Intel® AVX State Transitions: Migrating SSE Code to AVX

Submitted by Chris Kirkpatrick (Intel) (https://software.intel.com/en-us/user/510359) on August 2, 2012

Share (https://www.facebook.com/sharer/sharer.php?u=https://software.intel.com/en-us/articles/intel-avx-state-transitions-migrating-sse-code
Tweet (https://twitter.com/intent/tweet?text=Intel%C2%AE+AVX+State+Transitions%3A+Migrating+SSE+Code+to+AVX%3A&url=https%3A%2
Share (https://plus.google.com/share?url=https://software.intel.com/en-us/articles/intel-avx-state-transitions-migrating-sse-code-to-avx)

Introduction

Intel® Advanced Vector eXtensions (AVX) are the latest instruction set addition to the IA-32 and IA-64 architectures. They provide enhanced 256-bit SIMD operations on 8-wide floating-point vectors for Intel® 2nd Generation Core®™ processors code named Sandy Bridge and later processors. When porting or optimizing applications for AVX, one must pay careful attention to the transitions between SSE code and AVX code. Failure to do so can result in performance penalties. This document will detail these performance penalties, as well as an overview of the tools available for helping to detect and avoid situations where these performance penalties are encountered. This document assumes familiarity with both SSE and AVX. For an introduction to Intel® AVX, see the document “Intel® Advanced Vector Extensions” at http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions (/en-us/articles/introduction-to-intel-advanced-vector-extensions)

Instruction Set Definitions

Throughout this document, the following abbreviations will be used to refer to instruction sets.

- The SSE instruction sets will be referred to as “legacy SSE”
- The Intel® AVX 128-bit instructions will be referred to as AVX-128
- The Intel® AVX 256-bit instructions will be referred to as AVX-256

Intel® AVX Operating States

The Intel® AVX instruction set expands the existing 128-bit XMM register set to 256-bit YMM registers. This creates a situation where partial registers can be accessed, for example accessing the 128-bit lower half of a 256-bit YMM register via legacy SSE instructions. Correctness requires that the processor preserve the upper 128-bits of a 256-bit YMM register while executing 128-bit legacy SSE code and restore this state when transitioning back to 256-bit AVX code. The action of preserving and restoring of the upper state can incur a performance penalty unless the proper bits are known to be zero.

The upper 128-bits of the 256-bit YMM registers can operate in three possible states:

- State A: The upper half of ***ALL*** YMM registers is known to be zero. This can be considered to be the program starting state prior to executing any AVX-256 instructions.
- State B: The upper half of ***ANY*** YMM register is not known to be zero. This state occurs as a result of executing an AVX-256 instruction.
- State C: The upper half of ***ANY*** YMM register is not known to be zero whilst executing a legacy SSE instruction. The upper half off ALL YMM registers must be saved/restored by internal hardware as required.

Figure 1 depicts the three operating states. The transitions from each respective state will be described in the following sections.

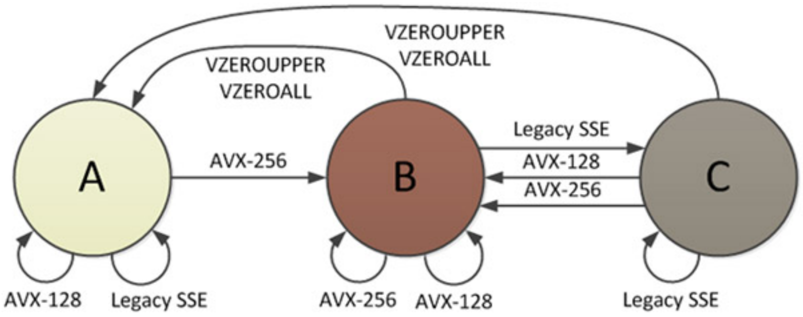


Figure 1: Register File Operating States

State A

When operating in State A, the upper half of ***ALL*** the YMM registers is known to be zero. As long as 128-bit instructions are being executed-whether those instructions are legacy SSE instructions or AVX-128 instructions-no performance penalty will occur. Execution of any AVX-256 instruction will remove the known zero status of the upper 128-bits of the instruction's target YMM register, causing a transition into State B.

State B

When operating in State B, the full 256-bit YMM register is in use. AVX-256 and AVX-128 instructions can be executed without penalty. AVX-128 instructions execute penalty free because the upper half of the instruction's target register is zeroed upon execution.

If legacy SSE instructions are executed while in State B, a transition to State C occurs. Because the zero state of the YMM registers is not known, ***the transition to State C happens regardless of whether the upper 128-bits of the YMM registers are zero***. Transitions to/from State C are undesirable because the upper 128-bits of ALL YMM registers must be stored in an internal register buffer before execution of the legacy SSE instruction can begin. The same penalty is paid when transitioning out of State C. The cost of the penalty is on the order of 50-80 clock cycles on Sandy Bridge hardware.

The correct method for transitioning to legacy SSE code from AVX code is to clear the upper 128-bits of ***ALL*** YMM registers, forcing a transition to State A. AVX provides two instructions to accomplish this:

- VZEROALL: Zero out the contents of ***ALL*** YMM registers
- VZEROUPPER: Zero out the upper 128-bits of ***ALL*** YMM registers.

The penalty paid for transitions utilizing these instructions is only the execution cost of the instruction-a single cycle on modern hardware.

State C

Operating in State C is undesirable. Whenever control is transferred from State B into State C, the upper half of ***ALL*** YMM registers must be saved to an internal register buffer. The same penalty occurs when transferring control out of State C as the contents of the register buffer must be restored into each respective YMM register. While in this operating state, any number of legacy SSE instructions can be executed. The penalty paid is only the cost for the two transitions. A summary of the transition penalties between the three operating states is displayed in Figure 2.

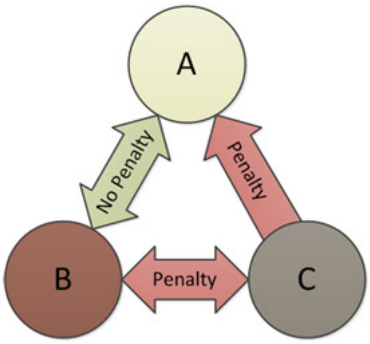


Figure 2: Performance of Register State Transitions

Software Tools for Detecting AVX/SSE Translation Penalties

As mentioned previously, Intel® AVX was designed for uniform blocks of legacy SSE or AVX code. As always, it is best to profile applications to detect whether translation penalties are a bottleneck of the program.

Intel®VTune Amplifier XE

Intel® VTune Amplifier XE provides two hardware events for tracking Intel® AVX transitions within programs. It comes in both precise (_PS) and non-precise versions, depending on the hardware utilizing the profiler. The hardware events are:

- OTHER_ASSISTS.AVX_TO_SSE: The number of penalty transitions from AVX-256 to legacy SSE
- OTHER_ASSISTS.SSE_TO_AVX: The number of penalty transitions from legacy SSE to AVX-256

For a detailed walkthrough of using Intel® VTune Amplifier XE to discover transition penalties, see section 2.2 of the document “Avoiding AVX-SSE Transition Penalties” at http://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties (/en-us/articles/avoiding-avx-sse-transition-penalties)

Intel Software Development Emulator

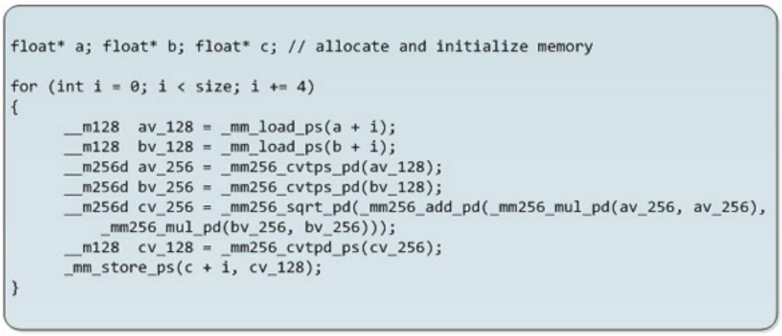
The Intel® Software Development Emulator (SDE) comes with a built-in AVX/SSE transition checker. Because it is an emulator, some real-time applications may run poorly inside the emulation environment. When testing specific areas of large real-time systems, it is better to utilize Intel® VTune Amplifier XE.

The Intel® Software Development Emulator can be downloaded free for Windows and Linux from http://software.intel.com/en-us/articles/intel-software-development-emulator (/en-us/articles/intel-software-development-emulator)

The example below will use the source code found in figure 1 of “Avoiding AVX-SSE Transition Penalties” at http://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties (/en-us/articles/avoiding-avx-sse-transition-penalties). The source code found in the article is duplicated below for convenience.

The source code in figure 3 iterates over each pair of elements from two given arrays and calculates the distance between each pair of values using the following formula:

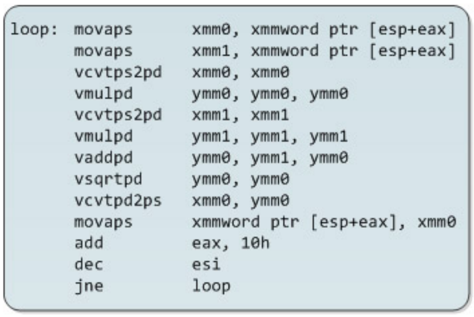
$$c[i] = \sqrt{(a[i])^2 + (b[i])^2}$$



```
float* a; float* b; float* c; // allocate and initialize memory

for (int i = 0; i < size; i += 4)
{
      __m128  av_128 = _mm_load_ps(a + i);
      __m128  bv_128 = _mm_load_ps(b + i);
      __m256d av_256 = _mm256_cvtps_pd(av_128);
      __m256d bv_256 = _mm256_cvtps_pd(bv_128);
      __m256d cv_256 = _mm256_sqrt_pd(_mm256_add_pd(_mm256_mul_pd(av_256, av_256),
          _mm256_mul_pd(bv_256, bv_256)));
      __m128  cv_128 = _mm256_cvtpd_ps(cv_256);
      _mm_store_ps(c + i, cv_128);
}
```

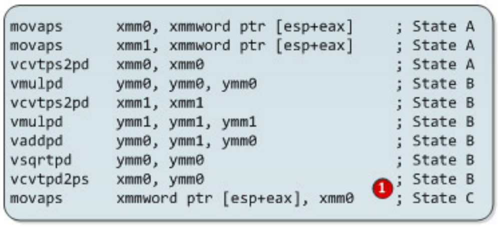
Figure 3: Calculates the distance between pairs of values $(a[i], b[i])$



```
loop:  movaps      xmm0, xmmword ptr [esp+eax]
       movaps      xmm1, xmmword ptr [esp+eax]
       vcvtps2pd   xmm0, xmm0
       vmulpd      ymm0, ymm0, ymm0
       vcvtps2pd   xmm1, xmm1
       vmulpd      ymm1, ymm1, ymm1
       vaddpd      ymm0, ymm1, ymm0
       vsqrtpd     ymm0, ymm0
       vcvtpd2ps   xmm0, ymm0
       movaps      xmmword ptr [esp+eax], xmm0
       add         eax, 10h
       dec         esi
       jne         loop
```

Figure 4: Assembly code listing for figure 3

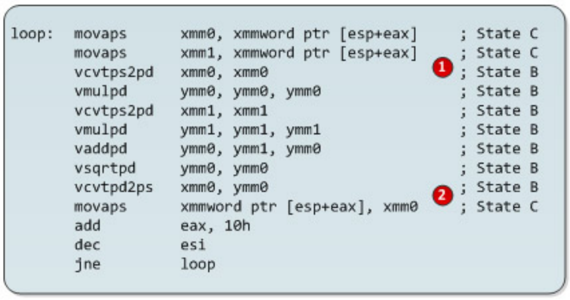
Assume that upon entry to the loop, the program is in State A. The state transitions occur as follows:



```
movaps      xmm0, xmmword ptr [esp+eax]    ; State A
movaps      xmm1, xmmword ptr [esp+eax]    ; State A
vcvtps2pd   xmm0, xmm0                     ; State A
vmulpd      ymm0, ymm0, ymm0               ; State B
vcvtps2pd   xmm1, xmm1                     ; State B
vmulpd      ymm1, ymm1, ymm1               ; State B
vaddpd      ymm0, ymm1, ymm0               ; State B
vsqrtpd     ymm0, ymm0                     ; State B
vcvtpd2ps   xmm0, ymm0                   ❶; State B
movaps      xmmword ptr [esp+eax], xmm0    ; State C
```

A single transition penalty for $\mathrm{AVX} \rightarrow \mathrm{SSE}$ occurs at location ❶ in the code listing.

The second, and all subsequent, iterations of the loop execute as follows:



```
loop:  movaps      xmm0, xmmword ptr [esp+eax]    ; State C
       movaps      xmm1, xmmword ptr [esp+eax]  ❶ ; State C
       vcvtps2pd   xmm0, xmm0                     ; State B
       vmulpd      ymm0, ymm0, ymm0               ; State B
       vcvtps2pd   xmm1, xmm1                     ; State B
       vmulpd      ymm1, ymm1, ymm1               ; State B
       vaddpd      ymm0, ymm1, ymm0               ; State B
       vsqrtpd     ymm0, ymm0                     ; State B
       vcvtpd2ps   xmm0, ymm0                   ❷ ; State B
       movaps      xmmword ptr [esp+eax], xmm0    ; State C
       add         eax, 10h
       dec         esi
       jne         loop
```

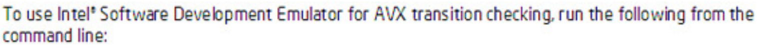
Note that because the first iteration of the loop leaves execution inside State C, subsequent loops suffer two transition penalties: one penalty for moving from SSE ? AVX, and one for moving from AVX -> SSE.

For this example, assume an array size of 4000 elements. The loop operates on 4 elements at a time. Then it is expected that the number of AVX -> SSE transitions should be equal to the number of iterations of the loop-1000 penalties.

Because during the first iteration of the loop, no SSE ? AVX penalty is paid, there should be one less penalty than in the AVX ? SSE case for a total of 999 penalties. Below this expected behavior will be verified using Intel® Software Development Emulator.

1. Run SDE on the Compiled Module

To use Intel® Software Development Emulator for AVX transition checking, run the following from the command line:



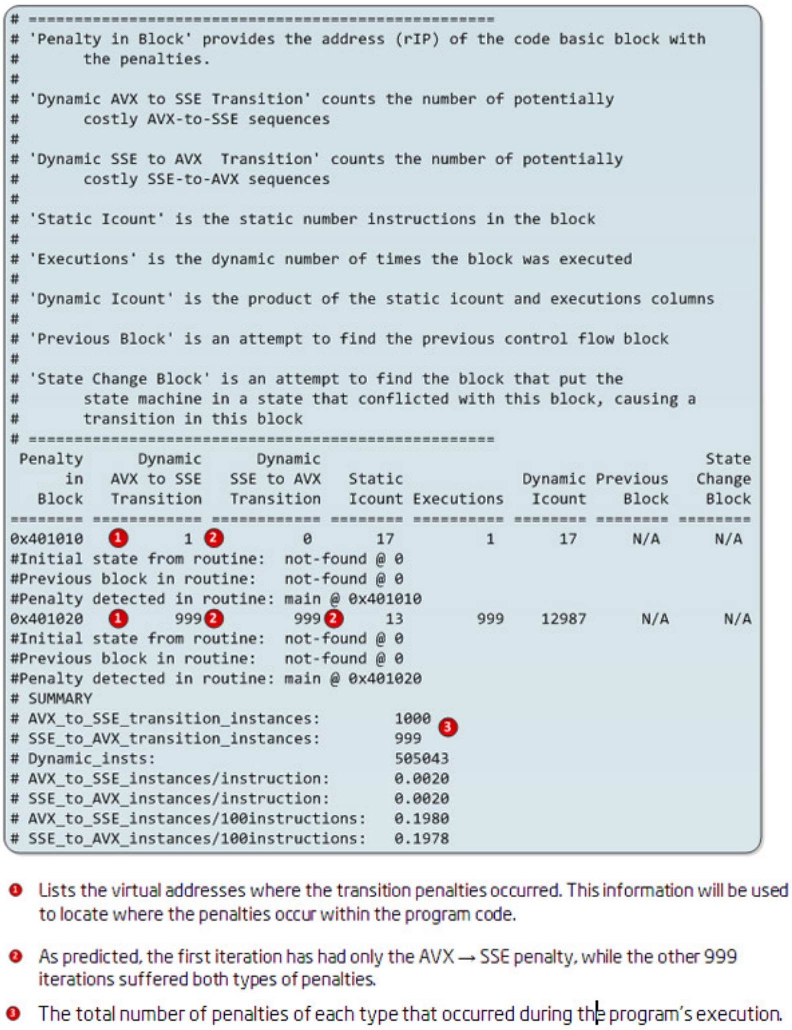
To use Intel® Software Development Emulator for AVX transition checking, run the following from the command line:

This will run the transition checker, and produce an output file in the directory of execution.

2. Analyze the Output

Open in any available text editor. The file will detail the locations of transition penalties that occurred during the execution of the program.

Below is the output from SDE on the sample code given above using the array size of 4000.



```
# ==================================================
# 'Penalty in Block' provides the address (rIP) of the code basic block with
#       the penalties.
#
# 'Dynamic AVX to SSE Transition' counts the number of potentially
#       costly AVX-to-SSE sequences
#
# 'Dynamic SSE to AVX  Transition' counts the number of potentially
#       costly SSE-to-AVX sequences
#
# 'Static Icount' is the static number instructions in the block
#
# 'Executions' is the dynamic number of times the block was executed
#
# 'Dynamic Icount' is the product of the static icount and executions columns
#
# 'Previous Block' is an attempt to find the previous control flow block
#
# 'State Change Block' is an attempt to find the block that put the
#       state machine in a state that conflicted with this block, causing a
#       transition in this block
# ==================================================
 Penalty      Dynamic      Dynamic                                      State
    in      AVX to SSE   SSE to AVX   Static             Dynamic Previous  Change
   Block    Transition   Transition   Icount Executions   Icount   Block    Block
======== ============ ============ ======== ========== ======== ======== ========
0x401010  ①        1 ②          0       17          1       17      N/A      N/A
#Initial state from routine:  not-found @ 0
#Previous block in routine:   not-found @ 0
#Penalty detected in routine: main @ 0x401010
0x401020  ①      999 ②        999 ②     13        999    12987      N/A      N/A
#Initial state from routine:  not-found @ 0
#Previous block in routine:   not-found @ 0
#Penalty detected in routine: main @ 0x401020
# SUMMARY
# AVX_to_SSE_transition_instances:        1000 ③
# SSE_to_AVX_transition_instances:        999
# Dynamic_insts:                          505043
# AVX_to_SSE_instances/instruction:       0.0020
# SSE_to_AVX_instances/instruction:       0.0020
# AVX_to_SSE_instances/100instructions:   0.1980
# SSE_to_AVX_instances/100instructions:   0.1978
```

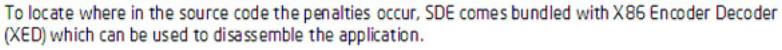
① Lists the virtual addresses where the transition penalties occurred. This information will be used to locate where the penalties occur within the program code.

② As predicted, the first iteration has had only the AVX → SSE penalty, while the other 999 iterations suffered both types of penalties.

③ The total number of penalties of each type that occurred during the program's execution.

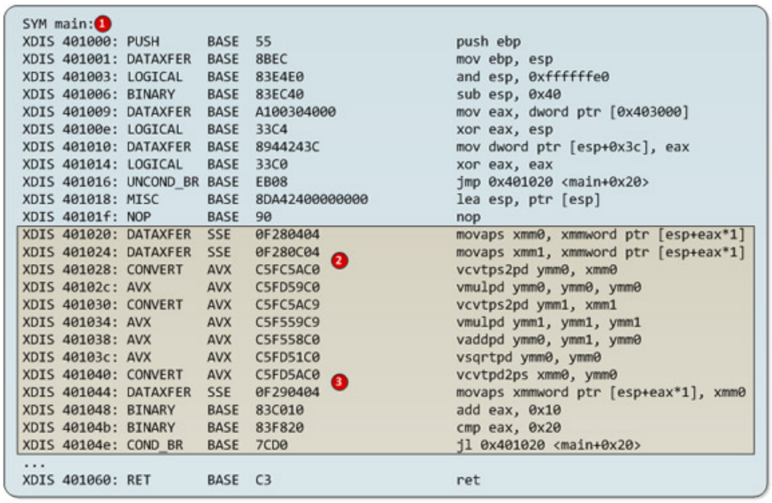
3. Locating Penalties Using Virtual Addresses

To locate where in the source code the penalties occur, SDE comes bundled with X86 Encoder Decoder (XED) which can be used to disassemble the application.



To locate where in the source code the penalties occur, SDE comes bundled with X86 Encoder Decoder (XED) which can be used to disassemble the application.

The output of XED for the sample program above is listed below. Only the relevant block of addresses are included.



```
SYM main:❶
XDIS 401000: PUSH       BASE  55                       push ebp
XDIS 401001: DATAXFER   BASE  8BEC                     mov ebp, esp
XDIS 401003: LOGICAL    BASE  83E4E0                   and esp, 0xffffffe0
XDIS 401006: BINARY     BASE  83EC40                   sub esp, 0x40
XDIS 401009: DATAXFER   BASE  A100304000               mov eax, dword ptr [0x403000]
XDIS 40100e: LOGICAL    BASE  33C4                     xor eax, esp
XDIS 401010: DATAXFER   BASE  8944243C                 mov dword ptr [esp+0x3c], eax
XDIS 401014: LOGICAL    BASE  33C0                     xor eax, eax
XDIS 401016: UNCOND_BR  BASE  EB08                     jmp 0x401020 <main+0x20>
XDIS 401018: MISC       BASE  8DA42400000000           lea esp, ptr [esp]
XDIS 40101f: NOP        BASE  90                       nop
XDIS 401020: DATAXFER   SSE   0F280404                 movaps xmm0, xmmword ptr [esp+eax*1]
XDIS 401024: DATAXFER   SSE   0F280C04  ❷              movaps xmm1, xmmword ptr [esp+eax*1]
XDIS 401028: CONVERT    AVX   C5FC5AC0                 vcvtps2pd ymm0, xmm0
XDIS 40102c: AVX        AVX   C5FD59C0                 vmulpd ymm0, ymm0, ymm0
XDIS 401030: CONVERT    AVX   C5FC5AC9                 vcvtps2pd ymm1, xmm1
XDIS 401034: AVX        AVX   C5F559C9                 vmulpd ymm1, ymm1, ymm1
XDIS 401038: AVX        AVX   C5F558C0                 vaddpd ymm0, ymm1, ymm0
XDIS 40103c: AVX        AVX   C5FD51C0                 vsqrtpd ymm0, ymm0
XDIS 401040: CONVERT    AVX   C5FD5AC0  ❸              vcvtpd2ps xmm0, ymm0
XDIS 401044: DATAXFER   SSE   0F290404                 movaps xmmword ptr [esp+eax*1], xmm0
XDIS 401048: BINARY     BASE  83C010                   add eax, 0x10
XDIS 40104b: BINARY     BASE  83F820                   cmp eax, 0x20
XDIS 40104e: COND_BR    BASE  7CD0                     jl 0x401020 <main+0x20>
...
XDIS 401060: RET        BASE  C3                       ret
```

❶ "SYM" identifies an exported symbol. The sample code was placed within a **main** method for simplicity. By searching for the Virtual Address in the XDE output and locating the nearest symbol above the found location, the method enclosing the transition penalty can be identified.

❷ Note the transition from SSE to AVX that SDE found.

❸ Again, note the transition from AVX to SSE.

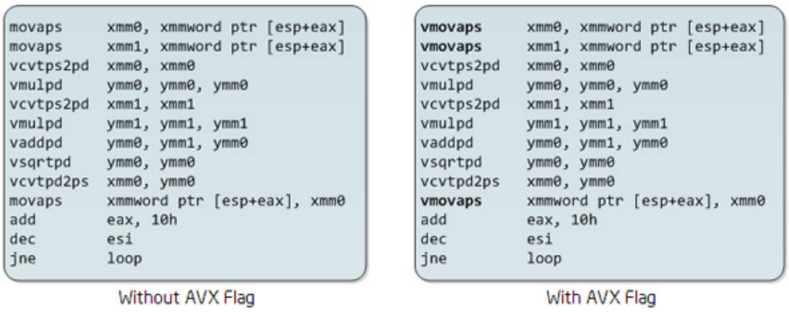
Avoiding Transition Penalties

In order to achieve maximum performance, minimizing transitions into State C is a priority. This can be achieved by keeping the upper bits of the YMM registers in the known-zero state provided by State A. Transitioning into State A is done by using the two instructions provided by AVX-VZEROALL, and VZEROUPPER.

When transitioning between blocks of legacy SSE and AVX-256 instructions, ensure that the upper half of ***ALL*** YMM registers is known to be zero when:

- Inside a module that utilizes AVX code then calling code using legacy SSE instructions
- Returning from code utilizing AVX code to legacy SSE instructions

Figure 5 shows the comparison between compiling with and without the AVX flag. Note how the legacy SSE instructions have been changed to use the AVX-128 instructions.



```
movaps     xmm0, xmmword ptr [esp+eax]
movaps     xmm1, xmmword ptr [esp+eax]
vcvtps2pd  xmm0, xmm0
vmulpd     ymm0, ymm0, ymm0
vcvtps2pd  xmm1, xmm1
vmulpd     ymm1, ymm1, ymm1
vaddpd     ymm0, ymm1, ymm0
vsqrtpd    ymm0, ymm0
vcvtpd2ps  xmm0, ymm0
movaps     xmmword ptr [esp+eax], xmm0
add        eax, 10h
dec        esi
jne        loop
```

Without AVX Flag

```
vmovaps    xmm0, xmmword ptr [esp+eax]
vmovaps    xmm1, xmmword ptr [esp+eax]
vcvtps2pd  xmm0, xmm0
vmulpd     ymm0, ymm0, ymm0
vcvtps2pd  xmm1, xmm1
vmulpd     ymm1, ymm1, ymm1
vaddpd     ymm0, ymm1, ymm0
vsqrtpd    ymm0, ymm0
vcvtpd2ps  xmm0, ymm0
vmovaps    xmmword ptr [esp+eax], xmm0
add        eax, 10h
dec        esi
jne        loop
```

With AVX Flag

Figure 5

Note: If the module is compiled with Intel® Composer XE, the compiler will also replace calls to legacy SSE instructions inside inline assembly blocks to the newer AVX-128 instructions automatically. For the behavior of other compilers, check their respective documentation.

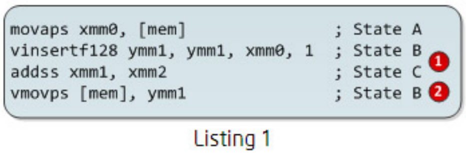
For a more detailed look at AVX Transition penalties when using Intel® Composer XE, see the document "Avoiding AVX-SSE Transition Penalties" at http://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties (/en-us/articles/avoiding-avx-sse-transition-penalties).

Example Transition Penalties

Note: The examples below are examples of where the transition penalties occur. By using modern compilers and compiler intrinsics, legacy SSE instructions will automatically be updated to the newer AVX-128 instructions. Therefore these are not examples of real code modifications.

For the following code listings, assume starting in State A.

The code found in listing 1 shows a block of assembly instructions which contains two transition penalties.

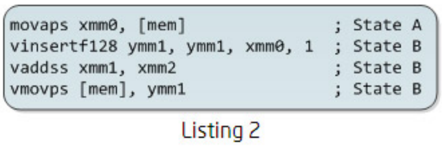


```
movaps xmm0, [mem]                 ; State A
vinsertf128 ymm1, ymm1, xmm0, 1    ; State B ❶
addss xmm1, xmm2                   ; State C
vmovps [mem], ymm1                 ; State B ❷
```

Listing 1

Listing 1

To remove these penalties, the legacy SSE instruction ADDSS is updated to the AVX-128 VADDSS instruction as shown in listing 2.

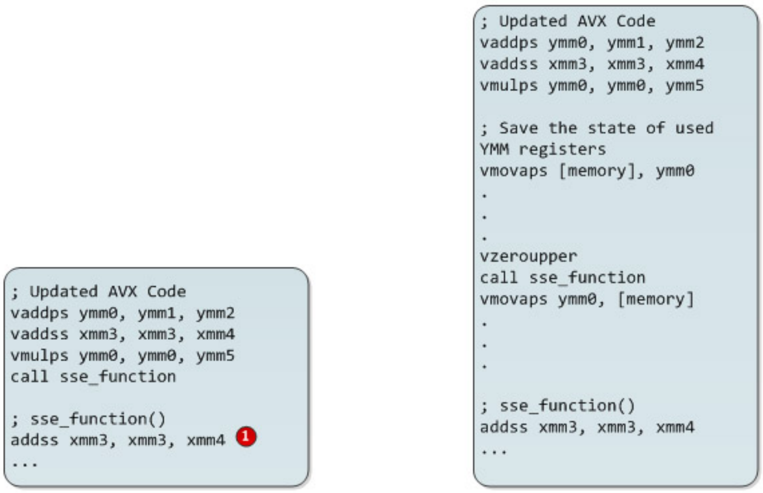


```
movaps xmm0, [mem]                 ; State A
vinsertf128 ymm1, ymm1, xmm0, 1    ; State B
vaddss xmm1, xmm2                  ; State B
vmovps [mem], ymm1                 ; State B
```

Listing 2

Listing 2

For cases where legacy SSE code is called by code which has been updated to use AVX instructions, the state of the upper 128-bits of the YMM registers must be known to be zero to avoid transition penalties. An example of this exists in listing 3.



```
; Updated AVX Code
vaddps ymm0, ymm1, ymm2
vaddss xmm3, xmm3, xmm4
vmulps ymm0, ymm0, ymm5
call sse_function

; sse_function()
addss xmm3, xmm3, xmm4 ❶
...
```

```
; Updated AVX Code
vaddps ymm0, ymm1, ymm2
vaddss xmm3, xmm3, xmm4
vmulps ymm0, ymm0, ymm5

; Save the state of used
YMM registers
vmovaps [memory], ymm0
.
.
.
vzeroupper
call sse_function
vmovaps ymm0, [memory]
.
.
.

; sse_function()
addss xmm3, xmm3, xmm4
...
```

Listing 2

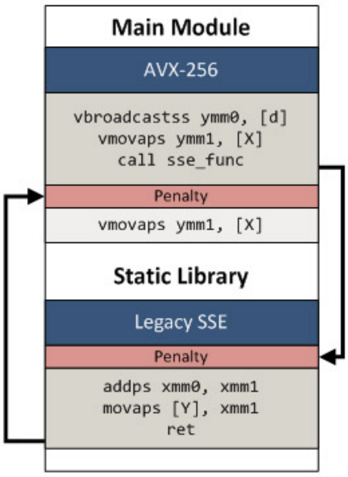
To remove the penalty, a call to VZEROUPPER is inserted before calling the legacy SSE code. In addition, the state of the YMM registers must be manually saved so they can be retrieved afterward.

Similar to the case above, functions utilizing AVX-256 instructions should terminate with a call to VZEROUPPER/VZEROALL to potentially avoid penalties caused by legacy SSE instructions in the calling code.

Linking with Existing SSE Code

In simple cases such as those detailed above, proper use of compiler intrinsics, as well as compiling the module for the AVX instruction will remove transition penalties. However, when a project links with external code which is not recompiled to utilize AVX instructions, the transition penalties may, or may not, be automatically removed by the compiler.

In cases such as this, care must be taken to ensure the upper 128-bits of the YMM registers is managed properly to avoid transitions into State C. Figure 6 details an example of a main module interacting with a statically linked library.



```
vbroadcastss ymm0, [d]
vmovaps ymm1, [X]
call sse_func
```

```
vmovaps ymm1, [X]
```

```
addps xmm0, xmm1
movaps [Y], xmm1
ret
```

Listing 6

With the main module compiled to take advantage of the AVX instruction set, calling library functions which utilize legacy SSE will cause transition penalties to occur. One penalty will occur on entry to the library function and another when AVX instructions are executed upon return to the calling code.

These transition penalties are removed by saving the state of the YMM registers and inserting a call to VZEROUPPER before calling the legacy SSE function as was seen in listing 3.

Intel® Composer XE will automatically insert the necessary calls to remove the transition penalties when interacting with statically linked libraries. However, in cases where the compiler does not automatically insert the calls to VZEROUPPER/VZEROALL, the intrinsic _mm256_zeroupper is used to manually return to execution State A before calling legacy SSE code.

When the external library is compiled to use AVX instructions while the main module uses legacy SSE code, ensure that the library code calls VZEROUPPER/VZEROALL before returning to the calling code to remove transition penalties.

Conclusion

When upgrading high-performance applications to utilize the new Intel® AVX instruction set, it is important to watch for critical loops involving costly transitions between AVX and legacy SSE code. When deciding to integrate AVX code into a project, keep the following in mind:

- Leave the YMM registers in a known zero state prior to executing legacy SSE code by using the VZEROUPPER or VZEROALL instructions.
- If the external code utilizes AVX instructions, set the YMM registers to a known-zero state before returning to the caller by calling VZEROUPPER or VZEROALL.

For a more detailed look at the Intel®AVX instruction set, see the document "Intel®Advanced Vector Extensions Programming Reference" at http://software.intel.com/en-us/avx (http://software.intel.com/en-us/avx)

For a more detailed look at Intel®Architecture, see the "Intel® 64 and IA-32 Architectures Software Developer Manuals" at http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html)

For another look at avoiding transition penalties using Intel® Composer XE see the document "Avoiding AVX-SSE Transition Penalties" at http://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties (/en-us/articles/avoiding-avx-sse-transition-penalties)

About the Author

Chris Kirkpatrick is an intern Software Engineer in the Software and Services Group of Intel Corporation where he enjoys specializing in computer graphics and software optimization. When in leisure, Chris enjoys reading and writing music.

For more complete information about compiler optimizations, see our Optimization Notice (/en-us/articles/optimization-notice#opt-en).

Categories: Intel® Advanced Vector Extensions (/en-us/search/site/field_technology/intel_advanced_vector_extensions-20857/language/en) , Intel® Streaming SIMD Extensions (/en-us/search/site/field_technology/intel_streaming_simd_extensions-20861/language/en)

Add a Comment

^Top

(For technical discussions visit our developer forums. For site or software product issues contact support.)

Please sign in to add a comment. Not a member? Join today >

Rate Us

Terms of Use *Trademarks F

English >

English ▾

We want your feedback to improve our website! This is for Intel® Developer Zone feedback only. If you need support for technical issues please post to **forums**, for non-technical site/program/account issues contact **front line support**.

Would you recommend Intel® Developer Zone to a friend?

Not at all likely Extremely likely

0 1 2 3 4 5 6 7 8 9 10