Hyper-Threading:

Squeezing Bubbles Out of the Processor's Pipeline

Introduction

For decades, the goal of the computer has been to use clever tricks to fool the user into thinking that the processor is running two processes at once. We often are running two or more applications on our computers, believing that the operating system is always running our processes. However, this is not the case! Even in the case of music playing on our computers, each process is given only a small time slice with which it can give instructions to the processor. This means that there is a switching that happens between open processes quickly enough that we don't actually notice the change.

This isn't to say that the processes aren't all loaded into memory at the same time, but rather that only a single process can be running on a processor at a given time. For years, it was through "smoke and mirrors" in which the processor appeared to be able to execute more than one process at a time. There have been many tricks in the past used to make the processor seemingly more efficient or run more jobs in the same amount of time, but it hasnâ€™t been until relatively recently (late 2002) that Intel came out with technology capable of running more than one process at a time on a processor known as simultaneous multithreading (SMT) or hyper-threading. Before going into hyper-threading, we will look at the evolution of the processor as it made its progression towards SMT.

Wikipedia: Hyper-Threading

Pre-emptive and Cooperative Multitasking

In the past, in order to allow for more than one process to use the processor over a given range of time, the system employed "cooperative multitasking". Cooperative multitasking relied on the kind-hearted nature of the processes to give up the processor once their designated time slice was over, however, there was no

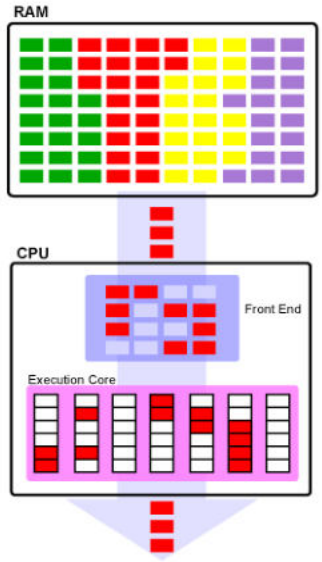
enforcement by the operating system and each program was simply expected to follow this loose rule. But just like in society, a few "bad eggs" can ruin it for everyone and you can imagine how well that system worked... In an ideal situation, cooperative multitasking was very useful and provided meaningful time slices to important processes, but without the explicit cooperation between the OS and the processes, this style of multitasking did not work.

Then along came pre-emptive multitasking! Using this type of multitasking, the operating system has explicit control over the processes running on the processor and when a process's time slice is over, the operating system gives the process the boot just like the bouncer at a nightclub.

Pre-emptive multitasking also uses **memory protection**, a system which prevents other processes that will run on the processor from using memory that has not been specifically allocated for them. This "walls" off the processes from one another and prevents them from overwriting each other's data in RAM.

Single Threaded Processor: Time is a Terrible Thing to Waste

Not too long ago, our computer systems each had a single core and a single thread that could be executed on during any given clock cycle. This means that the CPU was limited to how much data it could process based almost entirely on its processing speed. However, the hardware paired with the operating system made it appear to the user of the system that many processes were all being executed simultaneously, while the processes believed that they were the only program being run and had full execution time for the processor. These assumptions are both false, but to a human being, the few nanoseconds that a process is not executing on the processor is far too small a time chunk for us to recognize. To the process, it is unaware of the other processes on the system do to the memory protection used by the operating system and simply presumes to have the whole system to itself.



In the above image, you will notice that the processes (each a different color in RAM), actually populate all of the shown RAM in this example. While all of them are loaded into memory, they have their allocated memory space and that is all that they are aware of in this environment. In the image, you can see that only one process at a time can execute on this single threaded CPU. Only the red process is executing and the rest must wait their turns before being executed on the CPU.

You should also notice that surrounding the red boxes in the execution core are white boxes. These white boxes represent **pipeline bubbles** which are missed opportunities in which some small amount of work might have been able to be accomplished during that pipeline stage. There also similar white boxes in the CPU's frontend, which represents the CPU's ability to execute four instructions per clock cycle, but as a single threaded processor, it never reaches that limit in this example.

The process happily executes until its time slice is done at which time, the processor must perform a **context switch** in which it saves the context of the process to memory and forces the program to quit executing. The context of a process is a set of metadata regarding this process including how much it has executed, how long it has executed, values in registers, etc. Anything that was known by the process or created during execution is stored until next it can be executed on the processor. By saving the process's context, when the CPU next runs the process, it appears to the process as if it has been running the entire time without interruption.

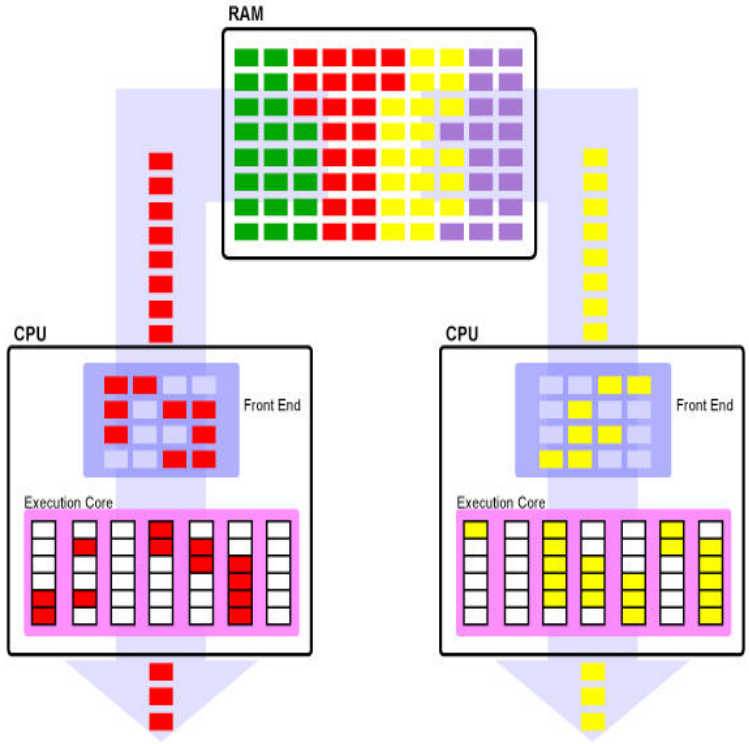
Clearly, depending on the process having its context saved, a context switch can be a costly operation taking many clock cycles before the next process can execute at all. These can also leave large bubbles in the pipeline of the processor, which could have been used for processing data.

ARS Technica Article on Hyper-Threading and CPU Design

Symmetrical Multiprocessing: Two Heads are Better than One

One solution to the problem of expensive context switching is to have more than one core per processor. This allows for two or more threads to be executing, one on each core meaning that there will be approximately half the number of necessary context switches than on a single threaded CPU. This is known as symmetrical multiprocessing (SMP) and has become a very common arrangement in the processors. Most processors that have been made in the past few years for home computers are at least dual-core machines and in fact there are eight-core home PCs, which allow for eight total threads to be executed at a time.

One thing to note is that each program can be made of multiple threads. When a program is broken into multiple threads, if these threads are independent of one another, it is possible to have many threads of the same program executing on the multiple CPUs at the same time. This means that developers that took the time to write multithread code for a single-core single threaded CPU design, will see an increase in performance on SMP CPUs without any additional work.



In the above image, we can see that with two cores, two processes can be running at any given time and since the red and yellow processes are independent of one another, they don't have to wait for one another. These CPUs are still only single threaded CPUs though, so while you are capable of running two processes at the same time, it still has the same amount of bubbles in the pipeline per core. In fact, while you reduce the number of context switches per core by half, the number of bubbles in the pipeline is nearly doubled! This means that in theory we can get twice as much work done in the same amount of time, but the efficiency in the pipeline is not changed and there are still a number of missed opportunities that propagate through the pipeline.

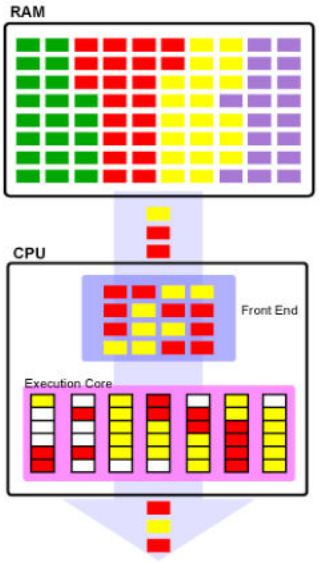
PC Stats Article on Hyper-Threading

Simultaneous Multithreading: Made to Pop the Bubbles

Simultaneous multithreading (SMT) or hyper-threading has been a concept used in Intel processors like the Pentium 4 since 2002. It has helped to significantly reduce the amount of wasted pipeline stages by allowing for more than one thread to be executed per core. At this point in time, per single physical core, two "logical" processors are recognized for a hyper-threaded CPU.

The way this is accomplished is by duplicating the portion of the CPU that is used to store the **architectural state** . In other words, the registers are duplicated in such a way that it allows for both threads to use the same

number of registers as they would have originally had on a single threaded machine, but it still has the same number of execution resources. This means that while one process is stalled by waiting for memory accesses or branches to be resolved, the other process can be using the execution resources not being used that would have remained unused in a single threaded CPU. Remarkably, the cost in processor die size is fairly small for this change, but its improvement on multithreaded code efficiency is astounding.



Again, in the above picture, you can see an example of the hyper-threaded CPU processing data. It is executing data from both the red and yellow processes simultaneously and filling in the pipeline stages as best it can to allow for all of the execution stages to be busy as frequently as they can be. You'll notice that there are far fewer bubbles in the both the front end (instructions for the CPU) and the pipeline itself. While exactly the same amount of work has been done in this example as the single-threaded SMP design, there is much less wasted pipeline stages and around half the resources being used as there is still only one set of execution resources.

The cool part about SMT is that there is no special logic required by the operating system to use SMT natively. As long as the operating system is capable of working with SMP processors, it will recognize CPUs with hyper-threading capabilities as twice the number of physical processors. For example, if you are fortunate enough to own a six-core Intel i7 processor, your computer will read twelve logical processors for use!

One final thing to note about this is that if the programs you are running on your system are not multithreaded programs, you will see absolutely no performance boost over a comparable single-threaded CPU. The good news is that you won't see a performance decrease if the program is not multi-threaded either as the SMT processors are designed to work just as well on single-threaded code, but only one logical processor will be doing the work.

PDF Copy of Intel's White Paper on Hyper-Threading

Conclusion

Hyper-threading technology was and still is a very exciting movement forward in CPU design, as more efficiency is gained through a relatively simple change to the processor. By adding more registers to mirror a single-threaded processor, we are allowing for twice the amount of work to be done per core (in ideal cases).

While this performance boost is fantastic in terms of efficiency in the pipeline stages, some manufacturers have stated that they see SMP technology as being more valuable than SMT technology for a few reasons. One, they claim that while the die size for a dual-core SMP processor will be larger than a single-core SMT processor, the SMT processor will in fact use "46% more power" than by the dual-core processor. They also state that while there will be greater efficiency in the pipeline over the SMP processor, they have found that an SMT processor will have "42% more cache thrashing" over the SMP design. As we have learned in previous architecture and operating system classes, memory thrashing can be very detrimental to the performance of the processor.

ARM's Stance on SMT in 2006

In the end, hyper-threading is a powerful technology that over the years has been tweaked and improved to reduce the severity of these issues. It appears as though we will be seeing a lot more multi-core SMT processors in the future as Intel attempts to squeeze as much performance out of their processors as they possibly can.

Thanks for your kind attention!