Lecture 4 notes

Continued review of pipelines.

- We analytically calculate the impact of pipeline stalls on performance. Ideal pipeline speedup is equal to the pipeline depth. For each class of instructions that may cause a stall, we calculate the additional CPI (cycles per instruction). Dividing the pipeline depth with the overall CPI provides us with an idea of the slowdown due to stalls
- We take a closer look at branches: In the 5-stage pipeline, a branch requires 3 stall cycles, two of which are wasted until we find out if the branch is taken or not taken and one is wasted to update the PC. A trick to reduce the branch stall from 3 cycles to 1 is to move the comparison of the register values in the second stage (i.e. during the second cycle of instruction execution, ID) of the pipeline. Here, the clock cycle needs to be long enough to decode the instruction, read register contents and compare the two registers for equality.
- The 1-cycle pipeline stall on branches may still be unacceptable. Can we avoid it?
    - We can predict the outcome of the branch as not taken. If we are right and we keep fetching instructions without stalling, the program execution is correct. If we are wrong, we have to undo (squash) the one instruction we issued during the cycle that would otherwise be a stall.
    - We can predict the outcome of the branch as taken. This does not buy us anything, since we cannot know the target address of the taken branch until the second cycle (ID), therefore we do not eliminate the stall cycle!
    - We can use the stall cycles for executing an instruction, which is not control-dependent on the branch, i.e. neither on the taken, nor on the non-taken path. This technique is called a branch delay slot. Compilers are typically very good at eliminating branch stalls, by utilizing branch delay slots.

Review of caches:

Memories are organized in hierarchies. Using a hierarchy was motivated by the concepts of locality and working sets. Empirically, programs spend most of their execution time in a small subset of their instructions, and programs access both instructions and data in regular patterns. One of this pattern is “spatial”: the loads and stores issued by programs in program order are to adjacent memory locations. This is called “spatial locality”. Another pattern is “temporal”: the programs access a word W and then reaccess (reuse) W many times with short intervals between consecutive accesses. This is called “temporal locality”. A set of addresses that the program reuses frequently during a window of its execution time is called the "working set".

When we design a memory hierarchy, we look into putting some fast storage on chip, i.e. close to the processor execution units. On-chip storage is typically provided by registers (1 cycle latency, extremely expensive), L1 cache (2-3 cycles latency in modern machines, very expensive), L2 cache (ca. 10 cycles latency, expensive). Off-chip storage includes oftentimes an L3 cache, off-chip DRAM, the disk, and archival storage. As we move down the memory hierarchy, the latency increases by orders of magnitude, the bandwidth decreases linearly, but the cost/bit decreases also, therefore capacity increases. In the end, we are looking to have a system where all memory accesses are served in 1 cycle (register or sometimes L1 cache speed), and the memory capacity is as much as the disk's.

We defined the following terms: hit, miss, hit rate, miss rate, miss penalty and average memory access time. Miss penalty involves both the time to access the next layer of the memory hierarchy (further away from the processor), and the time to fetch the data to the layer that missed. The latter depends on the data bandwidth available between the two layers.

When we design a given layer of the memory hierarchy, we need to answer questions: Where is a block of data coming from the next level of the memory hierarch placed? When the processor tries to access a block, how do we find the block in a given layer of the memory hierarchy (or how do we find if the block is missing from the specific layer)? When a block is brought in a layer and there is no free space in the layer for the block, how do we select a block to replace (evict)? Finally, when we write to a block in a given layer of the memory hierarchy, how do we propagate the updated data to the lower layers?

Virtual memory:

Virtual memory and virtual address spaces are conveniences to simplify memory management, so that programmers do not have to write code which uses directly physical addresses and is constantly aware of the memory size. Virtual memory provides many advantages: The programmer sees a consistent view of memory, although the actual memory allocated in the system may be scattered; the programmer does not need to worry about the size of physical memory (especially if the programmer is not performance-aware…); it becomes easier to write multi-threaded programs with threads sharing data in a virtual address space than a physical address space; it simplifies dynamic memory allocation; it protects data within programs, programs from each other and the operating system from programs; it enables easy sharing of data between programs,

We discussed the concepts of page translation, page tables, and TLBs. We focused on the problem of how to achieve a TLB organization which delivers the translation in time for the cache to be accessed in one cycle. We outlined suboptimal solutions that limit the cache size, or increase cache associativity (hence complexity, hit time), or restrict translations using software. We discussed the option of using virtual addresses to access caches. There is always a heated debate whether virtual addressing of caches is better than physical addressing. There are merits in both solutions and the opinions of experts may change over the years, because of technology changes and/or changes in the applications and usage mode of computer systems.