CMPSCI 630 Systems **Spring 2018**

Lecture 7

Lecturer: Emery Berger *Scribe: Arjun Karuvally*

7.1 Exploiting Parallelism

ALU - fetches and decodes into micro ops.Depending on the instruction the ALU changes state. In earlier days, the processor manual would contain information about how many cycles each instruction took. So, the cost of memory was fixed. At present, the introduction of caches have changed this making the cost of an instruction unpredictable and based on where the memory would be stored. Possible ways to cope with latency: speculation - predict branch taken or not taken based on history and buffer instructions. The program counter goes into speculative mode during the time - All writes and accessses will be in speculative until that is accessed during execution which means that the speculation was correct and the states can stay as it is. If speculation is wrong, rollback has to be done. Other possible way is having more to do-SMT(simultaneous multi threading)(hyperthreading) having multiple ALU, while waiting for a block or branch the processor can pick up other instructions which it has not processed yet. Like having multiple threads. This is not OS level, but hardwarre level. This does not happen during a context switch, instead when there is stalling. Hide latency by scheduling a whole bunch of threads. Fundamental technique used in systems - Hide latency using concurrency - exploiting parallelism across threads.

7.1.1 Intrathread Parallelism(ILP- Instruction Level Parallelism)

Intrathread Parallelism(ILP-Instruction level parallelism) - Two types of dependencies - data dependence and control dependence. Using dependence a graph(DAG-Directed Acyclic Graph) which denotes these dependencies, we can point out what portion can be parallelised. Whenever the processor detects a portion of code can be done in parallel, it executes them in parallel. For this, the processor has a complicated set of mechanisms. Typically multiple ALUs and FPUs exist to exploit this type of parallelism.

7.1.2 Notes on Itanium

Digression: Intel Itanium; Intel had a trend of ever deeper pipelines. They had a high number of stages(28 stages!), so had very complicated architecture. Rather than change IA-32 architecture into IA-64, entirely new architecture was concieved - Itanium(nicknamed Itanic). Based on the architectural concept, VLIW(Very Large Instruction Word), take all instruction and bundle into a very large instruction. Studies into how many instructions executed before branch shows on average 7 instructions. So the concept of VLIW fails as most of the instructions will have a lot of NOOPs and this leads to wastage of memory bandwidth. Itanium resurrected VLIW - PaRISC and created EPIC(Explicitly Parallel Instruction Coding) architecture. There is no fetch complexity, the compilers figure out the parallelism and set the parallel operations and the chip does not have to figure out parallelism. Alias analysis is the issue. In C, two pointers can point to the same location. In principle a pure functional programming language that does not modify state can execute instructions in parallel, but directly modifying state is faster in the processor-So in order to make use of the pure functional programming improvement, there has to be sufficient parallelism that the overhead due to working with non modifiable state can be overcome. Itanium-1 had in order execution what is required

is OOO(Out Of Order Execution), so Itanium-2 was released and the architecture was complicated. In branches, the architecture added predication bits, that means for each(branch taken or not taken), there was a bits that were set to indicate them so, each instruction had an additional if checking. Each bit contains each info on each branch.

The main issue is that the program is dynamic(at any point, the compiler is not able to do static analysis on dynamic programs). Hardware prefetching is very advanced and efficient that compiler cannot beat this.

7.2 Locks and Concurrency

Every single Java object is a Monitor(a Mesa Monitor). Every object can be locked(using synchronise), has condition variables(NOTIFY and WAIT). The locks are recursive locks(a counter). There are cases where there are a bunch of objects waiting for another object. The other object can NOTIFY ALL but only one thread can execute at a time(Thundering herd problem). Mesa style monitors guarantee progress property. The dual of progress is safety - nothing bad ever happens. Typically we require safety and progress. One of the main issue of programming with threads is the safety property. All issues come down to non-determinism. Sometimes non-determinism is not the issue(eg. socket communication). Stop non determinism implies stop races(non determinism based on timing). We can use atomic instructions-an instruction that cannot be broken down into multiple instruction(this is a hardware level guarantee). If there is no happens before relationship then there is a race. One Big Lock is a bad idea because there is no concurrency. Typically we have blocking locks instead of spinning locks. Issue with these locks are that they are unfair(problem of starvation). In order to avoid this, use of exponential block is used - used on ethernet - best should be adaptive. Randomized exponential backoff can be applied to locks which makes the strategy asymptotically optimal. Optional read: MCS a queueing lock that is fair and efficient.