[TD 2570-CR] EV6 Interlocked Memory Instructions for Alpha Compilers - BLITZ

Unique Id: 009D0AFA-95035BA0-1C02A1

SOURCE: Compaq Computer Corporation INFORMATION BLITZ

INFORMATION BLITZ TITLE:

Ensuring Proper Use of Interlocked Memory Instructions for Applications to Run on Alpha 21264 (EV6) processors.

DATE: December 11th 1998
INFORMATION BLITZ #: 2570-CR

AUTHOR: Karen Marion
TEL#: 603-884-1455
DTN: 264-1455
EMAIL: karen.marion@digital.com
DEPARTMENT: OpenVMS Engineering

==========

PRODUCT NAME(S) IMPACTED: Compilers for Use on OpenVMS Alpha:
BLISS V1.1
DEC C V5.x
DEC C++ V5.x
DEC Pascal V5.0-2
MACRO-32 V3.0
MACRO-64 V1.2

PRODUCT FAMILY:		PRODUCT NUMBERS:
Storage	___	_____
Systems	___	_____
Networks	___	_____
PC	___	_____
Software	_X_	_____

PROBLEM STATEMENT:

The Alpha Architecture Reference Manual, Third Edition (AARM) describes strict rules for using interlocked memory instructions. The forthcoming Alpha 21264 (EV6) processor and all future Alpha processors are more stringent than their predecessors in their requirement that these rules be followed. As a result, code that has worked in the past despite noncompliance may now fail when executed on systems featuring the new 21264 processor. Occurrences of these noncompliant code sequences are believed to be rare.

NOTE: The 21264 processor is not supported prior to OpenVMS

Alpha Version 7.1-2.

PROBLEM SYMPTOM:

The result can be a loss of synchronization between processors when interprocessor locks are used or an infinite loop when an interlocked sequence always fails. This has occurred in some code sequences in programs compiled on old versions of the BLISS compiler, some versions of the MACRO-32 compiler and the MACRO-64 assembler, and in some DEC C and C++ programs.

The affected code sequences use LDx_L/STx_C instructions, either directly in assembly language source or in code generated by a compiler. Applications most likely to use interlocked instructions are complex, multithreaded applications or device drivers using highly optimized, hand-crafted locking and synchronization techniques.

SOLUTION:

OpenVMS recommends that code that will run on the 21264 processor be checked for these sequences. Particular attention should be paid to any code that does interprocess locking, multithreading, or interprocessor communication.

The SRM_CHECK tool (named after the System Reference Manual, which defines the Alpha architecture) has been developed to analyze Alpha executables for noncompliant code sequences. The tool will detect sequences that may fail, report any errors, and display the machine code of the failing sequence.

The SRM_CHECK tool can be downloaded from the following location:

www.openvms.digital.com/openvms/srm_check.exe

ADDITIONAL INFORMATION:

This section contains information about:

- Using the SRM_CHECK tool
- Characteristics of noncompliant code
- Coding requirements
- Compilers

Using SRM_CHECK.EXE to Analyze Code

To run the SRM_CHECK tool, define it as a foreign command (or use the DCL$PATH mechanism) and invoke it with the name of the image to check. If a problem is found, the machine code will be displayed and some image

information will be printed. The following example illustrates how to use the tool to analyze an image called myimage.exe:

```
$ define DCL$PATH []
$ srm_check myimage.exe
```

The tool supports wildcard searches. Use the following command line to initiate a wildcard search:

```
$ srm_check [*...]* -log
```

Use the -log qualifier to generate a list of which images have been checked. The -output qualifier can be used to write the output to a data file, as in the following example that writes to a file named check.dat.

```
$ srm_check 'file' -output check.dat
```

The output from the tool can be used to find the module that generated the sequence by looking in the image's MAP file. The addresses shown correspond directly to the addresses that can be found in the MAP file.

The following example illustrates the output from using the analysis tool on an image named system_synchronization.exe

```
 ** Potential Alpha Architecture Violation(s) found in file...
 ** Found an unexpected ldq at 00003618
 0000360C   AD970130     ldq_l          R12, 0x130(R23)
 00003610   4596000A     and            R12, R22, R10
 00003614   F5400006     bne            R10, 00003630
 00003618   A54B0000     ldq            R10, (R11)
 Image Name:    SYSTEM_SYNCHRONIZATION
 Image Ident:   X-3
 Link Time:      5-NOV-1998 22:55:58.10
 Build Ident:   X6P7-SSB-0000
 Header Size:   584
 Image Section: 0, vbn: 3, va: 0x0, flags: RESIDENT EXE (0x880)
```

The MAP file for system_synchronization.exe contains the following:

```
EXEC$NONPAGED_CODE       00000000 0000B317 0000B318 (      45848.) 2 **  5
SMPROUT                  00000000 000047BB 000047BC (      18364.) 2 **  5
SMPINITIAL               000047C0 000061E7 00001A28 (       6696.) 2 **  5
```

The address 360C is in the SMPROUT module (which contains the addresses from 0-47BB). By looking at the machine code output from the module, you can locate the code and use the listing line number to identify the corresponding source code. If SMPROUT had a nonzero base, it would be necessary to subtract the base from the address (360C in this case) to find the relative address in the listing file.

Note that the tool reports potential violations in its output. Although SRM_CHECK can normally identify a code

section in an image by the section's attributes, it is possible for OpenVMS images to contain data sections with those same attributes. As a result, SRM_CHECK may scan data as if it were code, and occasionally report a "false positive" when a block of data appears to be a noncompliant code sequence. This has also been found to be quite rare. This circumstance can be detected in the same way the noncompliant source code is found, by examining the MAP and listing files.

Characteristics of Noncompliant Code

The areas of noncompliance detected by the SRM_CHECK tool can be grouped into the following four categories. Most of these can be fixed by recompiling with new compilers. In rare cases, the source code may need to be modified. See Section 4.4 for information about compiler versions.

- Some versions of OpenVMS compilers introduce noncompliant code sequences during an optimization called "loop rotation." This problem can only be triggered in C or C++ programs which use LDx_L/STx_C instructions in assembly language code that is embedded in the C/C++ source using the ASM function, or in assembly language written in MACRO-32 or MACRO-64. In some cases, a branch was introduced between the LDx_L and STx_C instructions.

  This can be addressed by recompiling.

- Some code compiled with very old Bliss, MACRO-32, or DEC Pascal compilers may contain noncompliant sequences. Early versions of these compilers contained a code scheduling bug where a load was incorrectly scheduled after a load_locked.

  This can be addressed by recompiling.

- The MACRO-32 compiler may generate a noncompliant code sequence for a BBSSI or BBCCI instruction in rare cases where there are too few free registers.

  This can be addressed by recompiling.

- Incorrectly coded MACRO-64 or MACRO-32 and incorrectly coded assembly language embedded in C or C++ source using the ASM function.

  This requires source code changes. The new MACRO-32 compiler will flag noncompliant code at compile time.

If the SRM_CHECK tool finds a violation in an image, the image should be recompiled with the appropriate compiler (see Section 4.4). After recompiling, the image should be analyzed again. If violations remain after recompiling, source code must be examined to determine why the code

scheduling violation exists. Modifications should then be made to the source code.

Coding Requirements

The Alpha Architecture Reference Manual describes how an atomic update of data between processors must be formed. The Third Edition, in particular, has expanded greatly on this topic. In this edition, Section 5.5, "Data Sharing", and Section 4.2.4, which describes the LDx_L instructions, detail the conventions of the interlocked memory sequence.

The following two requirements are the source of all known noncompliant code:

- There cannot be a memory operation (load or store) between the LDx_L (load locked) and STx_C (store conditional) instructions in an interlocked sequence

- There cannot be a branch taken between a LDx_L and a STx_C instruction. Rather, execution must "fall through" from the LDx_L to the STx_C without taking a branch.

  Any branch whose target is between a LDx_L and matching STx_C creates a noncompliant sequence. For example, any branch to "label" in the following would result in noncompliant code, regardless of whether the branch instruction itself was within or outside of the sequence:

```
                LDx_L  Rx, n(Ry)
                ...
          label: ...
                STx_C  Rx, n(Ry)
```

Therefore, the SRM_CHECK tool looks for the following:

- Any memory operation (LDx/STx) between a LDx_L and a STx_C.

- Any branch which has a destination between a LDx_L and STx_C.

- STx_C instructions that do not have a preceding LDx_L instruction.

  This typically indicates that a backward branch is taken from a LDx_L to the STx_C. Note that hardware device drivers that do device mailbox writes are an exception, and use the STx_C to write the mailbox. This is only found on early Alpha systems, and not on PCI based systems.

- Excessive instructions between a LDx_L and STxC.

  The AARM recommends that no more than 40 instructions

appear between a LDx_l and STx_c. In theory, more than 40 instructions can cause hardware interrupts to keep the sequence from completing. There are no known occurrences of this.

To illustrate, the following are examples of code flagged by SRM_CHECK.

```
        ** Found an unexpected ldq at 0008291C
        00082914   AC300000     ldq_l          R1, (R16)
        00082918   2284FFEC     lda            R20, 0xFFEC(R4)
        0008291C   A6A20038     ldq            R21, 0x38(R2)
```

In the above example, a LDQ instruction was found after a LDQ_L before the matching STQ_C. The LDQ must be moved out of the sequence, either by recompiling or by source code changes.

```
* Backward branch from 000405B0 to a STx_C sequence at 0004059C
00040598   C3E00003     br             R31, 000405A8
0004059C   47F20400     bis            R31, R18, R0
000405A0   B8100000     stl_c          R0, (R16)
000405A4   F4000003     bne            R0, 000405B4
000405A8   A8300000     ldl_l          R1, (R16)
000405AC   40310DA0     cmple          R1, R17, R0
000405B0   F41FFFFA     bne            R0, 0004059C
```

In the above example, a branch was discovered between the LDL_L and STQ_C. In this case, there is no "fall through" path between the LDx_L and STx_C, which the architecture requires.

_____ Note _____

This branch backward from the LDx_L to the STx_C is characteristic of the noncompliant code introduced by the "loop rotation" optimization.

_____

The following MACRO-32 source code demonstrates code where there is a fall through path, but that is still noncompliant because of the potential branch, AND a memory reference in the lock sequence.

```
 getlck: evax_ldql  r0, lockdata(r8)  ; get the lock data
         movl       index, r2         ; and the current index
         tstl       r0                ; If the lock is zero,
         beql       is_clear          ; skip ahead to store
         movl       r3, r2            ; Else, set special index
is_clear:
         incl       r0                ; increment lock count
         evax_stqc  r0, lockdata(r8)  ; and store it
         tstl       r0                ; did store succeed?
         beql       getlck            ; retry if not
```

To correct this code, the memory access to read the value of INDEX must first be moved outside the LDQ_L/STQ_C sequence. Next, the branch between the LDQ_L and STQ_C, to the label IS_CLEAR, must be eliminated. In this case, it could be done using a CMOVEQ instruction. The CMOVxx instructions are frequently useful for eliminating branches around simple value moves. The following example shows the corrected code.

```
        movl       index, r2          ; Get the current index
getlck: evax_ldql  r0, lockdata(r8)   ; and then the lock data
        evax_cmoveq r0, r3, r2        ; If zero, use special index
        incl       r0                 ; increment lock count
        evax_stqc  r0, lockdata(r8)   ; and store it
        tstl       r0                 ; did write succeed?
        beql       getlck             ; retry if not
```

Compiler Versions

This section contains information about versions of compilers that may generate noncompliant code sequences and the recommended versions to be used when recompiling.

Table 1 contains information for OpenVMS compilers.

Table 1 OpenVMS_Compilers

Old Version	Recommended Minimum Version
BLISS V1.1	Bliss V1.3
DEC C V5.x	DEC C V6.0
DEC C++ V5.x	DIGITAL C++ V6.0
DEC Pascal V5.0-2	DEC Pascal V5.1-11
MACRO-32 V3.0	V3.1 for OpenVMS Version 7.1-2 V4.1 for OpenVMS Version 7.2
MACRO-64 V1.2	See below

Current versions of the MACRO-64 Assembler may still encounter the loop rotation issue. However, MACRO-64 does not perform code optimization by default, and this problem can only arise when optimization is enabled. If SRM_CHECK indicates a noncompliant sequence in MACRO-64 code, it should first be recompiled without optimization. If the sequence is still flagged when retested, the source code itself contains a noncompliant sequence and must be corrected.

```
<>UPDATE /TEXT_UPDATE/UNIQUE_IDENTIFIER="009D0AFA-95035BA0-1C02A1"-
 /TITLE="[TD 2570-CR] EV6 Interlocked Memory Instructions for Alpha Compilers - BLITZ
 /BADGE=(AUTHOR="999997",ENTER="913696",MODIFY="913696",-
 EDITORIAL_REVIEW="913696",TECHNICAL_REVIEW="999997")-
 /NAME=(AUTHOR="MARION KAREN",ENTER="SPAINHOWER JOE",-
 MODIFY="SPAINHOWER JOE",EDITORIAL_REVIEW="SPAINHOWER JOE",TECHNICAL_REVIEW="MARION K
 /DATE=(AUTHOR="14-DEC-1998",ENTER="14-DEC-1998",-
 EXPIRE="14-DEC-2000",FLASH="15-JAN-1999 10:48:03.91",MODIFY="15-JAN-1999",-
 EDITORIAL_REVIEW="14-DEC-1998",TECHNICAL_REVIEW="14-DEC-1998")-
 /GEOGRAPHY="USA"/SITE="EIRS"/OWNER="TIM-BLITZ"-
 /FLAGS=(USA_CUSTOMER_READABLE,NOPOST_MESSAGE_DISPLAY,NOLOCAL,-
 EUR_CUSTOMER_READABLE,GIA_CUSTOMER_READABLE,NOINIT_MESSAGE_DISPLAY,-
 EDITORIAL_REVIEWED,FIELD_READABLE,FLASH,TECHNICAL_REVIEWED,READY)
```