  47FF0400      CLR            R0
01A0611C  C3FFFFF9      BR             #XFFFFFF9 -> 1A06104
```

Interpretation engine

Oracle Rdb was ported to run on the Microsoft Windows NT environment running on Intel x86 and Alpha processors (this product was, however, never released for production use). At that point in time, in order to rapidly complete the porting effort for the Intel x86 platform, an interpretation engine was created that could interpret those portions of the Alpha instruction set generated at run-time by Oracle Rdb. This approach allowed a single piece of code (the interpretation engine) to be written and, more importantly, debugged without having to change the instruction generation machinery within the Oracle Rdb database engine (which continued to generate subroutines using the Alpha instruction set).

Over time, an expanded set of “rich” instructions were added to the code generation capabilities on the Intel x86 platform. These instructions were intended to perform more complex actions as one “pseudo” instruction, replacing, in some cases, a large number of Alpha instructions in the code stream. Execution of these “rich” instructions could be more optimized as compared to individually executing long sequences of individual instructions. Approximately 150 of these “rich” instructions were eventually implemented.

Though not used on Alpha and VAX systems (the supported platforms for Oracle Rdb), this interpretation engine remained part of the Oracle Rdb source code and lay “dormant” for many years.

Itanium Emerges

With the advent of OpenVMS for the Integrity Server platform, Oracle chose to port the Rdb database engine to the Integrity Server for the OpenVMS operating system. Though native language compilers were available (primarily, in the case of Oracle Rdb, BLISS, C++, C and MACRO32), there was no immediate capability for the Oracle Rdb engine to create executable instructions for the Itanium architecture.
At this point, the interpretation engine was pressed in to service again. Most of the code had not even been compiled in over 10 years. But with a bit of effort (mostly correcting issues related to improved C compilers with enhanced detection for latent bugs), it was able to successfully execute once again. Debugging effort was required to get it working completely properly but it did prove to be a valuable tool that allowed a significantly more rapid production delivery and deployment of Oracle Rdb Release 7.2 on the Itanium platform.

Performance

Overall, we anticipated that, while the performance of the interpreted code would never be as good as a native executable code subroutine, the Integrity system, as a whole, would perform at least comparable to "equal" Alpha systems. This was the case for the vast majority of applications and systems that we analyzed. CPU, memory and IO performance tended to provide a balanced system that performs very well when running customer applications. A few applications, however, spent a significant amount of time executing the run-time generated code and these applications were, in some cases, significantly slower than we, and our customers, would have preferred.
In particular, one major customer application was generally as good or better performing on Itanium systems than on Alpha systems. But several significant queries of the application were both frequently executed and much slower once migrated from Alpha to Itanium. Analysis revealed that most of the additional CPU time was spent in the interpretation engine while running particular parts of the application.

A major effort was spent in analysis and tuning of the interpretation engine itself. This tuning yielded performance improvements of over 20% in some cases. This was, however, not nearly enough (and, regrettably, not even in the same order of magnitude required). Further analysis indicated that there was likely no way to make the interpretation model execute fast enough to meet our customers' needs in all cases.

A New Direction

It was felt that the investment required to enhance the Oracle Rdb database engine to add another set of code generation capabilities (in addition to VAX, Alpha and "rich" instructions) for native Itanium would consume significant resources for development and testing and likely could not be completed in time for this particular customer's production deployment schedule. There were too many locations in the code that would be required to be changed to produce instructions for yet another architecture. Our experience with the port to Alpha indicated that there would be substantial human resources required to produce and debug the resultant code.

Based on this analysis, the concept of "compiling", at run time, a complete subroutine from a mixture of Alpha and "rich" instructions in to native Itanium executable code was born. The design that we arrived at is not dissimilar to the JAVA machinery's "Just In Time" (aka JIT) compiler available on many platforms: Input a stream of generic and platform-independent instructions and create platform-specific executable code which is expected to perform much better than interpreting the "pseudo" instructions.

Initial prototypes were developed to create and call an executable stream of Itanium instructions. The success of these tests supported the idea that it was viable for Oracle Rdb to be able to create native subroutines and to call such code at run time while on the Itanium platform.

High Level Design

The basic operation for what we originally called the "interp compiler" (based on the idea that this was a compiler to replace the interpretation engine) was to pass a pointer to a complete subroutine of compiled Alpha and "rich" code and then attempt to compile it completely in to a native Itanium instruction subroutine. If the compilation was successful (all instructions were able to be compiled) then a pointer to a procedure descriptor for the generated routine was returned with the low bit set (i.e., an odd value). If, however, the compilation could not be complete (if instructions were found that were not able to be compiled), the original routine address was returned (with the low bit clear as the routine had been originally allocated on a longword boundary).

Later, when the subroutine was to be called, the low bit of the routine's address was first evaluated. If clear, the existing interpretation engine was called to execute the subroutine. If the routine address was odd (indicating that the low bit was set), the routine was called directly (after clearing the low bit) to be executed "native".
In this way, the "interp compiler" could start small (only able to compile a few instruction types) and then grow (by adding the ability to compile more and more instructions and addressing modes and so on) all while the database engine continued to operate correctly (presumably as more and more subroutines could be compiled, execution performance would continue to improve). This made it possible to continue to execute and test Oracle Rdb while the "interp compiler" was being actively developed. Without this model it was have been a much slower process in that the "interp compiler" would have had to be entirely complete before we could even begin to test it.

Itanium Architecture

Significant attributes of the Itanium architecture that pertain to the "interp compiler" include:

- 128 bit "bundle" containing 3 instructions that will all be executed
- Multiple execution unit type combination selected via a "template" within a bundle
- Predicate registers that control if an instruction will have an effect or not

Producing code for the Itanium architecture is a fair measure more complex than, for example, code generation on the Alpha architecture. A significant set of rules and requirements are imposed in terms of which instruction type may be used in which bundle slot depending on the specified template, the use of "stops" to indicate that the results of prior instructions are required by following instructions, and so on. A larger number of registers provides major benefits in regards to having more scratch registers available for intermediate results. And the use of predicates can, in some cases, drastically reduce the number of branches taken which can, in turn, improve performance by reducing "wasted" processor cycles due to "bubbles".

Additional steps were required after creating code. Because the Itanium instruction cache (I cache) and data cache (D cache) are not synchronized, after new executable code is created, the hardware must be notified by flushing the instruction cache for the memory addresses of the newly created code. This was accomplished most easily by calling the SYS$PAL_IMB system service specifying the starting address and the length of the generated code. The system service invalidates each I cache line as needed and ensures that the data and instruction caches are correctly synchronized prior to attempting to execute the new instructions.

Starting Simple

The Oracle Rdb "interp compiler" is implemented as a routine written in BLISS (the primary implementation language utilized by Oracle Rdb for both ease of development and resultant product performance). Generation of instructions is accomplished though a set of macros that implement primitive operations that are generally produced as one or more instructions within one or more bundles. The original project goal was to have macros that would create one instruction per bundle. Over time, more and complex macros were created to perform different functions and to create bundles with more instruction sequences to help produce faster and denser code streams.

A simple macro might, for example, produce a single ld4 instruction (to fetch 4 bytes from memory) alone in a single bundle (nop instructions would occupy the remaining two slots). Another level of complexity might be a single macro to create a pair of memory load and store instructions in a bundle. A more complex macro may implement a call sequence where output parameters are created, registers are saved, a procedure descriptor read, the routine called, and then after the call registers are restored. This sequence would require a modest number of bundles to implement.

Branches

One area of complexity is forward and backward branches within the code. The input subroutine may contain both "rich" and Alpha instructions that change the flow of control via conditional and unconditional branches. Branches are self-relative within the context of the input routine. To preserve the correct branch destination, a table is constructed that contains the address of the bundle containing the branch along with the original branch offset. Another table is maintained to associate the original instruction location along with the location of the generated code. After code generation is complete, branches are "fixed up" to adjust the destination offset to the correct destination bundle.

For performance, the Itanium architecture includes "hints" for most types of branch instructions. These hints allow a compiler to indicate additional information to the hardware in regards to how branches are expected to execute. The hardware, in turn, may use this information to predict how the flow of control is expected to operate and can allocate resources more efficiently and, ultimately, execute the whole of the code stream faster. For example, one such branch "hint" type would be "dynamic, predicted not taken". This hint implies that the compiler expects that the conditional branch will generally not be taken but the hardware should allocate prediction resources (such a history of branches taken or not taken at this location).

Based on both performance testing and research papers evaluated, the "interp compiler" utilizes these branch hints in the generated instruction stream. Unconditional branches are specified as "static, taken", most conditional branches are specified as "dynamic, predicted not taken". Exceptions to these rules are backward (typically involved in a loop) branches which likely are specified as "dynamic predicted taken".

Exception Handling

In order to allow OpenVMS exception handling mechanisms to function properly, the "interp compiler" must "register" each generated routine with the operating system. This registration includes identifying any "unwind" information specifically regarding the routine's first and last instruction, the length of the routine's prologue and registers and stack usage. Because the created code will be both created and executed in the processor's executive mode, a kernel mode image exit handler is utilized to un-register the generated code during image run down. Without having such unwind information registered with the operating system, exception handling is not possible; otherwise an exception from the generated code, or code that is called by the generated code, cannot be handled and results in, depending on the mode and context, either process or image termination.

The OpenVMS calling standard uses a variant of the common Itanium standard which includes a moderately complex set of rules for representing unwind information. This scheme includes a compressed variable-length fields and a dense structure. Within the interp compiler, this "signature" information is produced at the end of executable code creation for each routine.

Simple Code Sequence Examples

The following code sequence shows the original "rich" instruction (indicated by longword address and content fields at the left part of the line) CLR_Q (clear quadword) along with its single operand followed by the generated Itanium instructions (indicated by quadword content and instruction addresses) created for the "rich" instruction.
The operation's addressing mode is evaluated as an offset from the global register r2 (this register maps to Alpha register R12 within Oracle Rdb). The offset is created by adding 8000 to the value 0040 and then sign extending from 16 to 64 bits. Next, r0 (which is always read as the value zero) is written to the destination address, thus clearing it.

It would obviously be possible to combine these two instructions in to a single bundle. However, the construction of the interp compiler is such that the addressing steps are evaluated first and then the operation steps are produced. While it would be viable to perform a second pass to combine the instructions in to a single bundle, it has not yet been a high priority for execution optimization. An additional concern for such optimization is that significant amounts of time could be spent in the interp compiler that could exceed the potential benefits for performance improvements of the generated code. In this case, for example, the stall caused by the memory reference will dramatically overshadow any other optimizations possible for the two instructions which still require a stop between them (as the first updates r14 which is used as input to the second).

```
s037E54B4  04000157      CLR_Q
037E54B8  D0000040              dst    quad*
                              { .mfi
013807E80380  0000000080366190        add         r14 = 3F8040, r2
000008000000  0000000080366191        nop.f       000000
000008000000  0000000080366192        nop.i       000000 ;; }
                              { .mfi
008CC0E00000  00000000803661A0        st8         [r14] = r0
000008000000  00000000803661A1        nop.f       000000
000008000000  00000000803661A2        nop.i       000000 ;; }
```

In the next example, the MOV_Q (move quadword) “rich” instruction has two operands (source and destination address information). The source location is indicated as an offset (00000050) from a register (Alpha register R16 which is translated as Itanium register r32; the first input parameter to the routine as specified in the OpenVMS calling standard). The destination is an offset (0040) from register r2 (strictly, the offset is FFFF8040 from register R30). The interp compiler detects that both source and destination addresses are likely to be at least quadword aligned and produces a single ld8 instruction to read the source quadword and a single st8 instruction to write to the destination.

```
037E54C4  04000145      MOV_Q
037E54C8  00000010              src    char*   (R16)
037E54CC  00000050              offset long
037E54D0  D0000040              dst    quad*
                              { .mfi
0108020A0380  00000000803661C0        add         r14 = 0050, r32
000008000000  00000000803661C1        nop.f       000000
000008000000  00000000803661C2        nop.i       000000 }
                              { .mfi
013807E803C0  00000000803661D0        add         r15 = 3F8040, r2
000008000000  00000000803661D1        nop.f       000000
000008000000  00000000803661D2        nop.i       000000 ;; }
                              { .mmi
0080C0E00400  00000000803661E0        ld8         r16 = [r14] ;;
008CC0F20000  00000000803661E1        st8         [r15] = r16
000008000000  00000000803661E2        nop.i       000000 ;; }
```

More Complex Examples

In the following example generated code, the “rich” instruction MOV_NB_BR_CLR is used to move a null bit (an indication of a database field within a row containing a value) to a byte and then branch if the bit was clear (indicating in this case that the field was not null). Note that there are 4 operands to the “rich” instruction. The interp compiler turns this “rich” instruction in to 8 Itanium instructions stored in three bundles.

The first two instructions add the offset 00CA to r32 (the first input parameter to the subroutine) and then fetch a byte from the resultant location. The next two instructions move the offset 3F8030 to r14 and then add r30 to r14 to result in the output address of the null bit.

The fifth and sixth instructions first extract the bit specified in the first operand of the rich instruction and then test the bit to determine if it is set or clear. The extr.u instruction extracts one bit from the specified position (4 in this case) and stores the result starting at bit 0 in the register r15. In the next instruction (tbit.z), the predicate register p6 will be set if the null bit is equal to zero and will be cleared if the null bit is not equal to zero.

Finally the resultant null byte is stored. If the null bit is clear (indicating that the database field has a value), a branch is to be taken. The branch displacement is a sign-extended 21-bit value indicating a number of longwords. Here, it is a forward branch of 4 longwords. In the instruction stream, if predicate p6 is true (which indicates that the null bit was not set), a relative branch is taken. Otherwise, if predicate p6 is false, the branch is not taken and execution continues at the first instruction of the following bundle.

```
037E546C  0400019E      MOV_NB_BR_CLR
037E5470  00000004               bitNum ubyte
037E5474  000000CA               nulByt ulong
037E5478  D0000030               res    ubyte*
037E547C  FFE00004               brOff  ulong   (037E5490)
                               { .mmi
01080A0943C0  00000000803660D0        add         r15 = 00CA, r32 ;;
008000F003C0  00000000803660D1        ld1         r15 = [r15]
013807C60380  00000000803660D2        mov         r14 = 3F8030 ;; }
                               { .mii
010001E1C380  00000000803660E0        add         r14 = r14, r30
00A400F103C0  00000000803660E1        extr.u      r15 = r15, 04, 01 ;;
00A038F00180  00000000803660E2        tbit.z      p6, p7 = r15, 00 }
                               { .mfb
008C00E1E000  00000000803660F0        st1         [r14] = r15
000008000000  00000000803660F1        nop.f       000000
008400009006  00000000803660F2   (p6) br.cond.dptk.many 0000040 ;; }
```

Within the database environment, string operations (moving, changing and comparing) are common. The following example demonstrates the compiled code for the CMP_S "rich" instruction which is used to compare two fixed the length strings. The first operand is the number of bytes to compare. The second operand is the address of the first string and the third operand is the address of the second string. CMP_S returns either -1, 0 or 1 to the return status register (r8 which maps to Alpha R0) depending on the relationship (less than, equal, greater than) of the two strings.
The loop count application register ac.lc is used in conjunction with the br.cloop (branch counted loop) instruction to implement the main loop construct. Within the body of the loop, two bytes are fetched with a post increment of the source registers. Then the cmp.eq instruction is used to compare the values of the bytes for equality. Predicate register p7 is set if the comparison detects inequality. Un-equal values result in a branch out of the loop. Otherwise (in the case of the bytes being equal to each other), a backwards branch is taken by the br.cloop instruction to the prior bundle to fetch the next bytes.

When the bytes are known not equal, they are compared to each other with the cmp.lt instructions. If the strings are equal (when the loop executes to completion and no different bytes had been detected), r8 remains as zero. If the last bytes fetched are not equal (indicating that the loop did not complete and a difference was found), r8 is set to either -1 or 1. Note that within the final two bundles, the comparisons are done in parallel (the instructions can execute simultaneously because they do not depend on each other) and then the two moves are executed in parallel. The moves to r8 can be executed simultaneously because at most one of them will produce a result because predicates p6 and p7 are mutually exclusive – in no case will both be set. It is possible that neither is set (when the strings are equal) and r8 will remain 0.

```
037F22FC  0400013B      CMP_S
037F2300  0000001F               srcLen uword
037F2304  D0000288               src1   byte*
037F2308  025B5950               src2   byte*
                               { .mfi
000008000000  0000000080372090        nop.m       000000
000008000000  0000000080372091        nop.f       000000
00005413C000  0000000080372092        mov.i       ar.lc = 1E }
                              { .mfi
01382FE10380  00000000803720A0        add         r14 = 3F8288, r2
000008000000  00000000803720A1        nop.f       000000
000008000000  00000000803720A2        nop.i       000000 }
                              { .mlx
000008000000  00000000803720B0        nop.m       000000
000000000009  00000000803720B1        movl        r15 = 00000000025B5950 ;;
00C596CA03C0 }
                              { .mmi
00A000E02400  00000000803720C0        ld1         r16 = [r14], 001
00A000F02440  00000000803720C1        ld1         r17 = [r15], 001
000008000000  00000000803720C2        nop.i       000000 ;; }
                              { .mbb
01C039120180  00000000803720D0        cmp.eq      p6, p7 = r16, r17
008600003007  00000000803720D1   (p7) br.cond.dpnt.many 0000010
0091FFFFE140  00000000803720D2        br.cloop.sptk.few 1FFFFF0 ;; }
                              { .mii
010800000200  00000000803720E0        mov         r8 = r0
018001120180  00000000803720E1        cmp.lt      p6, p0 = r16, r17
0180010221C0  00000000803720E2        cmp.lt      p7, p0 = r17, r16 ;; }
                              { .mfi
013FFFCFE206  00000000803720F0   (p6) mov         r8 = 3FFFFF
000008000000  00000000803720F1        nop.f       000000
012000002207  00000000803720F2   (p7) mov         r8 = 000001 ;; }
```

In the case of the SET_T (set text) instruction, one or more bytes of a constant value are written to memory starting at a specified location. The interp compiler attempts to optimize these memory writes by performing overlapped operations and performing as few writes as possible by promoting the size of the memory reference based on the minimum alignment of the read and write stream pointers. Two pointers are used, offset by 8 bytes, to allow multiple st8 instructions to be executed in parallel. Post-increment instruction modes are used to update the output pointers in order to avoid additional instructions that would otherwise be required in order to increment the pointers. “Tail” writes of one, two or four bytes are used to complete the sequence.

If the byte count for the fill was larger, a loop would have been generated to perform the fill. In addition, the interp compiler produces, as needed, code to perform one, two or four byte writes prior to the loop and then again after the loop in order to align the output pointer on an 8 byte boundary so that as few memory writes as possible are created.

```
037E5550  04000151      SET_T
037E5554  00000020               src    byte
037E5558  D0000060               dst    char*   #XFFFF8060(R12)
037E555C  0000001F               dstLen word
                              { .mfi
013807EC0380  0000000080366340        add         r14 = 3F8060, r2
000008000000  0000000080366341        nop.f       000000
000008000000  0000000080366342        nop.i       000000 ;; }
                              { .mlx
010800E10400  0000000080366350        add         r16 = 0008, r14
008080808080  0000000080366351        movl        r15 = 2020202020202020 ;;
00C2002403C0 }
                              { .mmi
00ACC0E1E400  0000000080366360        st8         [r14] = r15, 010
00ACC101E400  0000000080366361        st8         [r16] = r15, 010
000008000000  0000000080366362        nop.i       000000 ;; }
                              { .mmi
00ACC0E1E200  0000000080366370        st8         [r14] = r15, 008 ;;
00AC80E1E100  0000000080366371        st4         [r14] = r15, 004
000008000000  0000000080366372        nop.i       000000 ;; }
                              { .mmi
00AC40E1E080  0000000080366380        st2         [r14] = r15, 002 ;;
00AC00E1E040  0000000080366381        st1         [r14] = r15, 001
000008000000  0000000080366382        nop.i       000000 ;;
```

Accessing Unaligned Data

Both the Alpha and Itanium systems impose a severe performance penalty when the processor attempts to perform an unaligned memory reference. An unaligned reference, for example, would be to attempt to fetch a longword (4 bytes) from a virtual address where the two lowest bits are not clear (i.e. not aligned on a 4 byte boundary). And the penalty on OpenVMS Itanium systems is significantly higher than it is on Alpha systems. Thus, avoiding alignment faults has an even greater benefit (for all processes on the system) on Itanium systems.
The "interp compiler" attempts to detect memory references that are not naturally aligned and produces a longer code sequence to perform the memory read or write operation without the overhead of an alignment fault.
For "rich" instructions, the assumption is made that register addresses are naturally aligned on quadword (8 byte) boundaries. Offset values can be then evaluated to determine if the resultant memory address is aligned or not. When an unaligned reference is predicted, a sequence of instructions can be generated to avoid the fault. For example, a load of a quadword that is located on a longword boundary an be accomplished by fetching the two longwords and then merging them together with the "mix4.r" instruction:

```
                              { .mmi
013807CF8E40  000000008033E180       mov         r57 = 3F807C ;;
000008000000  000000008033E181       nop.m       000000
01000393CE40  000000008033E182       add         r57 = r30, r57 ;; }
                              { .mmi
00A083908900  000000008033E190       ld4         r36 = [r57], 004 ;;
008083900380  000000008033E191       ld4         r14 = [r57]
000008000000  000000008033E192       nop.i       000000 ;; }
                              { .mfi
000008000000  000000008033E1A0       nop.m       000000
000008000000  000000008033E1A1       nop.f       000000
00F88241C380  000000008033E1A2       mix4.r      r14 = r14, r36 ;; }
```

The following sequence (adapted from analysis of code generated by the HP GEM compiler backend) is a longword store where the destination is predicted to not be naturally aligned. The least significant bit of the address (presented in r3) is tested. If it is set, the address is byte aligned and p7 is set; otherwise the address is word aligned and p6 is set. In the case of byte alignment, a single byte is stored and the address is incremented (thus aligned on a word boundary) and the output value is shifted 8 bits to the right. A word is then stored and the output is shifted right 16 bits. Finally, if the original address was word aligned, the final word is written, otherwise the final byte is written. This sequence results in either two (for word alignment) or three (for byte alignment) memory writes.

```
00A072000180     0611              tbit.z  pr6, pr7 = r3, 0
00AC0031004E     0621        (pr7)  st1     [r3] = r8, 1
00A5B882020E     0622        (pr7)  shr.u   r8 = r8, 8
00AC40310080     0630               st2     [r3] = r8, 2
00A578840200     0631               shr.u   r8 = r8, 16
008C40310006     0640        (pr6)  st2     [r3] = r8
008C0031000E     0641        (pr7)  st1     [r3] = r8
```

Optimizations

In some situations, the generated Itanium code sequences will execute faster than corresponding sequences on Alpha. For example, in cases of filling or comparing a relatively few bytes of memory, the code generated for Itanium includes a sequence of memory stores or fetches in-line while the Alpha code calls to the operating system routines OTS$FILL or OTS$CMP variants. The overhead of the call in some instances will be greater than the actual memory references.

In other cases, the Itanium instruction set provides instructions that perform operations that require a sequence of instructions on the Alpha platform. For example, the "mux1@rev" instruction can be used to reverse the order of bytes within a quadword. Within Oracle Rdb on the Alpha platform, this operation is accomplished in a series of independent shift and mask instructions. This byte reversal is, for example, used when constructing index keys so performance is an important consideration as this may be a commonly executed sequence.

Optimization does tend to be a repetitive and, at least based on our observations, a never-ending process. Over time, code sequences are compressed and improved with a goal of reducing latency in regards to the CPU clock rate and memory access latencies.

For example performance analysis both "by eye" and by processor cycle sampling lead to reductions in code steams by often "combining" addressing operands in to a single bundle as in the following example; initially the operands (moving the address values to r14 and r15) would have required two bundles.

```
06CE1B68  04000148      MOV_B
06CE1B6C  D00002D0               src    byte*
06CE1B70  D00010F1               dst    byte*
                              { .mfi
01382FEA0380  00000000805326F0       add         r14 = 3F82D0, r2
000008000000  00000000805326F1       nop.f       000000
01390FEE23C0  00000000805326F2       add         r15 = 3F90F1, r2 ;; }
                              { .mmi
008000E00400  0000000080532700       ld1         r16 = [r14] ;;
008C00F20000  0000000080532701       st1         [r15] = r16
000008000000  0000000080532702       nop.i       000000 ;; }
```

Instruction Execution Frequency

As part of a performance analysis sub-project, we created an instrumented interpretation engine that sent, via an OpenVMS mailbox, instruction execution information from all processes on a system to a separate collector process that captured instruction execution counts during a portion of an Oracle Rdb regression test run. The following table includes the top 20 instructions and the number of times each instruction was executed. In the table, the indication "RICH" indicates a "rich" instruction and "EVAX" indicates an Alpha instruction.

Instruction Mnemonic	Execution Count
RICH_MOV_B	60,974,337
RICH_MOV_NB_BR_CLR	60,537,063
RICH_MOV_L	47,099,034
RICH_MOV_Q	42,723,231
RICH_B_BR_SET	32,023,634
RICH_BRANCH	23,386,664
RICH_MOV_S	21,233,064
EVAX_LDA	21,169,625
EVAX_BIS	17,947,775
RICH_MOV_W	16,446,120
RICH_CMP_L	16,401,014
EVAX_JSR	15,925,377
RICH_MOV_B_BR_SET	14,723,867
RICH_EXE_ACTION	14,336,902
RICH_MOV_NBIT_BR_SET	14,138,224
EVAX_BR	13,827,371
RICH_OR_B_BR_SET	10,937,403
RICH_CNV_SORT_N	8,293,658
RICH_STALL	7,429,742

EVAX_LDAH	6,818,905

This data was, in turn, used as a guide for which instructions should be first considered for increased optimization by the interp compiler. The idea is that an instruction executed several times an hour has a marginal impact on performance as compared with an instruction executed thousands of times per second.

Database Performance Improvements

The creation and optimization of the Oracle Rdb "interp compiler" has been an iterative affair. Initial performance improvements from the interp compiler allowed applications running on Itanium systems to run at least as fast as on Alpha systems. Further optimizations (including reducing memory references, eliminating unneeded "stops", avoiding alignment faults, and so on) have dramatically improved code quality and yielded even better performance. In some cases, application performance has improved by a factor of 3 due to the interp compiler generating native instructions.

And Oracle continues to measure and analyze performance of the Oracle Rdb database product family on the HP OpenVMS operating system for the Integrity Server platform. An extensive set of regression tests are continuously run in our development environment to help ensure correctness of the generated code. We are also in constant contact with our customer based to help understand their performance challenges. This input helps us decide where to focus our optimization efforts to everyone's benefit.

Models and Examples Followed

A number of different resources were referenced in regards to code generation. In addition to the (voluminous) Intel documentation of the Itanium architecture, we also utilized the compiler machine code listings from high level language compilers on OpenVMS (for example, BLISS, C and MACRO32 which all use GEM code generator and the C++ compiler which uses an Intel code generator).

Both the OpenVMS debugger and system dump analyzer include the ability to format an instruction stream which helped significantly when we were learning the intricate details of the Itanium architecture.

The OpenVMS listings include the MACRO2000 facility which implements the MACRO32 compiler. This was used in many cases as a template for code generation for complex alpha instructions (such as ZAP and MSK). The internet also proved to be an excellent resource for example instruction streams and discussions of Itanium performance in regards to the use of the architecture.
The Intel and OpenVMS documentation was referenced extensively while we were creating the unwind information tables for generated code. And the OpenVMS calling standard manual was invaluable in regards of register usage rules.

Credit and Thanks

A large number of people devoted a great many hours to this project of developing the Oracle Rdb "Just In Time" code generator for the Itainum systems. It is not possible to remember or credit everyone who was involved. But special thanks and recognition are due to engineering members of both HP and Oracle including: John Reagan, Jeanie Leab, Guenther Froehlin, Greg Jordan, Christian Moser, Burns Fisher, Ian Smith, Martin Ramshaw, and Richard Bishop.

For more information

The Oracle Rdb web site is accessible on the internet at www.oracle.com/rdb. For more information about the Intel Itanium architecture and instruction set, visit www.intel.com. For more information about the HP OpenVMS system, visit www.hp.com.