Deeper Pipelining

Prof. Loh
CS3220 - Processor Design - Spring 2005

April 25, 2005

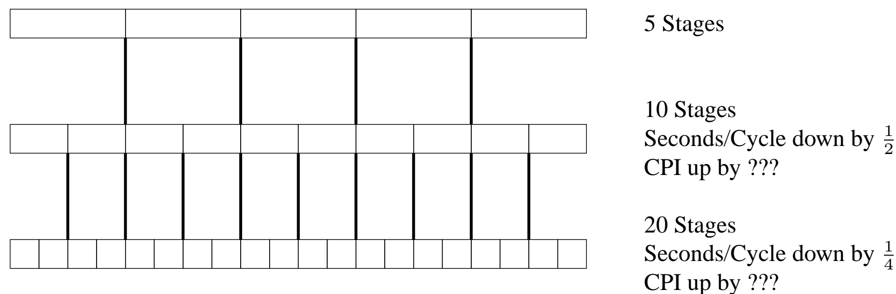
1 Why Make Deeper Pipelines?

The performance of a processor really comes down to how fast it can run a program. Therefore program runtime is the final metric in measuring computer performance. The runtime is determined by several factors:

$$\text{runtime} = \text{insts} \times \frac{\text{cycles}}{\text{inst}} \times \frac{\text{seconds}}{\text{cycle}}$$

For a given program, we typically think of the total number of instructions as a constant. If you have the option to recompile the code, then it is possible to reduce the total number of instructions through compiler optimizations or using more expressive instructions (for example replacing several additions with a single SIMD addition instruction). The remaining two terms are often in conflict such that decreasing one results in an increase of the other.

To improve performance, reducing the cycle time (or increasing clock frequency) helps the last term. For example, consider the classic 5-stage pipeline shown below. By taking each stage and sub-dividing into two smaller stages, we can theoretically increase the clock frequency by a factor of two which would providing a doubling of performance. A second doubling (or a quadrupling over the original 5-stage pipeline) would yield a 20-stage pipeline that should provide a 4x improvement in clock frequency and therefore performance.



2 Latch Overhead Impact on Frequency

Unfortunately there are some factors that prevent the ideal speedup. For example, it is not possible to perfectly divide a single stage of work (delay) into two stages that each have exactly half of the work (delay). One problem is that for every clock cycle, there is some portion of that cycle where the processor does not do any "useful" work. Typically there is some amount of the clock cycle dedicated to *latch overhead* which consists of the gate delay of the latches that separate pipeline stages. For each latch, there is also a *setup* and *hold* time during which the input to the latch cannot change. This represents additional time during each cycle that does not perform useful work.

Consider for example a processor with a 1GHz clock frequency and a 100ps latch overhead. In one cycle (1000ps), there are 900ps worth of "real" work and another 100ps worth of overhead. If we try to increase the clock frequency by doubling the number of pipeline stages, we would divide the 900ps of work into two equal portions of 450ps each.

For each of these new pipeline stages, we still need to pay for the 100ps of latch overhead. This means that the final cycle time will be 450ps+100ps = 550ps or a clock frequency of 1.81 GHz. So even though we doubled the number of pipeline stages, the clock frequency has only improved by 81.8% (as opposed to an ideal 100%).

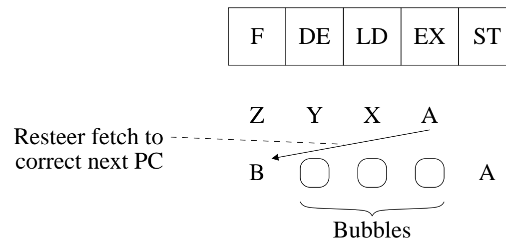
If we attempt to further pipeline the machine (quadruple pipeline), then we would take our 900ps worth of work and divide that over four pipeline stages that each contained 225ps of useful work. When you add in the latch overhead, this results in a cycle time of 325ps or a clock frequency of 3.08GHz. This is almost 25% less than the ideal clock frequency of 4GHz that one would obtain with perfect pipelining with no latch overhead. We can see that just due to the latch overhead, further pipelining a processor provides rapidly diminishing returns.

3 CPI Impact

Increasing the pipeline depth may enable higher clock frequencies, but it also comes at a price of greater CPI penalties.

3.1 Branch Misprediction

Consider the five stage pipeline shown below.

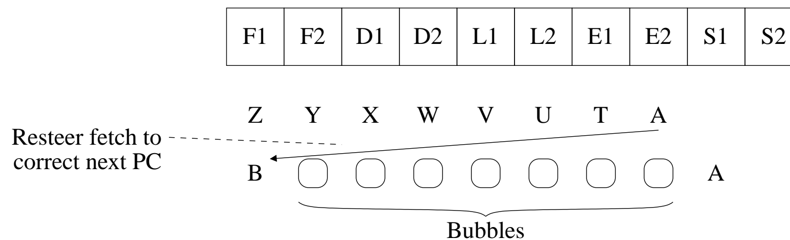


If instruction A is a mispredicted branch, then we will have to insert three bubbles between instruction A and the correct next instruction B. Assuming a branch occurs once every five instructions (i.e. 20% of instructions are branches) and the dynamic branch predictor achieves a prediction rate of 90% (or a misprediction rate of 10%), then the branch misprediction CPI penalty is:

$$\underbrace{0.2}_{\text{Fraction Branches}} \times \underbrace{0.1}_{\text{Misprediction Rate}} \times \underbrace{3 \text{ cycles}}_{\text{Additional Cycles}} = 0.06 \text{ CPI}$$

If all other instructions take 1 CPI, then the total CPI is 1.06.

Now consider a ten stage pipeline, where we will now need to insert seven bubbles on a branch misprediction:



This results in a CPI penalty of:

$$\underbrace{0.2}_{\text{Fraction Branches}} \times \underbrace{0.1}_{\text{Misprediction Rate}} \times \underbrace{7 \text{ cycles}}_{\text{Additional Cycles}} = 0.14 \text{ CPI}$$

For an overall CPI of 1.14. For a twenty stage pipeline (not shown), the number of bubbles increases to 15, which provides a CPI penalty of:

$$\underbrace{0.2}_{\text{Fraction Branches}} \times \underbrace{0.1}_{\text{Misprediction Rate}} \times \underbrace{15\text{ cycles}}_{\text{Additional Cycles}} = 0.30\text{ CPI}$$

As the pipeline depth increases, the frequency increases, but overall performance does not scale perfectly linearly with the frequency increase. The additional CPI penalty of the larger branch misprediction penalty erodes some of the performance gains. In practice, the CPI penalty is even greater because at a faster clock frequency, there is less time to performa branch prediction. This results in a small branch predictor that then in turn results in a higher branch misprediction rate. If for example, this smaller predictor resulted in only 85% prediction accuracy, then the overall CPI penalty would be:

$$\underbrace{0.2}_{\text{Fraction Branches}} \times \underbrace{0.15}_{\text{Misprediction Rate}} \times \underbrace{15\text{ cycles}}_{\text{Additional Cycles}} = 0.45\text{ CPI}$$

3.2 Cache Miss Penalty

The latency of the memory subsystem also affects the CPI. Typically when you increase the clock frequency by increasing the pipeline depth, the latency to memory (absolute latency in terms of seconds, not cycles) does not change. For example, consider a 60ns memory access. With a 1 GHz clock, that translates into a memory latency of 60 cycles. If 20% of instructions are loads and stores, and we have a 2% cache miss rate, then the CPI penalty associated with cache misses is:

$$\underbrace{0.2}_{\text{Fraction Memory Insts}} \times \underbrace{0.02}_{\text{Cache Miss Rate}} \times \underbrace{60\text{ cycles}}_{\text{Additional Cycles}} = 0.24\text{ CPI}$$

Now consider doubling the pipeline depth to 10 stages. As we saw earlier, this increases the clock frequency to 1.8GHz. At 1.8 GHz, there are only 550ps per cycle. Therefore the 60ns memory latency now takes 110 cycles. The cache miss CPI penalty for the 10-stage 1.8 GHz pipeline is thus:

$$\underbrace{0.2}_{\text{Fraction Memory Insts}} \times \underbrace{0.02}_{\text{Cache Miss Rate}} \times \underbrace{110\text{ cycles}}_{\text{Additional Cycles}} = 0.44\text{ CPI}$$

Taking it one step further to our 20-stage pipeline with a 3.08Ghz clock speed, our 60ns memory access now takes 60ns/325ps = 185 cycles. The cache miss CPI penalty is thus:

$$\underbrace{0.2}_{\text{Fraction Memory Insts}} \times \underbrace{0.02}_{\text{Cache Miss Rate}} \times \underbrace{185\text{ cycles}}_{\text{Additional Cycles}} = 0.74\text{ CPI}$$

Similar to branch predictors, as the clock frequency increases and the cycle time decreases, the largest sized cache that can be accessed in a single cycle also decreases. This in turn increases the miss rate of the cache, causing more loads and stores to have to stall for a slow main memory access. For example, let us assume that at 3.08GHz, the cache size must be reduced such that the miss rate increases from 2% up to 5%. The cache miss CPI penalty is now:

$$\underbrace{0.2}_{\text{Fraction Memory Insts}} \times \underbrace{0.05}_{\text{Cache Miss Rate}} \times \underbrace{185\text{ cycles}}_{\text{Additional Cycles}} = 1.85\text{ CPI}$$

This is a substantial penalty. An alternative is to use not reduce the size of the cache but instead increase the latency. If we maintain a 2% miss rate but increase the cache latency to 2 cycles, our cache-related CPI penalty is now:

$$\underbrace{0.2}_{\text{Fraction Memory Insts}} \times \underbrace{1\text{ cycles}}_{\text{Additional Cycles}} + \underbrace{0.2}_{\text{Fraction Memory Insts}} \times \underbrace{0.02}_{\text{Cache Miss Rate}} \times \underbrace{185\text{ cycles}}_{\text{Additional Cycles}} = 0.94\text{ CPI}$$

The equation is broken down into two parts. The first adds a cycle to *all* load and store instructions because all such instructions must access the data cache just to determine whether there was a hit or a miss. The additional cycle is only 1 even though the cache latency is 2 because we assume that in the base case all instructions already take 1 cycle; the 2-cycle cache latency *adds 1 extra cycle*. The second term covers the memory instructions that do not hit in the cache and adds the corresponding memory latency. This demonstrates that sometimes you are better off using a slightly slower cache if that helps you avoid a larger number of even slower memory accesses.

3.3 Putting It Together

Assuming a constant number of instructions to execute, our program runtime will be determined by our clock frequency and our CPI. To summarize our previous examples:

Stages	Frequency	Branch CPI	Cache/Memory CPI	Total CPI
5	+0%	+0.04	+0.24	1.28
10	+81.8%	+0.14	+0.44	1.58
20	+208%	+0.30	+0.74	2.04
20’	+208%	+0.45	+0.94	2.39

The row labeled 20’ corresponds to the slightly more realistic configuration where the branch predictor has been made smaller (resulting in more mispredictions) and the cache latency has been increased to 2 cycles. At the end of the day, we are interested in the overall runtime of the program. By combining CPI and frequency, we can compute the seconds per instruction (which for a constant number of instructions is proportional to the runtime):

$$\frac{\text{runtime}_5}{\text{Insts}} = 1.28 \text{ CPI} \cdot 1\frac{\text{ns}}{\text{cycle}} = 1.280\frac{\text{ns}}{\text{inst}}$$

$$\frac{\text{runtime}_{10}}{\text{Insts}} = 1.58 \text{ CPI} \cdot 0.55\frac{\text{ns}}{\text{cycle}} = 0.869\frac{\text{ns}}{\text{inst}}$$

$$\frac{\text{runtime}_{20}}{\text{Insts}} = 2.04 \text{ CPI} \cdot 0.325\frac{\text{ns}}{\text{cycle}} = 0.663\frac{\text{ns}}{\text{inst}}$$

$$\frac{\text{runtime}_{20'}}{\text{Insts}} = 2.39 \text{ CPI} \cdot 0.325\frac{\text{ns}}{\text{cycle}} = 0.777\frac{\text{ns}}{\text{inst}}$$

Going from 5 stages to 10 stages reduces the runtime per instruction from 1.28 down to 0.87, which is a speedup of 32%. Going from 10 stages to 20 stages reduces the runtime from 0.87 to 0.66 which only provides an additional 23.7% speedup. When you consider the fact that the branch predictor and cache performance will suffer at the higher clock frequencies, the 20-stage pipeline only gives you 0.78 ns/instruction which only represents a 10.6% performance increase over the 10-stage pipeline.

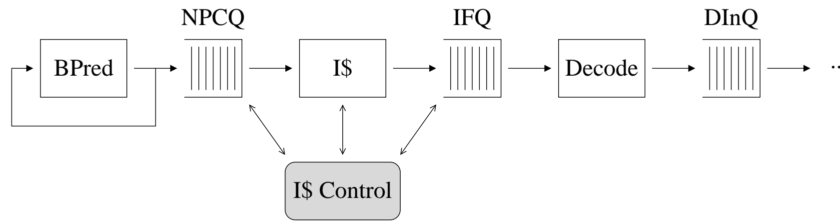
The diminishing returns makes scaling the clock frequency less desirable. The 20-stage pipeline requires twice as many pipestages which represents a large increase in the design complexity. This in turn requries more engineers and designers and more time to design, implement and debug the processor. The larger number of stages also increases the power spent on both pipeline latches and clock distribution (to clock all of those latches). More stages also implies more control logic to coordinate all of the pipeline stages.

4 Control Explosion

Typically the control logic for a pipeline is illustrated as some magical bubble or cloud that generates all of the correct control signals for all of the logic and functional units in the processor. In a deeply pipelined processor, it may no longer even be possible to send a signal from the last pipeline stage to the first in under a cycle. In such a scenario, how can the control logic track everything at once and generate, for example, the correct stall signals for every stage? The answer is that the control logic cannot do it. One solution is to slow the clock speed down so that the control logic can receive all of its inputs and compute all of its outputs in a single cycle. While this would result in a correctly

fucntioning processor, this would reduce the clock frequency to the point where it was pointless in the first place to pipeline the processor so deeply.

Another approach is to sub-divide the processor into smaller control domains. The figure below shows the first few stages of the pipeline where different sections are separated by a buffer or queue. For example, the branch prediction stage uses the current PC to predict the next fetch address, and then uses that to predict the next-next fetch address and so on. These fetch addresses are enqueued in a NPCQ (Next PC Queue). A pipelined instruction cache checks the NPCQ on each cycle to determine if there is a new group of instructions to fetch. If so, the I$ dequeues the fetch address from the NPCQ and starts the corresponding fetch. When the I$ read completes (or if there was a cache miss and then the bytes are returned from L2 or main memory), the resulting instruction bytes are then written into the IFQ (Instruction Fetch Queue). The decode logic checks the IFQ each cycle to see if there's any instructions to decode. If so, the decode logic starts the decode process (which might be pipelined over a few cycles) and then inserts the decoded instrutions into a DInQ (Decoded Instruction Queue).



The advantage of breaking up the pipeline into these smaller pieces is that the processor can have individual and local control logic for each piece. For example, the control logic for the I$ section only needs to check the queue before it (is there work to do?), the queue after it (is there somewhere to place my completed work?), and its own pipestages (which only consisted of the pipelined I$). In this fashion, the control logic is significantly simpler which lends itself to faster clock speeds, easier implementation, and easier debugging.