Computer Architecture/Parallel Computing

Tyler Chang
April 14, 2018

Because of significant overlap in material, I have combined my notes for ECE/CS 5504 (Computer Architecture) and CS 5234 (Parallel Computing).

Basic Terminology

- C: capacitive load, V: (operating) voltage, f: (clock) frequency, I: (static) current
- *Dynamic Power* : $P_{dyn} = \frac{1}{2}CV^2f$
- *Static Power* : $P_{stat} = IV$
- *Latency* : Time for a response
- *Bandwidth* : The rate at which data can flow (not necessarilly proportional to latency)
- *Relative Speedup* : $\frac{\text{Execution time of } x}{\text{Execution time of } y}$
- *Wall Clock Time* : Time including all overheads
- *CPU Time* : Number of instructions $\times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{second}}{\text{cycles}}$
- *Amdahl's Law* : Law of diminishing returns
- *Temporal Locality* : Will probably reuse things that were recently used
- *Spatial Locality* : Will probably use things "near" things we are currently using

The Cache

Each cache is divided into sets, ways, and blocks. A *cache block* contains a single block of contiguous memory. Arranging cached memory into blocks is a means for exploiting spatial locality since any memory address is pulled in a block with its surrouding memory. A *set* is an ordered list of blocks. Each block is assigned a set number, and can only be loaded into the cache in its designated location. The number of sets in the cache is referred to as the number of *ways*. An E-way cache with S sets is shown below:

	Way 1	Way 2	...	Way E
Set 1	Block of Set 1	Block of Set 1	...	Block of Set 1
Set 2	Block of Set 2	Block of Set 2	...	Block of Set 2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Set S	Block of Set 3	Block of Set 3	...	Block of Set 3

Therefore, the total size of the above cache is Block Size $\times$ E $\times$ S.

Note that if there is only one set (fully associated cache), then loading blocks into the cache takes longer since each block in the cache must be checked when determining which to vacate, but the cache space can be used most efficiently. However, if there is only one way (directed cache), then loading blocks into the cache can be done quickly, but the cache space is often used inefficiently since 2 blocks of the same set can never be in the cache at the same time, even if both are needed. In most current Intel machines, the L1 cache is 4 ways, and the L2 and L3 caches are both 16 ways.

When data is requested by a process that is not in the cache, it is called a *cache miss*. There are three reasons for a cache miss:

- A compulsory or cold miss occurs the first time a program requests data, since it cannot be in the cache.
- A capacity miss occurs when the cache is overfilled.
- A conflict miss occurs when data from Set k is requested, but all the cache slots associated with Set k are filled.

The average memory access time (AMAT) of the cache is given by:
Hit time + (Miss Rate $\times$ Miss Penalty).

There are many models for maintaining the cache hierarchy, but the most common is the *inclusive* cache structure. By this structure, whenever some data is in L1, it must be in L2 and L3 and the main memory as well. By contrast, the *exclusive* cache structure does not require that L1 be a subset of L2, etc. When data is written to the cache, there are two models for how it can make its way back to main memory. In the first model, *write through*, the data is always written down through all levels of the cache to the main memory. This is expensive but easy to maintain. By contrast, the *write-back* method only writes to the top level of the cache, then when the cache block is removed or when there is I/O down time, this value is carried down to update the main memory.

Virtual vs. Physical Memory

Note that one of the fundamental issues with multitasking is that no process should be allowed to accidentally access another process's data, despite the fact that they may share both virtual and physical resources. This is called the *aliasing issue*, and in the context of the cache, it has two possible solutions:

1. Track the process ID (PID) associated with each memory location.
2. Flush the cache every time there is a context switch.

Obviously, the second solution is significantly more expensive, so the first solution is preferred. To this end, we introduce the concept of *virtual* vs. *physical* address space.

Each block of memory has a physical address corresponding to its location in the DRAM (main memory). However, on a multitasking machine, the physical memory is often fragmented as it is divided between various processes. For the sake of abstraction, it is convenient for each physical process to own continuous memory addresses. To this end, each process accesses its data in virtual address space, where all of its data appears to be in a continuous block of memory addresses. To manage the virtual memory abstraction, a *page table* must sit between the virtual and physical memory maintaining the correspondence between each process's virtual address space and the physical address space. The *translation lookaside buffer* (TLB) catches requests from the virtual address space, looks up the corresponding physical addresses in the page table, and then services those requests from the physical memory.

Always, the CPU must operate in virtual address space to maintain the virtual memory abstraction. Similarly, the main memory must operate in physical address space since this is how the memory is stored. The cache, however, can be either virtually or physically addressed. In the former case, the TLB sits under the cache making cache misses much more expensive. In the latter case, the TLB sits between the CPU and the cache making cache hits slightly more expensive.

Instruction Level Parallelism

Instruction level parallelism (ILP) focuses on improving the performance of a single execution thread through parallelism. The classic solution to this problem has been pipelining, in which the five steps in an instruction are carried out in parallel in a staggered fashion:

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Execute (EX)
4. Memory access (MA)
5. Register write back (WB)

The idea is that while the current instruction is being written back, the next instruction can carry out its memory access, the instruction after that can be executed, the instruction after that can be decoded, and the instruction after that can be fetched. In this way, once the pipeline has been filled, a single instruction can be executed every clock cycle, even though it would take more than one clock cycle to perform all 5 steps.

Unfortunately, there are 3 types of *hazards* that can cause this execution pipeline model to "stall."

- Structural hazards: When there are not enough hardware resources to execute each operation concurrently.

- Control hazards: When the next instruction cannot be fetched until a branch instruction (such as an IF-statement) has been resolved.
- Data hazards: When an instruction depends on previous values.
  - Anti (false) dependency: When the same register is being used to store two independent values. Typically manifests as a write-after-read (WAR) data hazard.
  - Output dependency: When the same register is being used to write two independent values. Typically manifests as a write-after-write (WAW) data hazard.
  - True dependency: When the next instruction actually depends on the current value. Typically manifests as a read-after-write (RAW) data hazard. Of all the data hazards, this is the only type that can actually stall the pipeline.

All of the above hazards can be addressed through (static) compiler optimizations (such as loop unrolling), or through (dynamic) hardware optimizations, such as *out of order execution*.

The cornerstone of out of order processing is *Tomasulo's algorithm*, which incorporates instruction reordering, register renaming, and speculative execution through its in-order issue, out-of-order execution, in-order commit model. The idea is that when an instruction is issued, it is queued up in the *reservation station*, where it sits and waits until all its data dependencies are resolved. As soon as space becomes available in an execution unit (structural hazards are cleared), the first instruction whose true dependencies (data hazards) have been resolved is loaded for execution. To keep track of data dependencies, *register renaming* is used, where a register file tracks which registers depend on which outputs. When an instruction completes execution, it broadcasts on a *common data bus* so that the register file can be appropriately updated.

To address control hazards, branch predictors guess (through extremely trivial algorithms) which path each branch will take and speculatively execute based on these predictions. Of course, if the branch predictor was wrong, execution must be rolled back to where the wrong branch was taken and the entire out of order pipeline must be flushed. To allow for this, every speculative result is stored in a temporary location and the result of each instruction is committed in order, so that if a branch is missed, all future executions can be rolled back.

Single Instruction Multiple Data/Threads Parallelism

We now move away from the single instruction, single data (SISD) model for achieving parallelism, which has been characterized by ILP.

Single instruction multiple data (SIMD) can be used when the same instruction is to be carried out (independently) on many different data. A classic example is a FOR loop with no inter-iteration dependencies. The classic solution to this problem was vector architectures, in which multiple data were stored in enlongated segmented registers, and special vector processing units were then able to carry out the same operation on all the loaded data at once.

More recently, the single instruction multiple thread (SIMT) model has become popular. This is the model used by GPUs. The idea here is that instead of applying the same instruction to a vector of registers, CPUs have become cheap enough that we can now have an entire vector of threads performing this same instruction concurrently. The NVIDIA (CUDA) model for this sort of execution is outlined below:

- A *CUDA core* is a single in-order processor (making them cheap and energy/heat efficient, but slow).
- To make context swapping fast, each CUDA core's register file is large enough to hold data for many independent threads.
- The entire GPU consists of numerous *streaming multiprocessors* (SMs), each of which has many (usually 32 or 64) CUDA cores.
- A *warp* is a group of (usually 32 or 64) threads which executes the same instruction concurrently (on the same SM).
- A *thread block* is a block of threads which are given the same series of instructions to run on the same SM (where they are divided up into warps). Since there is no out-of-order processing, each individual warp stalls frequently, but context swapping is instant, so this is not an issue since another warp can be swapped in while the stall is resolved.
- A *grid* is an array of thread blocks which is distributed over all SMs.
- The on-chip memory for each processor is shared between all threads, making it extremely fast.
- The *global memory* that is shared between SMs must be located off-chip, so it is quite slow.
- Each individual thread can request its own *private memory*, but this is also located off-chip and is consequently very slow.
- The latency to access memory for GPUs is often slow as well, since it is not optimized for single thread speed. However, the *throughput* is extremely high, since many threads can request data at once.

Multiple Instruction Multiple Data Parallelism

Multiple instruction multiple data (MIMD) is in a sense the strongest notion of parallelism. In this model, there are multiple independent threads performing concurrent operations on multiple independent data. Examples include multi-core CPUs, distributed clusters, and warehouse supercomputers. The main concerns for this model are the issues of *coherency* and *consistency*.

Memory Coherency

Coherency refers to the process of migrating or replicating data, such that all the processors see the same data. For example, if P1 writes $X = 0$ to some memory address, then if P2 accesses that memory address, P2 should see $X = 0$. On a single multicore CPU, cache coherency is an issue, since each core has a private cache, but all the cores share the same main memory. If the cache has a write-back policy, then other processes must have some mechanism for knowing when their data is invalid. Generally, this is achieved through *bus snooping*. The idea is that all the caches share a common data bus, and every write is broadcasted along the bus, so that other CPUs know when their cache is "dirty."

There are numerous protocols for how much information about the write to broadcast, and what the protocol should be for resolving the coherency issues. Some common protocols are MSI, MESI, and MOESI.

It should be noted that cache coherency can negatively impact the cache miss rate:

- A *true sharing* miss occurs when one processor writes to a shared data value in a shared block, then another processor attempts to access that data. Of course, since the data has been altered, a cache miss occurs while coherency is being restored.
- A *false sharing* miss occurs when one processor writes to a private data value in a shared block, then another processor attempts to access a different data value in that same block. Of course, the second processor does not need the first processor's data to perform the operation, but because the cache is maintained at the block granularity, a miss still occurs while coherency is falsely restored.

On a final note, for distributed systems, the above problem is exhacerbated since not just cache but also main memory coherency must be maintained. Also, for non-homogeneous systems and distributed data, we must consider the issue of *non-uniform access times* (NUMA). An alternative to bus snooping, is directory based coherency, in which the sharing status of each block is maintained in a shared directory. This approach is generally more expensive for a few processors, but more scalable for large systems and better for NUMA architectures.

Memory Consistency

Memory consistency is another issue for MIMD, particularly for distributed systems. The strongest notion of memory consistency is *sequential consistency*. By this model, even though operations happen concurrently, it must appear to each processor that all the operations happened in some valid sequential order. This can be achieved by synchronizing at every write, but a steep performance price is paid.

A more relaxed model puts the weight on the programmer, to annotate their code by placing locks and synchronizations where necessarry to maintain sequential consistency. Some common low-level tools for this include:

- Open MP for shared memory systems allows users to annotate their code with compiler directives to tell which loops and blocks can be parallelized.
- Posix(P)-threads for shared memory systems allow users more control at the cost of greater complexity.
- MPI is an interface for distributed memory MIMD.
- OpenACC provides a single cohesive interface (based on directives) for shared memory, distributed memory, and GPU computing (including non-NVIDIA).