OpenVMS Technical Journal V9

Alignment Faults – What Are they and Why Should I Care?

Guy Peleg, Director of EMEA Operations, BRUDEN-OSSG

Overview

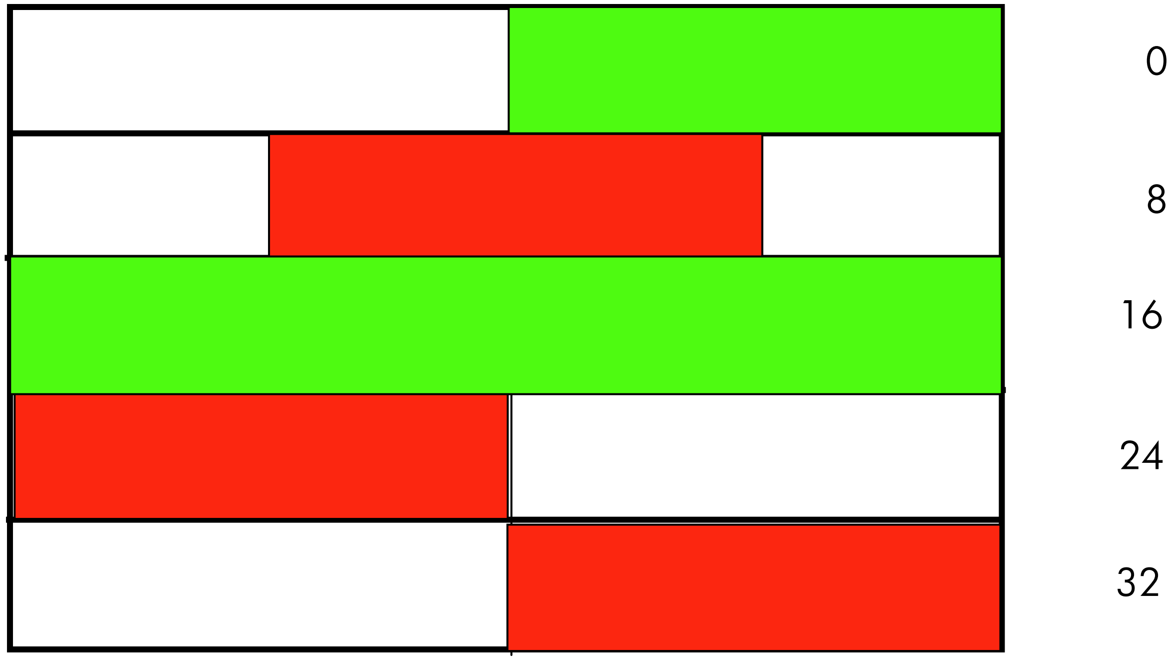
The article explains what alignment faults are, describes how alignment faults impact application performance, presents ways to detect alignment faults on a running system, and provides a few ideas on fixing alignment faults.

What is an Alignment Fault?

AlphaServer and Intel® Itanium® 2 processors provide fast access to naturally aligned data. To be naturally aligned, a word datum must be on a word boundary, a longword datum must be on a longword boundary, and a quadword datum must be on a quadword boundary.

When an attempt is made to load or store a quadword, longword, or word to or from a memory location that does not have a naturally aligned address, the processor transfers control to a special routine (PALcode on AlphaServer systems and an operating system routine on Intel® Itanium® 2 systems) to execute a series of instructions to perform the unaligned access. The step of executing a special set of routines to access unaligned data is referred to as alignment fault.

The following diagram illustrates the difference between aligned and unaligned memory access:



In the first row, we access a longword starting with address 0 that is naturally aligned so all is well. In the second row we attempt to access a longword starting at address 10. This address is not naturally aligned (10 divided by 4 does not yield a remainder of 0). Alignment fault will occur in this case. In the third row, we attempt to read a quadword starting at address 16 that is naturally aligned (16 divided by 8 yields a remainder of 0) so all is well. In the fourth row, we attempt to access a quadword starting at address 28. Address 28 is not quadword aligned so an alignment fault will occur.

Okay…I understand Alignment faults but why should I care?

When the compiler can detect misaligned data, what would normally take three instructions on an AlphaServer system will take fifteen. As not all of these instructions access memory, the aggregate degradation in performance is an instruction stream that is three times slower. When the compiler cannot correct the problem, a run time alignment fault is incurred. The alignment handler is about ten to twenty times slower than accessing naturally aligned data.

The behavior of an Intel® Itanium® 2 system is similar to the AlphaServer, except that alignment faults are hundreds to thousands of times slower than accessing naturally aligned data, as alignment faults are handled by the operating system itself instead of PAL code (firmware). There is also a system-wide impact for resolving alignment faults. This impact is due to the requirement for spinlock (MMG) and associated MP synchronization time.

Let’s take a look at a small example. The following program allocates 1 GB of virtual memory in P2 space and randomly increments 50,000,000 quadwords.

```
$ ty aligned.c
#include <far_pointers>
#include <gen64def>
#include <ints>
#include <starlet>
#include <stdio>
#include <stdlib>
#include <lib$routines.h>
#include <unistd.h>
```



```
#include <stsdef>

#define random_key(upper_bound) (abs (random () % upper_bound))

void main()
{
int             NumberOfBytes    =      1000000000;     // 1GB using marketing bytes
int             status;
VOID_PQ         MappedVA;
INT64_PQ        RandomVA;


        lib$init_timer();               // initialize timer

        //
        // Allocate 1GB from P2 space
        //
        status = lib$get_vm_64 (&NumberOfBytes, &MappedVA);

        if (!$VMS_STATUS_SUCCESS(status))
        {
                lib$signal (status);
                return;
        }

        RandomVA = MappedVA;

        for (int i=0; i<50000000; i++)
        {

                // Increment a random Quadword
                RandomVA [random_key((100000000/8) -1)] ++ ;

        }

        //
        // Free VM
        //
        status = lib$free_vm_64 (&NumberOfBytes, &MappedVA);

        if (!$VMS_STATUS_SUCCESS(status))
        {
                lib$signal (status);
                return;
        }

        lib$show_timer();
}

$! Run the program – rx2600 1.3 GHZ
$ cc/pointer=long aligned
$ link aligned
$ r aligned
```



```
 ELAPSED:    0 00:00:18.97  CPU: 0:00:18.97  BUFIO: 0  DIRIO: 0  FAULTS: 713808
$
```

Incrementing 50,000,000 random quadwords on a 1.3 GHz Integrity rx2600 Server took 18.97 seconds.

Now, let's force the above program to increment 50,000,000 quadwords using unaligned pointers:

```
$ ty not_aligned.c
#include <far_pointers>
#include <gen64def>
#include <ints>
#include <starlet>
#include <stdio>
#include <stdlib>
#include <lib$routines.h>
#include <unistd.h>
#include <stsdef>

#define random_key(upper_bound) (abs (random () % upper_bound))

void main()
{
int             NumberOfBytes    =      1000000000;     // 1GB using marketing bytes
int             status;
VOID_PQ         MappedVA;
INT64_PQ        RandomVA;


        lib$init_timer();               // initialize timer

        //
        // Allocate 1GB from P2 space
        //
        status = lib$get_vm_64 (&NumberOfBytes, &MappedVA);

        if (!$VMS_STATUS_SUCCESS(status))
        {
                lib$signal (status);
                return;
        }

        //
        // Force the pointer to become unaligned
        //
        RandomVA = (INT64_PQ)((char *) MappedVA + 1);


        for (int i=0; i<50000000; i++)
        {
```



```
                // Increment a random Quadword
                RandomVA [random_key((100000000/8) -1)] ++ ;

        }

        //
        // Free VM
        //
        status = lib$free_vm_64 (&NumberOfBytes, &MappedVA);

        if (!$VMS_STATUS_SUCCESS(status))
        {
                lib$signal (status);
                return;
        }

        lib$show_timer();
}

$ cc/pointer=long not_aligned.c
$ link not_aligned
$ r not_aligned
 ELAPSED:    0 00:03:45.62  CPU: 0:03:45.53  BUFIO: 0  DIRIO: 0  FAULTS: 200027
$
```

The same 1.3 GHz Integrity rx2600 Server increments 50,000,000 unaligned quadwords in 3 minutes and 45 seconds.

For our small test program, performance degrades by more than *12 times* when accessing unaligned data.

Detecting Alignment Faults

Now that you are all convinced that alignment faults are bad for performance, let's take a look at various tools provided by OpenVMS for detecting alignment faults:

- MONITOR ALIGN (V8.3)
- FLT extension in SDA
- Symbolic Debugger

MONITOR ALIGN

OpenVMS V8.3 introduced a new class for the monitor utility. The align class monitors alignment faults currently occurring throughout the system and breaks out the output per mode.

The following display was generated while running the NOT_ALIGNED program:

```
$ monitor align/int=1

                              OpenVMS Monitor Utility
                             ALIGNMENT FAULT STATISTICS
                                   on node IT13
```



```
                          21-NOV-2006 01:50:13.26

                                  CUR        AVE        MIN        MAX


   Kernel Fault Rate             0.00       0.44       0.00       4.00
   Exec   Fault Rate             0.00       0.00       0.00       0.00
   Super  Fault Rate             0.00       0.00       0.00       0.00
   User   Fault Rate        445492.00  220809.67       0.00  445492.00

   Total  Fault Rate        445492.00  220810.12       0.00  445492.00
```

Our test program generates more than 445,000 alignment faults per second, all in user mode.

MONITOR ALIGN provides a high-level overview of alignment faults currently occurring on the system. It helps detect alignment faults and warns that the system is suffering from alignment faults. But MONITOR ALIGN does not provide any information about which process or program generated the alignment faults. MONITOR ALIGN is intended to help and determine if you are suffering from alignment faults. Different tools should be used to determine what is generating the faults. Note that MONITOR ALIGN is currently available on Intel® Itanium® 2 systems only.

FLT Extension in SDA

Once you determine that your system is prone to alignment fault issues, the next step is to determine where the faults are coming from. The FLT extension in SDA is a very powerful tool for detecting and logging alignment faults. For each alignment fault that occurs while logging is enabled, it logs the time the fault occurred, the CPU encountering the fault, the unaligned Virtual Address, access mode, and process id. This information allows the developer to determine the exact location in the application which generated the alignment fault. The FLT extension is available on both AlphaServer and Intel® Itanium® 2 systems.

Here are few examples demonstrating the use of FLT

```
$ ana/sys

OpenVMS system analyzer

Load the SDA extension

SDA> flt load
FLT$DEBUG load status = 00000001

Start tracing …

SDA> flt start trace
Tracing started...

Look at the summary display

SDA> flt show trace/sum
```

```
Fault Trace Information: (at 21-NOV-2006 02:07:21.87, trace time 00:00:00.190015)
-------------------------------------------------------------------------------
Exception PC              Count   Exception PC                          Module
Offset
----------------  ------------  -------------------------------------  -------------------------------
-
00000000.000103D1         39384   SYS$K_VERSION_16+00391
00000000.000103E1         39383   SYS$K_VERSION_16+003A1
```

Two Program Counters are displayed pointing to PC 103D1 and 103E1, each PC generated more 39834 faults. Let's find our culprit, instead of looking at the summary output we can look at individual entries in the trace buffer for more information:

```
SDA> flt show trace

Unaligned Data Fault Trace Information:
---------------------------------------
Timestamp              CPU  Exception PC                                   Unaligned VA      Access
EPID    Trace Buffer
--------------------- ---  ---------------------------------------------  ----------------  ------  ----
----  ----------------
21-NOV 02:08:22.002794  00  00000000.000103E1 SYS$K_VERSION_16+003A1        00000000.840BECF9  User
2160057F  FFFFFFFF.7E4E86C0
21-NOV 02:08:22.002791  00  00000000.000103D1 SYS$K_VERSION_16+00391        00000000.840BECF9  User
2160057F  FFFFFFFF.7E4E8658
21-NOV 02:08:22.002789  00  00000000.000103E1 SYS$K_VERSION_16+003A1        00000000.84617049  User
2160057F  FFFFFFFF.7E4E85F0
21-NOV 02:08:22.002786  00  00000000.000103D1 SYS$K_VERSION_16+00391        00000000.84617049  User
2160057F  FFFFFFFF.7E4E8588
21-NOV 02:08:22.002784  00  00000000.000103E1 SYS$K_VERSION_16+003A1        00000000.8252A0E1  User
2160057F  FFFFFFFF.7E4E8520
21-NOV 02:08:22.002781  00  00000000.000103D1 SYS$K_VERSION_16+00391        00000000.8252A0E1  User
2160057F  FFFFFFFF.7E4E84B8
21-NOV 02:08:22.002779  00  00000000.000103E1 SYS$K_VERSION_16+003A1        00000000.850E3241  User
2160057F  FFFFFFFF.7E4E8450
21-NOV 02:08:22.002776  00  00000000.000103D1 SYS$K_VERSION_16+00391        00000000.850E3241  User
2160057F  FFFFFFFF.7E4E83E8
21-NOV 02:08:22.002774  00  00000000.000103E1 SYS$K_VERSION_16+003A1        00000000.84CD53D1  User
2160057F  FFFFFFFF.7E4E8380
21-NOV 02:08:22.002771  00  00000000.000103D1 SYS$K_VERSION_16+00391        00000000.84CD53D1  User
2160057F  FFFFFFFF.7E4E8318

……
```

All the entries are pointing to process with ID 2160057F, let's look at the process to find out what image it is executing:

```
SDA> set proc/id=2160057F
SDA> show proc/image

Process index: 017F   Name: Faulty          Extended PID: 2160057F
--------------------------------------------------------------------

                           Process activated images
                           ------------------------

               Image Name                    Type       IMCB          GP
--------------------------------------- ------------ -------- -----------------
```

```
NOT_ALIGNED                           MAIN        7FE89290 00000000.00240000
DCL                                   MRGD     SHR 7FE88BD0 00000000.7B0D8000
LIBRTL                                GLBL     SHR 7FE8BC10 00000000.7B546000
LIBOTS                                GLBL     SHR 7FE8A690 00000000.7B560000
CMA$TIS_SHR                           GLBL     SHR 7FE88010 00000000.7B73C000
DPML$SHR                              GLBL     SHR 7FE88270 00000000.7B904000
DECC$SHR                              GLBL     SHR 7FE883A0 00000000.7BB10000
SYS$PUBLIC_VECTORS                    GLBL         7FE886C0 FFFFFFFF.8CA00400
SYS$BASE_IMAGE                        GLBL         7FE88920 FFFFFFFF.8CA24E00

Total images = 9                      Pages allocated = 322
SDA> map 0103E1
Image                                       Base              End          Image Offset
NOT_ALIGNED
    Code                              00000000.00010000 00000000.0001059F 00000000.000103E1
SDA>
```

We found out that all the alignment faults are generated by process "Faulty" executing the NOT_ALIGNED image. Next step would be to look at the listing and determine the offending code in offset 103E1.

Before we look at the listing, the FLT extension can interpret the location of the faulting PC if the image contains traceback information and if it lives in system space. Now, let's install NOT_ALIGNED.EXE as resident image, it will force the image to be copied into system space:

```
SDA> flt stop trace
SDA> spawn instal add/resi SYS$SYSDEVICE:[PELEG]NOT_ALIGNED
SDA> flt start trace
Tracing started...
SDA> flt show trace/summ

Fault Trace Information: (at 21-NOV-2006 02:13:23.77, trace time 00:00:00.190637)
-------------------------------------------------------------------------------
Exception PC              Count   Exception PC                          Module
Offset
----------------   ------------   ------------------------------------  --------------------------------
-
FFFFF802.11EFE3D1         39384   NOT_ALIGNED+103D1                     NOT_ALIGNED
000103D1
                                  NOT_ALIGNED + 000003D1 / main + 000002D1
FFFFF802.11EFE3E1         39383   NOT_ALIGNED+103E1                     NOT_ALIGNED
000103E1
                                  NOT_ALIGNED + 000003E1 / main + 000002E1
SDA>
```

We start tracing again, now the summary display show the exact location in the image that generated the fault. In our example this is routine main+2D1 and main +2E1 in NOT_ALIGNED.EXE.

Let's look at relevant portion of the listing in NOT_ALIGNED.LIS

```
001000000046     0240          (pr6)  break.m 1048577
00C7080121C0     0241                 setf.sig f7 = r9
```

```
018402242200    0242                cmp4.lt pr8, pr0 = i, r34 ;;           // pr8, pr0 = r33, r34
// 023707
                         }
                         { .mfi
00C708006180    0250                setf.sig f6 = r3                      // 023711
000008000000    0251                nop.f   0
000008000000    0252                nop.i   0 ;;
                         }
                         { .mfi
000008000000    0260                nop.m   0
0000E000E240    0261                fcvt.xf f9 = f7
000008000000    0262                nop.i   0
                         }
                         { .mfi
000008000000    0270                nop.m   0
0000E000C200    0271                fcvt.xf f8 = f6
000008000000    0272                nop.i   0 ;;
                         }
                         { .mfi
000008000000    0280                nop.m   0
000630910280    0281                frcpa.s1 f10, pr6 = f8, f9
000008000000    0282                nop.i   0 ;;
                         }
                         { .mfi
000008000000    0290                nop.m   0
018448A021C6    0291          (pr6) fnma.s1 f7 = f10, f9, f1
000008000000    0292                nop.i   0 ;;
                         }
                         { .mfi
000008000000    02A0                nop.m   0
010438A142C6    02A1          (pr6) fma.s1  f11 = f10, f7, f10
000008000000    02A2                nop.i   0
                         }
                         { .mfi
000008000000    02B0                nop.m   0
010438700186    02B1          (pr6) fma.s1  f6 = f7, f7, f0
000008000000    02B2                nop.i   0 ;;
                         }
                         { .mfi
000008000000    02C0                nop.m   0
0104508001C6    02C1          (pr6) fma.s1  f7 = f8, f10, f0
000008000000    02C2                nop.i   0 ;;
                         }
                         { .mfi
000008000000    02D0                nop.m   0
010430B16286    02D1          (pr6) fma.s1  f10 = f11, f6, f11
000008000000    02D2                nop.i   0 ;;
                         }
                         { .mfi
000008000000    02E0                nop.m   0
0184389102C6    02E1          (pr6) fnma.s1 f11 = f9, f7, f8
000008000000    02E2                nop.i   0
                         }
                         { .mfi
000008000000    02F0                nop.m   0
018448A02186    02F1          (pr6) fnma.s1 f6 = f10, f9, f1
000008000000    02F2                nop.i   0 ;;
                         }
```

```
                        { .mfi

main + 2D1 and main + 2E1 point to line number 23711 in the source:

     1   23707        for (int i=0; i<50000000; i++)
     2   23708        {
     2   23709
     2   23710               // Increment a random Quadword
     2   23711                   RandomVA [random_key((100000000/8) -1)] ++ ;
     2   23712
     1   23713        }

The next step logical step would be fixing the program to avoid unaligned memory access.
```

Symbolic Debugger

The symbolic debugger can be used for detecting alignment faults. The SET BREAK/UNALIGN command will cause the debugger to break each time an alignment fault occurs. The faulting Virtual Address, the current PC, and the source line that generated the fault will be displayed:

```
$ run/debug not_aligned

         OpenVMS I64 Debug64 Version V8.3-009

%DEBUG-I-INITIAL, Language: C, Module: NOT_ALIGNED
%DEBUG-I-NOTATMAIN, Type GO to reach MAIN program

DBG> set break/unaligned
DBG>
* SRC: module NOT_ALIGNED -scroll-
source*****************************************************************************************
****
 23703:         // Force the pointer to become unaligned
 23704:         //
 23705:         RandomVA = MappedVA + 1;
 23706:
 23707:         for (int i=0; i<50000000; i++)
 23708:         {
 23709:
 23710:                 // Increment a random Quadword
->3711:                 RandomVA [random_key((100000000/8) -1)] ++ ;
 23712:
 23713:         }
 23714:
 23715:         //
 23716:         // Free VM
 23717:         //
 23718:         status = lib$free_vm_64 (&NumberOfBytes, &MappedVA);
 23719:
```



```
* OUT -
output*****************************************************************************************
**
Unaligned data access: virtual address = 0000000081E0E7E1, PC = 00000000000103E2
break on unaligned data trap preceding NOT_ALIGNED\main\%LINE 23711+402
 23711:                RandomVA [random_key((100000000/8) -1)] ++ ;



DBG>


NOTE: SET BREAK/UNALIGNED can not be used while the FLT utility is in use. When FLT is
running, attempting to use the debugger for reporting alignment faults will fail with the
following error:


DBG> set break/unalign
%SYSTEM-E-AFR_ENABLED, alignment fault reporting already enabled
-FOR-W-NOMSG, Message number 00189E80
DBG>
```

Guidelines for Fixing Alignment Faults

The perfect application avoids alignment faults completely; however life is not always perfect. Alignment faults are likely to be encountered when a module that declared unaligned data calls a routine in another module that does not anticipate receiving unaligned data. Remember that alignment faults are bad on AlphaServer systems, but are *really* bad on Intel® Itanium® 2 systems.

Some alignment faults are easy to fix, some are very hard, and some are close to impossible. Here are the most popular ways of fixing alignment faults:

- Align the data.
- Hint to the compiler that the data about to be accessed is (or may be) unaligned.
- Copy the data to an aligned buffer.

Align the Data

Aligning the data is the best solution for avoiding alignment faults.

Today's compilers are smart enough to detect alignment faults problems most of the time and add code to access the data through multiple loads, shifts, and masks.
Sometimes it is not possible or not practical to align the data. Such examples would be when transferring data between systems or when reading/write from/to fixed record layout in a file.

Make sure fields within data structures are naturally aligned. Some compilers like C and C++ do this by default. In MACRO, use .align [quad|long]. In SDL, use basealign [quad|long]

Hints to the Compiler

Programming languages may support declaration modifiers that will cause predicated code to be generated that will test for unaligned data and operate on it in such a way as to preclude alignment faults.
Language support includes:

- __unaligned (C)
- .set_registers unaligned=<Rx> (Macro)
- align(x) (Bliss32/Bliss64)
- aligned(x) (Pascal)

Using the options will eliminate the alignment faults. However, code accessing aligned data will be slower than normal.

Remember – the extra code generated when giving hints to the compiler that data maybe unaligned will *perform much better than hitting an alignment fault*.

Let's modify the NOT_ALIGNED program to declare that the pointer for the random data is unaligned:

```
$ ty not_aligned.c
#include <far_pointers>
#include <gen64def>
#include <ints>
#include <starlet>
#include <stdio>
#include <stdlib>
#include <lib$routines.h>
#include <unistd.h>
#include <stsdef>

#define random_key(upper_bound) (abs (random () % upper_bound))

void main()
{
int             NumberOfBytes    =      1000000000;     // 1GB using marketing bytes
int             status;
VOID_PQ         MappedVA;
INT64_PQ        RandomVA;


        lib$init_timer();               // initialize timer

        //
        // Allocate 1GB from P2 space
        //
        status = lib$get_vm_64 (&NumberOfBytes, &MappedVA);

        if (!$VMS_STATUS_SUCCESS(status))
        {
                lib$signal (status);
                return;
        }

        //
        // Force the pointer to become unaligned
        //
```

```
        RandomVA = (INT64_PQ)((char *) MappedVA + 1);

        for (int i=0; i<50000000; i++)
        {

                // Increment a random Quadword – pointer now declared unaligned
                __int64 __unaligned *MyData = &RandomVA [random_key((100000000/8) -1)];
                *MyData = *MyData + 1;

        }

        //
        // Free VM
        //
        status = lib$free_vm_64 (&NumberOfBytes, &MappedVA);

        if (!$VMS_STATUS_SUCCESS(status))
        {
                lib$signal (status);
                return;
        }

        lib$show_timer();
}
$ cc/pointer=long not_aligned.c
$ link not_aligned
$ r not_aligned
 ELAPSED:    0 00:00:20.74  CPU: 0:00:20.67  BUFIO: 0  DIRIO: 0  FAULTS: 703741
$
```

Now our program completed in 20.74 seconds…this is a big improvement comparing to 3 minutes and 45 seconds when the compiler was not expecting unaligned data.

Copying the Data

The last option for fixing alignment faults is to copy the data to an aligned buffer. This approach is useful when the data itself is aligned but the buffer containing the data is not.

If the amount of data that needs to be moved is small and many references are made to it, then copying the data is a good idea. However, if the quantity of data to be moved is large and only a small number of references are made to it, then it is better to take a few alignment faults and leave the data alone.

Summary

From a performance standpoint, Alignment faults are expensive on AlphaServer systems but are VERY expensive on Intel® Itanium® 2 systems. For achieving good performance on the latter, alignment faults need to be resolved. OpenVMS allows monitoring alignment faults using the MONITOR ALIGN command, the FLT extension in SDA, and the debugger.

To avoid alignment faults, naturally align the data, declare pointers to be unaligned, or copy the data to an aligned buffer where it makes sense.

For more information

To get to the latest issue of the OpenVMS Technical Journal, go to:
http://www.hp.com/go/openvms/journal

Author Bio

Guy Peleg joined BRUDEN-OSSG last September, he is a Senior Member of the Technical Staff and Directorof EMEA Operations. Prior to joining BRUDEN-OSSG, he was a software engineer in the OpenVMS Engineering group working on the various utilities. He was part of the team ported OpenVMS to Integrity Server Platforms (IPF), he lead the LMF port to IA64, the EDCL project and various virtualization projects on IPF. Before joining Engineering, Guy provided customer support and consulting with Compaq/DEC in their field offices. He is known worldwide for his commitment to the OpenVMS customer. He has given numerous technical presentations and has been published in the OpenVMS Systems Technical Journal. His presentations are entertaining and highly informative.

