Lecture 7 notes

We ask ourselves the question what is the theoretical limit of ILP that can be exploited in a program. What is the maximum number of instructions per cycle that we can issue and graduate in typical programs, assuming that we have infinite resources? Interestingly enough, this question has been given many different (and conflicting) answers!

We also ask ourselves what mechanisms are there beyond the ILP techniques we have seen so far to increase ILP. Clearly, vector instructions (instructions performing the same operation over multiple data elements of the same type in one cycle), are an option in this direction.

The common belief is that although both compilers and hardware techniques can overcome some of the limitations of ILP, in general, realistic hardware can not achieve ideal ILP. In addition, no hardware or software can go over the theoretical ILP limits.

We discuss the limits of ILP in a theoretical computer with unlimited registers for renaming, perfect branch prediction, perfect return address and jump prediction, and perfect disambiguation of memory addresses, so that we can safely move loads before stores if loads are independent form these stores. We also assumed perfect caches and unlimited instructions issued per cycle. We then proceed to incrementally relax these assumptions and see what is the impact on ILP.

We conclude that with even very aggressive support not available today (e.g. 10x instruction issue width, perfect memory disambiguation, 128 renaming registrers, and the best predictor available to date on real processors, and 256-instruction windows, ILP can not exceed 10-15 in integer programs and 50 in FP programs.

Most architects and engineers foresee that it makes no sense to further attempt to increase ILP. Increasing the instruction issue bandwidth and introducing enough resources to support high ILP complicates dramatically the design of the processor, increase dramatically power and provides only diminishing returns in performance. Memory disambiguation seems to be one of the hardest obstacles.

How we can go beyond the current limits of ILP? We explore two strategies: Thread-Level parallelism and data-level parallelism. We will look for instructions across many programs, many independent instruction streams in the same program, or across instructions performing the same operation over long streams of data of the same type.

In principles, the goal of Thread-Level Parallelism (TLP) is to provide many instruction streams and an associated execution mechanism, so that either each program in isolation is accelerated (via parallelization) or the throughput of many simultaneously executing programs is improved. Many architects believe that TLP

is more cost-effective to exploit than ILP. The idea of TLP and threads is obviously not new. The techniques has been used extensively in the past for overlapping latencies (in software and in a few cases, hardware) and for parallelizing HPC codes.

Multithreading can be exploited on a processor which already exploits ILP, via small improvements. We can have multiple threads sharing the functional units of 1 processor via overlapping the execution of their respective instructions. The processor can support this by duplicating the independent state of each thread, e.g. the register file, the PC, and if threads run on different address spaces, the page table, or other information used for address translation. Memory between threads in the same address space or different address space can be shared via the virtual memory mechanism. The context switch for threads running on a processor can also be assisted by hardware. Current software context switching latencies remain high (in the 1000s of clock cycles), whereas hardware can accomplish a context switch in perhaps tens or hundreds of cycles).

Multithreading is often characterized by the thread switching frequency. In the one extreme, threads switch upon every instruction. This is called fine-grain context switching. Alternatively, threads switch upon stalling for a long-latency event, such as a cache miss. The latter strategy is called coarse-grain context switching.

In fine-grain multithreading machines, thread switch on each instruction, and this enables the interleaving of multiple threads. Threads typically run in a round-robin fashion, skipping any stalled threads. Other policies (perhaps fairer, or more efficient) are possible here. The processor must be able to manage thread states to switch in every clock cycle. The advantage of fine-grain multithreading is that it can handle effectively both short and long stalls, assuming that there are enough threads to "feed" the processor with instructions. The disadvantage of fine-grain multithreading is that it slows down the execution of each thread individually, since any thread, even without stalls, may be delayed from other threads. Fine-grain multithreading is used in a state-of-the-art processor, Sun's Niagara.

With coarse-grain multithreading, the processor switches only on costly stalls, such as L2 cache misses. The processor does not need to have a lightning-fast context-switch mechanism in this case. Furthermore, individual threads are not slowed down while they are executing without stalls. One disadvantage is that when a thread suffers many short stalls, the thread is penalized many times by pipeline restarting/reloading costs. This happens because the processor issues instructions from one thread and when a stall occurs, the pipeline must be emptied or frozen, and the new thread must refill the pipeline. Because of this case, coarse-grain multithreading is better for reducing the penalty of stalls with high cost, where the refill time is relatively small compared to the stall time.

A compromise between fine-grain and coarse-grain multithreading is the following: We can use a multiple-instruction-issue processor, where we can fetch from more than one threads simultaneously. In a theoretical sense, TLP and ILP exploit two different kinds of parallel structure in a program. With ILP only, functional units are

often left idle because of stalls or dependencies in the code. We can look for instructions in other threads to fill out these empty instruction slots and improve the utilization of functional units. This leads us to the idea of SMT, or simultaneous multithreading.

The insight behind SMT is that dynamically scheduled processors already have many HW mechanisms to support multithreading. They have a large set of virtual registers that can be used to hold the register sets of independent threads, they have register renaming mechanism which provides unique register identifiers, so instructions from multiple threads can be mixed in the datapath without confusing sources and destinations across threads, and they have out-of-order completion, which allows the threads to execute out of order, and get better utilization of the HW. Just adding a per thread renaming table and keeping separate PCs should suffice to convert an ILP processor to a TLP processor. Independent commit of instructions can be supported by logically keeping a separate reorder buffer for each thread

What are the design challenges of SMT? Since SMT makes sense only with fine-grained implementation, there can be a significant impact of fine-grained scheduling on single thread performance. Individual thread performance, depends critically on the selection of threads to fetch instructions from and the number of instructions that we fetch from each thread. Different policies are possible here, some may sacrifice throughput, some may sacrifice individual thread performance, hard to find the right balance. If in general, the processor uses a “preferred” thread approach, throughput may suffer. Otherwise, latency of individual threads may suffer. SMT requires a large register file, and needs to support wide instruction issue and multiple instruction completion effectively (in essence, this challenge is also faced by ILP processors). Furthermore, a key design decision is whether to share caches and TLBs between threads or not. Depending on the design of the memory hierarchy (shared/private), conflicts between threads on shared resources can make a huge difference in performance