Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon™ Processor

Version 2.1

05/01

Order Number: 248674-002

Table of Contents

1 Introduction 5
2 Problem Description and Solution 5
2.1 Issues with Spin-wait Loops 5
2.1.1 Mechanics of Using the Pause Instruction 7
2.2 Data Placement and Cache Line Size 7
3 Identifying Problem Areas 9
4 Summary 10
5 Examples 10
5.1 A Sample Spin-wait Lock 10
5.2 Another Spin-wait Sample 10

Revision History

Revision	Revision History	Date
2.1	Updated to change code name to product name	05/01
2.0	Updated to use the approved terminology of the Pentium® 4 Processor	07/00
1.0	Original publication of document	02/00

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

Intel Corporation, *Intel® C/C++ Compiler User's Guide*, document number 741901, 1999.

1 Introduction

Parallel programs with multiple threads must use synchronization techniques in order to insure correct operation. Generally, synchronization operations use shared synchronization variables and "spin-wait" loops that check on the values of those variables. Starting from the Intel® Pentium® 4 and Xeon™ processors, Intel® IA-32 architecture provides a new instruction to address the performance issues associated with spin loops.

This application note addresses two important optimization issues for multi-threading computations involving high-speed processors: spin loop and shared-data management. Specifically, these optimizations include the use of the new `PAUSE` instruction in spin-wait loops and the placement of shared and non-shared data on different 128-byte cache lines. Intel strongly recommends using the new `PAUSE` instruction in spin-wait loops as soon as possible since it is backward compatible with all earlier IA-32 architecture. This document describes in detail the recommended changes and the reasons behind these changes.

2 Problem Description and Solution

When working with a multi-threaded system, two issues must be addressed in order to obtain good performance for synchronization. The first is the way that spin-wait loops that check the value of the shared synchronization variable are programmed, and the second is the alignment of (or actual location in memory of) the synchronization variable. In fact, this alignment issue of synchronization variables can be generalized to include the alignment of all shared data in the program, and the way non-shared data is mixed with shared data. Addressing these issues helps you avoid a severe degradation of application performance and needless consumption of system resources. By following the suggestions in the following sections you can increase performance and decrease power consumption.

2.1 Issues with Spin-wait Loops

The spin-wait loop that checks the value of the synchronization variable typically looks something like this:

```
wait_loop: cmp   eax, sync_var
           jne   wait_loop
```

The synchronization variable is in memory location `sync_var`, and its value is read once during each loop iteration. The memory location is checked over and over again, until the desired value is written into that memory location by some other thread. The other thread shares this memory address space but has an independent execution.

On a modern microprocessor, a loop like this results in the issue of multiple simultaneous read requests and these requests often execute out-of-order. On detection of a write by another processor to any load that is in progress, the processor must guarantee that no violations of memory order can occur. In doing so, the processor suffers a severe penalty. In other words, *a spin-wait loop such as this will always take a severe penalty upon exiting the loop*. This penalty also occurs on the Pentium Pro processor, the Pentium II processor and the Pentium III processor. However the penalty in these processors is modest compared with the penalty on the Pentium 4 processor where the performance penalty of exiting this loop is about 25 times more severe. On

Pentium 4, Xeon, and future processors, the penalty can be avoided by inserting a PAUSE instruction in the loop as shown:

```
wait_loop: pause
           cmp   eax, sync_var
           jne   wait_loop
```

The PAUSE instruction introduces a slight delay in the loop and effectively causes the memory requests to be issued at approximately the maximum speed of the memory system bus, approximately equal to the highest speed at which the sync_var value can be changed by another processor. There is no point in trying to issue requests any faster than this. The net effect of this usage is to improve spin-wait performance significantly. Inserting the PAUSE instruction has the added benefit of significantly reducing the power consumed during the spin-wait because fewer system resources are used.

Since this point may seem counter-intuitive, it is worth repeating: *the PAUSE instruction slows down the spin-wait loop so that overall performance can be increased*. The spin-loop does not need to be executed quickly since it is checking for an event that happens slowly. Checking the synchronization variable too quickly needlessly wastes system resources, and causes the processor to suffer a relatively severe penalty upon loop exit.

In order to avoid executing the PAUSE instruction when the synchronization is already satisfied, the wait loop can be coded as follows:

```
            cmp   eax, sync_var
            je    next_inst
wait_loop:  pause
            cmp   eax, sync_var
            jne   wait_loop
next_inst:  ...
```

It should be noted that these code fragments all have a subtle interaction with branch prediction. Modern microprocessors can execute instructions faster than they can figure out whether or not a branch will be taken. Therefore, most modern microprocessors will predict the outcome of a conditional branch instruction (like 'JE' or 'JNE') before the outcome of the branch is evaluated. The processor then executes instructions speculatively until it knows for certain whether its prediction is correct. If the processor predicted incorrectly, then all speculatively executed instructions are canceled. Since a spin-wait loop often protects shared data that another thread is using, there is a possibility that this shared data will be speculatively accessed while another thread is using it. However, it should be stressed that shared data is never modified or acted upon by speculative accesses.

There are some other techniques for avoiding problems with spin-wait loops. First, one can use standard Operating System (OS) timing services so that a lock variable is only checked periodically. This avoids the frequent checking of a synchronization variable. Another technique can be (and often is) employed within the OS itself. The OS can force the waiting thread to execute a HALT instruction, so that the thread does not consume any resources. When the OS makes the determination that the synchronization should be satisfied, the thread is 'woken up' so that it can resume processing. Neither of these solutions requires the use of a PAUSE instruction. *If you expect that a thread will wait for a significant period of time, then using a HALT instruction via an OS call is the preferred solution*.

2.1.1 Mechanics of Using the Pause Instruction

The PAUSE instruction is first introduced for the Pentium 4 processor. Technically, it is a hint to the hardware that says that the code being executed is a spin-wait loop. What the hardware does with this hint is specific to a particular processor. On the Pentium 4 processor, the instruction behaves as described above, resulting in a net performance improvement. However for all known existing IA-32 processors the instruction is interpreted as a NOP (no operation) and does not affect the program in any way. It has been verified that PAUSE is a NOP for all known Intel® architectures prior to the Pentium 4 processor. It is even known to behave as a NOP on the non-Intel x86 family processors that were available at the time of testing.

For this reason, it is highly recommended that you insert the PAUSE instruction into all spin-wait code immediately. Using the PAUSE instruction does not affect the correctness of programs on existing platforms, and it improves performance on Pentium 4 processor platforms.

There are three ways to insert the PAUSE instruction into your code:

- Use the mnemonic PAUSE, as depicted in the examples in section 5.
- Use the _mm_pause intrinsic provided by the Intel® C/C++ compiler. For more complete details on intrinsics, see the *Intel C/C++ Compiler User's Guide*.
- If all else fails, insert the instruction byte codes directly. A handy macro for doing this is:

```
#define _MM_PAUSE  {__asm{_emit 0xf3};__asm {_emit 0x90}}
```

2.2 Data Placement and Cache Line Size

Since Intel IA-32 architecture has a coherent cache, the way shared data is accessed can have a significant impact on overall performance. At first, one would think that this issue is limited to data that is clearly operated upon by multiple threads but this is not the case. Since every shared data item that is accessed is placed in cache, any data that is on that cache line is also shared. This can lead to a phenomena known as "false sharing," where two threads each access two non-shared data items that happen to reside on the same cache line. Effectively, the accesses will appear to be to shared data, since the system's unit of sharing is one cache line. This section describes the implications of cache line size and the placement of data on the performance of a multithreaded application.

When a processor wants to modify shared data it must obtain exclusive "ownership" to the cache line that contains the data. To do this, the processor issues a bus transaction, which will ensure that all other processors with copies of the cache line will invalidate the line in their caches. This bus transaction can be either a "read-for-ownership", or "invalidate" transaction. When the other processors observe one of these bus transactions, they invalidate their copy of the cache line. Subsequently, when any of those processors wants to access any data in that cache line, they must re-read the cache line using the system bus. Thus, the processor with the latest copy of the line updates memory and the requesting processor gets a copy of that updated data. In this way, every processor always works with the most recent data, and never a "stale" copy of the data.

When several processors are writing shared data, "ownership" of the cache line must pass from processor to processor. This can cause excessive bus activity unless the programmer pays careful attention to the layout of the shared data. In this document, we restrict our attention to two categories of shared data: the synchronization variable protecting a critical section and the

data accessed in that critical section. However, it should be noted that the issues highlighted in this section are generally applicable to any shared data situation.

In the best case, the cache line on which a synchronization variable resides is transmitted across the system memory bus only when the synchronization variable is updated. In the worst case, many spurious transfers occur, wasting precious system bus bandwidth. Multiple synchronization variables on the same cache line can cause some of the worst problems, although more subtle interactions are possible. Following the few simple rules outlined in this section will avoid the worst of these problems, and yield the best performance.

The ideal situation occurs when each synchronization variable resides alone on its own cache line. Additionally, there should be no other data on that cache line. To see why this is, consider the following scenario of a typical four-processor system where all four processors are competing for a lock protecting a critical section. Processor A uses a synchronization variable to lock a critical section. When processor A modifies the synchronization variable, it first obtains ownership for the cache line over the system bus, forcing processors B, C, and D to invalidate their copy of that cache line. Subsequently, processors B, C, and D re-read the cache line, find that they cannot access the critical section, and begin a spin-wait loop. Since B, C, and D have their own cached copy of the synchronization variable, and no further transactions are required on the system bus until processor A decides to release the critical section. When processor A modifies the synchronization variable, the cache line containing the variable must be written to the system bus forcing processors B, C, and D to invalidate their copies of the cache line (for the second time). Processors B, C, and D then re-read the cache line, and they all attempt to write it. One of those processors “wins” the write attempt, and forces the other processors to invalidate (yet again) and then re-read the cache line. The processors that “lose” return to their respective spin-wait loops.

Now consider the repercussions of having data from the critical section on the same cache line as the synchronization variable. While in the critical section, processor A obtains exclusive access in order to modify the data. Since processors B, C, and D are constantly reading the synchronization variable, and since the synchronization variable is on the same line as the data, processor A will lose exclusive access immediately after modifying the data. In other words, because A needs exclusive access to the *data*, it must obtain exclusive access to the *synchronization variable*. Since processors B, C, and D need shared access to the synchronization variable, the cache line containing both items must change state over and over. This situation is called “false sharing”, and it results in many wasted transactions on the system bus that can seriously impact overall system performance. This situation is avoided if the synchronization variable is on one cache line, and the critical section data on a different cache line. Then, the first cache line remains shared between the processors while processor A maintains exclusive access to the second cache line. No changes in ownership of either cache line are required until processor A releases the critical section.

When multiple synchronization variables are on the same cache line, a similar scenario involving many wasted bus transactions occurs. The cache line that is common to all these synchronization variables would be ‘owned’ by a succession of processors, and each change in ownership would involve multiple bus transactions. These situations should be studiously avoided.

The scenarios above illustrate why you need to be aware of the location of the all shared data with respect to their position on cache lines. In particular, frequently accessed synchronization variables and other shared data should be located in different cache lines. To exert control over the placement of the data, you must know the cache line size for the target platform architecture. On a Pentium III processor, the cache line size is 32 bytes. However, for a Pentium 4 processor,

the line size is 128 bytes (sub-divided or "sectored" into two 64-byte pieces). Thus, the optimal organization for a Pentium 4 processor is to have the synchronization variable alone on a 128-byte line. If that is not feasible, then having it alone on a 64-byte line would be the next best thing.

Padding the synchronization structures out to the size of a cache line is not the end of the story. You should also ensure that the structures be aligned on the cache boundary. Otherwise, the synchronization variable is not guaranteed to be alone on the cache line though it is guaranteed not to be on the same line as another synchronization structure. To force alignment, there are two techniques that can be employed. For dynamic memory, use a code fragment such as:

```
struct syn_str { int s_variable; };
void *p           = malloc ( sizeof (struct syn_str) + 127 );
syn_str * align_p = (syn_str *)( (((int) p) + 127) & -128 );
```

When using the Intel C/C++ compiler, the following fragment also works:

```
_declspec(align(128)) struct syn_str aligned_structure;
```

3 Identifying Problem Areas

The remaining task is to locate the spin-loops and synchronization variables in your application code. Often, applications rely on a standard library to provide synchronization. In such cases, you need to check for proper `PAUSE` usage and correct synchronization variable cache line alignment. Occasionally, synchronization calls are littered through the application code, and must be ferreted out one at a time. In such a context, use the VTune™ Performance Analyzer to help locate the problem areas. Once the VTune analyzer points out a problem area, a visual inspection can determine whether synchronization or some other issue is to blame.

To make it easier to find the problems, develop a test case that has a lot of work to do, uses a lot of processors, and places maximum pressure on all critical sections in the application. Then, use the setup wizard of the VTune analyzer on the application, setting up for time-based sampling (TBS). Examine the hot-spot report, organized by functions. Double-click on all functions that take a significant fraction of total application time to obtain a source level view (or assembly code view). The presence of an interlocked exchange instruction (`XCHG`) is a strong indicator of synchronization code. Also, unexpected cache activity for no apparent reason may be the result of subtle interactions between synchronization variables.

Other indications of the presence of synchronization codes are an application that has a high number of context switches (as shown by the Windows NT† Performance Monitor) and high CPU utilization, but poor scalability with the number of processors. This often points to the presence of a spin-wait loop. Another warning sign is a high number of context switches, but low CPU utilization. The low CPU utilization is likely to be due to the presence of a `HALT` instruction in the waiting loop.

4 Summary

Synchronization between threads frequently involves the use of spin-wait loops. These loops should make use of the PAUSE instruction to maximize performance and minimize power consumption. The PAUSE instruction can be added to application code now, as it is ignored on all known existing Intel architectures. Further, care should be taken to insure that synchronization variables are the only data on a cache line to minimize system bus traffic. For the Pentium 4 processor, the preferred cache size to honor is 128 bytes. If that is not possible, then honoring a 64-byte line size is the next best thing.

5 Examples

Two examples of generic spin-wait loops are given here.

5.1 A Sample Spin-wait Lock

```
get_lock:   mov   eax, 1
            xchg  eax, A          ; Try to get lock
            cmp   eax, 0          ; Test if successful
            jne   spin_loop
critical_section:
            <critical section code>
            mov   A, 0           ; Release lock
            jmp   continue
spin_loop:  pause                ; Short delay
            cmp   0, A           ; Check if lock is free
            jne   spin_loop
            jmp   get_lock
continue:
```

5.2 Another Spin-wait Sample

```
  // Come here if we didn't get the lock on the first try.
   for (;;)
   {
      for (int i=0; i < SPIN_COUNT; i++)
      {
         if (    (i & SPIN_MASK) == 0
               && m_dwLock == UNLOCKED
               && InterlockedExchange( &m_dwLock, LOCKED )==
UNLOCKED)
            return;
         #ifdef _X86_
           _mm_pause();
         #endif
      }
      SleepForSleepCount( cSleeps++ );
   }
```