CS 6290 High-Performance Computer Architecture Lecture Notes

Jie Wu, Jack

Fall 2022

1 Introduction

1.1 What is Computer Architecture

Computer architecture is a design of computer that is well-suited for its purpose.

1.2 Why Do We Need Computer Architectures

improve performance: speed, battery life, size, weight, energy efficiency

improve abilities: 3D graphics support, debugging support, security

Computer architecture achieves these goals by leveraging improvements in fabrication technology and circuit designs.

1.3 Computer Architecture & Tech Trends

We must anticipate future technology when designing a computer to prevent it from becoming obsolete upon completion, for which we must be aware of technology trends

1.4 Moore's Law

Moore's law states that the number of transistors that can fit into the same chip area is doubled every 18 to 24 months. Computer architects should strive to translate it into

- the processor speed being doubled every 18 to 24 months
- the energy per operation is halved every 18 to 24 months
- memory capacity is doubled every 18 to 24 months

1.5 Memory Wall

While processor speed in terms of instructions per second roughly doubles every 2 years, memory latency improves roughly only 10% every 2 years. It implies that the gap between processor speed and memory speed would widen as time goes. This problem is often called **memory wall**. We have been using caches as a sort of stairs for the memory wall, and only those rare accesses that are missing the cache will end up going to the slow memory.

1.6 Processor Speed, Cost, Power

One of the jobs computer architects need to figure out is that given the technology can either double the speed while keeping the cost and power the same, or halving the cost and power while slightly improving the speed and the infinitely many possible combinations between these two extremes, which one to choose.

1.7 Power Consumption

There are two kinds of power that processors consume

active power: power consumed by actual activities in an electronic circuit

static power: power consumed when the circuit is powered on but idle

1.8 Active Power

It is given by

$$P = \frac{1}{2}CV^2 \cdot f \cdot \alpha$$

where C is the total capacitance of the circuit which is roughly proportional to chip area, V is the power supply voltage of the circuit, f is the clock frequency, and α is the *activity factor* which is roughly the percentage of the circuit that is active in a clock cycle. If we reduce the size of the processor by half but double the number of processors on the chip, then the capacitance and hence the active power remains the same. In reality smaller transistors allow increase in clock frequency and reduction in power supply voltage. Due to the squaring of the power supply voltage, it is the key to active power. For example if the clock frequency is increased by 10% while the voltage is reduced by 10%, then the active power is reduced by 10%.

1.9 Static Power

Transistors are like electronic faucets with valve controlled by power supply voltage. Leakage increases with lower voltage (looser faucet) which is one component of the static power consumption. Therefore while reducing power supply voltage decreases active power it increases static power. Hence the optimal voltage is not too high nor too low.
By the way this faucet leakage is not the only leakage that results in static power consumption. The valve itself has leakage.

1.10 Fabrication Cost and Fabrication Yield

Since the cost of the manufacturing process up to cutting the chips is almost fixed, the size of the chip significantly affects the cost. The more chips in the standard 12-inch wafer the lower the fabrication cost per chip. Hence the cost of a single chip should be linearly proportional to its size. However in reality it is worse than that because of the *fabrication yield*—the percentage of chips on the wafer that are good and thus can be sold—is inversely proportional to chip size, the larger the chip the smaller percentage of the yield. This can be explained by the fact that

- there are almost a certain number of defects on a wafer and the number of defects is more or less a constant (of course with larger chips more than one defects can fall into one single chip, thus the number of bad chips is smaller than the number of defects)

- doubling the chip size results in more than halving the number of chips in a wafer, because wafer is round but chips are square we lose some chips along the edge

As a result any slight change in chip size will lead to significant change in manufacturing cost per chip. When combining it with Moore's law we have two choices of benefit

- smaller chips with the same performance but lower cost (smartphone)
- same chip size with better performance but the same cost (high-performance computing)

2 Metrics and Evaluation

2.1 Performance

There are two aspects of performance

latency: how long does it take from when we start till when we are done

throughput: how many things can we do per second

From the description one may think that throughput = 1/latency, which is not necessarily the case. Take a car assembly line for example, suppose the latency is 4 hours and the assembly line consists of 20 steps each taking the same amount of time i.e. 12 mins, then the throughput is actually 5 cars per hour ($\Rightarrow$ 1/12 mins, do not count the latency of the first car) instead of $1/4 = 0.25$ cars per hour.

2.2 Latency and Throughout Quiz

Another way to break the throughput = 1/latency relationship is to replicate, in which case throughput = number of replications/latency.

2.3 Comparing Performance

When X is N times as fast as Y we say the speedup of X is N. Then it matters whether we use latency or throughput as the measure of speed, because

$$N = \frac{\text{throughput}(X)}{\text{throughput}(Y)} \qquad vs. \qquad N = \frac{\text{latency}(Y)}{\text{latency}(X)}$$

2.4 Speedup

As a sanity check a speedup > 1 means improved performance, whereas a speedup < 1 means worse performance or slowdown. This helps to ascertain whether the denominator and numerator are correct.

2.5 Measuring Performance

We can express performance as 1/execution time. But on what workload the execution time is measured matters, and using actual user workload has the following shortcomings

- we have to run many different programs
- the workload may not be representative of other users
- even if it is representative, how do we get the workload data

Hence we usually use benchmark instead of specifying workload.

2.6 Benchmarks

Benchmarks are programs and input data that users or companies agree on for performance measurement. Usually instead of a single benchmark we have a benchmark suite consisting of multiple programs each of which is representative of some type of applications (e.g. one compression program).

2.7 Types of Benchmarks

real applications: the most representative but the most difficult to set up, should be used for comparing actual machines

kernel: the most time-consuming part of applications typically a loop, usually for prototype

synthetic benchmarks: behave similarly to kernel but simpler to compile, an abstraction of a kernel, usually good for design studies but not good for actual performance reporting

peak performance: usually only good for marketing

2.8 Benchmark Standards

Usually there are standard organizations that take input from manufacturers, user groups, and experts in academia, and produce standard benchmark suites for use in comparing performance along with instructions on how to produce representative measurements. Common benchmark suits include TPC (for database, web servers, and other transaction processing), EEMBC (for phones, video players, and other embedded processing), SPEC (which include GCC for compilation, LBM for fluid dynamics, PERL for string processing, most processor oriented).

2.9 Summarizing Performance

We can use average execution time to summarize the performance on a benchmark suite. But in order to obtain average speedup we should use the geometric mean of execution times rather than the simple average of execution time ratios, the ratio of which gives the geometric mean of speedup because

$$\frac{\sqrt[n]{t_{1,Y} \times t_{2,Y} \times \cdots \times t_{n,Y}}}{\sqrt[n]{t_{1,X} \times t_{2,X} \times \cdots \times t_{n,X}}} = \sqrt[n]{\frac{t_{1,Y}}{t_{1,X}} \times \frac{t_{2,Y}}{t_{2,X}} \times \cdots \times \frac{t_{n,Y}}{t_{n,X}}}$$

As a rule of thumb we should not use simple average of ratios albeit taking ratio of averages is OK.

2.10 Iron Law of Performance

The processor time can be expressed as

$$\begin{aligned}\text{CPU time} &= \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}} \\ &= \#\text{ of instructions per program} \times \#\text{ cycles per instructions} \times \text{clock cycle time}\end{aligned}$$

instructions per program: affected by algorithm, compiler, and instruction set

cycles per instruction (CPI): affected by instruction set and processor design

clock cycle time: affected by processor design, circuit design, and transistor physics

Computer architecture affects

instruction set: a choice between more complex but fewer instructions which take more cycles and simpler but more instructions which take fewer cycles

processor design: a choice between shorter cycles but more cycles per instruction and longer cycles but fewer cycles per instruction

A good architecture design will strike a balance between the two choices.

2.11 Iron Law for Unequal Instruction Times

The processor time can be expressed as (where the subscript i stands for the ith type of instructions)

$$\text{CPU time} = \left(\sum_i \#\text{ of instructions per program}_i \times \#\text{ cycles per instructions}_i \right) \times \text{clock cycle time}$$

2.12 Amdahl's Law

Amdahl's law quantifies the overall speedup when only part of the program experiences speedup

$$\text{speedup} = \frac{1}{(1 - \text{frac}_{\text{ENH}}) + \frac{\text{frac}_{\text{ENH}}}{\text{speedup}_{\text{ENH}}}}$$

where frac_{ENH} is the percentage of original execution time that is affected by enhancement NOT the percentage of instructions that is affected by enhancement.

2.13 Amdahl's Law Implications

It is usually better to have a smaller speedup on most of your execution time than a huge speedup on a very small part of it. Amdahl's law suggests us to make the common case fast (note that improvement in clock frequency improves all instructions thus $\text{frac}_{\text{ENH}} = 100\%$).

2.14 Lhadma's Law

As the reverse of Amdahl's law, Lhadma's law (note the spelling is reversed) reminds us that we should not mess up uncommon case too badly. Because with a huge worsening on even a small portion of the execution time the overall speedup would not be good (in fact if the slowdown is $1/(1 - \text{frac}_{\text{ENH}})$, then even infinite speedup on the common case would not result in any overall speedup).

2.15 Diminishing Returns

Continuing improving on the same subset of the instructions will result in smaller and smaller speedup, because on the one hand the improved part constitutes smaller and smaller portion of the execution time, on the other hand it is usually the case that initial improvement is not hard but further improvement becomes more and more difficult. Therefore computer architects should reconsider what is the dominant part in the execution time after each improvement.

3 Pipelining

3.1 Pipelining

The idea is that although the latency is long for delivering the first unit, there will be no waiting time for the delivery of subsequent units.

3.2 Pipelining in a Processor

The pipeline of a traditional processor pipeline is depicted in Figure 1 below. Suppose the five stages

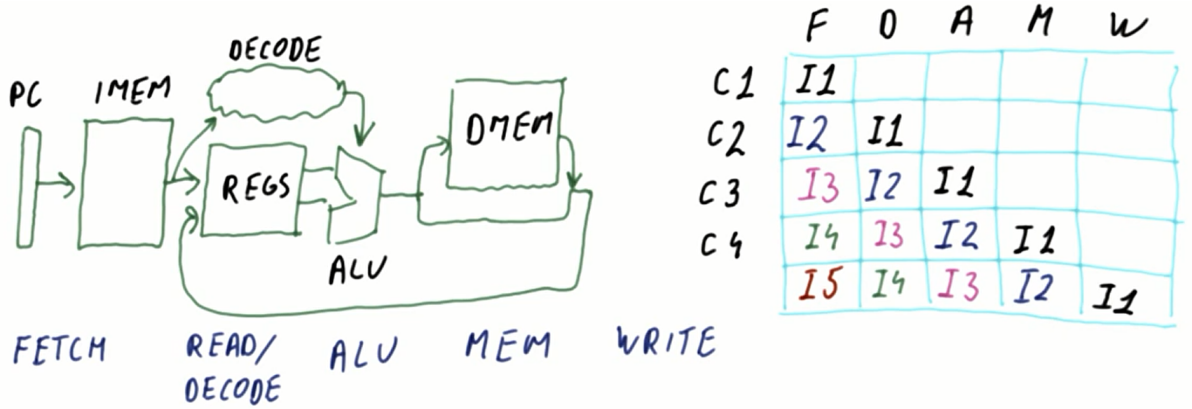

Figure 1: A typical processor pipeline

take 20ns with each taking 4ns, then the latency is 20ns per instruction but the throughput is 1/4ns.

3.3 Instruction Pipelining Quiz

Assuming each stage takes the same amount of time/number of cycles. The total amount of time/number of cycles for n instructions to complete is given by the latency of the initial instruction + $(n-1)\times$ the amount of time/number of cycles to complete one stage/the last stage.

3.4 Pipeline CPI

Pipeline CPI is greater than 1 not because of initial latency since CPI would approach 1 as the number of instructions going through the pipeline approaches infinite, but rather due to pipeline stalls—a job breakdown in an intermediate stage that stalls the inflow of subsequent jobs.

3.5 Pipeline Stalls and Flushes

Instruction pipeline stalls occur for example when we need to increment a counting variable, because its value should first be read from memory and then written to register for the decoder to retrieve and add. Thus there are at least two cycles of pipeline stalls in this case.
Besides stalls instruction pipelines may also need to be flushed from time to time, further increasing the CPI. Because suppose the instruction fetched is a jump instruction, the problem is that we don't know what instruction it is supposed to jump to until the end of the decoding cycle. After we know what instruction we should jump to, we need to flush the instructions that follow the jump instruction to fetch the correct instruction that should have been fetched right after the jump instruction, which creates pipeline bubbles.

3.6 Control Dependencies

For the 5-stage instruction pipeline example above, suppose 20% of the instructions are branches or jumps of which 50% will be taken, and on average we figure out the branches or jumps on the third stage which implies that on average branches and jumps create 2 pipeline bubbles, then the CPI would be $1 + 20\% \times 50\% \times 2 = 1.2$. For modern pipelines that are much deeper, the average number of pipeline bubbles created by branches and jumps would be much larger, which calls for branch prediction techniques to reduce the percentage of branches and jumps taken.

3.7 Data Dependencies

There are three types

read-after-write (RAW): also called flow dependence, the result of the previous instruction needs to be read for executing the subsequent instruction

write-after-write (WAW): also called output dependence, two instructions attempt to write results to the same variable

write-after-read (WAR): also called anti dependence (anti to RAW), the value needs to be read for executing the previous instruction before the subsequent instruction can write result into it

RAW is true dependence whereas WAW and WAR are false dependencies.

3.8 Data Dependencies Quiz

Note that there is no RAW dependence for I1 $\rightarrow$ I4, because the value of $R1$ that is read by I4 is written by I3 not I1.

3.9 Dependencies and Hazards

A dependence is the property of the program alone without regard to what the pipeline looks like. In a particular pipeline some dependencies will cause problems while others will not. Consider again the 5-stage instruction pipeline example above, WAW is never a problem because the dependence will be naturally satisfied by the pipelining order, so is WAR because the decoding stage precedes the writing stage. However RAW will cause problem if there are fewer than three instructions between the producing instruction and the consuming instruction so that the decoding may precede the writing. The situation when a dependence can cause a problem in a pipeline is called a **hazard**. In other words a hazard occurs when a dependence results in incorrect execution of instruction(s). A hazard is the property of both the program and the pipeline.

3.10 Handling of Hazards

Once a hazard is detected we can

control dependencies: flush is the only solution

data dependencies: either stall or forward to replace the wrongly read value, forwarding is preferred over stalling but forwarding is not always possible, because by the time the consuming instruction needs the value to compute, the value may still not be available in the pipeline

3.11 Flushes, Stalls, and Forwarding Quiz

After one stall cycle we can use forwarding to avoid stalling more.

3.12 How Many Stages

On the one hand more stages leads to more hazards which increases CPI. On the other hand more stages means less work per stage which decreases the cycle time required. Since by the iron law the execution time is the product of the number of instructions, CPI, and cycle time, the number of stages should be chosen to balance the increase in CPI and the decrease in cycle time. The dependence of

the execution time on the number of stages exhibits a parabolic shape, with a minimum occurring at around 30–40 stages. Therefore from the performance perspective the optimal number of stages is 30 to 40. However recall that power consumption is proportional to clock frequency thus the dependence of power consumption on the number of stages is almost linear. Balancing good performance and low power consumption the optimal number of stages for modern processors is 10 to 15.

4 Branches

4.1 Branch in a Pipeline

It never pays off not to fetch an instruction after a branch because it would guarantee to generate two pipeline bubbles. However the problem is when we are fetching an instruction we don't know whether it is a branch but we already face the challenge of guessing where it is to take us to.

4.2 Branch Prediction Requirements

Branch prediction means using only the knowledge of where we fetch the current instruction from (the program counter or PC of the current instruction) to guess the PC of the next instruction to fetch. A branch predictor must correctly guess (1) is this a taken branch (2) if it is taken what is the target PC.

4.3 Branch Prediction Accuracy

CPI is given by

$$\text{CPI} = 1 + \frac{\text{misprediction}}{\text{instruction}} \times \frac{\text{penalty (in terms of cycle)}}{\text{misprediction}}$$

where

$$\frac{\text{misprediction}}{\text{instruction}} = \%\text{ instructions that are branches} \times \%\text{ misprediction of taken branches}$$

$$\frac{\text{penalty (in terms of cycle)}}{\text{misprediction}} = \text{the order of the stage the branch is resolved} - 1$$

Misprediction per instruction is determined by predictor accuracy, whereas penalty per instruction is determined by pipeline. The deeper the pipeline the more dependent it is on having a good predictor for good performance. That is the reason why research in branch vectors continues to these days, even though the prediction accuracy is significantly better than 90% nowadays.

4.4 Branch Prediction Benefit Quiz

Assume the 5-stage pipeline as above, consider the following loop

$$\begin{aligned} \text{ADDI} \quad & R1 \leftarrow R1, -1 \\ \text{ADD} \quad & R2 \leftarrow R2, R2 \\ \text{BNE2} \quad & R1, \text{loop} \end{aligned}$$

If we fetch nothing until we are sure of what to fetch, then the two add instructions would cost us two cycles each, one for fetching and the other for discovering that it is not a branch instruction in the decoding stage, and the branch instruction would cost us three cycles because where it is taking us to is resolved in the ALU stage (check $R1 \neq 0$). Overall we would spend 7 cycles per iteration of the loop. With a perfect predictor each of the three instructions would cost us one cycle, thus overall we only need 3 cycles per iteration of the loop. Hence the speedup of having a perfect predictor is $7/3 = 2.33$.

4.5 Performance with Not-taken Prediction

If we always predict a branch is not taken, then a non-branch instruction would cost one cycle while a branch instruction would cost one (if indeed not taken) or three cycles (still using the 5-stage pipeline), compared to not making any prediction where a non-branch instruction would cost two cycles and a branch instruction would cost three cycles. Therefore it is always better off to make some prediction, no matter whether it is accurate or not.

4.6 Multiple Predictions Quiz

Suppose there are two branch instructions one immediately following another, and we are using a non-taken predictor. The second branch instruction would not cause additional penalty because it would be flushed before it causes flush.

4.7 Predict Non-taken

Making non-taken prediction amounts to just incrementing the PC. If there are 20% instructions that are branches and 60% of the branches are taken, then the % misprediction of the non-taken predictor is $20\% \times 60\% = 12\%$.

4.8 Why We Need Better Prediction

We have seen that the deeper the pipeline the more dependent it is on having a good predictor for good performance. It would be more important if the ideal CPI is less than 1. The number of instructions per cycle is typically four in modern processors, for which the speedup of a perfect predictor (CPI $= 0.25$) over the non-taken predictor (CPI $= 0.25 + 0.12 \times 10 = 1.45$) would reach 5.8 if the branch instruction is resolved at the 11th stage!

4.9 Predictor Impact Quiz

The impact of a $p\%$ increase in % misprediction on CPI = % instructions that are branches $\times$ $p\%$ $\times$ penalty per misprediction.

4.10 Why We Need Better Prediction Part 2

If the ideal CPI is 0.25 or equivalently 4 instructions per cycle, then the penalty per misprediction in terms of the number of instructions flushed is $4 \times 10 = 40$ instead of 10 if the branch instruction is resolved at the 11th stage.

4.11 Better Prediction: How?

Better prediction can be achieved by leveraging the history of how the current PC behaved in the past.

4.12 Branch Target Buffer

The simplest predictor that uses history is called the **branch target buffer** (BTB). It uses the current PC of the branch to index into a table, from which we read out our best guess of what the next PC would be. This table is fetched along with the branch instruction. Later on we will discover the correct PC. We will then compare it with the predicted one, and if they disagree we will index the PC of the branch with this correct PC.

Since we want to make a prediction in one cycle (the fetch cycle), BTB needs to have a 1-cycle latency thus should be small. Hence we cannot have a dedicated entry in the BTB for every PC in a program.

4.13 Realistic BTB

We don't need to index every PC in a program. It suffices to index only those that are likely to execute soon. In addition we want the map from PCs to BTB to be simple, because any delay in computing the mapping function means that we need an even smaller BTB so that the prediction can be done in one cycle. The way we do this is to take a few least significant bits of the PC addresses as the indices in our BTB (say 10 out of 64 for a 64-bit machine for a BTB with 1024 entries).

4.14 BTB Quiz

Suppose a BTB has 1024 entries meaning it uses 10 bits for indexing, and all instructions have 4-byte fixed size and are word aligned, meaning all the addresses need to begin at a divisible-by-four address (trailing 00). Assume it is a 32-bit machine i.e. PC addresses are 32 bits long. Which BTB entry is used for a PC at 0x0000AB0C? Note that since all PC addresses are divisible by 4 i.e. with the trailing two bits equal 00. If we use the least significant 10 bits, we can only index 256 out of the 1024 entries of the BTB. To avoid this waste of entries we should instead use the least significant 10 bits beyond the training 00 bits. For this 0x0000AB0C the entry should be 1011000011 or 0x2C3 since B0C = 101100001100.

4.15 Direction Predictor

To make direction prediction we will use a **branch history table** (BHT), indexing PCs into it with a one-bit entry indicating whether it is a not-taken branch (bit 0) in which case we simply increment the PC, or it is a taken branch (bit 1) in which case we write the destination address into the BTB.

4.16 BTB and BHT Quiz

Assume we have a perfect 16-entry BHT and a perfect 4-entry BTB. Since the loop is to be executed 100 times, the BHT is accessed 100 times for each instruction within the loop. However since the loop termination condition needs to be checked, the BHT is accessed 101 times for the loop instruction.

4.17 BTB and BHT Quiz 2

Since all the PC addresses are word aligned meaning all the addresses need to begin at a divisible-by-four address (trailing 00) and the BHT is 4 bits (16 entries), the BHT entries should take the four least significant bits beyond the trailing 00 bits.

4.18 BTB and BHT Quiz 3

Since the BHT is perfect, the BTB is not accessed for all the non-branch instructions. Since the loop is to be executed 100 times, the BTB is accessed 100 times for the branch instruction inside the loop. The BTB is accessed only once for the loop instruction when the loop termination condition is fulfilled.

4.19 BTB and BHT Quiz 4

Again since all the PC addresses are word aligned meaning all the addresses need to begin at a divisible-by-four address (trailing 00) and the BTB is 2 bits (4 entries), the BTB entries should take the two least significant bits beyond the trailing 00 bits.

4.20 BTB and BHT Quiz 5

Suppose the BHT is a 1-bit predictor and initially it predicts not taken i.e. bit 0. It will only make one misprediction for the inner branch instruction and one misprediction for the loop instruction when the loop termination condition is fulfilled.

4.21 Problems with 1-Bit Prediction

As we have seen above 1-bit predictors perform well for programs with a lot of iterations, actually no matter whether it initially predicts taken or not taken. The problem with 1-bit predictors is that each anomaly (minority outcome, taken or not taken) results in two mispredictions, one for the anomalous behavior and one for the ensuing normal behavior. Therefore if the number of taken outcomes and the number of not-taken outcomes are comparable, then 1-bit predictors would not perform well. 1-bit predictors also do not perform well for short loops i.e. just a few iterations, because there would be two mispredictions, one at the start of the loop trained by the preceding loop (taken) and one at the termination of the loop (not taken).

4.22 2-Bit Predictor

The more significant bit of the predictor tells us what the prediction should be, while the less significant bit of the predictor is called the hysteresis or conviction bit, which tells us how confident we should predict according to the prediction bit. Usually 00 represents strong not-taken state, 01 represents weak not-taken state, 10 weak taken state, and 11 strong taken state. With these four states one anomaly would only change our conviction on the prediction bit. Only two consecutive anomalies would change our actual prediction. As a result each anomaly would result in only one misprediction because the ensuing normal behavior would not lead to misprediction.

4.23 2-Bit Predictor Initialization

Suppose we have a dominant outcome (taken or not taken). Contrary to strong state, if we start in a weak state we won't have any misprediction if we are right, and we will have only one misprediction if we are wrong. However starting from a weak state has the risk of always mispredicting in the case of alternating outcomes. Fortunately alternating cases are very rare. Even if we have a dominant outcome the two mispredictions of starting from a strong state does not matter either, because it will almost always be correct after the mispredictions. Therefore in practice the accuracy of 2-bit predictors is affected very little by the choice of starting state, thus we may just start from 00 as it is the easiest to initialize.

4.24 2BP Quiz

Every single predictor has the worst case scenario of always mispredicting. To achieve the worst case scenario we just need to design an alternating sequence of outcomes that makes the predictor go into an infinite flowchart loop. A good predictor would not have this worst case behavior for sequences that are likely to happen.

4.25 1BP to 2BP

We have seen that moving from 1-bit predictors to 2-bit predictors improves performance. It is natural to ask whether it is worth upgrading to 3-bit or even 4-bit predictors. Although a more bit predictor does perform better when anomalous outcomes come in streaks, the cost increases in proportion to the number of bits. Moreover the situation in which anomalous outcomes come in streaks does not happen frequently. Therefore a 3-bit predictor may be worth the cost but definitely not a 4-bit predictor.

4.26 History Based Predictors

Alternating outcomes that cause headache for 1-bit and 2-bit predictors have repeated patterns thus are actually predictable. History based predictors can learn repeated patterns by rewinding the sequence to see the correlation between the ensuing outcome with its immediate history.

4.27 1-Bit History with 2-Bit Counters

For a history based predictor with a 1-bit history and 2-bit counters, its BHT has a 1-bit history in each entry, which depending on its value points to either one of the two 2-bit counters. Both the 2-bit counters and the 1-bit history are updated according to the actual outcome (for the 1-bit history if the previous outcome was not-taken/taken then it will point to the first/second counter). The repeated pattern in an alternating sequence $(NT)^*$ can be captured by updating the first 2-bit counter to strong taken 11 and the second 2-bit counter to strong not-taken 00.

4.28 1-Bit History Quiz

For an alternating sequence $(NNT)^*$ the 1-bit history with 2-bit counters predictor makes a misprediction (of T) every three predictions, thus the accuracy is only 2/3.

4.29 2-Bit History Predictor

For a history based predictor with a 2-bit history and 2-bit counters, its BHT has a 2-bit history in each entry which depending on its value points to either one of the four 2-bit counters. It is able to capture the repeated pattern in the alternating sequence $(NNT)^*$ that the 1-bit history predictor fails to achieve. Because after two not-takens the 2-bit history will point to counter 0 (i.e. 00) which will predict taken, after one not-taken followed by one taken the 2-bit history will point to counter 1 (i.e. 01) which will predict not taken, and after one taken followed by one not-taken the 2-bit history will point to counter 2 (i.e. 10) which will predict not taken, hence after a while there will be one specific counter that predicts one of the outcomes in the repeated pattern and there will be no mispredictions. However notice that we notice that we have used only 3 of the 4 counters. The waste of counter is even more severe when we use the 2-bit history predictor to predict the $(NT)^*$ alternating sequence.
In general an n-bit history predictor will correctly predict all patterns of length $\leq n+1$. However it will cost $n + 2 \times 2^n$ bits per entry, and $2^n - (n+1)$ counters will be wasted. Nevertheless we do need a long history predictor because the length of a repeated pattern is usually determined by the length of a loop.

4.30 History Predictor Quiz

For a nested loop $i = 0$ to 7 inside which $j = 0$ to 7, we need at least four entries because we have 4 branches—2 loop-backs and 2 loop termination conditions. For the outer loop branch we need an 8-bit history predictor because we have a repeated pattern $(NNNNNNNNT)^*$.

4.31 History with Shared Counters

The idea of reducing the waste of counter in n-bit history predictors is to share 2-bit counters between entries. We take some of the lowermost bits of PCs to index a **pattern history table** (PHT), the entries of which keep the history bits alone for a particular branch. Hence if we have a 11-bit history then the PHT will have 11 bits per entry. These 11 bits combined with the 11 bits of the PC in some way, usually XOR, index the entries in the BHT each of which is a single 2-bit counter. When the actual outcome is known we use the same history and PC bit combination to index back to the same counter, incrementing it or decrementing it according to the actual outcome, and then we shift that pattern in the PHT entry so that the history is ready for the next prediction on this branch (same PC bits). The consequence is that the number of counters used is equal to the number of bits required to capture the repeated pattern. Note that it is possible for another PC when combined with its history bits via XOR to point to the same counter. But the possibility of conflict is very low if we have a large array of 2-bit counters e.g. there are 2^{11} counters for a 11-bit PHT and the overall cost is $2^{11} \times 11 + 2^{11} \times 2 = 26$KB. In addition the number of bits of PCs used to index PHT needs not equal to the number of bits per entry in the PHT.

4.32 PShare

What is described above is a PShare predictor where "p" stands for private, meaning that each branch has its private history in the PHT. This type of predictors is good for alternating patterns and loops with only a few iterations, or more general whenever the branch's future behavior is predictable based on its own previous behavior.
Another type of predictors is GShare predictor where "g" stands for global, meaning that it has only one single global history that is used to predict all branches, so that every branch decision, regardless of what the PC it has, shifts the history bit in the PHT entry. This type of predictors is good for correlated branches such as if else.

4.33 PShare vs GShare Quiz

Consider the following C code snippet

```
for (int i = 1000; i!=0; i--)
    if (i%2)
        n+ = i
```

which translates to the following assembly language

```
Loop:
    BEQ   R1, zero, Exit
    AND   R2, R1, 1
    BEQ   R2, zero, Even
    ADD   R3, R3, R1
Even:
    ADD   R1, R1, −1
    B Loop
Exit:
```

For PShare we only need one history because each of the three branches, the two BEQs and the B Loop each require only 1-bit history (the second BEQ is an $(NT)^*$ alternating sequence), whereas for GShare

we need 3-bit history, because considering all the three branches the repeated pattern is (NTTNNT)*. Hence GShare can do the same job as PShare, except that it requires longer history but along with the benefit of handling correlated branches that PShare cannot handle.

4.34 GShare or PShare

Many early processors would choose either GShare or PShare. But people soon realized that we need both, GShare for correlated branches and PShare for self-similar branches.

4.35 Tournament Predictor

Tournament predictor combines PShare predictor and GShare predictor using a meta-predictor which is an array of 2-bit counters. The counter tells the prediction of which predictor, GShare or PShare, to use. The PC is used to index all three predictors. The meta-predictor is trained by how accurate the predictions of the PShare predictor and GShare predictor are. The counter doesn't change if both are correct or both are wrong. If the PShare predictor is correct while the GShare predictor is wrong, then the counter is shifted to choose the PShare predictor more often for the next prediction; if the GShare predictor is correct while the PShare predictor is wrong, then the counter is shifted to choose the GShare predictor more often for the next prediction. Note that each branch has its own meta-predictor entry so the choice of predictor is branch dependent.

4.36 Hierarchical Predictor

While a tournament predictor combines two good predictors, a hierarchical predictor combines a good predictor with an OK predictor (e.g. a single 2-bit predictor). While a tournament predictor updates both predictors based on each outcome, a hierarchical predictor only updates the OK predictor based on each outcome, and updates the good predictor only when the OK predictor does not perform well. The idea is to combine very expensive but few entries for the very few complicated branches and very cheap but many entries for the majority simple branches. With such a better resource allocation a hierarchical predictor usually outperforms a tournament predictor. In fact a hierarchical predictor can combine more than two predictors.

4.37 Hierarchical Predictor Example

Intel's Pentium M processor has three predictors (1) a cheap predictor that contains many 2-bit counters (2) a local history predictor (3) a global history predictor. There is also a tag array associated with both the local history predictor and global history predictor that stores the branch ownership information. The prediction is formed by first checking whether the global history predictor has a prediction for the branch of interest i.e. whether the branch was insert into the tag array of the global history predictor. If the global history predictor doesn't have a prediction for the branch of interest, then we check whether the local history predictor has a prediction for it, again by checking whether the branch was inserted into the tag array of the local history predictor. If the local history predictor doesn't have a prediction for the branch either, we will then use one 2-bit counter in the cheap predictor to make a prediction for it. If the 2-bit counter mispredicts then we will insert the branch into the tag array of the local history predictor. Similarly if the local history predictor mispredicts a branch, that branch will be inserted into the tag array of the global history predictor. To escalate a branch to the local or the global history predictor, some bits of the branch are inserted into the tag array. Note that some upper bits of the branch are used as well to ensure that it is indeed the branch that needs escalation.

4.38 Multi-Predictor Quiz

Suppose we have a program for which a 2-bit predictor works well for 95% of the instructions, a PShare predictor works well for the same 95% instructions plus 2% additional instructions, a GShare predictor works well for the same 95% instructions plus the remaining 3% instructions. We can form a perfect hierarchical predictor that chooses between the 2-bit predictor and a tournament predictor which itself chooses between the PShare predictor and the GShare predictor.

4.39 Return Address Stack

For conditional branches both the direction prediction (taken vs. not taken) and the target prediction (which PC to return) can be well performed (the various predictors for the direction, and BTB for the target), so are jump and function call instructions. However for function return instructions although the direction prediction is easy to make the target prediction is not, because a function can be called at many places in a program and the return target depends on the calling place. To solve this problem we will use a **return address stack** (RAS) predictor dedicated to predicting function returns, which is a small stack in hardware. When we have a function call, the return address will be pushed into the stack. When we reach the return instruction, this address will be popped out of the stack and used as the return target. Note that it is different from the program stack, because it needs to be on chip very close to where the rest of the branch prediction is happening thus needs to be very small.

4.40 RAS Full Quiz

When a RAS is full we have two options, don't push any more to avoid overwriting or wrap around and continue pushing to overwrite. It turns out that the wrap around approach is much better. Because the don't push any more approach only predicts correctly when the large main function finally returns but mispredicts for the many intermediate small functions, whereas the wrap around approach predicts correctly for the many intermediate small functions and only mispredicts for the very few large main functions. Note that both inevitably make mispredictions but that is allowed for predictors.

4.41 But How Do We Know It's A Return

We can use a RAS to predict the target of a function return. But at the outset we must know whether an instruction is a function return, which however is not decoded yet when we fetch it. One way to tackle this problem is to use a predictor and train it on whether the instruction fetched is a return instruction or not. It is a very accurate predictor because if we saw a PC is a return instruction before, it is likely to be still a return instruction when we see it again. Another and more common approach is pre-decoding return instructions, which annotates return instructions with an extra bit in cache (33 bits for a 32-bit machine). In fact pre-decoding can annotate not only return instructions but also branch instructions and even the length of instructions.

5 Predication

5.1 Predication

Besides branch prediction, predication is another way to deal with control hazard. Unlike branch prediction which predicts the direction, predication does the work of both directions and gets rid of one upon knowing the actual outcome, thus the waste is always 50%. Because of this even though the penalty of misprediction is huge in modern processors (because the pipeline is very deep), we should

still use branch prediction for loop instructions, function call and return instructions, and large if-then-else instructions. Nonetheless for small if-then-else instructions predication may be better, since the waste may be smaller than the misprediction penalty (e.g. waste 3 vs. 8% $\times$ 50 penalty).

5.2 If Conversion

A compiler creates the code that will execute along both directions through if conversion illustrated as follows

$$
\begin{array}{l}
\text{if (condition)} \\
\quad x = A[i]; \\
\quad y = y + 1; \\
\text{else} \\
\quad x = A[j]; \\
\quad y = y - 1;
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
x1 = A[i]; \\
x2 = A[j]; \\
y1 = y + 1; \\
y2 = y - 1; \\
x = \text{condition}?x1 : x2; \\
y = \text{condition}?y1 : y2
\end{array}
$$

5.3 Conditional Move

The simplest supporting hardware for the above conditional assignment of x and y is conditional move. In the MIPS instruction set, there is MOVZ which moves the source to the destination if the zero condition is met and MOVN which moves the source to the destination if the non-zero condition is met. The x86 instruction set has a whole set of CMOV instructions such as CMOVZ, CMOVNZ, CMOVGT.

5.4 MovZ MovN Performance

Consider the following branch instruction and its if conversion

$$
\begin{array}{l}
\text{BEQZ} \quad R1, \text{Else} \\
\text{ADDI} \quad R2, R2, 1 \\
\text{B} \quad \text{End} \\
\text{Else:} \\
\text{ADDI} \quad R3, R3, 1 \\
\text{End}
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
\text{ADDI} \quad R4, R2, 1 \\
\text{ADDI} \quad R5, R3, 1 \\
\text{MOVN} \quad R_2, R4, R1 \\
\text{MOVZ} \quad R3, R5, R1
\end{array}
$$

Suppose the branch end is perfectly predicted and the branch start is predicted with an 80% accuracy with a 40-instruction penalty for misprediction. The if conversion takes 4 instructions, whereas the branch prediction takes 2.5 (3 for the first branch and 2 for the second, assuming each is 50% taken) + 20% $\times$ 40 = 10.5 instructions, which is inferior than the if conversion.

5.5 MOVc Summary

To summarize conditional move

- we need compiler support to do if conversion, thus it is not fully backward compatible
- it can remove hard-to-predict branches
- if conversion typically uses more registers than the original code, because it needs to store results from both paths and
- more instructions will be executed, because it executes both paths and conditional move needs to be executed to select the actual result

To avoid the need of more registers and extra conditional move, we can make all instructions conditional. This approach is called **full predication**.

5.6 Full Predication Hardware Support

Full predication adds a condition bit(s) to every instruction which tells us the condition of writing to the destination register. For example Intel Itanium instruction set uses a 1-bit so-called qualifying predicate to implement full predication. An Itanium instruction has 41 bits with the least significant 6 bits specifying where the qualifying predicate is stored.

5.7 Full Predication Example

The full predication of the above branch instruction is as follows (note that the predicates $P1$ and $P2$ are always set together and opposite to each other)

$$\begin{array}{l} \text{BEQZ} \quad R1, \text{Else} \\ \text{ADDI} \quad R2, R2, 1 \\ \text{B} \quad \text{End} \\ \text{Else:} \\ \text{ADDI} \quad R3, R3, 1 \\ \text{End} \end{array} \quad \Longrightarrow \quad \begin{array}{l} \text{MP.EQZ} \quad P1, P2, R1 \\ (P2)\ \text{ADDI} \quad R2, R2, 1 \\ (P1)\ \text{ADDI} \quad R3, R3, 1 \end{array}$$

It takes 4 instructions if we use if conversion and conditional move, but only 3 instructions if we use if conversion and full predication instead. Moreover we can use the same registers because we no longer need $R4$ and $R5$ and there is no additional work of conditional move.

5.8 Full Predication Quiz

Consider the following branch instruction and its full predication

$$\begin{array}{l} \text{BEQZ} \quad R2, \text{Else} \\ \text{ADDI} \quad R1, R1, 1 \\ \text{B} \quad \text{End} \\ \text{Else:} \\ \text{ADDI} \quad R1, R1, -1 \\ \text{End} \end{array} \quad \Longrightarrow \quad \begin{array}{l} \text{MP.EQZ} \quad P1, P2, R2 \\ (P2)\ \text{ADDI} \quad R1, R1, 1 \\ (P1)\ \text{ADDI} \quad R1, R1, -1 \end{array}$$

With a CPI of 0.5 without misprediction, the full predication would take $0.5 \times 3 = 1.5$ cycles. On the other hand the branch prediction would take $0.5 \times 2.5 = 1.25$ cycles without misprediction. Suppose the penalty of misprediction per branch is 10 cycles, then a 2.5% misprediction would add another 0.25 cycle to the total cost of the branch prediction. Therefore the full predication would outperform the branch prediction if the branch prediction accuracy is less than 97.5%.

6 ILP

6.1 ILP All in the Same Cycle

The ideal situation of **instruction level parallelism** (ILP) is that all instructions can go through the pipeline in the same stage. However RAW dependencies prevent us from doing so.

6.2 The Execute Stage

One way to deal with RAW is forwarding, but it requires the write to finish its execution first so that the required value is available in the pipeline. Thus the consuming instruction needs to be stalled and executed in later cycle.

6.3 RAW Dependencies

We would have a CPI of 0 if we have a very large number of instructions with no dependencies. We would have a CPI of 1 if we have any number of instructions that all depend on previous instructions.

6.4 WAW Dependencies

The final value is determined by the write that is stalled the most, which is not necessarily the last write in order.

6.5 Dependency Quiz

Consider again the 5-stage pipeline and forwarding is available to deal with RAW dependencies. Forwarding requires the result to be available in the pipeline, meaning that the producing instruction must either finish the execution stage if it is an arithmetic instruction or the memory stage if it is a load instruction. We also need to be careful about the right order of assigning values to the same variable to observe WAW dependencies.

6.6 Removing False Dependencies

The false dependencies occur when we use the same register for different results. They can be removed by taking care of multiple values in the same register.

6.7 Duplicating Register Values

One way to remove false dependencies is to duplicate registered values i.e. store all possible values of a variable, and any instruction that needs it will search through all these values and find the latest one that is before it.

6.8 Register Renaming

Another way to remove false dependencies is register renaming, which separates architectural registers that are used by programmers and compilers from physical registers which are all the places that can store a value. Register renaming amounts to rewriting the program to use physical registers. It refers to the places that the needed values are actually stored through a **register allocation table** (RAT), which stores the information about which physical register has value for which architectural register.

6.9 RAT Example

In a RAT a process renames an instruction by replacing variable names with the names of the physical registers their values are stored, e.g. ADD $R1$, $R2$, $R3$ is renamed to ADD $P1$, $P2$, $P3$. In this way different values of the same variable bear different names because they are stored in different places, and the processor is able to go to the place that stores the latest version of the variable value.

6.10 Instruction Level Parallelism (ILP)

The ILP can be defined as the instructions per cycle (IPC) when we have a processor that can do (1) an entire instruction (fetch, decode, execute, write result) in one cycle (2) any number of instructions in the same cycle although it still has to obey true dependencies. ILP is what an ideal processor can do subject only to true dependencies. The steps to get ILP are to first rename all registers and then

pretend executing all instructions without true dependencies in one cycle. Note that ILP is a property of a program not a processor, because to compute ILP we always assume an ideal processor.

6.11 ILP Example

To compute ILP we can simply check off the instructions that can be executed when counting cycles. One trick is to work directly on the program without renaming registers while ignoring all false dependencies.

6.12 ILP with Structural & Control Dependencies

As far as ILP is concerned there are no structural dependencies, because structural dependencies occur when we don't have enough hardware to do things in the same cycle, but when computing ILP we always assume ideal hardware.
As for control dependencies we assume for ILP perfect same-cycle branch prediction i.e. branches are predicted in the same cycle in which we fetch them, so we see all correct instructions after a branch. As a result the target instruction can be executed before the branch instruction, and there will be no delay resulted from control dependencies. Therefore to compute ILP we can also ignore control dependencies and treat branches as instructions that produce no result.

6.13 ILP vs IPC

IPC $\leq$ ILP because ILP is for a perfect processor, while IPC is for a real processor. While ILP is not a property of a processor, IPC is.
Note that n-issue means a processor can execute n instructions in the same cycle, while out-of-order means it doesn't need to execute instructions exactly in program order.

6.14 ILP & IPC Quiz

For an in-order processor, once we stop executing an instruction we cannot execute instructions that follow it.

6.15 ILP & IPC Discussion

We define ILP as the IPC on an ideal out-of-order processor that has perfect branch prediction and sufficient resource. The IPC on a narrow-issue in-order processor is usually limited by the narrow issue, whereas the IPC on a wide-issue in-order processor is usually limited by the in-order requirement. To achieve an IPC that is close to ILP we need a wide-issue out-of-order processor, and it should be able to fetch and execute many instructions in a single cycle and be able to eliminate false dependencies.

7 Instruction Scheduling

7.1 Improving IPC

We have seen that ILP can be significantly larger than 1 and usually way over 4. But to achieve ILP we need to handle

control dependencies: via branch prediction

WAR & WAR data dependencies: via register renaming

RAW data dependencies: via out-of-order execution

structural dependencies: via investing in wider-issue processors

In this lesson we will focus on how to do register renaming and out-of-order execution in a way that can be amenable to hardware implementation.

7.2 Tomasulo Algorithm

Tomasulo algorithm is one of the first techniques for out-of-order execution. It determines which instructions have inputs ready for execution. It also includes a form of register renaming. It is very similar to what modern processors use today as far as out-of-order execution is concerned. The differences between Tomasulo algorithm and the techniques used today are

Tomasulo algorithm	today
only for floating point instructions	for all instructions
consider fewer instructions in future time window	>100 future instructions
no support for exception handling	exception handling supported

7.3 The Picture

The overall structure of Tomasulo algorithm consists of

instruction queue: stores floating-point instructions to be executed

reservation station: there is more than one reservation station, instructions fetched from instruction queue will be put into one of them, waiting for their inputs to become ready

register file: where the floating-point registers are, the floating-point values that are ready to be input to the instructions will be entered into the reservation stations from the register file

execution unit: once an instruction is ready to execute, it goes to an execution unit, each execution unit has a separate reservation station

common data bus: once the execution of an instruction has produced a result, the result will be broadcast on the common data bus (CDB) and routed to the register file so that subsequent instructions know where to fetch their inputs, at the same time the result will be broadcast to the reservation stations for the instructions waiting there for input

address generation unit: if the instruction is not an arithmetic but a load/store instruction (load from or store into the register file), then the instruction will go to the address generation unit (not the execution units because address computation is an integer operation) and be inserted into a load buffer (only address) or a store buffer (address and value), when the load operation completes, the value loaded is also broadcast to the CDB and routed to the register file as well as the store buffer
unlike arithmetic instructions which can be executed out of order, loads and stores are done in order, we will see later how modern processors reorder loads and stores, so this is much simpler than modern processors

For later discussion we call

issue: fetching an instruction from the instruction queue and putting it into a reservation station

dispatch: sending an instruction from a reservation station to an execution unit for execution

broadcast: broadcasting a result on the CDB

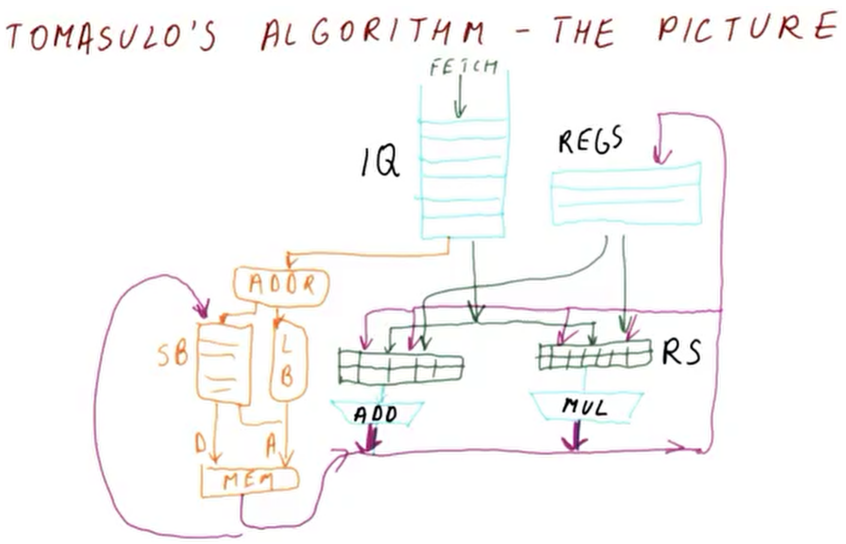

Figure 2: An overview picture of Tomasulo algorithm

7.4 Issue

What happen during issue are

1. take the next instruction in program order (must be in order!) from the instruction queue (LIFO)
2. determine where inputs of these instructions come from (in the register file? yet to be broadcast? which instructions to produce it?)
3. get an available reservation station of the correct type, if all are being used then we simply don't issue the instruction in this cycle (in Tomasulo algorithm one instruction is issued per cycle)
4. put the instruction in the chosen reservation station
5. tag the destination register of the instruction so that once the result is produced it will go there, and future instructions that need the result will know which is the producing instruction

7.5 Issue Example

For each of the floating-point registers there is a **register alias table** (RAT) to keep which instruction is to store result in that register. If the RAT is empty then it means that the result is already available in the register file and we can fetch it from there. After an instruction is fetched from the instruction queue, we look at the RAT to find out whether the required inputs are ready or not. Finally for register renaming we put the name of the reservation station into the RAT to indicate where the result is to be produced. Once filled future reference to the result will come from that reservation station instead of the register file. If the RAT is already filled with some other reservation station, then that reservation station will be overwritten.

7.6 Dispatch

Dispatching needs to latch results that are produced and decide which instructions are ready to execute. In the same cycle we usually have both of them. First we match the tag of the freed reservation station with the operands of the remaining reservation stations and deliver the result to them. This completes result latching. After that we search for the reservation stations that have all the required inputs and send them to the appropriate execution unit for execution. This completes the dispatch.

7.7 More Than One Instruction Ready

One criterion is the oldest first, because all other things being equal it is more likely that more instructions are waiting for the older's result. An alternative is the most dependencies first—check how many other instructions need the result, but obviously it is much more difficult to implement and costs more power. Thus typically we will just follow the oldest first. Of course a third option is just to randomly select among the ready instructions.

7.8 Dispatch Quiz

Tomasulo algorithm is out-of-order execution, thus that there is another instruction that is older cannot be the reason why an instruction is not dispatched.

7.9 Write Result (Broadcast)

First we put the tag of the reservation station and the result on the bus which are then broadcast. After that we write the result to the register file according to which RAT entry matches the tag (overwriting may be necessary). We also need to update the RAT to make sure that after writing the result to the register file the matched entry now points to the register file (make the entry empty). Finally we need to free the reservation station whose result is broadcast.

7.10 More Than One Broadcast

One straightforward solution is to have more than one bus, but that requires doubling or tripling the matching hardware. If there is only one bus, then one common heuristic is to give priority to the slower unit (multiplication/division is slower than addition/subtraction). Because longer execution implies more dependencies would likely accumulate over time.

7.11 Broadcast Stale Result

Stale result may occur when there is another instruction that produces a new value for the same variable and updates the RAT immediately after register renaming. When we broadcast this stale result, the reservation station filling is done as usual. However we don't update the RAT and don't write the stale result to the register file, because this stale result would never be used by future instructions (all the instructions that need this stale result are already in reservation stations which are filled already).

7.12 Review

All of the issue, capture (reservation station filling), dispatch, and write result are happening during every cycle. It is just that they apply to different instructions. Because all of these activities can happen every cycle, it brings in the following questions

can we do same-cycle issue + dispatch if it doesn't need to capture: no, because during the issue cycle the reservation station we just wrote in is not ready for dispatch, nonetheless it is possible to design a processor in a way that allows same-cycle issue and dispatch

can we do same-cycle capture + dispatch: no, because during the capture cycle the reservation station that captured updates its status from operands missing to operands available but not dispatchable, again it is possible to design a processor in a way that allows same-cycle capture and dispatch

can we update the RAT for both issue and write result: yes, because the entry should be that of issue to reflect the latest update which subsequent instructions will use

7.13 Load and Store Instructions

Just like data dependencies that go through registers, there can be dependencies through memory. Because loads and stores are the only instructions that can have dependencies through memory, the data dependencies through memory can be defined as follows (sw = store work, lw = load work)

RAW: sw to some address followed by lw from that address

WAR: lw from some address followed by sw to that address

WAW: two sw to the same address

There are two options to deal with this memory-based data dependencies. One is to load and store in order. This is the option adopted by Tomasulo algorithm. Another is to identify the dependencies and reorder them accordingly, which turns out to be more complicated to do than for eliminating register-based data dependencies and is the reason why Tomasulo chose not to follow. Nevertheless modern processors do identify dependencies and reorder even for loads and stores.

8 Reorder Buffer

8.1 Exceptions in Out-of-Order Execution

The problem of handling exceptions precisely (inputs have already been changed after the exception handler is returned) is the major drawback of Tomasulo algorithm.

8.2 Branch Misprediction

Another problem with Tomasulo algorithm is that it can take a long time to realize that the branch is mispredicted, and by then many instructions that should not have been executed have been completed. A final problem with Tomasulo algorithm is *phantom exceptions*—the exceptions that should not have been raised but happen because of branch misprediction.

8.3 Correct Out-of-Order Execution

We should execute out of order, broadcast out of order, but deposit values to register in order so as to prevent from discovering that one of the earlier instructions shouldn't have been done. But Tomasulo algorithm deposits values to register out of order, which is the reason why it has the above mentioned problems. To solve this problem we need a structure called **reorder buffer** (ROB), which remembers the program order even after issue and retains the results until it is safe to write them to register.

8.4 ROB

We keep at least three fields for ROB—value field, done flag, and destination register. To keep instruction results in program order we need two pointers

issue pointer: tell which entry the next issued instruction to go to

commit pointer: tell which entry to write to register after the last write is completed

With ROB Tomasulo algorithm is modified as follows

issue: in addition to getting a free reservation station we also need a ROB entry that the issue pointer points to, instead of making the RAT point to the reservation station we should make it point to the ROB entry

dispatch: we still need to check whether an instruction is ready for dispatch and if yes send it to an appropriate execution unit, but now we can free the reservation station immediately after sending it to an execution unit instead of waiting for the result to be produced and broadcast as in Tomasulo algorithm, because previously it is the reservation station that serves the name of the result, but now the result is named after the ROB entry

broadcast: capture is exactly as before except that we use the ROB entry instead of the reservation station as the tag of the result, but now instead of being broadcast to the register file the result is broadcast to the ROB first, as a result we don't need to update the RAT to make it point to the register file at the moment

commit: during each cycle we check whether the instruction the commit pointer points to is done or not, if yes we then write the result to the destination register, in addition we need to update the RAT to make it point to the register file after the commit, because we will free the ROB entry for future dispatched instruction

8.5 Free Reservation Station Quiz

Given the same number of reservation stations, that instructions cannot be issued because there is no available reservation station is less likely to happen in a ROB-based processor which frees reservation stations sooner.

8.6 Hardware Organization with ROB

The ROB has a head and a tail pointer. The instructions that are currently in execution are between these two pointers. The head pointer points to where to put the next issued instruction, while the tail pointer points to the next instruction to check for commit. The RAT entries that don't point to the register file now point to the ROB entries between the head and tail pointers instead of the reservation stations.

8.7 Branch Misprediction Recovery

We do branch misprediction recovery before we restart from the correct PC pointed to by the branch instruction. Branch misprediction recovery consists of

1. commit the branch normally and annotate the ROB entry corresponding to the mispredicted branch that there is a misprediction, thankfully since none of the instructions after the branch have updated the registers, the registers contain the correct value at the point of the branch
2. reverse the previous issuing by freeing the ROB entries after the one corresponding to the mispredicted branch, and making both the issue pointer and commit pointer point to the ROB entry next to the one corresponding to the mispredicted branch
3. empty the RAT so that each entry points to the register file
4. free the reservation stations and stop the execution units from broadcasting

8.8 ROB and Exceptions

There are two problems arisen from exceptions. One is that division can take a long time, and divided by zero is realized after several ensuing instructions have been executed. To deal with it we treat the exception just like any other result. That is we mark the corresponding ROB entry as an exception. When the commit point reaches it, we flush everything including the division and jump to the exception handler. Similarly for a load that would encounter page fault, when the commit reaches the ROB entry of the load, we flush everything including the load and jump to the exception handler.
The second problem associated with exceptions is phantom exceptions—exceptions that should have not been raised otherwise but occur due to branch misprediction. ROB handles this by marking the corresponding entry as an exception, and after the misprediction and the correct PC have been figured out cancels the instruction that triggers the exception.
In summary the idea with exception handling is simply to treat an exception just as a result and delay the handling of the exception until the instruction that triggers the exception commits. At that point we know exactly where the resume point is for the exception handler.

8.9 Outside View of Executed

The programmer never sees the execution of any wrong instructions. Thus commit is kind of the official execution, whereas the actual execution before we broadcast the result is kind of the internal state of the processor.

8.10 Exceptions with ROB Quiz

When we have an exception, the correct state of the processor before jumping to the exception handler should be that, all the instructions preceding the instruction that triggers the exception should be committed (thus we need to wait for some of them to commit before passing the control to the exception handler), while all the instructions following the instruction that triggers the exception should be flushed and become unexecuted.

8.11 RAT Updates on Commit

The RAT entry should point to the ROB entry that reflects the latest update for an variable. Only after the instruction that produces the latest update is committed will it be updated (vs. Tomasulo algorithm). By contrast the register file is updated each time an instruction is committed no matter whether that reflects the latest update among the uncommitted instructions or not, so that it is up to date at any commit point and ready for jumping to the exception handler. Nevertheless after updating the register file we will check whether that reflects the latest update, and if yes we will also update the RAT so that it now points to the register file and the ROB entry it previously pointed to will be freed. As a result if we no longer have any instruction in the ROB, then the entire RAT should point to the register file.

8.12 ROB Quiz 3

Because we assume that capture and dispatch can be done in the same cycle, we should be careful about whether the result broadcast during the cycle can be captured and enable the dispatch of some other instruction(s).

8.13 ROB Quiz 5

To find out what would happen in a particular cycle, it is advised to check all instructions in the order of issue, dispatch, broadcast, and commit.

8.14 ROB Quiz 6

We don't update the RAT when broadcasting the result of an instruction as in Tomasulo algorithm. We update it only when issuing an instruction or committing an instruction.

8.15 ROB Timing Example

An instruction can be committed if (1) it has finished broadcasting its result (2) all previous instructions have been committed.
Besides committing instructions in order we also have to issue instructions in order.

8.16 ROB Timing Quiz 3

Remember to check whether we can commit multiple instructions in one cycle or not.

8.17 Unified Reservation Stations

Unified means all reservation stations are identical and can dispatch instructions to either add/subtract execution unit or multiply/divide execution unit. The benefit of having unified reservation stations is that issuing instructions would no longer be delayed by lack of the reservation stations for a particular type of execution units. The drawback is that the logic for dispatching instructions becomes more complicated, because we need not only to decide which type of execution units to dispatch but also to dispatch one instruction to each type of execution units in a single cycle.

8.18 Superscalar

A real superscalar processor needs to

- fetch > 1 instruction per cycle
- decode > 1 instruction per cycle, which amounts to having more than one decoder
- issue > 1 instruction per cycle, which still need to be in order
- dispatch > 1 instruction per cycle, which can be enhanced by having more add/subtract execution unit than multiply/divide execution unit because we usually have more adds than multiplies
- broadcast > 1 instruction per cycle, which requires reservation stations to be able to check their tags of results against multiple buses
- commit > 1 instruction per cycle, which still need to be in order

The performance of a superscalar processor is limited by the bottleneck along this pipeline—the smallest number of instructions per cycle that can be processed in a particular stage.

8.19 Terminology Confusion

Sometimes issue is called allocate or even dispatch instead, dispatch is called execute or even issue instead, and commit is called complete, retire, or graduate instead.

8.20 Out of Order

Actually only dispatch (following data dependencies instead of program order), execution, and broadcast are done out of order in an out-of-order processor. Fetch, decode, issue, and commit are still done in order.

8.21 In Order vs. Out of Order Quiz

Since commit is done in order, so is the release of ROB entry.

9 Memory Ordering

9.1 Memory Access Ordering

We have

- eliminated control dependencies via branch prediction
- eliminated false data dependencies on registers via register renaming
- obey true data dependencies on registers via Tomasulo-like scheduling

If a store is followed by a load, there could also be a dependence between the memory written by the store and the value read by the load if the two addresses are the same. This is a memory dependence.

9.2 When Does Memory Write Happen

Memory write or the store happens at commit, because any instruction that has not yet been committed is subject to cancellation. However it does not mean that the load also has to wait until commit. We can use a structure called **load-store queue** (LSQ) to get the required data as early as possible.

9.3 Load-Store Queue

LSQ is a structure like ROB but only for load and store instructions. It contains four fields: (1) a bit that tells whether an instruction is a load or a store (2) the address this instruction is accessing (3) the value the instruction should store or load (4) a bit that flags whether the instruction is completed or not. Instructions are placed in program order. But we check for every load whether the address it computes matches any store in LSQ. If there is not a match then we go to memory; if there is a match then we do what is called the *store-to-load forwarding*, where we take the value from the store without going to memory.
Of course store-to-load forwarding assumes that the store has been completed. It is possible that the store doesn't have the address when the load computes its address. One way to handle this is to make loads and stores executed in order. A better option is to wait for all previous stores to complete. A third option is to just let the load go anyway by ignoring the stores that are still being executed even though there might be a match. But if there is indeed a match (checked by the store that completes later), then we have loaded the wrong value and we need to reload and redo all the subsequent instructions. Most modern processors choose the go-anyway option, because it produces the best performance and there are entire schemes that try to predict wrong loads and wait for those loads that often make mistake.

9.4 Out-of-Order Load/Store Execution

If we allow loads and stores to be executed out of order, it is possible that the subsequent load finishes before the previous store writes to the same address.

9.5 In-Order Load/Store Execution

One solution to this problem is to execute other instructions out of order but loads and stores in order.

9.6 Store-to-Load Forwarding

For loads we need to figure out which earlier store to get value from, because there could be multiple stores to the same address the load is going to. For stores we need to figure out which later load to give value to so as to wake it up, because there could be multiple loads the store needs to give value to. We figure both out from LSQ.

9.7 LSQ

For store-to-load forwarding we search for every load the most recent store that precedes it and has the matched address. A load commits by depositing the value to the register file. A store commits by putting the value to the cache or memory.

9.8 LSQ, ROB, and RS

To issue a non-load/store instruction we need (1) a ROB entry (2) a reservation station, while to issue a load/store instruction we need (1) a ROB entry (2) a LSQ entry that serves as a reservation station for loads and stores.
There are two steps of the execution of a load/store instruction: (1) compute address (2) produce value. For stores these two steps can happen in any order. Broadcast only applies to loads while writing to memory only applies to stores. After committing a load/store we need to free both the ROB entry and the LSQ entry.

10 Compiler ILP

10.1 Can Compilers Help Improve IPC

The ILP of a program is limited by the dependence chains in it. In addition the hardware has a limited window into the program, so that a real processor may not be able to see independent instructions because they are so far apart beyond the available ROB entries. The compiler can help us put these independent instructions closer to each other so that the IPC achieved by the processor is closer to the available ILP.

10.2 Tree Height Reduction

One technique to improve the ILP of a program is *tree height reduction*. For example suppose we want to calculate $R8 = R2 + R3 + R4 + R5$. We have two possible ways to do the summation.

$$\begin{array}{lll} \text{ADD} & R8, R2, R3 & \\ \text{ADD} & R8, R8, R4 & \xrightarrow{\text{tree height reduction}} \\ \text{ADD} & R8, R8, R5 & \end{array} \begin{array}{lll} \text{ADD} & R8, R2, R3 \\ \text{ADD} & R7, R4, R5 \\ \text{ADD} & R8, R8, R7 \end{array}$$

On the left we have a dependence chain of length three, while on the right the dependence chain consists of only two steps if we are able to execute the first two independent instructions in parallel.
However tree height reduction cannot always be done, because it relies upon the associativity of the operation and not all operations are associative.

10.3 Tree Height Reduction Quiz

$$\begin{aligned} \text{before tree height reduction:} \quad & R10 = R1 + R2 - R3 + R4 - R5 + R6 - R7 \\ \text{after tree height reduction:} \quad & R10 = \underbrace{(R1 + R2)}_{R10} + \underbrace{(R4 + R6)}_{R11} - \underbrace{(R3 + R5 + R7)}_{R12} \end{aligned}$$

While the sequential execution takes 6 cycles, the tree height reduction approach takes only 3 cycles.

10.4 Make Independent Instructions Easier to Find

A real processor can only see a limited number of instructions ahead. We will look at three techniques to help a real processor find independent instructions

- instruction scheduling for simple branch-free sequences of instructions
- loop unrolling followed by instruction scheduling
- trace scheduling

10.5 Instruction Scheduling

The idea of instruction scheduling is to find instructions that can be done in place of stalls to reduce the number of stalls required for execution (further reduction is possible if the instruction moved up itself requires stall). But remember to modify the instructions affected by the move.

10.6 Instruction Scheduling Quiz

While moving an instruction up to a stall cycle required by another instruction, we may need to rename the destination register if there is name conflict.

10.7 Scheduling and If Conversion

With if conversion both paths are executed albeit predicated differently, we can move instructions of one path to stall cycles of another, we can also move instructions of both paths to stall cycles above the if branch. Overall with if conversion we get a lot more opportunities for filling stall cycles. Therefore if conversion not only helps avoid branch prediction but also helps compilers in instruction scheduling.

10.8 If Convert A Loop

A loop is not suitable for if conversion because we need a new predicate for each iteration.

10.9 Loop Unrolling

We unroll a loop by making each iteration of the new loop do more than one iteration of the old loop. Unrolling once amounts to the following

$$\begin{array}{l}\text{for } (i = 1000; i! = 0; i = i - 1)\\ \quad a[i] = a[i] + s;\end{array} \xrightarrow{\text{unroll once}} \begin{array}{l}\text{for } (i = 1000; i! = 0; i = i - 2)\\ \quad a[i] = a[i] + s;\\ \quad a[i-1] = a[i-1] + s;\end{array}$$

Translating to assembly it is

$$\begin{array}{lll}\text{Loop:} & \text{LW} & R2, 0(R1)\\ & \text{ADD} & R2, R2, R3\\ & \text{SW} & R2, 0(R1)\\ & \text{ADDI} & R1, R1, -4\\ & \text{BNE} & R1, R5, \text{Loop}\end{array} \xrightarrow{\text{unroll once}} \begin{array}{lll}\text{Loop:} & \text{LW} & R2, 0(R1)\\ & \text{ADD} & R2, R2, R3\\ & \text{SW} & R2, 0(R1)\\ & \text{LW} & R2, -4(R1)\\ & \text{ADD} & R2, R2, R3\\ & \text{SW} & R2, -4(R1)\\ & \text{ADDI} & R1, R1, -8\\ & \text{BNE} & R1, R5, \text{Loop}\end{array}$$

Note that unrolling twice amounts to doing three iterations at a time, unrolling three times amounts to doing four iterations at a time and so on.

10.10 Loop Unrolling Benefits ILP

The first benefit loop unrolling provides is that it reduces the looping overhead and thus the number of instructions to execute. In the previous example, without unrolling the number of instructions for 1000 iterations is $5 \times 1000 = 5000$. With unrolling once the number of instructions for 1000 iterations is reduced to $8 \times 500 = 4000$. The decrease arises from the reduction of the looping overhead ADDI and BNE.

10.11 Loop Unrolling Benefits CPI

Another benefit of loop unrolling is that it also reduces CPI in addition to the number of instructions. This is because loop unrolling provides more instructions that can be reordered to achieve more parallelism through instruction scheduling. Hence the more we unroll the more parallelism we can achieve. Applying instruction scheduling to the previous example we get

$$\begin{array}{lll}\text{Loop:} & \text{LW} & R2, 0(R1)\\ & \text{ADDI} & R1, R1, -4\\ & \text{ADD} & R2, R2, R3\\ & \text{SW} & R2, +4(R1)\\ & \text{BNE} & R1, R5, \text{Loop}\end{array} \xrightarrow{\text{unroll once}} \begin{array}{lll}\text{Loop:} & \text{LW} & R2, 0(R1)\\ & \text{LW} & R6, -4(R1)\\ & \text{ADD} & R2, R2, R3\\ & \text{ADD} & R6, R6, R3\\ & \text{ADDI} & R1, R1, -8\\ & \text{SW} & R2, +8(R1)\\ & \text{SW} & R6, +4(R1)\\ & \text{BNE} & R1, R5, \text{Loop}\end{array}$$

For a 4-issue in-order processor with perfect branch prediction (so that the BNE of current iteration and the LW of next iteration can be executed together in the same cycle) it takes 2 cycles to complete 5 instructions (ADD followed by SW & BNE & LW & ADDI) without unrolling (CPI = 0.4), but 3 cycles to complete 8 instructions (ADD & ADD & ADDI followed by SW & SW & BNE & LW followed by LW) with unrolling (CPI = 0.375).

10.12 Unrolling Downside

The downside of unrolling include

- the code size grows quickly with the number of unrollings
- it cannot be applied if we don't know the number of iterations to take e.g. while loop
- it cannot be applied if the number of iterations is a prime number (no divisor)

10.13 Function Call Inlining

Function call inlining is an optimization similar to inlining. The benefits it brings in are very similar to loop unrolling, namely

- it reduces the number of instructions by eliminating the function call and return overhead as well as input and output preparation
- it reduces CPI by providing more instructions to reorder for more parallelism

10.14 Function Call Inlining Downside

Very similar to loop unrolling, the downside of function call inlining is that the code size grows quickly with the number of function calls. Thus we want to inline only small functions.

10.15 Function Inlining Quiz

We can call the function while its input is still loading.

10.16 Other IPC Enhancing Compiler Stuff

software pipelining: treat the loop as a pipeline and schedule the loop so that we do the third stage of iteration #1 while doing the second stage of iteration #2 and the first stage of iteration #3, in order to get the benefit of unrolling while avoiding significant increase in code size

trace scheduling: find the common path for a branch, put together the blocks on the common path without branch, and apply instruction scheduling between them, also need to check whether the common path is not being executed, if so branch out of the scheduled common path and correct the reordering in it

11 VLIW

11.1 Superscalar vs. VLIW

VLIW stands for very long instruction word processors. Its differences from out-of-order and in-order superscalar processors are

	out-of-order superscalar	in-order superscalar	VLIW
instructions per cycle	up to N	up to N	1 equivalent work of N
find indep. instructions	a window of $\gg N$	next N in order	just do next one
hardware cost	expensive	less expensive	cheapest
help from compiler	can help	need help	completely rely on

11.2 Superscalar vs. VLIW Quiz

For a VLIW that can package four instructions of an out-of-order processor into one, the corresponding program size can range from having the same size (if no dependence beyond four instructions) to four times as large (if all instructions belong to one single dependence chain).

11.3 The Good and the Bad

The pros are

- compiler does all the hard work, but it has plenty of time to figure out a good schedule, because we compile a program only once but execute it many times
- the hardware is simpler
- it can be more energy efficient, because the hardware has less to do per execution
- it works well on loops and regular code such as sweeping through arrays and multiplying matrices

The cons are

- latencies are not always the same (e.g. cache miss can result in much longer latency than what the compiler planned for)
- many applications are irregular (e.g. those involving decision making) and are thus not applicable
- code size can be larger when there are many dependencies thus lots of NOPs need to be inserted

11.4 VLIW Backward Compatibility Quiz

A processor that packs two instructions of a VLIW into one in order to double the IPC while maintaining backward compatibility with the VLIW is not a VLIW. Because in a VLIW the entire N-instruction word should be free of dependence, which is not guaranteed for more than one VLIW instruction.

11.5 VLIW Instructions

The instruction set for a VLIW processor typically has

- all the normal ISA opcodes so it can do whatever normal instructions can do in an out-of-order processor
- support for extensive predication, because it relies on the compiler to expose parallelism and one way the compiler achieves that is through instruction scheduling
- lots of registers, because many instruction scheduling optimizations require additional registers
- branch hints where the compiler can specify to the hardware what it thinks the branches will do
- instruction compaction—replacing NOPs with stop bits

11.6 VLIW Examples

Intel Itanium processor: very complicated ISA and hardware, still not good with irregular code

DSP processors: usually have excellent performance and energy efficient because digital signal processing involves lots of regular small loops iterated over many times

11.7 VLIW Target Market Quiz

Counting the number of elements of a linked list would probably involve lots of load instructions, which renders the compiler scheduling vulnerable to cache misses.

12 Cache Review

12.1 Locality Principle

Locality principle states that things that will happen soon are likely to be close to things that just happened.

12.2 Locality Quiz

Things that are unique thus do not repeat don't have locality property. In fact the exclusive property of uniqueness is opposite to locality principle.

12.3 Memory References

The locality property of memory references states that

temporal locality: if an address was accessed recently, it is likely that it will soon be accessed again

spatial locality: if an address was accessed recently, it is likely that the addresses close to it will also be accessed soon

12.4 Spatial Locality Quiz

When a program is compiled, usually there will be some spatial locality between related variables.

12.5 Locality and Data Accesses

A library represents a data repository that is large but slow to access, and the accesses to the information in the library have temporal and spatial locality. The best way to accesses information in the library that leverages temporal and spatial locality is to borrow the book. Main memory is just like a library, large but slow to access. Therefore when a processor needs to access main memory, it will bring in not only the content of the memory location of interest but also that of a few nearby memory locations. This small repository of information is called a cache.

12.6 Cache Lookups

For fast access we need to keep cache small, which means that not everything will fit the cache. Hence when a processor wants to access some content, we can have

cache hit: i.e. found it in the cache

cache miss: i.e. did not find it in the cache, in which case the processor has to access the slow main memory

The more locality the more cache hits. When we have a cache miss, motivated by temporal locality we should copy this location to the cache.

12.7 Cache Performance

The average memory access time or AMAT is defined as

$$\text{AMAT} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$$

To have a small AMAT we need to have a small hit time which requires a small but fast cache, and/or a small miss rate which requires a large and/or smart (typically slow) cache. Hence when we design caches we need to balance the hit time and the miss rate. For simple cache the miss penalty is simply the main memory access time which amounts to tens or even hundreds of processor cycles.
Sometimes we also use miss time, which is the overall access time when we have a cache miss and is defined as

$$\text{miss time} = \text{hit time} + \text{miss penalty}$$

Note that it coincides with AMAT when the miss rate = 100%, which suggests an alternative definition of AMAT

$$\text{AMAT} = (1 - \text{miss rate}) \times \text{hit time} + \text{miss rate} \times \text{miss time}$$

We usually use the first definition because the miss time typically includes checking whether the required content is in the cache. Hence no matter whether it is a hit or a miss we need to pay for the hit time. (⇒ the two definitions are equivalent, just substitute the definition of miss time into the second and take out the hit time common factor)

12.8 Hit Time Quiz

In a well-designed cache hit time should be much smaller than miss penalty, and thus miss time should be roughly the same as miss penalty.

12.9 Miss Rate Quiz

In a well-designed cache hit rate should be almost 1 and much larger than miss rate.

12.10 Cache Size in Real Processors

L1 cache directly services read/write requests. Its typically size is 16KB to 64KB, which is large enough to get close to 90% hit rate and small enough to achieve a hit time of 1 to 3 cycles.

12.11 Cache Organization

To quickly determine whether it is a hit or a miss, cache is designed to be a lookup table in which we index the entries with some bits from data address. How many bytes we have in each cache entry is called the **block size** or **line size**. We want the block size to be at least as large as the largest single access that we can do in the cache. In fact we want to be able to accommodate more than the single largest access to leverage spatial locality. Thus the typical block size is 32 to 128 bytes to make a good balance between leveraging spatial locality and saving storage for programs without locality.

12.12 Block Size Quiz

For a program that is to access N variables with lots of temporal locality but no spatial locality, the largest N that can result in a high hit rate is equal to the number of blocks available in the cache, i.e. N = cache size ÷ block size. Because without spatial locality each variable would occupy one separate cache entry. To accommodate even more variables without sacrificing hit rate, we need some variables

to be close to each other so that we can benefit from spatial locality, i.e. one cache block may contain more than one variable.

12.13 Cache Block Start Address

If anywhere is allowed as the starting address, then the possible overlap among blocks would complicate the writing, since we need to ensure that every block is updated accordingly for the overlapped content. Therefore we only have caches where the blocks start at block aligned addresses.

12.14 Blocks in Cache and Memory

Cache lines are basically slots where a memory block can fit in. To distinguish cache space and cache content, the space in a cache is called a **line** and the content we put there is called a **block**.

12.15 Cache Line Size Quiz

1B is not a good line size because it not only does not allow exploiting spatial locality but also forces multiple block accesses even for a single word access. 1KB is not a good line size either because the cache size is only 2KB. A good line size not only allows exploiting spatial locality but also results in a sizable number of lines in the cache.

12.16 Block Offset and Block Number

With a block size of 16B, the least significant 4 bits of a 32-bit address specify where within the block the data are located while the remaining 28 bits specify which blocks the data can be found. The 4 bits are called the **block offset** while the 28 bits are called the **block number**.

12.17 Cache Tags

In addition to data, cache also keeps tags that specify which block is stored in a given cache line. To determine whether it is a cache hit, the block number of a given address is compared with the tags of the cache. Therefore the block number of an address is also called the **tag** part of the address.

12.18 Cache Tag Quiz

A tag contains at least one bit from the block number even though in general it may not contain all the bits of the block number.
A block always begins at an aligned address, so the first bit of the block is always zero.

12.19 Valid Bit

When the processor is turned on, the initial value of a tag may match that of an actual address no matter what the initial value is, yet the data stored in the cache line is garbage. To prevent from loading the garbage data in an apparent cache hit, an additional *valid bit* is added to each cache line to indicate whether data retrieval from main memory is completed or not. Hence the cache hit condition becomes tag matching block number AND valid bit equal to 1.

12.20 Types of Caches

fully associative: a block can go to any cache line

set-associative: a block can go to any one of a set of N cache lines, where $1 < N \ll$ total number of cache lines

direct-mapped: a block can go to only one particular cache line

12.21 Direct-Mapped Cache

For any given block address there is only one cache line that the address can go to. The least significant bits of the address are still the block offset. But the block number now contains the **index** bits of the unique cache line the address can go to, with the remaining bits of the address contained in the tag. Note that the tag does not contain the full block number, because the specific cache line we are looking at determines the index bits thus we already know the index bits.

12.22 Upside and Downside of Direct-Mapped Cache

pro: only need to check one cache line thus faster, cheaper, and more energy efficient

con: a block can only go to one cache line which may result in conflict (multiple blocks may fight for one cache line) and hence increased miss rate

12.23 Direct-Mapped Cache Quiz

The key to solving this kind of problems is to break down the given address into offset, index, and tag bits. To determine whether a block conflicts with the given one, check

- whether the index is the same, note that for this quiz the index has 6 bits (as there are 16K/256 = 64 cache lines) so we need to check the trailing 6 bits of the 2-digit hex (e.g. 0x12341666 and 0x12345678 have the same index albeit their hex digits are different 16 vs. 56)
- whether the block number is the same, the same block number means that they should be in the same cache line thus there is no conflict (e.g. 0x12345677 and 0x12345678 have the same block number 123456)

12.24 Direct-Mapped Cache Quiz 2

Since the line size is 32B, the least significant 5 bits would be the offset; since there are 8 cache lines, the next 3 bits would be the index and the remaining bits would be the tag.
Since 0x3F2E and 0x3F2F have the same tag and index, they belong to the same cache line; since 0x3F1F and 0x3E1F have different tags but the same index, 0x3F1F would be replaced by 0x3E1F in cache line 0.

12.25 Set-Associative Caches

A cache is an N-way set-associative cache if a block can be in one of N lines. Do not confuse N with the number of sets.

12.26 Offset, Index, Tag for Set-Associative

Identical to direct-mapped caches, the least significant bits are the offset which is determined by the block size, and the tag does not need to include the full block number because the specific set we are looking at determines the index bits thus we already know the index bits. Different from direct-mapped caches, the index bits are those that specify the set of cache lines not the cache line that the block goes to. In addition since the number of sets is smaller than the number of cache lines, the index bits for set-associative caches are generally fewer than those of direct-mapped caches with the same number of cache lines.

12.27 2-Way Set-Associative Quiz

An N-way cache reduces the number of conflicts. However it requires more complicated tag check because now we need to search in N lines.

12.28 Fully Associative Cache

Because any block can go to any cache line, there is no need for index bits. Therefore a block address consists of only the offset and the tag.

12.29 Direct-Mapped and Fully Associative

Both direct-mapped and fully associative can be considered as special cases of set-associative—direct-mapped is 1-way associative whereas fully associative is M-way associative where $M =$ the total number of cache lines.
When we try to figure out which bits are the index bits, we first need to figure out which bits are the offset bits.

12.30 Cache Replacement

The situation when we need cache replacement is when we have a cache miss thus need to put the new data block in the cache, but the set of cache lines the block should go to is full. As a result we need to determine which existing block we should remove from the set to make room for the new one. Possible replacement policies are (1) random (2) FIFO (3) least recently used or LRU. It turns out that LRU is the best among the three but impractical to implement (see below). There are several practical policies that approximate LRU, one such being not most recently used or NMRU. NMRU tracks which block has been used the most recently and picks randomly among the remaining blocks.

12.31 Implementing LRU

LRU works well because it exploits (temporal) locality. But in order to track which block was recently used we need to have an LRU counter for each block. The LRU counter value ranges from 0 to $N-1$, with the least recently used block $= 0$ and the most recently used block $= N-1$. When we want to replace an existing block, we will replace the block whose count is currently 0. The LRU counter works as follows: for a set in an N-way associative cache, when a block is accessed its counter is set to $N-1$. To ensure that all the counters still have different values, the counters that initially have a larger value than the counter that is hit will be decremented by one, while the counters that initially have a smaller value than the counter that is hit will remain unchanged. Hence maintaining LRU is relatively complicated, because we need to ensure that the $N \log_2 N$-bit counters have different values all the time. Moreover maintaining LRU is also energy inefficient, because we need to modify up to N

counters on each access even when cache hits happen frequently. This motivates LRU approximations which try to keep fewer counters and make fewer updates.

12.32 Write Policy

The write policy of a cache consists of two parts

allocate policy: do we allocate cache for the block we are to write? there are two types according to this choice

write-allocate: bring into cache the block we are to write

not-write-allocate: does not bring into cache the block we are to write

most caches nowadays are write-allocate, because there is some locality between reads and writes i.e. if we write something now we are likely to read it later

write policy: write to just cache or also main memory? there are two types according to this choice

write-through: write to main memory immediately

write-back: write to cache only, write to main memory only when the block is to be replaced in the cache

we prefer write-back because in that way we can not only exploit write locality but also prevent main memory from being overwhelmed by many writes

There is in fact some relationship between the allocate policy and the write policy—if you want to have a write-back cache you also need a write-allocate cache to support the write-back.

12.33 Write-Back Caches

For write-back caches we can have a block that we did write since it was last brought in from main memory, we can also have a block that we never wrote since it was last brought in from main memory. In the latter case there is no need to write that block back to main memory when it is to be replaced. To distinguish them we add a *dirty bit* to every block in the cache to indicate whether we have written to the block since it was last brought in from main memory, with 0 representing clean blocks and 1 representing dirty blocks.

12.34 Write-Back Cache Quiz

If the valid bit is 0, then the dirty bit doesn't matter.

12.35 Cache Summary

In addition to the 64-bit address we need a valid bit, a dirty bit if it is a write-back cache, and a LRU counter if it is not direct-mapped and its replacement policy is LRU. Hence the actual size of the cache is larger than the address and the data. Note that we don't check whether the dirty bit is 0, we simply set it to 1 regardless of what it was.

12.36 Cache Summary Quiz 2

To determine whether it is a cache hit, we only check the tag ignoring the offset.

13 Virtual Memory

13.1 Why Virtual Memory

We want to have virtual memory as the programmer and the hardware have different views of memory

hardware view: the machine has some memory modules, if there are two modules and each of which is 2GB then the memory addresses range from 0 to $2^{32} - 1$

programmer view: the memory addresses range from 0 to $2^{64} - 1$ for a 64-bit machine and thus its memory usage may be much more than the actual memory, each concurrently running program holds the same view and accesses memory as if it is the only program that is running on the machine (i.e. no memory conflict)

Virtual memory is a way to reconcile how the programmer views memory and how the hardware views memory.

13.2 Processor's View of Memory

The processor sees physical memory which is the memory contained in the actual memory modules. In fact the amount of memory available to the processor is less than that in the actual memory modules. The memory addresses that the processor uses have a 1-to-1 mapping to the bytes/words in the physical memory.

13.3 Program's View of Memory

A program sees a huge amount of memory, ranging from 0 to $2^{64} - 1$ for a 64-bit machine. This is the program's virtual memory. Usually a program uses some contiguous regions of its virtual memory with stack on the top and an enormous region in the middle between the heap and the stack that it would never access unless the heap grows into it. Each program has its own virtual memory. Different virtual memory addresses of different programs may point to the same physical memory address e.g. for data sharing.

13.4 Mapping Virtual to Physical Memory

The program's memory is divided into equal-size chunks called **pages**. A typical page size is 4KB. Each page is aligned to the page size i.e. each page begins at a multiple of 4KB. The physical memory is divided into slots that can host pages which are called **frames**, thus the physical memory behaves like a cache for the virtual memory. The operating system creates a mapping from the pages of the virtual memory to the frames of the physical memory, and the mapping mechanism is called a **page table**.

13.5 Page Table Quiz

The number of page frames is given by

$$\text{number of page frames} = \frac{\text{total physical memory}}{\text{page size}}$$

The number of entries in each page table is given by

$$\text{number of page table entries} = \frac{\text{total virtual memory}}{\text{page size}}$$

13.6 Where is the Missing Memory

The missing memory is in the hard disk. Because the virtual memory of all applications can significantly exceed the size of the physical memory, some of the pages are stored on the hard drive, which cannot be directly accessed by the processor as the processor can only access through loads from the physical memory.

13.7 Virtual to Physical Translation

The processor divides the virtual address generated by a program into the **page offset** part that tells where in the page the data are located, and the **page number** part that tells which page we should look for the data. The page offset bits are the least significant bits while the remaining bits go to the page number. With a page size of 4KB the page offset bits are the least significant 12 bits. The virtual page number is used to index the page table entries, each of which contains the corresponding physical frame number. The physical frame number together with the page offset form the physical memory address that corresponds to the virtual memory address. Hence in virtual-to-physical memory translation, only the virtual page number is translated into the physical frame number, the page offset remains the same.

13.8 Address Translation Quiz

Since there are four page table entries, the two most significant bits are the virtual page number, which is translated into the physical frame number stored in the corresponding page table entry.
By the way the physical address seems to be 22-bit instead of 20-bit, because the page table entries are 8-bit while the page offset is $16 - 2 = 14$ bits ($\Rightarrow$ the leading two bits are truncated because the leading two bits of all the page table entries are 00).

13.9 Size of Flat Page Table

What we have discussed so far are the so-called **flat page table**, which has the following properties

- it has one entry for each page of the virtual memory, even for pages that the program never uses
- its entries contain the physical frame number plus a few bits that tells whether the corresponding page is in the physical memory or not, thus the entry size is similar to that of the physical address

Therefore the overall size of a page table is given by

$$\text{page table size} = \frac{\text{total virtual memory}}{\text{page size}} \times \text{size of page table entries}$$

Thus one problem with the flat page table is that the page table is large even if the program uses very little of its available virtual memory. Another problem is that even though for a 32-bit virtual memory a flat page table has a size of several MB which can be easily accommodated, for a 64-bit virtual memory a flat page table would be many TB thus is much larger than the available physical memory.

13.10 Page Table Quiz

There are two processes in the system thus there will be two page tables and the total page table size is the sum of the two. It doesn't matter how much memory the processes actually use. It doesn't matter how much physical memory we actually have either.

13.11 Multi-Level Page Tables

The problem of flat page tables is two-fold—its size is not proportional to the actual virtual memory usage and its size far exceeds the physical memory available for a 64-bit virtual memory. Multi-level page tables are designed to overcome these two problems. The idea comes from the observation that applications usually use only the bottom contiguous region of its virtual memory for code, data, and heap and the top contiguous region of its virtual memory for stack, leaving the much more virtual memory in between unused. Multi-level page tables adopt the flat page table's idea of using bits to index page tables but avoid indexing the unused virtual memory region between heap and stack.

13.12 Multi-Level Page Table Structure

A multi-level page table still divides a virtual address into page number and page offset. But instead of using the entire page number to access one huge table, it partitions the page number into the so-called outer page number and inner page number. The inner page tables are identical to the flat page table discussed above, while the outer page number indexes which inner page table the virtual address should use to translate its inner page number into the physical frame number.

13.13 Two-Level Page Table Example

If all virtual memory is occupied, the total size of all inner page tables would be the same as the flat page table and we also need to store outer page table. The savings come from the fact that if there is any unused contiguous region of virtual memory with a size exceeding page size, then we don't need an inner page table and a corresponding outer page table entry for it. As a result, since there is usually a large region of unused virtual memory between heap and stack, we would often have one reasonably large outer page table but only a small number of inner page tables.

13.14 Two-Level Page Table Size

Since both the top and bottom contiguous regions of the virtual memory used share the same outer page number, we only need two inner page tables, one for each. In general when we need to compute the required two-level page table size, we just need to find out the range of the outer page numbers to determine how many inner page tables are needed. But don't forget to add the outer page table size.

13.15 Four-Level Page Table Size Quiz

The easiest way to figure out how large the multi-level page table is to first find out how many innermost page tables are needed, and then work backward to determine how many upper-level page stables are needed.
All 64-bit x86 processors nowadays use at least three-level page tables.

13.16 Choosing the Page Size

Larger pages lead to smaller page tables but suffer from internal fragmentation, which occurs when an application requests only a small amount of memory (or a few pages plus a small amount) but we can only allocate memory in unit of pages resulting in waste of memory. Hence choosing page size requires a tradeoff between having small page tables and avoiding internal fragmentation. Practically a good compromise is reached at page sizes ranging from a few KB to a few MB. This is why x86 processors use 4KB pages. In fact at the time they were designed the preference was avoiding internal fragmentation because memory was precious, which is not the case nowadays.

13.17 Memory Access Time with V-P Translation

With virtual-to-physical memory translation, after computing the virtual address and page number the processor has to

1. compute the physical address of the page table entry
2. read the correct page table entry
3. compute the physical address by combining the translated frame number with the page offset

The two computations are fast. But since page tables can be large, the required page table may be stored in main memory. Thus reading it may require access to main memory which is just as slow as when we have a cache miss. Things are even worse if we have multi-level page tables, because we need to repeat the above three steps N times for a N-level page table. Hence it may cost even more than just accessing main memory that we want to avoid by using cache.

13.18 V-P Translation Quiz

Using N-level page tables that cannot be cached, the number of cycles for executing the LW $R1$ 4($R2$) instruction is

$$\begin{aligned}\text{total number of cycles} = {} & \text{number of cycles to compute virtual address} \\ & + \text{number of cycles to access main memory} \times N \\ & + \text{number of cycles to access cache} \\ & + \text{number of cycles to access main memory} \times \text{miss rate}\end{aligned}$$

If page table entries are cached just like data, then the number of cycles becomes

$$\begin{aligned}\text{total number of cycles} = {} & \text{number of cycles to compute virtual address} \\ & + [\text{number of cycles to access cache} \\ & + \text{ number of cycles to access main memory} \times \text{miss rate}] \times (N+1)\end{aligned}$$

where the N represents page table entry retrieval and the +1 represents data fetch. Hence the virtual-to-physical memory translation is expensive even when page table entries can be cached just like data.

13.19 Translation Look-Aside Buffer (TLB)

To speed up virtual-to-physical translation processors include a structure call the **translation look-aside buffer** (TLB), which is a cache for translations. It differs from conventional cache in two ways

- because cache stores not only translations but also data (in fact the vast majority of cache lines are devoted to data), cache is much larger than TLB which stores only translations, due to the much smaller size access to TLB can be much faster
- for translation via a multi-level page table, cache needs to be accessed for each level of the page tables, in contrast TLB stores only final translations thus needs to be accessed only once

Therefore if we have both TLB hit and cache hit, the load/store can be done in one or two cycles. If we have a TLB miss, then we need to translate using page table(s), and upon completion put the final translation in TLB.

13.20 What If We Have a TLB Miss Quiz

There are two ways to perform translation using page table(s) and put final translation in TLB

software TLB miss handling: the operating system reads page tables and updates TLB

pro: the operating system can use any sort of page table that it deems appropriate, it may even use tree or hash table instead of page table

con: slower than hardware handling because it needs to execute a program to fill TLB

adopted by embedded processors because hardware cost is high and TLB misses are rare due to mostly regular execution

hardware TLB miss handling: the processor reads page tables and updates TLB

pro: faster than software handling

con: the page tables need to be in a form that is easily accessible by the hardware (e.g. flat or multi-level) and more hardware is required

adopted by most high-performance processors like x86 because hardware is cheap nowadays

13.21 TLB Size Quiz

For a 32KB cache with a block size of 64B, suppose the page size is 4KB then we have 512 cache blocks and 8 pages. If the processor just needs to access the 32KB cache then the TLB only needs 8 entries to cover the same memory. However if the processor needs only partial data in each page then we need more pages to cover the same memory. In the worst case the 512 cache blocks belong to 512 different pages which requires 512 TLB entries. In reality the number of pages and thus TLB entries we need for the same hit rate in cache and TLB lies between these two extremes.

13.22 TLB Organization

Sine TLB is already fast, it tends to be fully or highly associative. Usually we don't have direct-mapped TLBs because that would sacrifice hit rate.
We also want a TLB to cover more memory than cache so as to achieve a hit rate that is close to cache hit rate. Usually we want a TLB to have 64 to 512 entries. If more entries are desired we may have a two-level TLB—the first level is small and fast, if it is missed we go to the second level which is large but slow (several cycles).

13.23 TLB Performance Quiz

Suppose we have a 1MB page-aligned array that is to be read one byte at a time from start to end for 10 times. Suppose we have no other memory access except for 4KB pages, a 128-entry L1 TLB and a 1024-entry L2 TLB, where both TLBs are direct-mapped and initially empty. Since 256 pages are required to cover the entire array, the L1 TLB would have 10×256 misses followed by $10 \times (4096 - 1) \times 256$ hits. Because after the final translation for the first byte of a page is put in the L1 TLB, the remaining 4095 bytes in the translated page can find their final translations in the L1 TLB as they belong to the same page.
The L2 TLB would have 256 misses followed by $(10 - 1) \times 256$ hits. Because the L2 TLB has enough entries to store the final translations for all the 256 pages, hence after the first sweep there would be no more miss. In contrast the final translations for the first 128 pages would have to be put in the L1

TLB again during the second sweep. On the other hand we access the L2 TLB only when we have a miss in the L1 TLB. Therefore we have the following equation

$$\text{L1 TLB misses} = \text{L2 TLB hits} + \text{L2 TLB misses}$$

14 Advanced Caches

14.1 Improving Cache Performance

Because the average memory access time or AMAT is given by

$$\text{AMAT} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$$

the methods for improving cache performance can be grouped into three categories: (1) reduce hit time (2) reduce miss rate (3) reduce miss penalty.

14.2 Reduce Hit Time

Methods that can reduce hit time include

- reduce cache size, which however would increase miss rate
- reduce cache associativity, which would also adversely affect miss rate due to more conflicts
- overlap one cache hit with another cache hit
- overlap cache hit with TLB hit
- optimize lookup for common cases
- maintain replacement state more quickly

We will look at the latter four less simple methods to reduce hit time.

14.3 Pipelined Caches

One way to overlap one cache hit with another is to pipeline a cache if accessing it takes multiple cycles. One example of a cache pipeline would be to have tag reading as stage one, hit checking, block number and valid bit reading as stage two, offset and data reading as stage three. Usually the hit time for L1 cache will be one, two, or three cycles, hence usually L1 cache would be pipelined.

14.4 TLB and Cache Hit

A cache that is accessed using physical address is called a physically accessed cache. Because TLB access and cache access have to happen in order, the hit time for a physically accessed cache would take more than one cycle.

14.5 Virtually Accessed Cache

We can improve the hit time by using a virtually accessed cache, which uses virtual address to access data and get virtual address translated into physical address by TLB only upon cache misses. The advantages of virtually accessed cache over physically accessed cache are that (1) the hit time is just the cache hit time (2) no TLB access on cache hit thus saves energy.
However in reality TLB contains not only final translation but also permission bits, thus we still need to access TLB even on cache hits and the second advantage does not exist in real processors. In fact the bigger problem associated with virtually accessed cache is that the virtual addresses of subsequent program may overlap with those of the previous which however map to different physical addresses. This requires flushing the cache on every context switch which would result in a burst of cache misses.

14.6 Virtually Indexed Physically Tagged

Virtually-indexed physically-tagged (VIPT) cache was invented to combine the advantages of physically accessed cache and virtually accessed cache. It got its name because the index bits of virtual address are used to find the set of cache lines, meanwhile the tag of virtual addresses are translated into physical frame number by TLB and it is this physical tag not the virtual tag that is used for tag check. This design brings in the following advantage

- since the cache access and the TLB access proceed in parallel and the TLB access is usually much faster, the hit time would equal to the cache hit time, this achieves the speed advantage of virtually accessed cache
- since the tag check is on physical tag not virtual tag, no cache flush is necessary upon context switch, this achieves the advantage of physically accessed cache

This design also prevents aliasing, which occurs when multiple virtual addresses in the same address space map to the same physical address. Because it uses virtual index bits for cache access, the virtual addresses may end up in different sets. It turns out that there would be no aliasing problem in VIPT cache as long as the cache is small enough.

14.7 Aliasing in Virtually Accessed Caches

Aliasing occurs when multiple virtual addresses in the same address space map to the same physical address, so that writing to one virtual address will not be reflected in the read from another virtual address that shares the same physical address. For a virtual accessed cache to prevent this from happening, it needs translation into physical address to check possibly different versions of the same physical data, defying the purpose of virtual access.

14.8 VIPT Cache Aliasing

Recall that in virtual-to-physical memory translation only the virtual page number is translated into the physical frame number, the page offset remains the same. Therefore as long as all the index bits come from the page offset, they remain the same after translation, which implies that the same physical data (with the same page offset) would stay in the same set of cache lines. This prevents aliasing in VIPT cache. However to ensure that all the index bits are from the page offset, the cache size must be small enough so that the number of sets available does not exceed 2^i where i = the number of page offset bits − the number of block offset bits. Otherwise the index bits may change with the physical frame number, making storing the same physical data in different sets of cache lines possible.

14.9 VIPT Aliasing Avoidance Quiz

The maximum VIPT cache size to avoid aliasing is given by

$$\begin{aligned}\text{max cache size} &= 2^i \times \text{block size} \times N\text{-way associativity} \\ &= \text{page size} \times N\text{-way associativity}\end{aligned}$$

The second equality suggests that the only way to increase cache size while avoiding aliasing is increasing associativity.

14.10 Real VIPT Caches

Pentium 4: 4-way associativity × 4KB page size = 16KB L1 cache

Core 2, Nehalem, Sandy Bridge, Haswell: 8-way associativity × 4KB page size = 32KB L1 cache

Skylake: 16-way associativity × 4KB page size = 64KB L1 cache

14.11 Associativity and Hit Time

With higher associativity we could have both fewer conflicts and larger VIPT caches, both reduce miss rate. However higher associativity leads to slower hit because there are more cache lines to check.

14.12 Way Prediction

One way of cheating on associativity to reduce hit time is **way prediction**—guess which cache line in the set is the most likely to hit. If our guess is correct then it gives us the hit time similar to that of a direct-mapped; if our guess is wrong then we resort to normal set-associative check which will have a longer hit time.

14.13 Way Prediction Performance

We can estimate an 8-way set-associative way prediction VIPT cache with a set size of 32KB by looking at a 4KB direct-mapped cache, because the 8-way cache would first behave as a direct-mapped cache with a $32/8 = 4$KB cache line. Suppose without way prediction the cache would have a 90% hit rate, a hit latency of 2 cycles, and a miss penalty of 20 cycles, a normal 4KB direct-mapped has a 70% hit rate, a hit latency of 1 cycle, then the 8-way cache with way prediction has a 90% hit rate, a hit latency of 1 or 2 cycles, and a miss penalty of 20 cycles. Its AMAT can be estimated as

$$\text{AMAT with way prediction} = (70\% \times 1 + 30\% \times 2) + (1 - 90\%) \times 20$$

14.14 Replacement Policy and Hit Time

For a random replacement policy, there is nothing to update on cache hit which results in faster hit, but it potentially increases miss rate because often the block that will be used very soon is kicked out accidentally. In contrast the miss rate for a LRU replacement policy is low, but we have to update up to N counters on each cache hit where N is the associativity. Even the most recently used block is hit, we still need to check whether the other counters are all smaller than it so as to ensure that no counter needs update. This results in slower hit and spends a lot of power. Hence we want a replacement policy that can achieve a miss rate which is very close to that of LRU and require little activity on cache hit.

14.15 NMRU Replacement

Not-most-recently-used or NMRU replacement policy tracks which block in the set is the most recently used (MRU) one and kicks off a non-MRU one on replacement. Since it tracks only the MRU it only needs one $\log_2 N$-bit pointer per set for tracking where N is the associativity. It does have a hit rate that is slightly lower than LRU, because it is unable to identify and kick off the least recently used among the non-MRU ones.

14.16 PLRU Replacement

The pseudo-LRU or PLRU replacement policy maintains one bit per line in a set of cache lines as opposed to $\log_2 N$ bits for LRU. Initially all the bits start at 0. After that every time a cache line is accessed we will set its bit to 1. If cache replacement is needed we pick among the blocks whose bits are 0. Eventually the last 0 bit is set to 1. At this time we set the remaining bits to 0. Thus PLRU behaves somewhere between LRU and NMRU—when there is only one 0 bit it behaves like LRU, when there is only one 1 bit it behaves like NMRU. Since PLRU maintains only one bit per line, both PLRU and NMRU have much less activity on a cache hit than LRU.

14.17 Reducing the Miss Rate

There are three causes of misses, the so-called 3Cs, which are

compulsory miss: the first time the data block is brought into cache, it would be a miss even for an infinite cache

capacity miss: the desired data block was evicted due to limited cache size, it can be avoided with a larger cache but would still be a miss even in fully associative cache of the same size

conflict miss: the desired data block was evicted due to limited associativity, it would not be a miss in a fully associative cache of the same size

A larger cache would reduce capacity miss. A larger associativity would reduce conflict miss. A better replacement policy would also reduce conflict miss. But all of these also affect hit time.

14.18 Larger Cache Blocks

With larger cache blocks more words can be brought in on a cache miss, This reduces miss rate if the spatial locality of these words is good, but increases miss rate if the spatial locality of these words is poor because we brought in more junks which reduces capacity. If we plot miss rate against block size, we would have a parabolic curve in which miss rate initially decreases with increasing block size and then increases. For a 4KB cache the bottom of the curve or the optimal block size occurs at 64B. The curve is lower and bottoms slower for larger caches, e.g. the bottom occurs at 256B for a 64KB cache. Because larger caches can accommodate more junks before having those with good spatial locality feel the capacity limit. Overall we can reduce miss rate by having larger cache block size.

14.19 Miss Rate Quiz

Larger cache block size reduces all three types of cache misses. Because with larger blocks there would be fewer blocks to be brought in for the first time, fewer blocks to be kicked out due to capacity limit, and fewer blocks conflicting with each other.

14.20 Prefetching

Prefetching guesses which blocks will be accessed soon and brings them in ahead of time. However while good guess reduces cache miss, bad guess leads to cache pollution which may create additional cache miss.

14.21 Prefetching Instructions

One way of implementing prefetching is to add prefetch instructions to a program and let the compiler figure out when to request prefetches. For example

$$\begin{aligned}&\text{for } (i = 0; i < 10000000; i++)\\ &\quad \text{prefetch } a[i + p]\\ &\quad \text{sum} = \text{sum} + a[i]\end{aligned}$$

The tricky part is what value should p be. Because if p is too small then the prefetched data may still hasn't arrived when it is accessed due to memory latency; if p is too large then the prefetched data may have been kicked out when it is accessed due to limited capacity. What is worse the appropriate p may change with the hardware due to different processor speed and memory latency.

14.22 Prefetch Instructions Quiz

For the following program

$$\begin{aligned}&\text{for } (i = 0; i < 1000; i++)\\ &\quad \text{for } (j = 0; j < 1000; j++)\\ &\quad\quad a[i] = a[i] + b[i][j]\end{aligned}$$

where the elements of a array and b array are 8-byte floating numbers. With a 16KB fully associative LRU cache and a 200-cycle latency memory, suppose that it takes 10 cycles to complete one iteration if there is no miss, the appropriate p value for prefetching b elements is $200/10 = 20$, while the appropriate p value for prefetching a element is $\lceil 200/(10 \times 1000) \rceil = 1$. For the latter we also need to check whether the prefetched $a[i + 1]$ would be kicked out during the remaining $10 \times 1000 - 200$ cycles due to cache capacity. If yes then we should not prefetch it. Luckily the inner loop loads only 1000 8-byte b elements and one 8-byte a element but we have a 16KB cache to store them.

14.23 Hardware Prefetching

Hardware prefetching requires no change to a program, and lets either the processor or the cache guess which blocks will be accessed soon. There are a number of hardware prefetchers that work reasonably well, e.g.

stream buffer: guess the next block(s) in sequence are likely to be accessed soon

stride prefetcher: if previous memory accesses are all at a fixed distance from each other, then the next with the same distance or a multiple of it in advance are likely to be accessed soon

correlating prefetcher: if accessing A is often followed by accessing B, then when A is accessed again we should prefetch B

14.24 Loop Interchange

Loop interchange is one of the compiler optimizations which transforms a code into another code that has better locality. For example the nested loop initialization on the left below can be transformed into an equivalent one on the right with better locality, because C is row-major thus for the latter each row block of a array is accessed entirely before switching to the next row, which not only improves spatial locality but also makes prefetching easy to implement.

```
for (j = 0; j < 1000; j++)          for (i = 0; i < 1000; i++)
    for (i = 0; i < 1000; i++)          for (j = 0; j < 1000; j++)
        a[i][] = 0                          a[i][j] = 0
```

However loop interchange is only possible when the outer loop and the inner loop are independent.

14.25 Overlap Misses

An out-of-order processor continues working on other instructions after fetching a missed data block from main memory is initiated. But eventually it would run out of resources and stop probably before the load comes back from main memory, resulting in memory latency added to the execution time. But if the cache is non-blocking meaning concurrent loads/stores are allowed, before running out of resources the processor can issue another load that will be a cache miss. This is called *miss-under-miss*. If we manage to find three or four loads overlap, then the penalty paid by a blocking cache would be three or four times more than a non-blocking cache. The property that the processor exploits is called **memory-level parallelism**.
The processor can also support *hit-under-miss*, meaning hits to other blocks can be serviced and data returned after initiating the fetch for a cache miss.

14.26 Miss-Under-Miss Support in Caches

To support miss-under-miss operation we need to have miss status handling registers or MSHRs. When we have a miss we check MSHRs to see if there is any match.

no-match/miss: it is a new miss, and we allocate to it a MSHR remembering which instructions to wake up when the requested data come back from main memory

match/half-miss: the requested data have already been ordered from main memory, and we just add to the matched MSHR the instructions to wake up when the requested data come back, after waking up the instructions the MSHR is released for future miss

Ideally we want to have a few dozen of MSHRs to support miss-under-miss to achieve a memory-level parallelism of 16 or 32.

14.27 Miss-Under-Miss Quiz

An application that has a miss every 1000 instructions would not benefit from miss-under-miss, because the processor would run out of ROB entries until it finds the next miss as the number of ROB entries is fewer than 1000. Thus we never need to do miss-under-miss.

14.28 Cache Hierarchies

Beside overlapping multiple misses, we can also use multi-level caches or cache hierarchy to reduce miss penalty. The principle is that a miss in L1 cache would go to L2 cache to fetch the requested data. As a result L1 miss penalty no longer equals memory latency but is given by

$$\text{L1 miss penalty} = \text{L2 hit time} + \text{L2 miss rate} \times \text{L2 miss penalty}$$

14.29 AMAT with Cache Hierarchies

With cache hierarchy the average memory access time or AMAT is given by

$$\begin{aligned}
\text{AMAT} &= \text{L1 hit time} + \text{L1 miss rate} \times \text{L1 miss penalty} \\
\text{L1 miss penalty} &= \text{L2 hit time} + \text{L2 miss rate} \times \text{L2 miss penalty} \\
\text{L2 miss penalty} &= \text{L3 hit time} + \text{L3 miss rate} \times \text{L3 miss penalty} \\
&\vdots \\
\text{LL miss penalty} &= \text{main memory latency}
\end{aligned}$$

The last cache whose misses go directly to main memory is called the last-level cache or LLC.

14.30 L1 vs. L2

Comparing L1 cache with L2 cache

- L1 capacity $<$ L2 capacity, because L2 cache needs to have hits for blocks that were missed in L1 cache
- L1 latency $<$ L2 latency, we search in L1 cache first not because it has a better chance of finding the requested data than L2 cache but because it has a lower hit latency
- L1 accesses $>$ L2 accesses, because all accesses go to L1 cache first and only those missed go to L2 cache
- L1 associativity $<$ L2 associativity, because L1 cache needs to have a low hit latency

14.31 Multi-Level Cache Performance

Suppose we have a 16KB cache with a 90% hit rate and a hit latency of 2 cycles, a 128KB cache with a 97.5% hit rate and a hit latency of 10 cycles, and the miss penalty for both is 100 cycles. We form a cache hierarchy with the 16KB being the L1 cache and the 128KB the L2 cache, which would have a hit latency of 2 cycles for L1 hits and 12 cycles for L2 hits, a 90% L1 hit rate but 75% L2 hit rate ($\Rightarrow$ as $90\% + (1 - 90\%) \times 75\% = 97.5\%$). Its AMAT is given by

$$\text{cache hierarchy AMAT} = 2 + (1 - 90\%) \times [10 + (1 - 75\%) \times 100] = 5.5$$

14.32 Hit Rate in L2, L3, Etc

In the above example when the 128KB cache is used alone it has a 97.5% hit rate, but when it is used as the L2 cache in the cache hierarchy it only has a 75% hit rate. We refer to the former as the **global hit rate** and the latter the **local hit rate** which is the hit rate actually observed by a cache. Local hit rate can be much lower than global hit rate as in the above example, because accesses with good locality are filtered out by the L1 cache thus never reach the L2 cache. Hence local hit rate is hardware and configuration dependent, and when we compare caches we usually use global hit rate.

14.33 Global vs. Local Hit Rate

Global hit rate can be defined as

$$\text{global hit rate} = \frac{\text{number of hits in this cache}}{\text{number of memory references made by the processor}}$$

whereas local hit rate can be defined as

$$\text{local hit rate} = \frac{\text{number of hits in this cache}}{\text{number of accesses to this cache}}$$

They are different because not all the memory references made by the processor can reach the cache. Another popular metric of hit rate is misses per 1000 instructions or MPKI, the normalizer of which is 1000 instructions made by the processor.

14.34 Global and Local Miss Rate Quiz

For the L1 cache its global miss rate and local miss rate are the same. For the L2 cache since only 10% of the memory references made by the processor reach it, its global miss rate is its local miss rate divided by 10, because the denominator should be increased by 10-fold for global miss rate.

14.35 Inclusion Property

In a multi-level cache, blocks that are in the L1 cache

inclusion: have to also be in the L2 cache

exclusion: cannot also be in the L2 cache

Unless we enforce the inclusion or exclusion property, we will get neither inclusion nor exclusion, i.e. blocks that are in the L1 cache may or may not be in the L2 cache. This is because blocks that are the most recently accessed in the L1 cache may not be the most recently accessed in the L2 cache, since the access goes to the L1 cache and never reaches the L2 cache. As a result that block gets replaced in the L2 cache but not in the L1 cache. To enforce inclusion we need to add a so-called inclusion bit to the L2 cache, which is 1 if that block is also in the L1 cache so that it is excluded from cache replacement even if it is not the most recently accessed.

14.36 Inclusion Quiz

For a two-level cache that maintains the inclusion property, if a dirty block was replaced from the L1 cache and we now need to write it back, this write-back will hit the L2 cache because by the inclusion property the block that was in the L1 cache is still in the L2 cache.

15 Memory

15.1 Memory Technology: SRAM and DRAM

SRAM/DRAM stands for static/dynamic random access memory. Random access as opposed to sequential access means that we can access any memory location by address without need to go through all the memory locations.

SRAM: static means SRAM retains its data as long as it is powered on, several transistors are needed for one-bit storage thus fewer SRAM per unit area and more expensive, but the speed is faster

DRAM: dynamic means DRAM will lose data even when connected to power supply unless we refresh the data, one transistor is needed for one bit storage thus more DRAM per unit area and cheaper, however the speed is slower

Both SRAM and DRAM will lose data when the power is switched off.

15.2 One Memory Bit in SRAM

In both SRAM and DRAM, a bit sits at the intersection of a *wordline* that passes by many bits and a *bitline* that also passes by many bits. The wordline is wired to the gates of the field-effect transistors or FETs so as to open or close their connections to the bitline. When we want to write data, the wordline makes the cell connected to the bitline via the FET and then we put the bitline at the value we want; when we want to read data, the wordline makes the cell connected to the bitline via the FET and we let go of the bitline so that it can sense the value of the bit.
In SRAM the memory cell consists of two inverters forming a positive feedback loop to keep the data, each of which has two transistors. To make the overwriting of the data inside the feedback loop easier we typically have two FETs connecting the cell to two bitlines with opposite values at both ends, thus there are altogether 6 transistors. Moreover by looking at the difference between these two bitlines we can more quickly detect what data the cell stores. However since the bitlines are long we cannot write very fast.

15.3 One Memory Bit in DRAM

In DRAM we also have a FET that is activated by the wordline and connects the cell to the bitline. However the memory cell is made out of one single capacitor. When we want to store 1 in the cell we use the bitline to charge the capacitor via the FET; when we want to store 0 in the cell we let the capacitor discharge into the bitline via the FET. The problem of storing data in a capacitor is two-fold

leakage: the FET is not a perfect insulator when it is deactivated, thus the charges in the capacitor will slowly leak into the bitline over time

destructive read: as read is carried out through discharging, once the read is completed the charges in the capacitor will be gone, this necessitates write-back i.e. refresh, which is one reason why DRAM is slower than SRAM

The structure of DRAM makes it look like its area is the sum of the area of the FET and the capacitor. But in modern technology the FET and the capacitor are built as one single transistor called *trench cell*, with the capacitor buried deep into the silicon substrate. This avoids the tradeoff between smaller DRAM area and larger capacitance (thus slower leakage, capacitance is proportional to capacitor area). Because DRAM consists of one single transistor and one single bitline, its area is smaller than that of SRAM.

15.4 DRAM Technology Quiz

Trench cell is harder to make because it requires us to bury something deep into the silicon substrate. By the same reason it is less reliable than a normal transistor + normal capacitor combination. However because trench cell occupies much less area than the transistor + capacitor combination, it lets us make cheaper chips even though it is more difficult to make.

15.5 Memory Chip Organization

There are a number of wordlines. A *row decoder* decides which wordline gets activated according to some bits of the address called *row address*. Only one wordline can be activated at a time. A memory cell exists at every intersection between a wordline and a bitline.
Since bitlines are long, for SRAM it would take the weak cell long time to raise or lower its voltage, and for DRAM the discharge of the small capacitor of a trench cell would change its voltage very little. Therefore bitlines are commonly connected to a device called *sense amplifier*, which senses and amplifies small changes in the voltage of bitlines. Due to its powerful circuitry sense amplifier is significantly larger than a single row of cells. But one sense amplifier is sufficient to support all bitlines. The signal that is produced by sense amplifier is then sent to a storage element called *row buffer* which stores the correct values of all bitlines. The row buffer then feeds the data it latched to another decoder called *column decoder*, which selects the correct bitline value according to some other bits of the address called *column address*.
Due to destructive read, after the sense amplifier determines what the correct values of the bitlines are, we need to reverse the direction and raise each bitline to its original value. This is another reason why DRAM is slower than SRAM. Note that we cannot rely on the processor to refresh every row of the memory. Because with cache some rows that are most often accessed by the processor are actually those that don't get refreshed this way since the data are kept in cache. Instead there is a *refresh row counter* which keeps track of the row that needs refresh, and the refresh period per row T/n is usually well under a second. Also note that we need to pause read/write when a refresh is going on.
Since a wordline activates a whole row of the memory, to write to the memory we need to read the values of the other bitlines, and change the bit of interest in the row buffer. After that the usual refresh step is carried out. Thus the write operation is also read-then-write.

15.6 Memory Refresh Quiz

Suppose a memory has 4096 rows and the refresh period is 500μs. The read timing is as follows

1. 4ns to select a row
2. 10ns for sense amplifier to get bit values
3. 2ns to put data in row buffer
4. 4ns for column decoder to pick the right bit
5. 11ns to write data from sense amplifier to memory row

Since step 5 can overlap with step 3 & 4, one read would take $4 + 10 + 11 = 25$ns, which means that the memory can potentially make 40 million reads per second. However among these 40 million reads there are $1/500\mu \times 4096 = 8.192$ million reads that need to be devoted to refresh. Hence the memory can only support $40 - 8.192 = 31.808$ million reads per second.

15.7 Fast-Page Mode

Since the row buffer contains the bits of an entire row of the memory, if we want to read/write a bit from/to the same row, we can just change the column address and read from/write to the row buffer. This row buffer shortcut is called **fast-page mode**.
The fast-page mode consists of the following steps

1. open up a *page* (an alias for an entire row of the memory) by selecting a row according to a row address, sensing and amplifying the bitline values, and latching the data into the row buffer

2. read from and/or write to the page

3. close the page by writing data from the row buffer to the memory row

15.8 DRAM Access Scheduling Quiz

Suppose the DRAM has 32 1-bit arrays. Each array is 16MB organized into 2^{12} rows $\times$ 2^{12} columns, with the upper 12 bits being the row address and the lower 12 bits the column address. Suppose we have cache miss for 7 addresses: 0xF00F00, 0xE00F00, 0xF00E04, 0xE04F00, 0xE00E00, 0xF00123, and 0x123F00. If it takes 10ns to open a page, 2ns to read from a page, and 5ns to close a page, then by scheduling the three reads from page F00 together and the two reads from page E00 together we can save $17 \times 7 - [(10 + 2 \times 3 + 5) + (10 + 2 \times 2 + 5) + 17 \times 2] = 45$ns.

15.9 Connecting DRAM to the Processor

The misses and write-back requests from the L3 cache are made over an external connection through a front-side bus to another chip called memory controller, which has memory channels connecting to many DRAMs. Thus the memory latency consists of not only the time to open, read, and close a page, but also the processing time of the memory controller, and the request communication and data transfer time through the front-side bus and the memory channels. Recent processor chips integrate the memory controller so as to eliminate the time spent on the front-side bus. But because the processor chip communicates directly with DRAMs, DRAMs need to be highly standardized.

16 Storage

16.1 Storage

Storage not only stores files (programs, data, settings etc.). In fact virtual memory is implemented by using storage. Some of the virtual pages are on disk. For both of these uses of storage the performance is mainly gauged with throughput and latency. Storage throughput is improving but not as quickly as processor speed. Storage latency is improving as well but very slowly, even slower than DRAM.
In addition to performance we are also concerned with reliability. Because if a processor failed we can simply reboot the system, but if a disk failed we lose all the data on it. Therefore we are more worried about the reliability of our storage than we are about most of the other elements of computer system. The types of storage are very diverse, which include magnetic disks (the traditional hard drives), optical disks, tapes, flash drives etc.

16.2 Magnetic Disks

Hard disk is magnetic disk. In fact the obsolete floppy disk is also magnetic disk. A magnetic disk has a spindle to which we can attach so-called *platters*. All platters are attached to the same spindle and rotate at the same speed. Data bits are stored on both sides of a platter and accessed with a magnetic head attached to an arm. All the magnetic heads are attached to a head assembly which moves all the heads in unison. Usually it is the platters that are rotating, the heads are staying in place. Each head accesses a circle on the platter surface called a *track*. All the tracks with the same distance from the spindle form a *cylinder*. The way we access different tracks is by moving the heads so that they become closer to or further away from the spindle. We don't store data continuously on a track. Instead a track is divided into *sectors* which will be the smallest unit that can be read. Each sector has a preamble that tells a head that it is the beginning of a sector, which is followed by the stored data and a check

sum and other information needed to correct errors.
Based on the structure mentioned above, disk capacity is computed as

$$\begin{aligned}\text{disk capacity} = 2 &\times \text{number of platters} \times \text{number of tracks per surface} \\ &\times \text{number of sectors per track} \times \text{number of bytes per sector}\end{aligned}$$

Usually we have only a few platters, but thousands of tracks per surface, tens to hundreds of sectors per track, and $\sim$ 1KB per sector.

16.3 Access Time for Magnetic Disks

Assume the disk is already spinning. If not then it takes several seconds to spin up the disk. Thus most of the time the disk would just keep spinning. If the disk is spinning, then the access time consists of

seek time: the time it takes to move the head assembly to the correct cylinder

rotational latency: the time it takes to wait for the right sector to be rotated into under the head

data read time: the time it takes to read until the end of the sector

controller time: the time it takes for the disk controller to check the checksum etc. is OK

I/O bus time: the time it takes to get the data from the disk controller

Note that unlike DRAM or cache where we can have multiple accesses in parallel, a magnetic disk only allows sequential access. Therefore the access time should also include queuing delay.

16.4 Disk Access Time Quiz

Suppose the disk has 1000 cylinders and 10 sectors per track, the head assembly is initially at Cylinder #0 and it moves at a speed of 10μs per cylinder, and the disk rotates 100 times per second. Ignoring the controller time and bus time, if there is no previous access request then the average access time is 11ms, which can be computed as follows

average seek time: 10μs $\times$ half of 1000 cylinders = 5ms

average rotational latency: 1/100s $\times$ half rotation = 5ms

data read time: 1/100s $\div$ 10 sectors = 1ms

16.5 Trends for Magnetic Disks

Capacity is doubled every 1 to 2 years. Seek time remains 5–10ms with very slow improvement through shrinking disk size. Rotation latency is reduced by increasing the speed of rotation from 5000rpm to 15000rpm. Controller and bus are improving at OK rate. Overall the bottleneck lies in the seek time and to a less degree rotation latency, and their improvements are not subject to Moore's law. Because smaller seek time requires better motion mechanics and rotation latency requires not only better motion mechanics but also better material for platters.

16.6 Optical Disks

Optical disk is similar to hard disk in that it also has a platter that rotates and we store data on the tracks of the platter. The difference is that data is read through the reflection of laser from the disk, thus the laser head does not need to be close to the disk and smudges or dusts are less of a problem. As a consequence optical disks do not need to be enclosed and are mostly portable. But the portability requires standardization which slows down the rollout of new products.

16.7 Magnetic Tape

Magnetic tape usually serves as secondary storage for backup. It has large capacity and is replaceable. But it allows only sequential access. It is gradually dying out due to low production volume thus high cost and cheaper backup alternatives such as USB drive.

16.8 Using RAM for Storage

Although the latency of DRAM is about 100000 times better than disk, the reason why we normally do not use memory for storage is that disk is about 100 times cheaper than memory. To combine the advantages of both people invented solid-state disk or SSD which isn't really a disk at all. One approach to build a SSD is to use DRAM + battery. It is faster and more reliable but more expensive than disk, and it is not good for backup because the battery would eventually run out. Another approach is flash memory which also uses transistors to store data. It consumes much lower power than disk although it is slower than DRAM and has a smaller capacity. Its advantage over DRAM + battery is that it keeps data live without power.

16.9 Hybrid Magnetic Flash

One popular approach to combine the advantages of magnetic disk and flash is to use flash effectively as cache for disk—most data is on the disk but the data that is frequently accessed is on the flash.

16.10 Flash vs. Disk vs. Both Quiz

Suppose a user plays a game for 2 hours which issues 2GB reads of data and writes another 10MB of data. After that the user switches to watching a movie for 2 hours which issues 1GB reads sequentially. Suppose the user repeats these two for 4 times

- for a disk with 100MB/s sequential access and 1MB/s random access, the total access time is (2000+1000)/100 + 10/1 = 40s each time or 160s total
- for a flash with 1GB/s access, the total access time is (2+1+0.01)/1 = 3.01s each time or 12.04s total
- for the disk in combination of the 4GB flash, the total access time is $40 \times 1 + 3.01 \times 4 = 52.04$s ($\Rightarrow$ 1 read for storing into flash thus altogether 4 reads for flash)

16.11 Connecting I/O Devices

We usually connect I/O devices through standardized I/O bus, which has limited rate of improvement due to the need of standardization. Instead of having only one type of bus typically we have a hierarchy of buses. We have a mezzanine bus such as PCI express that directly connects to fast devices such as processors and graphics. Then we have SATA or SCSI connected to PCI express which connects to disks. The purpose of this indirect device connection to PCI express through SATA is to leverage the slowly changing SATA standards. Another common indirect connection to PCI express is through USB hub, which is even slower than SATA but has longer-lived standards.

17 Fault Tolerance

17.1 Dependability

Dependability is a quality of delivered service that justifies relying on that system to provide that service. It concerns about whether the delivered or actual service matches the specified or expected service. A computer system has components called **modules**, each of which has a specified service.

17.2 Faults, Errors, and Failures

fault: some module deviates from specified behavior

error: actual behavior within the system differs from specified behavior

failure: the system deviates from specified behavior

17.3 Fault, Error, and Failure Example

fault: programming mistake e.g. 5 + 3 = 7, also called latent error

error: R1 = add(5,3), also called activated fault

failure: schedule a meeting 3 hours later at 7am

Note that not every fault becomes an error (need execution/activation), and not every error becomes a failure (if (add(5,3) > 6)).

17.4 Reliability, Availability

A system is always in one of the following two states: service accomplishment and service interruption. Measuring how long a continuous service accomplishment state will last until next failure, **reliability** can be quantified by mean time to failure or MTTF. Measuring what percentage of time the system is in service accomplishment state, **availability** can be quantified by

$$\text{availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

where MTTR stands for mean time to repair.

17.5 Kinds of Faults

We can classify faults by cause

hardware fault: hardware fails to perform as designed

design fault: software bugs, hardware design mistakes (Pentium FDIV bug)

operation fault: operator or user mistake (accidental shutdown)

environmental fault: fire, power outage, sabotage

We can also classify faults by duration

permanent: can no longer be corrected (disassemble a processor but can't reassemble it back)

intermittent: last for a while but recurring (overclock)

transient: last for a while and then disappear

17.6 Fault Classification Quiz

Design faults are typically permanent.

17.7 Improving Reliability and Availability

fault avoidance: prevent faults from occurring (e.g. no coffee in server room)

fault tolerance: prevent faults from becoming failures (e.g. error correcting code redundancy)

speed up repair: only improves availability (e.g. keep spare hard disk in drawer)

17.8 Fault Tolerance Techniques

checkpointing: (recover) save state periodically and restore state upon error detection, works well for many transient and intermittent faults, it can't take too long otherwise it would constitute a service interruption

2-way redundancy: (detect) two modules do the same work, compare and roll back if there is difference

3-way redundancy : (detect + recover) three modules do the same work, the majority vote is the correct result, expensive but can tolerate any fault in a module

17.9 N-Module Redundancy

Space shuttle uses 5-module redundancy. If there is one wrong result in a vote we can still resume normal operation. However if there are two wrong results in a vote then we have to abort the mission.

17.10 Fault Tolerance for Memory and Storage

2-way redundancy or 3-way redundancy is normally used for computational modules such as processors. They are overkill if applied to memory or storage because there are cheaper alternatives—error detection and/or error correction code (ECC). In ascending complexity the commonly used error detection/correction code are

parity: add one extra bit to the data bit which can be computed by XOR of all data bits, faults of 1-bit flip can be detected

ECC: a typical example is single error correction double error detection or SECDED, which can fix any 1-bit flip and detect any 2-bit flip

Reed-Solomon: used by hard drives to detect and fix multi-bit flips especially streaks of flipped bits

RAID: our focus and covered below

17.11 RAID

RAID stands for redundant array of independent disks. In RAID there are several disks that play the role of one disk so that they together appear to be a larger disk, a more reliable disk, or a larger and more reliable disk. The goal of RAID is to deliver better performance and normal read/write service accomplishment even when there is a bad sector or a disk failure that error correction code cannot fix (i.e. reliability). Not all RAID techniques will achieve both of these two goals, and they are numbered as RAID 0, RAID 1, etc.

17.12 RAID 0

RAID 0 uses a technique called *striping* to improve performance, which makes two disk behave like one disk. The tracks of one disk are stripe 0, 2, 4, ... of the combined disk while the tracks of the other disk are stripe 1, 3, 5, ... of the combined disk, which are named stripes to distinguish them from the tracks of physical disk. This improves the throughput by almost 2 fold resulting in less queuing delay, because we can read from both disks simultaneously.
However the reliability of RAID 0 is worse than one disk, because the failure rate of N disks is N times that of a single disk $f_N = N \cdot f_1$. Thus the MTTF, which is the reciprocal of failure rate, of N disks is $1/N$ of that of a single disk $\text{MTTF}_N = \text{MTTF}_1/N$.

17.13 RAID 1

RAID 1 uses a technique called *mirroring* to improve both performance and reliability, which saves the same data on both disks thus can detect and correct any fault that affects one of the disks. Since write can be done simultaneously, RAID 1 has the same write performance as a single disk; since read can be done on any disk, the throughput of RAID 1 is twice as that of a single disk.
As with RAID 0, the failure rate of two disks is twice that of a single disk $f_2 = 2f_1$ thus $\text{MTTF}_2 = \text{MTTF}_1/2$. But with one disk failure we are still covered by the other disk, thus the mean time to data loss or MTTDL is the sum of MTTF_2 and MTTF_1. In fact the reliability of RAID 1 is even better than this, because we can repair the failed disk. Suppose MTTR_1 is much smaller than MTTF_1 which is almost always the case, the probability of the second disk failure during the repair of the first disk is the ratio $\text{MTTR}_1/\text{MTTF}_1$. As a result MTTDL with failed disk repair is given by (⇒ it is better to comprehend the multiplication by thinking about failure rate)

$$\text{MTTDL} = \frac{\text{MTTF}_1}{2} \times \frac{\text{MTTF}_1}{\text{MTTR}_1} \gg \frac{\text{MTTF}_1}{2}$$

17.14 RAID 1 Quiz

Assuming equal number of reads and writes, with a 10MB/s throughput per disk the throughput of a two-disk RAID 1 is given by

$$\underbrace{\left(\frac{1}{3}\text{s}\right) \times (10 \times 2)\text{MB/s}}_{\text{read}} + \underbrace{\left(\frac{2}{3}\text{s}\right) \times 10\text{MB/s}}_{\text{write}} = \frac{20}{3}\text{MB} + \frac{20}{3}\text{MB} = \frac{40}{3}\text{MB}$$

17.15 RAID 4

RAID 4 uses a technique called *block-interleaved parity* to improve both performance and reliability, in which $N-1$ disks contain the data in the stripe form as in RAID 0 and one disk contains the parity blocks. For example if $N = 4$ then the first disk contains stripe 0, stripe 3, stripe 6 etc. and the fourth disk contains stripe 0 $\oplus$ stripe 1 $\oplus$ strip 2, stripe 3 $\oplus$ stripe 4 $\oplus$ strip 5, etc. With parity we can recover an error at a cost of $1/N$ capacity instead of $1/2$ capacity as in RAID 1.
Since to write to RAID 4 we need to read the old parity, update it, and write the new parity back to the parity disk in addition to writing to a data disk, although the read throughput of an N-disk RAID 4 is on average that of $N-1$ disks, its write throughput is only $1/2$ of that of a single disk. This is a significant disadvantage of RAID 4 and the primary motivation behind developing RAID 5.
Similar to RAID 1, the MTTF of an N-disk RAID 4 without repair is (⇒ two disk failures will not change the parity)

$$\text{MTTF}_N \text{ without repair} = \frac{\text{MTTF}_1}{N} + \frac{\text{MTTF}_1}{N-1}$$

while the MTTF of an N-disk RAID 4 with repair is

$$\text{MTTF}_N \text{ with repair} = \frac{\text{MTTF}_1}{N} \times \frac{\text{MTTF}_1}{(N-1)\text{MTTR}_1}$$

17.16 RAID 4 Write

To compute the new parity, instead of reading all data disks RAID 4 XORs the new data with the old data first to figure out which bits have changed, and then XORs this change with the old parity to produce the new parity, thus altogether 2 reads (old data and old parity) and 2 writes (new data and new parity). Since the parity disk is involved in every write, it is the bottleneck of RAID 4 and is the target for improvement of RAID 5.

17.17 RAID 4 Quiz

Assuming equal number of reads and writes, with a 10MB/s throughput per disk the throughput of a 5-disk RAID 4 is given by

$$\underbrace{\left(\frac{1}{9}\text{s}\right) \times (10 \times 4)\text{MB/s}}_{\text{read}} + \underbrace{\left(\frac{8}{9}\text{s}\right) \times \left(10 \times \frac{1}{2}\right)\text{MB/s}}_{\text{write}} = \frac{20}{3}\text{MB} + \frac{20}{3}\text{MB} = \frac{40}{3}\text{MB}$$

17.18 Parity Quiz

The approach of having extra DRAM arrays for parity protection is preferred over adding extra bits in each DRAM array for parity protection for two reasons. First we can use the same array design for both unprotected and protected memories. Second coarser grain redundancy allows the detection of local errors that may not be detected with local parity bits (e.g. all zeros).

17.19 RAID 5

RAID 5 uses a technique called *distributed block-interleaved parity* to improve both performance and reliability, which does block-interleaved parity just like RAID 4, but its parity is distributed among all disks. For example instead of storing stripe 0 $\oplus$ stripe 1 $\oplus$ strip 2, stripe 3 $\oplus$ stripe 4 $\oplus$ strip 5, etc. on the fourth disk, stripe 3 $\oplus$ stripe 4 $\oplus$ strip 5 is stored in the first disk, stripe 6 $\oplus$ stripe 7 $\oplus$ strip 8 is stored in the second disk and so on. As a result any disk plays the role of a data risk for some stripes and plays the role of the parity disk for other stripes, thus the read throughput of an N-disk RAID 5 is on average N times that of a single disk. In addition since the 2 reads and 2 writes can be distributed among all disks as well, its write throughput is $N/4$ times that of a single disk.
The reliability of RAID 5 is the same as that of RAID 4.

17.20 RAID 5 Quiz

Identical to RAID 4, one disk capacity needs to be sacrificed for storing parity.
Assuming equal number of reads and writes, with a 10MB/s throughput per disk the throughput of a 5-disk RAID 5 is given by

$$\underbrace{\left(\frac{1}{5}\text{s}\right) \times (10 \times 5)\text{MB/s}}_{\text{read}} + \underbrace{\left(\frac{4}{5}\text{s}\right) \times \left(10 \times \frac{5}{4}\right)\text{MB/s}}_{\text{write}} = 10\text{MB} + 10\text{MB} = 20\text{MB}$$

17.21 RAID 6

Instead of one check block RAID 6 has two check blocks, one parity block for single-disk failures and a different type of check block for two-disk failures. Thus compared to RAID 5, the protection overhead of RAID 6 is doubled and 6 instead of 4 accesses are required for each write. Hence RAID 6 is useful only when the chance of second disk failure during the repair of the first disk is high, which is not the case when disk failures are independent (one example of dependent failure is replacing the wrong risk).

18 Multi-Processing

18.1 Flynn's Taxonomy of Parallel Machines

type	instruction stream	data stream	example
SISD	1	1	uniprocessor
SIMD	1	> 1	vector, SSE/MMX for multimedia
MISD	> 1	1	stream processor? rare
MIMD	> 1	> 1	multiprocessor

18.2 Why Multiprocessors

On the one hand uniprocessor nowadays is already 4-wide, and we will get diminishing return from making it even wider (as the portion of execution time that can be accelerated through parallelism is already very small, ref. Amdahl's law). What is worse to make uniprocessor faster by increasing clock frequency we will need higher voltage and power consumption. Therefore to benefit from Moore's law we have to double the number of cores every 18 to 24 months.

18.3 Multicore vs. Single Core Quiz

Suppose a better uniprocessor consumes 75W instead of 100W at 2GHz clock frequency with an IPC of 3.5 instead of 2.5. Since power supply voltage is proportional to clock frequency, to make it work at 100W we can increase clock frequency by $2 \times (100/75)^{1/3} = 2.2$GHz. Thus the speedup is the product of 3.5/2.5 and 2.2/2 which is 1.54. In contrast using two of the original uniprocessors we can get a speedup of 2.

18.4 Multiprocessor Needs Parallel Programs

The disadvantages of going from uniprocessor to multiprocessor are

- sequential/single-threaded code is a lot easier to develop
- debugging parallel code is much more difficult
- performance scaling is very hard to achieve for parallel programs, usually the performance levels off as the number of cores increases further

18.5 Centralized Shared Memory

In this type of multiprocessors all cores have their own caches, but all of them are connected to the same bus so that they can access the same main memory and I/O devices. This type of multiprocessors is what today multicore processors look like. It is also called UMA which stands for *uniform memory access* (time) because memory is at the same distance from all cores, or SMP which stands for *symmetric multiprocessor* because any core is identical to any other core.

18.6 Multicore Quiz

To increase the number of cores the centralized shared memory architecture needs to deal with the following problems

- memory needs to be large for multiple accesses thus is necessarily slow
- memory may receive too many accesses per second

18.7 Centralized Main Memory Problems

memory size: need large memory to accommodate many accesses, but the larger the memory the further away it is from cores thus slower access

memory bandwidth: requests from many cores can lead to memory bandwidth contention

As a result centralized shared memory architecture works well only for small machines (2,4,8,16 cores).

18.8 Distributed Memory

In the distributed memory or nonuniform memory access (NUMA) architecture each core can access its own memory while others cannot. A network interface card connects each core to a network. If a core wants to access data in another core's memory, it needs to create a network message using a send primitive in the operating system to send a request. Thus communication between cores is explicit. This architecture is also called multi-computer or computer cluster because each core has the basic components of a computer, and a program is written as if the cores are independent machines communicating over a network. This architecture tends to scale to a large number of cores, because it forces programmers to explicitly think about minimizing communication through maximizing local accesses.

18.9 NUMA Memory Allocation Quiz

In the distributed memory architecture the operating system should put not only the stack of core N in the memory slice N but also all data pages mostly accessed by core N in the memory slice N.

18.10 A Message Passing Program

```
double myArray[ArraySize/NumProc];
double mySum = 0;
for (int i = 0; i <ArraySize/NumProc; i++)
    mySum += myArray[i];
if (myPID = 0){
    for (int p = 1; p < NumProc; p++){
        int pSum;
        recv(p, pSum);
        mySum += pSum;
    }
}
else
    send(0, mySum);
```

18.11 A Shared Memory Program

```
shared double allArray[ArraySize];
shared double allSum = 0;
shared mutex sumLock;
shared int barrier;
double mySum = 0;
for (int i =myPID*ArraySize/NumProc; i <(myPID+1)*ArraySize/NumProc; i++)
    mySum += allArray[i];
lock(sumLock);
allSum += mySum;
unlock(sumLock);
barrier;
```

18.12 Message Passing vs. Shared Memory

	message passing	shared memory
communication	programmer	automatic
data distribution	programmer	automatic
hardware support	simple network	extensive
code correctness	difficult	less difficult
code performance	difficult	very difficult

For message passing once you get correctness performance is not far away. However for shared memory correctness is not even half way done in terms of performance.

18.13 Message Passing vs. Shared Memory Quiz

While message passing requires extra code for data distribution, shared memory requires extra code for synchronization.

18.14 Shared Memory Hardware

We have on one extreme multiple cores sharing the same physical address space (e.g. UMA, NUMA) and on the other extreme multithreading by time sharing a single core. What is in between is hardware multithreading in a core, which can be

coarse-grain: change thread every few cycles

fine-grain: change thread every cycle

simultaneous multithreading (SMT): also called hyperthreading, execute instructions belonging to different threads in any cycle

The hardware support becomes more extensive as we move from coarse-grain to fine-grain to SMT.

18.15 Multithreading Performance

Without multithreading much of the time we are running a less than fully utilized processor pipeline, and the operating system may need to spend many cycles to figure out which thread is ready to run next upon the interruption of the current thread. The switching overhead may outweigh the benefit of having multiple threads runnable.
A chip multiprocessor may assign a core to each thread so that there is no switching overhead, but the cost is doubled if it uses two cores for two threads.
With one core and fine-grain mulithreading support (separate sets of registers for the threads and scheduling logic), we can fill in periods of idleness in one thread with instructions from another thread. The hardware cost is very small, because only the fetch stage needs to be slightly complicated and we need extra sets of registers for the extra threads.
With SMT we can populate the unused issue slots in one thread (say 2 out of the 4-wide issue) with instructions from another thread. The hardware cost is slightly higher than fine-grain threading but is still small, because the processor scheduler stays the same, only the scheduling and the commit need to figure out which instruction comes from which thread.

18.16 SMT vs. Dual Core Quiz

Suppose the SMT has one floating-point intensive thread and one integer-only thread, and the core is a 4-issue that can execute two floating-point and two integer instructions per cycle, so that the SMT can make full use of all 4 issue slots per cycle. In contrast one core of the equivalent dual-core becomes effectively 2-issue because it sees mostly floating-point instructions while the other also becomes effectively 2-issue because it never sees floating-point instructions. In this case we get more performance per dollar cost from the SMT.
However if the SMT has two integer-only threads, then the equivalent dual-core would be slightly more cost effective. Because it delivers the 2-issue performance in each core for twice the cost of one core, whereas the SMT delivers the performance of a single core as both threads are sharing the 2-issue slots (cannot use the floating-point slots) for a cost that is more than one core. Hence SMT does not always wins over multiple cores, which depends on what type of program we have.

18.17 SMT Hardware Changes

The SMT hardware changes are depicted in Figure 3 below. We need extra PC, RAT, and architectural

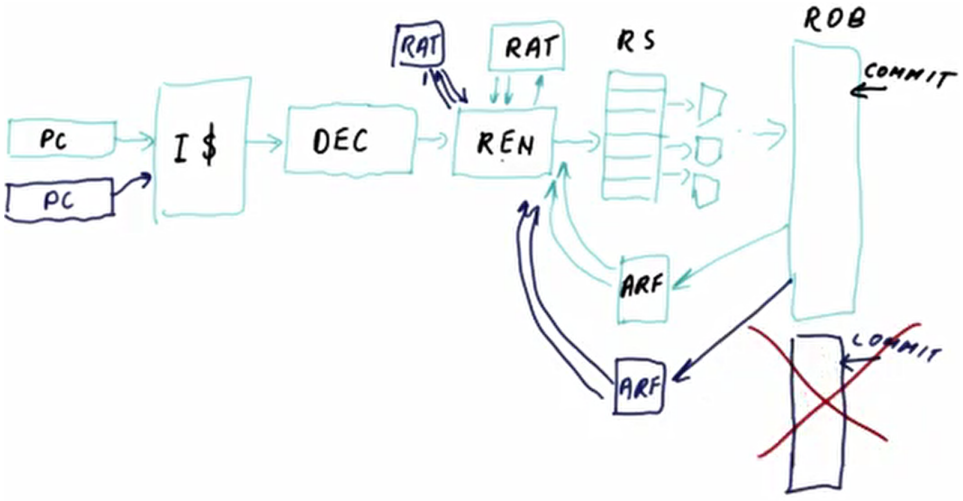

Figure 3: SMT hardware changes

register file (ARF) for the extra thread, but we don't need to modify the decode stage. The renamer, the reservation stations, and the execution units are simply shared by the threads. We can have extra ROB for the extra thread. But since ROB is large and complex we instead let the two threads share the ROB putting their instructions in interleaved order. This is less optimal but still works well. Hence the cost of two-threaded SMT is not twice the cost of uniprocessor. Note that SMT usually refers to the execution stage and most modern SMT processors use fine-grain mulithreading for the fetch stage.

18.18 SMT, Data Cache, TLB

For multithreading the problem with virtually indexed virtually tagged cache is that, the two threads may have different physical address spaces but share the same virtual addresses and the cache doesn't know which thread the virtual address it receives from. To avoid this aliasing problem coarse-grain multithreading flushes the cache before switching, which however is not an option for SMT. Hence for SMT we use virtually indexed physically tagged cache to avoid aliasing, in which the physical tag that is retrieved from the data cache is compared with those from the TLB. This requires the TLB to match not only the page number in the virtual address but also figure out which thread it is from. It can be done by adding a single bit to each TLB entry to store the thread ID.

18.19 SMT and Cache Performance

The cache of the core is shared by all SMT threads. This brings the following benefit and pitfall

benefit: fast data sharing—when one thread stores some variable and the other subsequently loads the same variable it is a cache hit, in general any communication between the threads that happens within a short span will be a cache hit

pitfall: since the cache capacity is shared by the threads, if the combined size (without overlap double counting) of the working set of one thread and that of the other exceeds the capacity then it is a cache miss, even though the size of individual working sets can fit the cache, this is called **cache thrashing** which would lead to much worse performance than single-threading

19 Cache Coherence

19.1 Cache Coherence Problem

Because a single large L1 cache for all cores would be too slow, we have private L1 cache for each core. This results in the possibility of **incoherence** in which different cores see different values at the same memory address.

19.2 Coherence Definition

There are three requirements for coherence

1. a read from an address by a core returns the value written by the most recent write to the same address by the same core if there is no other core that has written to the same address in between, i.e. if one core is operating then its read should get the most recent write, thus coherent behavior includes uniprocessor behavior as long as only one core is operating

2. if one core writes to an address and another reads from the same address after a sufficient time and there is no other write in between, then the read returns the value from the write

3. writes to the same address are serialized, that is any two writes to the same address must be seen to occur in the same order by all cores

19.3 Coherence Definition Quiz

Required by coherence both core see the same last write. Thus no matter which write they see, one of the cores would be released from its while loop which will in turn release the other from its while loop. As a result we get both prints.

19.4 How to Get Coherence

To achieve coherence property #2 we can

write-update: broadcast writes to update other caches

write-invalidate: writes prevent hits to other caches

To achieve coherence property #3 we can

snooping: all writes are broadcast on the shared bus, the order in which they appear on the bus is the order seen by all the cores

directory: each block is assigned an ordering point, different ordering points can be used for different blocks so that there is no contention, the ordering point is called directory

Thus altogether we have four possible combinations.

19.5 Write-Update Snooping Coherence

The serialization of the common broadcast bus guarantees that there is only one write at a time thus all writes are ordered (arbitration may be required in case there are simultaneous writes), while snooping the common bus ensures that all cores see every write so that they can update their private caches accordingly.

19.6 Write-Update Optimization #1: Memory Writes

The first optimization is avoiding memory writes and reads through *dirty bit*. Because memory is even slower than broadcast bus, it is the bottleneck of the write-update protocol. The solution to reducing memory traffic is the same as in uniprocessor—add a dirty bit to each block in each cache. With dirty bit write to memory occurs only when the dirty block is replaced, and read from memory occurs only when no cache has the block in dirty state.
There is one and only one block in dirty state. If one cache has the block in dirty state but another writes to it, then the dirty bit of the first cache is set to 0 while the dirty bit of the second is set to 1. In addition to updating memory upon replacement and snooping writes on the common bus, the cache with the block in dirty state is also responsible for snooping reads and responding before memory responds.

19.7 Write-Update Optimization #2: Bus Writes

With the addition of dirty bit memory traffic is significantly reduced and the common bus becomes the bottleneck. To reduce bus traffic we can add a *shared bit* which tells us whether the block is shared with other cache(s) or not. If it is shared then we do the usual write-update protocol. But if it is not shared then we can avoid the bus write.
Shared bit is updated through snooping. Each cache snoops reads and writes on the bus. If any one of them has the same block that is read or written, it then pulls the shared line on the bus to 1. This updates the shared bit of both the reading/writing cache and the snooping cache(s) to 1.

19.8 Write-Update Optimization Quiz

Suppose the two cores are executing a program where core 0 writes to A and then core 1 reads it and this repeats for 1000 times. After that core 0 replaces A and then core 1 also replaces it.

without optimization: the number of bus uses is 1000 + 1 where +1 is due to the read miss of core 1 (the rest 999 will be taken care by the update from core 0), while the number of memory writes is 1000

with dirty bit optimization: the number of bus uses is 1000 + 1 + 1 where the first +1 is due to the read miss of core 1 and the second +1 is due to the replacement in core 0, while the number of memory writes is 1

with dirty bit + shared bit optimization: the number of bus uses is still 1000 + 1 + 1 because the block is shared, while the number of memory writes is 1

19.9 Write-Invalidate Snooping Coherence

In the write-invalidate protocol, whenever a cache snooped a write on its block, instead of updating the value it simply changes the valid bit of the block to 0. At the same time the writing cache changes the shared bit of the block it writes to 0 because all other copies of the block will be invalidated. When this writing cache snooped a read of the block, it will respond with the data and update the shared bit to 1. Hence after every write a block becomes unshared, while after every read on bus a block becomes shared.
For blocks that are accessed by only one core, we have the same behavior for both write-update and write-invalidate. For blocks that are shared, write-update tends to result in more hits but does more broadcasts, whereas write-invalidate generates a miss on all readers after a write occurred but allows local write after the first write.

19.10 Write-Update vs. Write-Invalidate Quiz 1

Suppose the two cores are executing a program where core 0 writes to A and then core 1 reads it and this repeats for 1000 times. After that core 0 replaces A and then core 1 also replaces it.

write-update: the number of bus uses is 1000 + 1 where +1 is due to the read miss of core 1 (the rest 999 will be taken care by the update from core 0)

write-invalidate: the number of bus uses is 1000 + 1000 because every write of core 0 invalidates the block in core 1 resulting in cache miss

Hence write-update is much more efficient than write-invalidate when the pattern is one core producing data while the other consuming the data.

19.11 Write-Update vs. Write-Invalidate Quiz 2

Suppose the two cores are executing a program where core 0 reads A and then writes to A and repeats this for 500 times, after that core 1 reads A and then writes to A and repeats for 500 times.

write-update: the number of bus uses is 1 + 1 + 500 where the two +1's are due to the read miss of both cores while the 500 is due to the update by core 1 to the shared block in core 0 even though core 0 no longer needs it

write-invalidate: the number of bus uses is 1 + 1 + 1 where the first two +1's are due to the read miss of both cores while the last +1 is due to the invalidation by core 1

Hence write-invalidate is much more efficient than write-update when the pattern is one core using data first and then the other using it.

19.12 Update vs. Invalidate Coherence

If the application has a burst of writes to one address

write-update: each write sends an update, which is bad due to the bus contention created and power consumed

write-invalidate: first write invalidates and the remaining are hits, which is good

If the application writes different words in the same block

write-update: update is sent for each word, which is bad because one cache line worth of writes would result in many updates

write-invalidate: first write invalidates and the remaining are hits, which is good

If we have the producer-consumer pattern in which one core keeps producing data and the other keeps consuming it

write-update: producer sends updates leading to consumer hits, which is good

write-invalidate: producer invalidates leading to consumer misses, which is bad

It turns out that all modern processors use write-invalidate, and the decisive winning context of write-invalidate is when a thread moves from one core to another. In this common scenario the write-update protocol will continue updating the old core's cache which is horrible, whereas in the write-invalidate protocol the first write invalidates the block in the old core and there is no traffic after the invalidation. We will focus on write-invalidate in the following.

19.13 MSI Coherence

MSI is a simple write-invalidate coherence protocol. In MSI a block can be in one of the following three states. The following flowchart summarizes the possible transitions among the three states.

invalid state: a block is in the invalid state either when it is present in the cache but its valid bit is set to 0 or when it is not present in the cache (valid bit = 0)

- a local read sends the block to the shared state + puts a read request on the bus
- a local write sends the block to the modified state + puts a write request on the bus

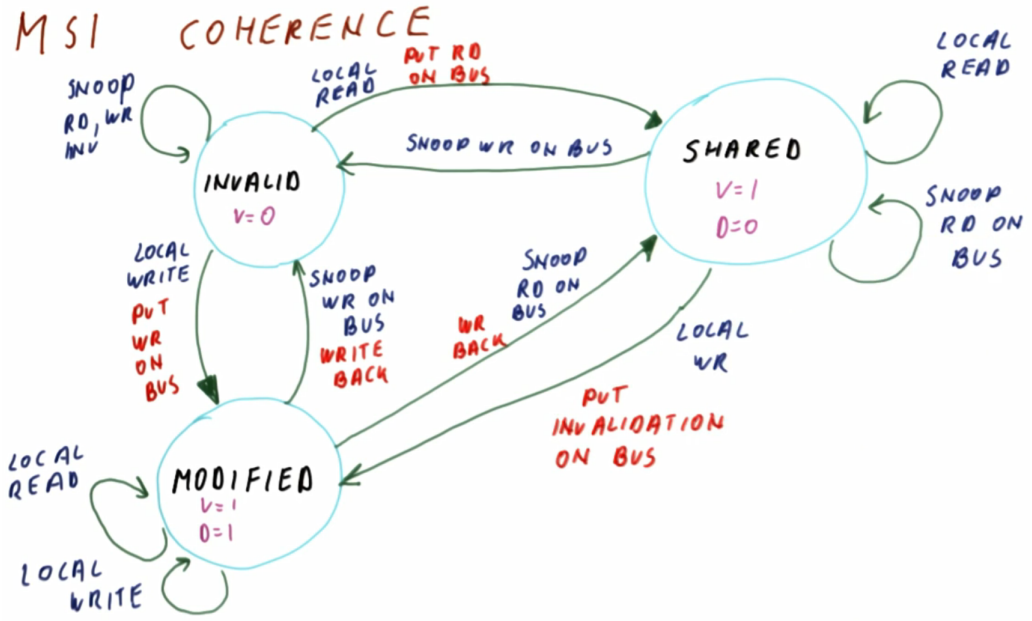

Figure 4: State transition flowchart of MSI coherence protocol

- a snooped read request on the bus does not do anything to the block
- a snooped write request on the bus does not do anything to the block

shared state: a block is in the shared state when it is present and valid in more than one cache (valid bit = 1, dirty bit = 0)

- a local read keeps the block in the shared state and does not do anything else
- a local write sends the block to the modified state + puts an invalidation request on the bus
- a snooped read request on the bus does not do anything to the block
- a snooped write request on the bus sends the block to the invalid state and nothing else

modified state: a block is in thet modified state when it is present and valid only in the cache and dirty (valid bit = 1, dirty bit = 1)

- a local read keeps the block in the modified state and does not do anything else
- a local write keeps the block in the modified state and does not do anything else
- a snooped write request on the bus sends the block to the invalid state + initiates a write-back of the block to the memory $\Rightarrow$ invalidation $\equiv$ replacement thus need write-back
- a snooped read request on the bus sends the block to the shared state + initiates a write-back of the block to the memory $\Rightarrow$ shared = clean thus need write-back

19.14 Cache-to-Cache Transfers

Suppose core 1 has block B in the modified state and core 2 puts a read request on the bus. There are two ways for core 1 to provide the data.

abort & retry: core 1 cancels the request of core 2 using some abort bus signal and then writes block B back to the memory, after that core 2 retries and gets the data from the memory
the disadvantage of this approach is that the read miss of core 2 has a latency that is twice the memory latency

intervention: core 1 tells the memory that it will supply the data using some intervention bus signal and then sends the data to core 2 which is also picked up by the memory (write-back is necessary because after the read request block B in both core 1 and core 2 will be in the shared state, i.e. the block is clean)
the disadvantage of this approach is that it needs more complex hardware support for the request intervention and the data pickup

Modern processors mostly use a variant of the intervention approach with fancier snooping protocols empowered by Moore's law.

19.15 MSI Quiz

Suppose initially block B is in the memory, then

core 1	core 2	state of B in core 1 cache	state of B in core 2 cache
read B		shared	invalid
	read B	shared	shared
write B		modified	invalid

Note that

- after the first read block B is not shared, but the shares state is a way of saying that the block is clean
- if a block is in the modified state in some cache, it is in the invalid state in any other cache; if a block is in the shared state in some cache, it is in either the invalid state or the shared state in any other cache

19.16 Avoiding Memory Writes on Cache-to-Cache Transfers

Since memory bandwidth is much smaller than cache bandwidth and memory reads and writes consume much more power than cache reads and writes, we want to avoid write-backs to memory as long as there is a cache that holds the most recent value of the block, and avoid memory reads as long as there is a cache that can provide the block. To achieve this we introduce another state called **owned state**. The cache in the owned state is responsible for providing data to other caches and eventually writing back to memory upon replacement.

19.17 MOSI Coherence

MOSI coherence behaves like MSI except that

modified state: a snooped read request on the bus sends the block to the owned state instead of the shared state, and the memory is not written back

owned state: similar to the shared state, except that it continues to provide the data like the modified state upon snooping a read request on the bus, and it writes the block back to the memory upon replacement

Essentially the owned state combines the property of the modified state (dirty) and that of the shared state (shared read access).

19.18 M(O)SI Inefficiency

Besides the memory access inefficiency that is solved by introducing the owned state, MSI (and MOSI) has another inefficiency that is related to thread-private data—data accessed by only one thread, e.g. data in single-threaded programs and stacks in multithreaded programs. The inefficiency occurs when we read the data and then write to it, which requires putting an invalidation request on the bus. In contrast in a uniprocessor after the read the write would be a cache hit and we only need to change the dirty bit to 1, which is much faster than invalidation. To avoid the slow invalidation for thread-private data we introduce another state called **exclusive state**.

19.19 The E State

The exclusive state is the combination of the property of the modified state (exclusive read/write access) and that of the shared state (clean) that is opposite to the owned state.

modified state: exclusive read/write access + dirty

shared state: shared read access + clean

owned state: shared read access + dirty

exclusive state: exclusive read/write access + clean

With the exclusive state we have cache hit instead of invalidation.

	MSI	MOSI	MESI/MOESI
read B	I → S with cache miss	I → S with cache miss	I → E with cache miss
write B	S → M with invalidation	S → M with invalidation	E → M with cache hit

19.20 MOESI Quiz

Suppose initially block B is in the memory, then

	state of B in core 0	state of B in core 1 cache	state of B in core 2 cache
core 0 read B	exclusive	invalid	invalid
core 1 read B	shared	shared	invalid
core 2 read B	shared	shared	shared
core 1 write B	invalid	modified	invalid

19.21 MESI vs. MOSI vs. MOESI Quiz

Suppose initially block B is in the memory, then we have

	MESI	MOSI	MOESI
mem reads	2	1	1
bus requests	5	6	5

because

c1 read B	c1: S for MOSI (mem+1, bus+1) or E for MESI/MOESI (mem+1, bus+1)
c1 write B	c1: M for MOSI (bus+1) or M for MESI/MOESI (none)
c2 read B	c2: S for all (bus+1), c1: O for MOSI/MOESI (none) or S for MESI (write-back)
c2 write B	c2: M for all (bus+1), c1: I for all (none)
c3 read B	c3: S for all (bus+1), c2: O for MOSI/MOESI (none) or S for MESI (write-back)
c1 read B	c1: S for all (bus+1, mem+1 for MESI), c2: O for MOSI/MOESI (none) or S for MESI (none)
c2 read B	c1, c3: S for all (none), c2: O for MOSI/MOESI (none) or S for MESI (none)

Thus the exclusive state saves one bus request, and the owned state saves one memory read.

19.22 Directory-Based Coherence

In snooping-based coherence we need to broadcast all requests to make them visible to all cores and establish ordering for all writes, thus we can have only one single bus to accommodate all requests and all accesses, which becomes a bottleneck. As a result snooping does not work well for more than 8 to 16 cores. The solution to this bottleneck is a non-broadcast network that makes requests visible and establishes write ordering. The structure that does these is called directory.

19.23 Directory

A **directory** is a distributed structure across cores instead of being centralized like the bus. Each slice of a directory operates independently serving a disjoint set of blocks thus together they provide a large bandwidth, where a slice is part of the directory that is next to a particular core. A slice has one entry for each block it serves. Each entry tracks which cache has the block in a non-invalid state. The order of access to a particular block is determined by the **home slice** of that block, which is the slice that has an entry for the block. Note that the cache states in the directory protocol are identical to those in the snooping protocol. The only difference is that requests and writes go through the directory instead of the bus.

19.24 Directory Entry

A directory entry has one dirty bit that indicates the block is or may be dirty, and one presence bit for each cache in the system that indicates whether the block is present and valid in that cache. Different blocks are distributed among different slices so that we get an even load among slices. We determine which is the home slice by looking at the address of the block. Upon receiving a read request the home slice gets the data from the memory or other cache, sends it to the requesting cache, tells the cache the state of the block (exclusive or shared), and updates the presence bit and the dirty bit (1 for exclusive state which may be dirty). Upon receiving a write request the home slice sends the write request for the block to the cache in which it is present, and upon receiving the invalidation confirmation from that cache updates the presence bit.

19.25 Directory MOESI Quiz

Suppose initially block B is in the memory, then the directory does (req = request, resp = respose)

c0 read B	req recv +1, resp sent +1 (data + E)	c0: E
c0 write B	none	c0: M
c1 read B	req recv +1, req fwd +1 (to c0), resp recv +1 (data), resp sent +1	c0: O, c1: S
c2 read B	req recv +1, resp sent +1 (look up mem)	c2: S
c3 read B	req recv +1, resp sent +1 (look up mem)	c3: S
c0 write B	req recv +1, req fwd +3 (to c1, c2, c3), resp recv +3, resp sent +1	c0: M, others: I

Note that we need a fancier directory to know how to ask the owned state to provide the data.

19.26 Cache Misses with Coherence

With coherence we have the fourth type of cache misses called **coherence miss** in addition to compulsory miss, conflict miss, and capacity miss, which happens when one core wants to read a block again but another core writes to it before the second read. It is so named because it would not be a miss

except for coherence.
There are in fact two types of coherence miss

true sharing; occurs when different cores access the same data (the example above)

false sharing: occurs when different cores access different data in the same block

19.27 False Sharing Quiz

Given word X, Y, Z, W belonging to the same block and the following executions occurring in sequence: c0 read X, c1 write X, c2 write W, c0 read X, there are three compulsory misses and one coherence miss which is of type true sharing rather than false sharing. Because X in c0 is invalidated by c1 write X instead of c2 write W.

20 Synchronization

20.1 Synchronization Example

The sections of code that need to be executed one at a time are called **atomic** or **critical sections**.

20.2 Synchronization Example: Lock

The type of synchronization we use for atomic sections is called mutual exclusion (mutex) or lock. This is implemented by inserting a lock before a critical section and an unlock after it. If the lock is open then we can enter the critical section; if the lock is locked then we will spin there until it becomes open. Once we leave the critical section we will unlock the lock. Note that mutex doesn't impose a particular order between threads. Whichever comes first will execute its critical section first.

20.3 Lock Variable Quiz

A lock is just another location in shared memory. It is a variable like any other variable and has a memory address.

20.4 Lock Synchronization

The following code that implements the lock function does not work

```
typedef int mutex_type;
void lock(mutex_type &lockvar){
    while(lockvar == 1);
    lockvar = 1;
}
```

because we need both the lock value checking loop and the lock value writing to be atomic.

20.5 Implementing Lock

One way is to use Lamport's bakery algorithm or some other algorithm which uses normal load and store instructions for lock variables. However they are complicated and slow. Another option is to use special atomic read/write instructions.

20.6 Atomic Instructions Quiz

Since a lock is just a variable in shared memory, to implement locks easily we need an instruction that both reads and writes the memory.

20.7 Atomic Instructions

There are three main types

atomic exchange: e.g. EXCH R1, 78(R2), which does both a load and a store at the same time, so that it swaps, the lock function can then be implemented as follows (lockvar = 0 means unlock)

```
R1 = 1;
while (R1 == 1)
    EXCH R1, lockvar;
```

the drawback of this approach is that it keeps writing to the memory location all the time even when the lock is locked by others, which is worse if we use write-invalidate for coherence

test-and-write: test a condition and write only when it is true, thus we don't write all the time and the iterative condition checking is on cached copy, the command TSET R1 Addr is defined as

```
if (mem[Addr] == 0){
    mem[Addr] = 1;
    R1 = 1;
} else
    R1 = 0;
```

load linked/store conditional: separate the load and store into two instructions thus avoid the need for multiple memory stage in pipelining (see below)

20.8 Test-and-Set Quiz

Using the TSET R1, Addr definition above, the lock function can be implemented as follows

```
lock(mutex_type &lockvar){
    R1 = 0;
    while(R1 == 0)
        TSET R1, lockvar;
}
```

20.9 Load Linked/Store Conditional

The motivation is that atomic read and write in the same instruction is bad for pipelining, because they cannot be completed in one access to memory. LL/SC separates the atomic read and write into two instructions

load linked: it behaves like a normal load except that it saves the address into a special link register

store conditional: it checks whether the address is the same as the one in the link register, if yes then it does a normal store and returns 1, otherwise it returns 0 without storing anything

20.10 How is LL/SC Atomic

We do LL R1, lockvar followed by SC R2, lockvar. The key step in between is that if we snooped a write to lockvar we put 0 in the link register. Thus LL/SC relies on coherence to achieve atomic operation. If the critical section involves only one variable, then we can implement LL/SC directly on that variable as follows

```
lock                          try:
var++;          ⟹                LL R1 var;
unlock                           R1 ++;
                                 SC R1, var;
                                 if (R1 == 0)
                                      go to try
```

20.11 LL/SC Lock Quiz

This critical section involves two variables thus there are two conditions that need to be checked.

```
void lock(mutex_type &lockvar){
    trylock:
        MOV R1, 1;
        LL R2, lockvar;
        SC R1, lockvar;
        BNEZ R2, trylock;
        BEQZ R1, trylock;
}
```

The link register is a hidden register. It can only be accessed implicitly through LL/SC.

20.12 Locks and Performance

With atomic exchange because the cores without the lock continue writing to lockvar, the cached copy of lockvar in those cores will keep on switching between the modified state and the invalid state. In addition the block containing lockvar will continue moving among those cores, causing high traffic on the common bus, which will slow down the cache miss handling of the core with the lock. Therefore this type of lock is not only power hungry but also slow.

20.13 Test-and-Atomic-OP Lock

One way to avoid the power consumption and useful work slowdown due to atomic exchange is to test whether the lock is free before the atomic write

```
R1 = 1;                          R1 = 1;
while (R1 == 1)          ⟹      while (R1 == 1){
    EXCH R1, lockvar;                while (lockvar == 1);
                                     EXCH R1, lockvar;
                                 }
```

Since the lockvar checking is a normal read, it will send the cached copy of lockvar in the cores without the lock to the shared state. After that the lockvar checking in those cores will be cache hits thus will no longer generate bus traffic.

20.14 Test-and-Atomic-OP Lock Quiz

Applying test-and-atomic-op to the LL/SC lock quiz we have

```
void lock(mutex_type &lockvar){
    trylock:
        MOV R1, 1;
        LL R2, lockvar;
        SC R1, lockvar;
        BNEZ R2, trylock;
        BEQZ R1, trylock;
}
```

$\Longrightarrow$

```
void lock(mutex_type &lockvar){
    trylock:
        MOV R1, 1;
        LL R2, lockvar;
        BNEZ R2, trylock;
        SC R1, lockvar;
        BEQZ R1, trylock;
}
```

The waiting for the lock to become free is done using the load link operation.

20.15 Unlock Quiz

The unlock function is called only by the thread that has the lock, thus we don't need to check whether the lock is busy and can just do a normal store to unlock it.

```
void unlock(mutex_type &lockvar){
    SW 0, lockvar
}
```

20.16 Barrier Synchronization

Barrier synchronization is used when there is a parallel section where several threads are doing the same thing independent of each other (e.g. summation). A **barrier** is a global wait to ensure that all threads have entered the barrier before any of them can proceed to pass the barrier. As such a barrier implementation requires two variables: a counter that counts the number of arrived threads and a flag that gets set when the counter reaches the total number of threads.

20.17 Simple Barrier Implementation Doesn't Work

The following simple barrier implementation can work only once

```
lock(counterlock);
if (count == 0)
    release = 0;
count++;
unlock(counterlock);
if (count == total){
    count = 0;
    release = 1;
} else
    spin(release == 1);
```

because the last thread can transcend the previous thread to leave and re-enter the barrier before the previous thread is released from the barrier. The consequence is a **deadlock**—the previous thread is waiting for the last thread to set release to 1, but the last thread is waiting for the previous thread to finish its job and re-enter the barrier.

20.18 Reusable Barrier

The idea of making a barrier reusable is that the value for releasing the barrier will not be the same for all instances of the barrier, even instances are released when release becomes 0 while odd instances are released when release becomes 1. To do this we flip the value of release instead of reinitializing it in each iteration.

```
localSense != localSense;
lock(counterlock);
count++;
if (count == total){
    count = 0;
    release = localSense;
}
unlock(counterlock);
spin(release == localSense);
```

21 Memory Consistency

21.1 Memory Consistency

While coherence defines the order of accesses to the same memory address which is necessary for data sharing, **consistency** defines the order of accesses to different memory addresses.

21.2 Consistency Matters

Suppose in the program order core 1 executes SW 1, D followed by SW 1, F and core 2 executes LW F, R1 followed by LW D, R2. In an out-of-order processor the execution order of core 2 could be LW D, R2 followed by LW F, R1 instead, because load reordering is permitted if the core is not writing to the addresses. The program order would never expect the R1 = 1 and R2 = 0 outcome, which however is possible for the execution order.

21.3 Consistency Matters Quiz

The problem is that although we are waiting for the print flag to be set to 1 before printing the data, in reality through branch prediction we may print the data at any point before the flag is set. Coherence doesn't prevent this arbitrariness.

21.4 Why We Need Consistency

Besides the data-ready flag synchronization mentioned above, consistency is also needed for thread termination, in which thread A creates thread B and waits for its completion of work and exit. Because the check of thread B exit may be branch predicted and confirmed while thread B is still doing its work.

21.5 Sequential Consistency

For programmers the most nature type of consistency is sequential consistency, which says that the result of any execution should be as if accesses executed by each processor were executed in-order,

although accesses among different processors can be arbitrarily interleaved. The simplest implementation of this is to have a core perform next access only when all previous accesses are completed, which works well but the performance greatly suffers.

21.6 Simple Implementation of Sequential Consistency Quiz

If we require issuing next access only when all prior accesses are completed, then the memory-level parallelism on each core is 1, because we cannot put together loads that are misses hoping to overlap them so as to reduce the total miss handling time.

21.7 Better Implementation of Sequential Consistency

Sequential consistency is violated by load reordering only when there is at least one store pertaining to the load that is ahead of its program order. Hence to detect such violation after the reordered load we need to monitor the coherence traffic produced by other cores. If a pertinent store is executed then the load should be replayed (which is possible because the ROB keeps the instructions in program order). In the worst case we can cancel the load and flush the instructions following it.

21.8 Relaxed Consistency

An alternative approach to better implementation of sequential consistency is to relax the consistency. There are four types of orderings: write A + write B, write A + read B, read A + write B, read A + read B. Sequential consistency requires execution to obey all these four types of orderings. In relaxed consistency some of these orderings need not be obeyed all the time. Usually the first type of ordering that needs not be obeyed is read A + read B.
With relaxed consistency we allow reordering for normal accesses and add special non-reorderable accesses. Programmers must use these non-reorderable accesses when ordering in the program matters. An example of such non-reorderable accesses is the x86 MSYNC instruction. No reordering across a MSYNC instruction is allowed. With MSYNC the data-ready flag synchronization can be written as

```
while (!flag);
MSYNC;
print(data);
```

21.9 MSYNC Quiz

Suppose the processor has a very relaxed consistency model in which all four types of reorderings are allowed. For the following lock-and-increment program the places that need MSYNC are highlighted

in red.

```
trylock:
    LL R2, lockvar;
    BNEZ R2, trylock;
    SC R1, lockvar;
    BEQZ R1, trylock;
    MSYNC;
    LW var;
    inc var;
    SW var;
    MSYNC;
    SW 0, lockvar;
```

In general a lock is considered to be an "acquire" type of synchronization and we put an MSYNC after it; a unlock is considered to be a "release" type of synchronization and we put an MSYNC before it. Similarly in flag synchronization waiting for a flag is an acquire and releasing the flag is a release. A barrier is where we have both an acquire and a release in the same operation, thus we need an MSYNC both before and after the barrier.
Note that even with the very relaxed consistency, the processor only allows reordering of accesses to different memory addresses. Cache coherence and correct uniprocessor behavior ensure that accesses to the same variable are still in order.

21.10 Data Races and Consistency

Data race occurs when there is a data dependence between accesses that are on different cores and the accesses are not ordered by synchronization, e.g. one core is reading/writing and another is writing. A data-race-free program is a program that behaves exactly the same as it would in sequential consistency even though it is running in any relaxed consistency model.
For debugging it would be very helpful if the processor can support sequential consistency. Thus some processors support flipping between sequential consistency and some relaxed consistency.

21.11 Consistency Models

Besides sequential consistency we have a family of relaxed consistency models such as weak consistency, processor consistency, release consistency, lazy release consistency, scope consistency etc. All of them support some form of synchronization to allow the enforcement of proper ordering.

22 Many Cores

22.1 Many-Core Challenges 1

The first challenges is that coherent traffic increases with the number of cores, because writes to shared addresses result in invalidations and subsequent cache misses, and both invalidations and the resulted misses go to the shared bus. Unfortunately the bus allows us to do only one request at a time, because we rely on the bus to produce ordering for writes to maintain coherence. To address this bottleneck we need a scalable on-chip network and directory coherence.

22.2 Network On Chip

One type of point-to-point network is *mesh* which is depicted in Figure 5 below. Because the links connecting adjacent chips are short, we can have a very large throughput for each link. Because we have multiple choices of path from one chip to another, the total available throughput of the network is several times that of the individual links. It is a scalable network because the number of links increases with the number of cores.
Another type of point-to-point network is *torus* which is also depicted in Figure 5 below. It can be created from a mesh by connecting the edges of the mesh to form loops in both dimensions.

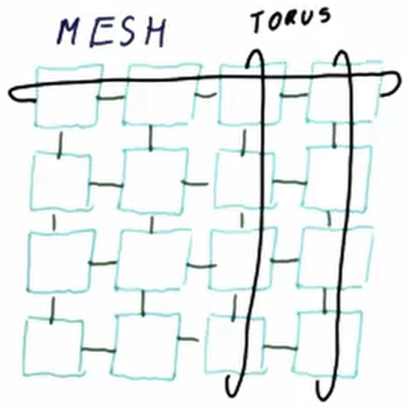

Figure 5: Mesh and torus point-to-point network

22.3 Mesh vs. Bus Throughput Quiz

Suppose we have 4 cores, each is going to send 1/4 of the messages to the other cores in the round-robin manner. Each core sends 10M messages/second. The bus can support 20M messages/second, while the mesh can support 20M messages/second per link. The cores cannot send in full 10M/s speed through the bus, because the bus bandwidth is 20M/s thus each core will send 5M messages/second totaling 20M messages/second. Hence the execution time of the bus is twice as long as each core in full speed. On the other hand the traffic per link on the mesh is $(1/2 + 1/2 + 1/6 + 1/6) \times 10 = 40/3$M messages/second ($\Rightarrow 2 \times 1/2$ for the traffic from the two connecting chips, $2 \times 1/6$ for the traffic from the two non-connecting chips), which is below the 20M/s bandwidth. Therefore each core can send in full speed on the mesh, thus the speedup of switching from bus to mesh is 2.

22.4 Many-Core Challenges 2

The second challenge is that the number of cache misses increases with the number of cores even though the number of cache misses per core stays the same. As a consequence the number of memory requests increases with the number of cores. Although the number of chip pins also increases, but its increase is not at all in proportion to the number of cores on chip. Thus off-chip throughput becomes a bottleneck. To address this bottleneck we need to reduce the number of memory requests per core. The way we achieve that is to make the size of the last level cache (LLC), which is L3 in most modern processors and shared equally by all cores, increase in proportion to the number of cores. However a single large LLC would be slow and has only one entry point for L2 cache misses which would become a bottleneck. Hence instead of having one large LLC we have distributed LLC.

22.5 Distributed LLC

A distributed LLC is logically a single cache in that a data block will not be replicated like it would be in private caches. It is sliced up so that each tile, which contains a core and possibly local caches, contains some slice of the LLC. As such the size of a distributed LLC is proportional to the number of cores, and there are multiple entry points one for each slice.
To help find out which slice contains the data, one way is to spread the sets of cache lines in a round robin manner among the slices by cache index. For example if there are 8 cores then slice 0 gets set 0, set 8, set 16 etc, slice 1 gets set 1, set 9, set 17 etc. and so on. The three least significant bits of an address specify which slice to look for it. However such spread may not be good for locality. Another way that can help with locality is to spread the data by page number instead of cache index, thus all the blocks that belong to the same page would end up in the same slice so that the operating system can map pages to make accesses more local.

22.6 Distributed LLC Quiz

Suppose we have 16 cores, each tile contains a core, its L1 cache, its L2 cache, and a slice of L3 cache. Suppose the L3 cache has a block size of 256 bytes and is distributed round robin by set. If loading address 0x12345678 is missed in both L1 and L2, we should send L3 request to tile 6, because the two least significant hex digits specify the block offset and the next specifies the index of the slice/tile.

22.7 Many-Core Challenges 3

The third challenge is that with many cores the coherence directory is too large to fit in the chip, because a traditional directory has one entry for each possible memory block and the memory can be many GB in size which would require billions of directory entries.

22.8 On-Chip Directory

For locality the home node of a block should be in the same tile as the LLC slice that contains the block. To fit in the chip we maintain a partial directory instead of keeping a directory entry for every memory block, partial in the sense that entries are allocated only for blocks that are present in the private cache of at least one core.

22.9 On-Chip Directory Quiz

If we run out of directory entries, we make the same replacement as we do for cache (e.g. LRU).

22.10 On-Chip Directory 2

We need to send an invalidation to each cache to reset every presence bit of the entry that we are replacing. Note that we are invalidating not because of coherence requirement but due to limited capacity of directory, thus this is yet another type of misses beyond the four types we have discussed before.

22.11 Many-Core Challenges 4

The fourth challenge is that power budget has to be split among cores, which implies that the power per core decreases with the number of cores. That means the clock frequency and supply voltage for each core also decrease. As a result a single-threaded program would run slower with more cores.

22.12 Multi-Core Power and Performance

Recall that active power is given by (where α is the activity factor)

$$P = \frac{1}{2}CV^2 \cdot f \cdot \alpha$$

With two cores, power spent per core is reduced by half. Since V is usually proportional to f, the $1/2$ reduction in power amounts to $\sqrt[1/3]{1/2} = 0.8$ or 20% reduction in clock frequency.

22.13 Performance vs. Number of Cores Quiz

Suppose the available parallelism for a program is 1 for 20% of the one-core execution time, 2 for 30% of the one-core execution time, 3 for 40% of the one-core execution time, and 4 for 10% of the one-core execution time, and suppose the total power supplied is fixed at 100W. If the clock frequency that can be achieved by one core with the 100W power supply is 5GHz, then the clock frequency that can be achieved by two cores with the 100W power supply is $5 \times 0.8 = 4$GHz, and the clock frequency that can be achieved by four cores with the 100W power supply is $4 \times 0.8 = 3.2$GHz.
With the same clock frequency, the speedup that can be achieved by two-core parallelism is $0.2+0.8/2 = 0.6$. Thus if the one-core execution time is 100 seconds, then it takes 60 seconds for two cores with the same frequency to complete the execution. With a 20% reduction in clock frequency the two-core execution time is $60 \div 80\% = 75$ seconds.
With the same clock frequency, the speedup that can be achieved by four-core parallelism is $0.2 + 0.3/2 + 0.4/3 + 0.1/4 = 0.508$. Thus if the one-core execution time is 100 seconds, then it takes 50.8 seconds for four cores with the same frequency to complete the execution. With a $0.8 \times 0.8 = 64\%$ clock frequency the four-core execution time is $50.8 \div 64\% = 79$ seconds, which is even slower than two-core parallelism. Therefore more available parallelism may not be able to compensate for the clock frequency reduction due to less power available for each core.

22.14 No Parallelism to Boost Frequency

The fixed power budget assumption in the quiz above is not true in reality, because we can allocate full power budget to boost the frequency of the active core(s) when there is no more parallelism available. Two examples of this clocking frequency boosting or "turbo frequency" are

- Intel core i7-4702MQ, number of cores = 4, design power = 37W, normal clock frequency = 2.2GHz, turbo frequency = 3.2GHz for one core, which is 1.45 times the normal frequency, if we assume active power is proportional to the cube of clock frequency then this amounts to 3 times the normal active power
  the reason why we cannot allocate all 37W to a single core (4 times the normal active power) is that the heat concentrated on one core would exceed the temperature limit thus might potentially damage the chip

- Intel core i7-4771, number of cores = 4, design power = 84W, normal clock frequency = 3.5GHz, turbo frequency = 3.9GHz for one core, which is 1.11 times the normal frequency, if we assume active power is proportional to the cube of clock frequency then this amounts to 1.38 times the normal active power
  the reason why the turbo frequency of core i7-4771 is much closer to its normal frequency than core i7-4702MQ is that, it is not a mobile processor thus has better cooling, which however implies that the heat generated at normal clock frequency is already a lot, this also illustrates that chips designed with lower power have more flexibility in power allocation

22.15 Many-Core Challenges 5

The fifth challenge is the confusion the solutions to the previous four challenges bring to the operating system.

22.16 SMT, Cores, Chips

Multithreading, multi-cores, multiple chips, it is not unusual to have a system that combine all these three forms of parallelism. We can have a dual-socket motherboard that hosts two chips with four cores on each chip and each core can run simultaneously two threads, which appear to the operating system as 16 threads that can run in parallel. But to maximize resource usage a smart operating system would first split tasks among threads from different chips and then map one thread to one core, and only after that would it use the second thread from the same core. This requires the operating system to know the multiple levels of parallelism granularity. Most well-known operating systems like Windows and Linux are able to figure out this hierarchy.

22.17 Many-Core Challenges

challenge: coherence traffic increases with the number of cores

solution: scalable on-chip network and directory coherence

challenge: off-chip traffic increases with the number of cores

solution: large shared distributed LLC

challenge: a single coherence directory would be too large to fit in the chip

solution: distributed partial directory

challenge: power budget must be split among cores resulting in less power spent on each core

solution: turbo frequency when using only one core

challenge: operating system confused by the multithreading, multi-cores, multiple chips hierarchy

solution: Windows, Linux