OpenVMS Technical Journal V9

Methodologies for Fixing Alignment Faults

Ajo Jose Panoor, OpenVMS DECnet Engineering

Overview

The OpenVMS operating system is one of HP's key operating systems and has been successfully ported to the Intel® Itanium® 2 architecture. As a strategy, one of the foremost initiatives is to provide performance improvements for the operating system including other layered components on the Intel® Itanium® 2 architecture. The "alignment faults" on OpenVMS were expensive and led to performance degradation Fixing this became a focus area for us.

Most of the layered products on OpenVMS were found to have alignment faults that had to be addressed. This paper describes the methods followed to fix the alignment faults in OpenVMS DECnet Itanium. DECnet-Plus being a networking protocol, any performance degradation arising from the protocol stack has major implications on overall system performance and utilization. Hence, it was mandatory to address the alignment faults. These methods can be adopted for use on any OpenVMS product on the Intel® Itanium® 2 architecture. Fixing alignment faults can significantly boost performance, not just of the application, but of the overall system as well.

This article describes the methods followed to fix the alignment faults in DECnet. DECnet-Plus V8.3 on the Intel® Itanium® 2 architecture is the most recent version and includes the fixes for alignment faults.

Memory Alignments and Alignment Faults

Based on the architecture, a processor requires its data/variables to reside at particular offsets in the system's memory. For example, a 32-bit processor requires a 4-byte integer to reside at a memory address that is evenly divisible by 4. This requirement is called "memory alignment." Thus, a 4-byte integer can be located at memory address 0x2000 or 0x2004, but not at 0x2001. Alignment is thus the aspect of a data item that refers to its placement in memory.

The mixing of byte, word, longword, and quadword data types can lead to data that is not aligned on natural boundaries. When accessing these unaligned data we end up generating a fault. (A fault normally occurs when the instruction cannot complete and the Program Counter is left pointing at the instruction, and we have to rectify this fault to resume execution.) Normally, for each of the alignment faults, a fault handler is called for retrieving the data from the unaligned address, thereby using more CPU cycles. Thus, an alignment fault is a concern for performance, not for program correctness.

On AlphaServer systems, we had the LDx_U and STx_U instructions that allowed an unaligned access to be handled efficiently with a few instructions. The PAL (Privileged Architecture Library) code manages the alignment issues with applications on the VAX and Alpha architectures. However the Intel® Itanium® 2 hardware does not support PAL and any memory access to unaligned data results in an alignment fault.

Alignment faults on the Intel® Itanium® 2 architecture introduce additional software overhead as access to memory is serialized with spin locks. This results in performance degradation. More software overhead and spin lock contention results in draining valuable CPU cycles. The challenge is to eliminate the generation of alignment faults for performance improvement.

Types of Alignment Faults

An alignment fault is normally generated while a read or write of 'N' bytes occurs on an address that is not an integral multiple of the size of the data in bytes. The occurrence can be generalized as follows:

- Reading/writing a word that is not on a word boundary.
- Reading/writing a longword that is not on a longword boundary.
- Reading/writing a quadword that is not on a quadword boundary.

Ideally there should not be any alignment faults. But, while porting some legacy code, we may have unaligned data structures.

Standard Techniques for Resolving Alignment Faults

Following are some of the standard techniques used for resolving alignment faults:

- The best solution is to align the data itself.
- The next option is to teach the compiler that the data is unaligned so that it may generate more instructions to avoid alignment faults.
- The final option is to copy the data to an aligned buffer and use it.

Techniques adopted to resolve DECnet Alignment faults

The following are the causes for alignment faults in DECnet:

- An unaligned structure.
- Comparison of unaligned data.
- Bit field access from unaligned address.
- DECnet fork blocks placed in unaligned addresses.

In most alignment fault hits, the faulty addresses generated are caused by complex calculations or from some runtime values. In such cases, we need to find the exact location where the unaligned address is generated and fix the problem appropriately. We made use of the TR_PRINT macro to find the exact location of alignment faults. Each case was treated separately and analyzed before adopting the appropriate technique.

Note: Because DECnet is a networking protocol, we had to take care not to modify the structures (or Protocol Data Units (PDUs)) that are sent through the wire. Moreover, the structure offsets which are accessed through symbols or constants in code (some bit fields offsets in BLISS sources) are also not modified in consideration of code stability.

Following are the methodologies used to resolve the alignment faults:

Structure Rearrangement

We had several structures that were unaligned. In one such case most of the members were a combination of "length" (2 bytes or word) and "address" (4 bytes or longword) and the arrangement was leading to subsequent address fields placed in unaligned addresses. Hence the fields are rearranged such that the address fields are aligned on longword boundaries. In such cases "fillers" can also be added to make proper alignment (Structure Padding). In locations where we modify the structures, we ensure that no structure members are accessed through hard-coded definitions in source code.

Using Unions

DEC[illegible] called IDNodeType. We store the low-order 4 bytes of Node ID in a longword and access them. But, in some runtime allocations, the node ID is placed in a word-aligned address and accessing it results in alignment faults.

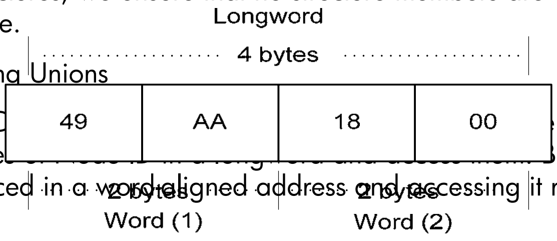

Figure 1 -- Union with one longword and two words

So a new union is introduced through which the Node ID can be accessed as a longword as well as two separate words as shown in figure 1. The longword access causing the fault is then replaced with two word accesses, thereby avoiding the alignment fault.

```
IDLow_u  UNION;

      IDLow      LONGWORD UNSIGNED;
      Low_s     STRUCTURE;
                   IDLow_1     WORD UNSIGNED;
                   IDLow_2     WORD UNSIGNED;
      END IDLow_s;
END IDLow_u;
```

Bit Field Access to get the Structure Fields

While extracting the network service access point (NSAP) length from the routing PDUs we always ended up in an alignment fault due to the following:

```
source_nsap = pdu [pdu$b_iso_address_part];
source_nsap_length =..source_nsap; ! - Alignment fault
```

In the second line for fetching a single byte we were making the compiler fetch 4 bytes (ld4 instruction) from an unaligned address and then used just one byte from it resulting in an alignment fault. Hence, using bit-field access, the fault is avoided. Here, the complier generates an ld1 instruction for fetching a single byte. (This method relies on the way instructions are generated by the complier, so behavior may vary based on the BLISS compilers)

```
source_nsap_length =.source_nsap[0,0,8,0];
```

Comparison of Unaligned Data

0000				49
0004	00	18	AA	00
0008	04			

Figure 2 – Unaligned data

Comparison of the System ID which is 6 bytes is usually done as a 4-byte and 2-byte comparison. In cases where the memory is allocated from the pool and used at runtime, there is a chance of data storage as shown in the Figure 23, i.e., the allocation starts from an unaligned offset. So the first 4-byte comparison and the second 2-byte comparison also cause an alignment fault. In order to avoid the alignment fault, we can make a byte-by-byte comparison to avoid the alignment fault. Similarly, in cases where a word alignment is found to be consistent, we adopted a word-by-word comparison to resolve the problem.

DECnet Fork Blocks placed in Unaligned Addresses

DECnet commands had a large number of alignment fault hits in the NET$FORK routine. (NET$FORK is a DECnet wrapper over the VMS Fork routine, EXE$FORK). The faulting address is found to be inside the VCRP structure allocated from the non-paged pool. (VCRP is a data structure used by network protocol stack to communicate between different layers). On formatting the address we found that the hitting address lies inside the fork block (FKB), which was placed from the start of the VCRP scratch area.

By default, VMS-FORK BLOCKS expects itself to be in a quadword-aligned address. But, the start of the VCRP Scratch area is not a quadword-aligned address and usage of the scratch area as a FORK BLOCK from that address caused the alignment fault. So, in order to make the fork block aligned, we moved the fork block to the nearest quadword-aligned offset from the beginning of the VCRP scratch area. Because VCRP is a common data structure used by other VMS networking products, we took extra care not to modify the existing structures but to move the fork block to the nearest quadword-aligned address without overflowing the scratch area. As a code fix, we generated a new symbol that has as its value the nearest quadword-aligned offset from the VCRP scratch address. The symbol VCRP$T_SCRATCH was the initial offset where the fork block was copied, and it has as its value 468 dec/1D4 hex which is not a quad word aligned offset. So a new structure is introduced for generating the symbol with its value as 472 dec/1D8 hex that is quadword aligned.

Teaching the Compiler

- Align (n) Qualifier for BLISS
  Align Qualifier tells the compiler that the data in the specified location is unaligned so that it generates extra code to handle this condition. At locations where the rearranging of structures or structure padding is not an option we normally go with teaching the complier to align naturally.

```
   local
      tpdu    : REF ALIGN(0) BLOCK[,BYTE];
```

Figure 3 – Align Qualifier for BLISS

  Here the ALIGN (0) qualifier tells the complier to perform a byte access to any of the succeeding references to the tpdu variable. The compiler generates more instructions to meet the byte access, which is one of the main disadvantages of using this method.

- CH$MOVE – BLISS inbuilt for Byte-by-byte copy
  Alignment faults due to bit accesses from unaligned addresses were fixed by using CH$MOVE function. CH$MOVE does a byte-by-byte copy to move the appropriate number of bytes from the unaligned address to a local variable. It then uses that variable for accessing the required bits, thereby avoiding the faults. A drawback is allocating more temporary storage. Adding Align (n) is more appropriate if the number of hits is less.

- BIND "Type Field" with Align (n) qualifier for BLISS
  The operation of associating an address with a name is called binding. Complex references causing the alignment faults can be replaced by bind variables locally and an align (n) qualifier can be added to avoid the alignment fault.

- Set_registers unaligned/ Set_registers aligned Qualifier for Macro32
  "Set_registers unaligned/ Set_registers aligned" are the counterparts for Align qualifier in Macro32. For example,

```
    .set_registers unaligned=<R3>
    MOVL    R1,(R3)+
    .set_registers aligned=<R3>
```

Figure 4 – Alignment Qualifier for Macro32

The content of R3 is an unaligned address and trying to access a longword from it generates an alignment fault. So the ".set_registers unaligned" directive is added to inform the complier that the content of register is unaligned. After accessing R3 the register is marked as an aligned one using the ".set_registers aligned" directive. This has to be reverted to “aligned” so that complier will not treat the register contents as unaligned for the rest of its access.

__unaligned	Directive used in C, indicates to the compiler that the data pointed to is not properly aligned
#pragma	nomember_alignment & member_alignment, for C structure padding
.align	Directive used in Macro32
Basealign	Directive used in SDL files

Figure 5 -- Other Compiler Directives

Tools and Utilities

- FLT SDA Extension

SDA provides a fault trace utility for finding the alignment faults. Loading and starting the FLT trace utility in SDA and running the program will provide the PC value of the instructions that causes an alignment fault.

- Monitor Align

The Monitor Align utility shows a statistical analysis of the alignment faults occurring in the system on different Execution Modes. The following table shows the alignment fault statistics taken over a small period of time after establishing almost one-hundred active DECnet connections.

MONITOR ALIGN	**CUR**	**AVE**	**MIN**	**MAX**
Kernel Fault Rate	27704.00	29114.74	26756.59	37276.94
Exec Fault Rate	0.00	0.07	0.00	6.80
Super Fault Rate	0.00	0.00	0.00	0.40
User Fault Rate	372.00	350.20	276.60	432.60
Total Fault Rate	28076.00	29465.02	27108.19	37613.87

Figure 6 -- Alignment fault statistics

After installing DECnet V8.3, where the alignment faults were fixed, the alignment fault rate reduced to a large extent for the same period of time.

MONITOR ALIGN	**CUR**	**AVE**	**MIN**	**MAX**
Kernel Fault Rate	23.00	21.42	12.99	99.00
Exec Fault Rate	0.00	3.70	0.00	164.50
Super Fault Rate	0.00	0.00	0.00	0.00
User Fault Rate	337.00	139.24	124.93	337.00
Total Fault Rate	360.00	164.37	137.93	373.50

Figure 7 -- Alignment fault statistics in DECnet V8.3

Performance Analysis

Significant performance improvements were observed after fixing the alignment faults. The following table shows the comparison between the DECnet kit with and without alignment fault fixes after establishing almost one hundred active DECnet connections with minimal disk access. The alignment hit count was reduced significantly and the CPU utilization was reduced by 35%.

For a period of 3 minutes	Unaligned	Aligned
Alignment hit count	78,039	37
CPU Utilization	95%	60%

Figure 8 -- DECnet kit with and without alignment fault fixes

References

- Alignment Faults and Performance - by Christian Moser.
- OpenVMS Internals – by Ruth Goldenberg/Lawrence Kenah/Denise Dumas/Saro Saravanan

For more information

Please contact the author with any questions or comments regarding this article at. To get to the latest issue of the OpenVMS Technical Journal, go to: http://www.hp.com/go/openvms/journal.

About the Author

Ajo Jose Panoor works for OpenVMS DECnet Engineering and was involved in fixing the alignment faults in DECnet. Ajo holds a bachelor's degree in Computer Science.