Perf Eval Scribe:

Little's Law

N = lamda * R (basic law of queueing theory)

lamda is average arrival rate of instructions
R is average service time of instructions
This law can be applied to any part of the processor (Whole processor or Fetch unit or ROB).

N - number of customers in line (IE 32 instructions in window)
R - Rate of issue
lamda - fetch rate (4 inst/cycle)

birth/death process - requests finishing and getting created. whats left is difference between whats alive and whats dead. Markov chain models birth/death rates via a state machine. single queue/single service.

Superscalar Paper:

People have hesitated to analyze superscalar processors and an approach was demonstrated here.

Calculates perfect CPI and then penalizes for loss of performance events.

Calculating Ideal CPI

1972 - Riseman & Foster - tries to develop an Instruction Window Characteristic. Given a window of W, how much issue rate (I) can I get?
Inspired by VI characteristics of transistors.
I = sqrt(W) instruction issue rate is square root of instruction window. Empirically shown. Done in 1972 (prog languages included fortran, C, pascal) There was work on dataflow (how much you could execute in an ideal case).

Paper presents a more complex IW characteristic. I = alpha*W^beta. It approaches Riseman characteristic for an alpha of 1 and beta of 0.5

alpha and beta vary depending on the program being analyzed. Derived by plotting log(I) vs log(W).

This method allows you to find the average performance in the ideal case.

Calculating Penalties

Shows how IPC drops depending on the event. Drain is a linear rate while recovery is logarithmic. References [6] and [7] show that drain is linear and this paper does not go into why.

Makes arguments for how isolated/multiple close branches behave. Drain cost and ramp up costs overlap for close branches. This calculation is important because you will frequently see overlapping branches with an basic block size of 4-5 instructions.

The authors note that the penalty can be larger than the number of pipeline stages. This is because sometimes you detect the branch very late (ie toward the end of the front-end pipeline).

Icache misses look the same but has a different effect. Penalty starts later than the miss detection process due to the buffer of instructions that are still in the icache. With icache misses, window drain is subtracted because delta I represents the amount of time under which we have detected a cache miss. The cache miss is detected but there is still a period of time where we are doing useful work.

Dcache misses are more complicated because you have to consider data dependencies, there are different penalties for L1/L2 misses, and the penalty depends on ROB size and the location of the instruction in the ROB.
RUU - register update unit.
Instruction window does not get full until ROB gets full.
The dominant effect from Dcache misses is ROB being full. If its an old instruction, the ROB fill is 0, otherwise the ROB fill length depends on the age of the instruction and the size of the ROB.