6.4 Hardware multithreading

(Original section[1])

A related concept to MIMD, especially from the programmer's perspective, is *hardware multithreading*. While MIMD relies on multiple *processes* or *threads* to try to keep many processors busy, hardware multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion to try to utilize the hardware resources efficiently. To permit this sharing, the processor must duplicate the independent state of each thread. For example, each thread would have a separate copy of the register file and the program counter. The memory itself can be shared through the virtual memory mechanisms, which already support multi-programming. In addition, the hardware must support the ability to change to a different thread relatively quickly. In particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles while a thread switch can be instantaneous.

Hardware multithreading: Increasing utilization of a processor by switching to another thread when one thread is stalled.

Thread: A thread includes the program counter, the register state, and the stack. It is a lightweight process; whereas threads commonly share a single address space, processes don't.

Process: A process includes one or more threads, the address space, and the operating system state. Hence, a process switch usually invokes the operating system, but not a thread switch.

There are two main approaches to hardware multithreading. *Fine-grained multithreading* switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle. To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle. One advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. The primary disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Fine-grained multithreading: A version of hardware multithreading that implies switching between threads after every instruction.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on expensive stalls, such as last-level cache misses. This change relieves the need to have thread switching be extremely fast and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall. Coarse-grained multithreading suffers, however, from a major drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the **pipeline** start-up costs of coarse-grained multithreading. Because a processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen. The new thread that begins executing after the stall must fill the pipeline before instructions are able to complete. Due to this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

PIPELINING

Coarse-grained multithreading: A version of hardware multithreading that implies switching between threads only after significant events, such as a last-level cache miss.

Simultaneous multithreading (SMT) is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled **pipelined** processor to exploit thread-level parallelism at the same time it exploits instruction-level parallelism (see COD Chapter 4 (The Processor)). The key insight that motivates SMT is that multiple-issue processors often have more functional unit parallelism available than most single threads can effectively use. Furthermore, with register renaming and dynamic scheduling (see COD Chapter 4 (The Processor)), multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

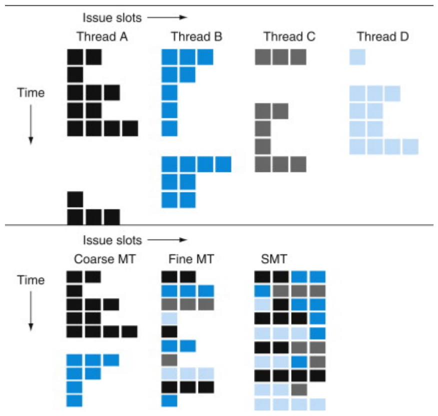
PIPELINING

Simultaneous multithreading (***SMT***): A version of multithreading that lowers the cost of multithreading by utilizing the resources needed for multiple issue, dynamically scheduled microarchitecture.

Since SMT relies on the existing dynamic mechanisms, it does not switch resources every cycle. Instead, SMT is *always* executing instructions from multiple threads, leaving it up to the hardware to associate instruction slots and renamed registers with their proper threads.

Figure 6.4.1: How four threads use the issue slots of a superscalar processor in different approaches (COD Figure 6.5).

The four threads at the top show how each would execute running alone on a standard superscalar processor without multithreading support. The three examples at the bottom show how they would execute running together in three multithreading options. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of gray and color correspond to four different threads in the multithreading processors. The additional pipeline start-up effects for coarse multithreading, which are not illustrated in this figure, would lead to further loss in throughput for coarse multithreading.



The figure above conceptually illustrates the differences in a processor's ability to exploit superscalar resources for the following processor configurations. The top portion shows how four threads would execute independently on a superscalar with no multithreading support. The bottom portion shows how the four threads could be combined to execute on the processor more efficiently using three multithreading options:

- A superscalar with coarse-grained multithreading
- A superscalar with fine-grained multithreading
- A superscalar with simultaneous multithreading

In the superscalar without hardware multithreading support, the use of issue slots is limited by a lack of **instruction-level parallelism**. In addition, a major stall, such as an instruction cache miss, can leave the entire processor idle.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, the pipeline start-up overhead still leads to idle cycles, and limitations to ILP mean all issue slots will not be used. In the fine-grained case, the interleaving of threads mostly eliminates idle clock cycles. Because only a single thread issues instructions in a given clock cycle, however, limitations in instruction-level parallelism still lead to idle slots within some clock cycles.

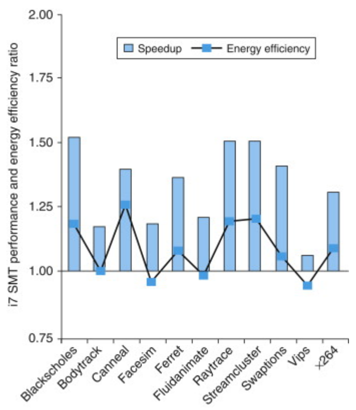
In the SMT case, thread-level parallelism and instruction-level parallelism are both exploited, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors can restrict how many slots are used. Although the figure above greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

The figure below plots the performance and energy benefits of multithreading on a single processor of the Intel Core i7 960, which has hardware support for two threads. The average speed-up is 1.31, which is not bad given the modest extra resources for hardware multithreading. The average improvement in energy efficiency is 1.07, which is excellent. In general, you'd be happy with a performance speed-up being energy neutral.

Figure 6.4.2: The speed-up from using multithreading on one core on an i7 processor (COD Figure 6.6).

Processor averages 1.31 for the PARSEC benchmarks (see COD Section 6.9 (Communicating to the outside world: Cluster networking)) and the energy efficiency improvement is 1.07

These data was collected and analyzed by Esmaeilzadeh et. al. [2011].



Now that we have seen how multiple threads can utilize the resources of a single processor more effectively, we next show how to use them to exploit multiple processors.

Check yourself

1. True or false: Both multithreading and multicore rely on parallelism to get more efficiency from a chip.
2. True or false: *Simultaneous multithreading* (SMT) uses threads to improve resource utilization of a dynamically scheduled, out-of-order processor.

Answer: 1. True. 2. True.

(*1) This section is in original form.

Provide feedback on this section