Avoiding Pipeline Stalls

At the beginning of this chapter, we discussed the performance penalty of pipeline stalls. Let us look at some specific situations that can cause stalls. These situations should be avoided. Many cannot be completely removed; however, developers should make the effort of reducing their frequency whenever possible.

Spin waits and the pause Instruction

The NetBurst architecture is particularly adept at spotting sequences of instructions that it can execute out of original program order, that is, ahead of time. These sequences are characterized by:

- having no dependency on other instructions;
- not causing side effects that affect the execution of other instructions (such as modifying a global state).

When the processor spots these sequences, it executes the instructions and stores the results. The processor cannot fully retire these instructions because it must verify that assumptions made during their speculative execution are correct. To do this, the assumed instruction path and context are compared with the correct path instruction path. If the speculation was indeed correct, then instructions are retired (in program order). However, if the assumptions are wrong, a lot of things can happen. In a particularly bad case, called a full stall, all instructions in flight are terminated and retired in careful sequence, all the pre-executed code is thrown out, and the pipeline is cleared and restarted at the point of incorrect speculation—this time with the correct path.

One common sequence that the processor frequently executes out of order is the spin wait. This tight loop generally consists of a handful of assembly instructions written here in pseudo-code:

```assembly
top_of_loop:
    load x into a register
    compare to 0
    if not equal, goto top_of_loop
    else . . .
```

The three instructions (load register, compare, jump) are ones the speculative execution engine is particularly good at recognizing and blazing through. It sees that the loop does not depend on any variables being calculated by other instructions and so the sequence can be executed without fear of disturbing other instructions. In addition, it knows that if x changes value while the loop is running, this change will be caught before the instructions are retired by the processor. As a result, it grabs this sequence and executes it numerous times and very quickly. In the process, it floods the processor’s store of instructions to be retired with the repeated iterations of the loop. With no reason to slow down, the speculative execution continues to crank out the instructions at full tilt. Finally, the variable being waited on changes value. The instruction-retirement logic recognizes this change and triggers the full pipeline stall: it discards all the pre-executed iterations of
the loop that are waiting to be retired, it retires all other instructions in flight, and it determines where the pipeline should resume and sets the pipeline to that instruction.

This pipeline stall, however, is not the only downside. Dozens of loop iterations were performed needlessly. This unnecessary work tied up execution units and it flooded the reorder buffer, the area inside the processor that holds speculatively executed instructions prior to their retirement. On a processor with Hyper-Threading Technology, this extra work has a serious, detrimental impact on the performance of the other thread: it starves the second logical processor of resources, so that both threads are effectively incapable of doing any work simply because the loop is spinning so fast.

It is clear that the loop variable cannot change faster than the memory bus can update it. Hence, there is no benefit to pre-execute the loop faster than the time needed for a memory refresh. By inserting a pause instruction into a loop, the programmer tells the processor to wait (literally to do nothing) for the amount of time equivalent to this memory access. On processors with Hyper-Threading Technology, this respite enables the other thread to use all of the resources on the physical processor and continue processing.

Inserting the pause instruction can be done in one of two ways. With embedded assembly language, it is simply:

\_asm
{ 
  pause
}

Using the intrinsics in the Intel C++ compiler and newer versions of the Microsoft C/C++ compiler, the instruction is \_mm_pause(). For example, a tight loop might be:

while ( x != synchronization_variable )
  \_mm_pause();

On processors that predate Hyper-Threading Technology, the pause instruction is translated into a no-op, that is a no-operation instruction, which simply introduces a one-instruction delay.

Spin-wait Instructions on the Prescott Architecture

The Pentium 4 processor released in early 2004 and code-named Prescott includes several new instructions. Two of them are directly aimed at spin-waits, located in the operating system. The first instruction, \texttt{monitor}, watches a specified area of memory for write activity. Its companion instruction, \texttt{mwait}, associates writes to this memory block to waking up a specific processor. Since updating a variable is a write activity, by specifying its memory address and using \texttt{mwait}, a processor can simply be suspended until the variable is updated. Effectively, this enables a wait on a variable without spinning a loop.

While these two instructions are accessible by application programs, in this first release, they are intended for use by the operating system. In addition, system BIOS’s must support these instructions for them to operate correctly. Their presence in the Prescott architecture, however, is important as it suggests that Intel is working on the problem of providing resources to avoid the costs associated with spin-waits. Analysts
expect that Intel will provide other resources in future processors. As a result, make sure to check your processor’s documentation to see whether chip-level alternatives to spin-waits are available.

**Serializing Events**

Some operations do not cause full stalls, they simply force the processor to throw away instructions that were pre-executed and execute remaining instructions in order. Invariably, these operations are ones that change the state of the processor in a fundamental way, such that identical instructions performed before and after the operation will generate different results. These partial-stall operations mostly affect floating-point calculations.

*Changing Floating-point Precision*

The NetBurst architecture enables the programmer to determine the number of bits of precision the processor should use in floating-point operations. The processor can operate at a standard precision of 32 bits (single precision), 64 bits (double precision) or 80 bits (extended double precision). These values relate to how the processor manipulates and rounds data internally. Regardless of how you define your floating-point values (float or double), the processor defaults to a precision of 80 bits. Such high precision, though, imposes a performance cost. For example, the `FDIV` instruction (floating-point division) requires 23 clock cycles at 32-bit precision, 38 cycles at 64 bits, and 43 clock cycles at 80 bits.

To avail your code of the fastest possible operation, the processor should be reset to use the lowest acceptable precision. Changing the precision control is done through the `_controlfp()` function in Windows. Note that the Windows documentation refers to precisions of 24, 53, and 64 bits. These are the precision of the mantissa and correspond to the 32, 64, and 80 bit numbers (respectively) described here. The Intel compiler enables the default to be set from the command line with the `-Qpcnn` command-line switch, where `nn` is the precision in bits using the 32-64-80 formulation. The Intel compiler defaults to 64 bits. Linux programmers desirous of changing floating-point precision, should consult the documentation on the `FSTCW` and `FLDCW` instructions. (Gerber 2000) also provides an example using assembly language.

The drawback of changing the precision of floating-point operations is the conflict it might create with libraries that expect a different (generally higher precision). In such cases, you might be obliged to reset the precision. This is where the problem comes in. Resetting floating-point precision produces a serializing event: When the reset instruction is spotted, the processor performs a partial stall and forces in-order execution of instructions. This expensive step assures that the floating-point precision of all calculations is performed strictly according to the programmer’s specification.

**Handling Denormals**

Denormals are numbers that cannot be expressed using the standard bit patterns for encoding floating-point numbers. Generally, they are numbers whose value is very, very close to zero. The NetBurst core enables the programmer to specify that denormals should be converted (or flushed) to zero when using the SIMD instruction sets (SSE and SSE2). This option greatly improves performance and frequently it costs nothing since values so close might not be of practical use in an application. Changing the denormal state, so that the processor flushes to zero is done by the `SMTXCSR` and `LDMXCSR`
assembly instructions. Changing the processor’s handling of denormals is a serializing event for the same reason that changing floating-point precision does: the processor must make sure it executes instructions using the correct floating-point computations.

**Other Considerations**

Elsewhere in this book, we have presented other issues that can harm performance; for example, 64KB alias problems and false sharing. To the extent that these problems force one thread to waste execution resources, they detrimentally affect the other processor. So, in a direct sense, it’s true that you should optimize threads individually before considering their interaction on processors that use Hyper-Threading Technology. Then, once they’re optimized, try to determine if there is a pressing need for using processor affinity to schedule threads on specific physical processors. Once this is done, look for places in which the threads can hurt each other’s performance, especially in the areas of cache and bus management. If this is done correctly, you will find that Hyper-Threading Technology imposes few constraints and requirements beyond those inherent in parallel programming, and yet it delivers visibly faster performance while using a single physical processor.