The modern world has an insatiable appetite for computation, so system architects are always thinking about ways to make programs run faster.

The running time of a program is the product of three terms: The number of instructions in the program, multiplied by the average number of processor cycles required to execute each instruction (CPI), multiplied by the time required for each processor cycle (t_CLK).

To decrease the running time we need to decrease one or more of these terms.

The number of instructions per program is determined by the ISA and by the compiler that produced the sequence of assembly language instructions to be executed.

Both are fair game, but for this discussion, let's work on reducing the other two terms.

As we've seen, pipelining reduces t_CLK by dividing instruction execution into a sequence of steps, each of which can complete its task in a shorter t_CLK.

What about reducing CPI?

In our 5-stage pipelined implementation of the Beta, we designed the hardware to complete the execution of one instruction every clock cycle, so CPI_ideal is 1.

But sometimes the hardware has to introduce "NOP bubbles" into the pipeline to delay execution of a pipeline stage if the required operation couldn't (yet) be completed.

This happens on taken branch instructions, when attempting to immediately use a value loaded from memory by the LD instruction, and when waiting for a cache miss to be satisfied from main memory.

CPI_stall accounts for the cycles lost to the NOPs introduced into the pipeline.

Its value depends on the frequency of taken branches and immediate use of LD results.

Typically it's some fraction of a cycle.

For example, if a 6-instruction loop with a LD takes 8 cycles to complete, CPI_stall for the loop would be 2/6, i.e., 2 extra cycles for every 6 instructions.

Our classic 5-stage pipeline is an effective compromise that allows for a substantial reduction of t_CLK while keeping CPI_stall to a reasonably modest value.

There is room for improvement.

Since each stage is working on one instruction at a time, CPI_ideal is 1.

Slow operations - e.g, completing a multiply in the ALU stage, or accessing a large cache in the IF or MEM stages - force t_CLK to be large to accommodate all the work that has to be done in one cycle.

The order of the instructions in the pipeline is fixed.

If, say, a LD instruction is delayed in the MEM stage because of a cache miss, all the instructions in earlier stages are also delayed even though their execution may not depend on the value produced by the LD.

The order of instructions in the pipeline always reflects the order in which they were fetched by the IF stage.

Let's look into what it would take to relax these constraints and hopefully improve program runtimes.

Increasing the number of pipeline stages should allow us to decrease the clock cycle time.

We'd add stages to break up performance bottlenecks, e.g., adding additional pipeline stages (MEM1 and MEM2) to allow a longer time for memory operations to complete.

This comes at cost to CPI_stall since each additional MEM stage means that more NOP bubbles have to be introduced when there's a LD data hazard.

Deeper pipelines mean that the processor will be executing more instructions in parallel.

Let's interrupt enumerating our performance shopping list to think about limits to pipeline depth.

Each additional pipeline stage includes some additional overhead costs to the time budget.

We have to account for the propagation, setup, and hold times for the pipeline registers.

And we usually have to allow a bit of extra time to account for clock skew, i.e., the difference in arrival time of the clock edge at each register.

And, finally, since we can't always divide the work exactly evenly between the pipeline stages, there will be some wasted time in the stages that have less work.

We'll capture all of these effects as an additional per-stage time overhead of O.

If the original clock period was T, then with N pipeline stages, the clock period will be T/N + O.

At the limit, as N becomes large, the speedup approaches T/O.

In other words, the overhead starts to dominate as the time spent on work in each stage becomes smaller and smaller.

At some point adding additional pipeline stages has almost no impact on the clock period.

As a data point, the Intel Core-2 x86 chips (nicknamed "Nehalem") have a 14-stage execution pipeline.

Okay, back to our performance shopping list… There may be times we can arrange to execute two or more instructions in parallel, assuming that their executions are independent from each other.

This would increase CPI_ideal at the cost of increasing the complexity of each pipeline stage to deal with concurrent execution of multiple instructions.

If there's an instruction stalled in the pipeline by a data hazard, there may be following instructions whose execution could still proceed.

Allowing instructions to pass each other in the pipeline is called out-of-order execution.

We'd have to be careful to ensure that changing the execution order didn't affect the values produced by the program.

More pipeline stages and wider pipeline stages increase the amount of work that has to be discarded on control hazards, potentially increasing CPI_stall.

So it's important to minimize the number of control hazards by predicting the results of a branch (i.e., taken or not taken) so that we increase the chances that the instructions in the pipeline are the ones we'll want to execute.

Our ability to exploit wider pipelines and out-of-order execution depends on finding instructions that can be executed in parallel or in different orders.

Collectively these properties are called "instruction-level parallelism" (ILP).

Here's an example that will let us explore the amount of ILP that might be available.

On the left is an unoptimized loop that computes the product of the first N integers.

On the right, we've rewritten the code, placing instructions that could be executed concurrently on the same line.

First notice the red line following the BF instruction.

Instructions below the line should only be executed if the BF is *not* taken.

That doesn't mean we couldn't start executing them before the results of the branch are known, but if we executed them before the branch, we would have to be prepared to throw away their results if the branch was taken.

The possible execution order is constrained by the read-after-write (RAW) dependencies shown by the red arrows.

We recognize these as the potential data hazards that occur when an operand value for one instruction depends on the result of an earlier instruction.

In our 5-stage pipeline, we were able to resolve many of these hazards by bypassing values from the ALU, MEM, and WB stages back to the RF stage where operand values are determined.

Of course, bypassing will only work when the instruction has been executed so its result is available for bypassing!

So, in this case, the arrows are showing us the constraints on execution order that guarantee bypassing will be possible.

There are other constraints on execution order.

The green arrow identifies a write-after-write (WAW) constraint between two instructions with the same destination register.

In order to ensure the correct value is in R2 at the end of the loop, the LD(r,R2) instruction has to store its result into the register file after the result of the CMPLT instruction is stored into the register file.

Similarly, the blue arrow shows a write-after-read (WAR) constraint that ensures that the correct values are used when accessing a register.

In this case, LD(r,R2) must store into R2 after the Ra operand for the BF has been read from R2.

As it turns out, WAW and WAR constraints can be eliminated if we give each instruction result a unique register name.

This can actually be done relatively easily by the hardware by using a generous supply of temporary registers, but we won't go into the details of renaming here.

The use of temporary registers also makes it easy to discard results of instructions executed before we know the

outcomes of branches.

In this example, we discovered that the potential concurrency was actually pretty good for the instructions following the BF.

To take advantage of this potential concurrency, we'll need to modify the pipeline to execute some number N of instructions in parallel.

If we can sustain that rate of execution, CPI_ideal would then be 1/N since we'd complete the execution of N instructions in each clock cycle as they exited the final pipeline stage.

So what value should we choose for N?

Instructions that are executed by different ALU hardware are easy to execute in parallel, e.g., ADDs and SHIFTs, or integer and floating-point operations.

Of course, if we provided multiple adders, we could execute multiple integer arithmetic instructions concurrently.

Having separate hardware for address arithmetic (called LD/ST units) would support concurrent execution of LD/ST instructions and integer arithmetic instructions.

This set of lecture slides from Duke gives a nice overview of techniques used in each pipeline stage to support concurrent execution.

Basically by increasing the number of functional units in the ALU and the number of memory ports on the register file and main memory, we would have what it takes to support concurrent execution of multiple instructions.

So, what's the right tradeoff between increased circuit costs and increased concurrency?

As a data point, the Intel Nehelam core can complete up to 4 micro-operations per cycle, where each micro-operation corresponds to one of our simple RISC instructions.

Here's a simplified diagram of a modern out-of-order superscalar processor.

Instruction fetch and decode handles, say, 4 instructions at a time.

The ability to sustain this execution rate depends heavily on the ability to predict the outcome of branch instructions, ensuring that the wide pipeline will be mostly filled with instructions we actually want to execute.

Good branch prediction requires the use of the history from previous branches and there's been a lot of cleverness devoted to getting good predictions from the least amount of hardware!

If you're interested in the details, search for "branch predictor" on Wikipedia.

The register renaming happens during instruction decode, after which the instructions are ready to be dispatched to the functional units.

If an instruction needs the result of an earlier instruction as an operand, the dispatcher has identified which functional unit will be producing the result.

The instruction waits in a queue until the indicated functional unit produces the result and when all the operand values are known, the instruction is finally taken from the queue and executed.

Since the instructions are executed by different functional units as soon as their operands are available, the order of execution may not be the same as in the original program.

After execution, the functional units broadcast their results so that waiting instructions know when to proceed.

The results are also collected in a large reorder buffer so that that they can be retired (i.e., write their results in the register file) in the correct order.

Whew!

There's a lot of circuitry involved in keeping the functional units fed with instructions, knowing when instructions have all their operands, and organizing the execution results into the correct order.

So how much speed up should we expect from all this machinery?

The effective CPI is very program-specific, depending as it does on cache hit rates, successful branch prediction, available ILP, and so on.

Given the architecture described here the best speed up we could hope for is a factor of 4.

Googling around, it seems that the reality is an average speed-up of 2, maybe slightly less, over what would be achieved by an in-order, single-issue processor.

What can we expect for future performance improvements in out-of-order, superscalar pipelines?

Increases in pipeline depth can cause CPI_stall and timing overheads to rise.

At the current pipeline depths the increase in CPI_stall is larger than the gains from decreased t_CLK and so further increases in depth are unlikely.

A similar tradeoff exists between using more out-of-order execution to increase ILP and the increase in CPI_stall caused by the impact of mis-predicted branches and the inability to run main memories any faster.

Power consumption increases more quickly than the performance gains from lower t_CLK and additional out-of-order execution logic.

The additional complexity required to enable further improvements in branch prediction and concurrent execution seems very daunting.

All of these factors suggest that is unlikely that we'll see substantial future improvements in the performance of out-of-order superscalar pipelined processors.

So system architects have turned their attention to exploiting data-level parallelism (DLP) and thread-level parallelism (TLP).

These are our next two topics.