ECE459: Programming for Performance Winter 2011

Lecture 6 — January 20, 2011

Patrick Lam *version 1*

Concurrency and Parallelism

(Discussed last time). Concurrency and parallelism both give up the total ordering between instructions in a sequential program, for different purposes.

Concurrency. We'll refer to the use of threads for structuring programs as concurrency. Here, we're not aiming for increased performance. Instead, we're trying to write the program in a natural way. Concurrency makes sense as a model for distributed systems, or systems where multiple components interact, with no ordering between these components, like graphical user interfaces.

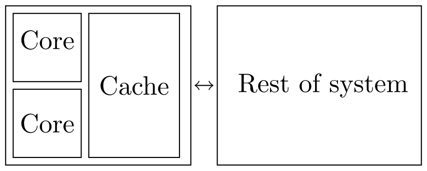
Parallelism. We're studying parallelism in this class, where we try to do multiple things at the same time in an attempt to increase throughput. Concurrent programs may be easier to parallelize.

Processor Design Issues

Recall that we listened to Cliff Click describe characteristics of modern processors in Lecture 2. In this lecture we'll continue our quick review of computer architecture and how it relates to programming for performance. Here's another reference about chip multi-threading; we are going to study some of the techniques in the "Writing Scalable Low-Level Code" section.

`http://queue.acm.org/detail.cfm?id=1095419`

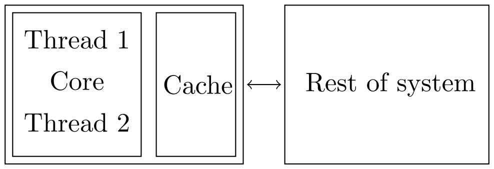
Implementing Hardware Threads. Last time, I talked about hardware threads. There are a number of ways to implement multiple hardware threads; for instance, one can dedicate one core to each thread. The cores might share a cache:



(credit: *Multicore Application Programming*, p. 5)

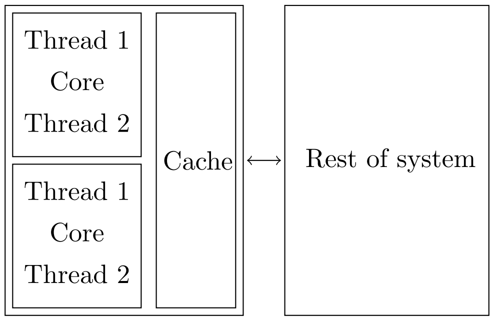
In this case, the cores don't need to change much, but they still need to communicate with each other and the rest of the system.

However, since we're talking processor design, we can also make one core execute two threads. The single core would keep two contexts and could 1) switch every 100 cycles; 2) switch every cycle; 3) fetch one instruction from each thread each cycle; or 4) switch every time the current thread hits a long-latency event (cache miss, etc.)



One would expect that executing two threads on one core might mean that each thread would run more slowly. It depends on the instruction mix. If the threads are trying to access the same resource, then each thread would run more slowly. If they're doing different things, there's potential for speedup.

It's possible to both use multiple cores and put multiple threads onto one core:



Here we have four hardware threads; pairs of threads share hardware resources. One example of a processor which supports chip multi-threading (CMT) is the UltraSPARC T2, which has 8 cores, each of which supports 8 threads. All of the cores share a common level 2 cache.

Using CMT effectively. Typically, a CPU will expose its hardware threads using virtual CPUs, which we've discussed last time. In current hardware designs, each of the hardware threads has the same performance.

However, performance varies depending on context. In the above example, two threads running on the same core will most probably run more slowly than two threads running on separate cores, since they'd contend for the same core's resources. Task switches between cores (or CPUs!) are also slow, as they may involve reloading caches.

Solaris "processor sets" enable that operating system to assign processes to specific virtual CPUs, while Linux's "affinity" keeps a process running on the same virtual CPU. Both of these features reduce the number of task switches, and processor sets can help reduce resource contention, along with Solaris's locality groups[1]

[1]Gove suggests that locality groups help reduce contention for core resources, but they seem to help more with memory.

Pipelining, speculation, superscalar execution. These were the focus of Lecture 2, so here's just a quick reminder. CPUs increase throughput by splitting up instructions and running a number of instructions at a time (*pipelining*). Pipelines break in the presence of branches, so the processor *speculates* on the direction of the branch, and flushes the pipeline when it mispredicts. Modern processors also contain multiple pipelines, which enable *superscalar execution.*

Caches

The next important architecture concept to know about is caches. Memory is slow, CPUs are fast. Caches help reduce the latency of memory accesses. Ineffective cache use slows down your program.

Both memory latency and bandwidth are important. The CPU techniques that we mentioned above help mask memory latency, as does running multiple hardware threads on the same core. Recall that Cliff Click said that modern CPUs are constantly waiting on memory latency. Memory bandwidth is also important in the presence of CMT, which multiplies the amount of bandwidth that a CPU can use.

To simplify processor design and to speed up many types of code, processors organize their caches into *cache lines.* Any load from memory results in the processor loading the entire surrounding area; on my E6300, a cache line is 64 bytes. Processors are sensitive to alignment: they will load starting from an address which is a multiple of 64.

Modern processors have multiple levels of cache: at least 2 and sometimes 3. Here are some typical stats on caches.

	L1	L2	L3	RAM
Size	32-64KB/core	2048–4096MB	6–8MB	big
Latency (cycles)	1–3	20–30	40–60	hundreds

Associativity. Recall that caches can store data in exactly one place in the cache (direct-mapped) or in multiple places (associative), which increases the effectiveness of the cache, at the cost of architecture complexity, up to a fully-associative cache. Higher associativity is particularly critical when multiple threads can potentially overwrite each others' cache lines.

Cache coherence. Maintaining a cache for a single core with a single hardware thread is not easy, but it's not too hard. However, the situation becomes more complicated for multicore and multi-processor systems[2].

The problem is that each core has to keep track of changes to memory made by other cores: if core 0 has location x in its cache, and core 1 writes to x, then core 0 had better flush its old value for x.

The traditional solution is for a core to announce its writes on a common bus. Any other core holding the same location is to flush its copy of the newly-written location. The problem (whose solution is beyond the scope of this class) is that announcing all the writes takes substantial bandwidth.

[2]For instance, there is a 2008 PhD thesis titled *Cache Coherence Techniques for Multicore Processors*, by Michael Marty at Wisconsin.

Virtual Memory

We need to talk about TLB misses. To talk about TLBs, I'll give a quick review on virtual memory. ("All problems in computer science can be solved by another level of indirection." – David Wheeler, first PhD in computer science).

Instead of having programs refer to physical memory addresses, all modern computers give each program a range of virtual memory addresses, and then translate these virtual memory addresses into physical memory addresses (or swap them in from secondary storage). Recall that VM is a handy way to implement memory mapping as well, where you access a file as if it's a set of memory accesses.

Because we've virtualized memory, we can use the same address in multiple applications, which simplifies the task of the compiler. Furthermore, these applications can all run at once, and reserve more memory than is physically present. The operating system can rearrange the memory allocations as needed, too.

Translation look-aside buffers. Going from a virtual address to a physical address can be expensive, so the processor maintains a TLB with recent lookups. (Actually, one for data and one for instructions). TLB misses will slow you down, which is why profilers can count the number and indicate the location of TLB misses.