Lecture 12 - Speeding up Single Threaded using OOOE

Gidon Rosalki

2025-06-22

This content will not appear in the exam this year!

1 Speeding up Single Threading

Our goal is to minimise the CPU time:

$$\text{CPU time} = \text{clock cycle time} \cdot \text{CPI} \cdot \text{IC}$$

We can minimise the clock cycle time by adding more pipe stages, minimise the CPI by using pipeline, super scalar, and minimise the IC with loop unrolling, or modifying the architecture (e.g. adding new instructions).

The CPI in a pipelined CPU, without hazards is simply 1, but with hazards is greater than 1. So what more can we do?

Well, as discussed last week, we can do n-way superscalar, where our CPU runs n instructions together, if they are independent of each other. We can statically (in the compiler) schedule instructions to reduce dependencies, and this can also be done dynamically by the CPU (e.g. OOOE later).

A few years ago, back in 2019 2020 we had CPUs that could fetch up to 5 instructions simultaneously, and execute up to 10 instructions simultaneously. However, programs are serial, and many instructions depend on each other. Therefore, how can so many instructions be executed simultaneously? Additionally, how can more instructions be executed than fetched?

Let us consider the following C code:

```c
for (int i = 0; i < N; i++) {
    A[i] = A[i] + 1;
}
```

This translates into the following MIPS 32b assembly (ints are 4 bytes)

```
LOOP:  lw     $t1, 0($a0)
       addi   $t2, $t1, 1
       sw     $t2, 0($a0)
       addi   $a0, $a0, 4
       addi   $a1, $a1, -1
       bne    $a1, $0, LOOP
```

If we are executing the program, using an unlimited umber of execution units, we can see that we can execute up to 3 instructions per cycle, with 1.5 instructions per cycle on average. This results in 4 cycles per iteration. Is this the best we can do? What if we could execute, as soon as the data is available, instead of the program order? Let us track the data dependency:

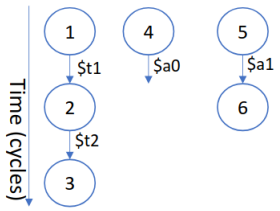

Figure 1: Data dependency tracking graph

This way, we can see that we can use up to 6 instructions together. Assuming N is larger, this is roughly 6 instructions per cycle on average, and roughly 1 iteration each cycle on average. However, execution is not in program order, what about branches?

To increase the performance, and be able to utilise more execution units, we **execute based on data dependency** instead of based only on the program order. This is called **Out of Order Execution**.

1.1 What about memory access latency

Let us consider a CPU, such that

	Size	Latency	Latency (clocks @ 4.8GHz)
L1 cache (data)	32KB	0.8ns	4
L2 cache	256KB	2.5ns	12
L3 cache	8MB	9.3ns	45
Memory	GBs	36.9ns	180

Table 1:

So, assuming the following characteristics for some program:

- 50% arithmetic / logic, 30% ld / st, and 20% control
- 10% of data memory operations miss with a 20 cycle miss penalty
- 2% of instructions miss instructions cache with a 20 cycle miss penalty
- Ideal CPI is CPI with no cache misses

So, since the PCI is the ideal CPI + average stalls per instruction, and our average stall per instruction is

$$30\% \cdot 10\% \cdot 20 + 2\% \cdot 20 = 1$$

Resulting in

	Ideal CPI (no misses)	CPI (with cache misses)	% time spent waiting for misses (stalling)
Pipelined single issue	1.1	1.1 + 1 = 2.1	48%
Superscalar dual issue	0.7	0.7 + 1 = 1.7	59%

Table 2:

So as we see, most of the time is spent stalling on cache misses, and as the CPU gets faster, a higher percent of the time is spent on stalls. So as we can see, there are many problems with program order execution, which brings us neatly into why we want OOOE.

2 Concepts in Out of Order Execution (OOOE)

Modern CPUs execute instructions out of order to increase performance. This means that instructions are executed in an order, not necessarily the same as the order specified by the program. However, program semantics are maintained (so the results are equivalent to in order execution). This method was proposed by Tomasulo in 1967. It is required for efficient use of multiple execution units, it improves the ILP (Instruction Level Parallelism, the ability to run instructions in parallel), and is required for performance for general purpose code, since it better utilises multiple execution units, and ensures execution when waiting for results (e.g. handling cache misses). However, building an OOOE CPU requires complex hardware, since we need to rename registers, reorder results, and have the CPU speculate instructions while also handling instances where it speculated incorrectly.

2.1 General scheme

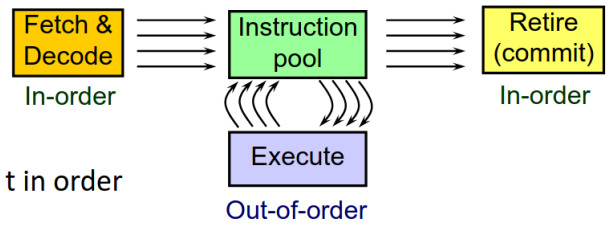

Figure 2: OOOE general scheme

Fetch and decode instructions in parallel, but in order. This fills the instruction pool. It also executes ready instructions from the instructions pool, where all data sources are ready, and needed execution resources are available. Once an instruction is executed, we signal to all dependent instructions that the data is ready. Next, commit instructions in parallel, but in order, and write back results (memory, registers). OOOE is typically superscalar, capable of fetching, executing, and committing multiple instructions every cycle.

Let us consider the following program:

```
(1) lw    $1, 0($4)  ; $1 <- load[$4]
(2) add   $8, $1, $2 ; $8 <- $1 + $2
(3) addi  $5, $5, 1  ; $5 <- $5 + 1
(4) sub   $6, $6, $3 ; $6 <- $6 - $3
(5) add   $4, $5, $6 ; $4 <- $5 + $6
(6) add   $9, $8, $4 ; $9 <- $8 + $4
```

So here, we have the following data flow graph:

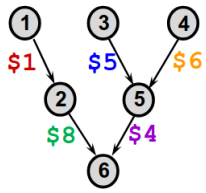

Figure 3: Data flow graph

Should we run in order, then we have to first run 1 (long time), followed by 2, 3, and 4 simultaneously, and finally 5 and 6 sequentially. However, if we use OOOE, and follow the data dependency order, we can execute 1, and then simultaneously 3, and 4, and then run 5 before 1 has even completed. We can then run 2, and then 6 sequentially.

We have a problem, false dependencies limit the number of instructions that can be fed into the instruction pool for OOOE. We can resolve this by using **register renaming**. Here we maintain a mapping table architectural register. For each instruction, we perform the following steps:

1. Source operands renaming: Check if source registers were mapped to physical register by a previous instruction and use it
2. Destination operand allocation: Allocate a destination register from the pool of physical registers, and update the mapping table

So, by following these for each instruction above, we achieve:

```
(1) lw    $2, 0($1) [→$2p1]   lw    p1, 0($1)
(2) add   $3, $2, $0 [→$3p2]  add   p2, p1, $0
(3) addi  $1, $0, 77 [→$1p3]  addi  p3, $0, 77
(4) add   $2, $3, $0 [→$2p4]  add   p4, p2, $0
(5) add   $2, $1, $1 [→$2p5]  add   p5, p3, p3
```

Where the first column is the instructions, the middle the mapping, and the third the resultant instructions after the mapping.

So as we can see, we have removed the false dependencies. Where earlier, both instructions 4, and 5, both wrote to the same register, they no longer do so.

2.2 A superscalar OOOE Machine

2.2.1 In order

Some steps still have to take place in order, but can be done making use of superscalar as discussed last week. These steps are:

1. Fetch and decode: Fetch and decode instructions in parallel, but in order
2. Rename: Data dependency analysis and structuring. Here we rename sources to physical registers, and allocate a physical register to the destination

The ROB (Reorder Buffer) is the pool of instructions fetched, and waiting for retirement. This keeps track of speculative instruction results before writing them to the architecture (register file or memory).

2.2.2 Out of order

1. Reservation Stations (RS): This is the pool of instructions waiting for execution. IT maintains per instance sources, and a ready / not ready status.
2. Execute: We track which instructions are ready, meaning they have all sources ready. Then we dispatch ready instructions to the execution ports in FIFO order. The RS handles only register dependencies, it does not handle memory dependencies. After execution, we write back the value to the RS, and mark more sources as ready. The RS then sends ”exe done” indication to ROB, and reclaims the RS entry.

2.2.3 In order

1. Retire (commit): Commit instructions received from ROB in parallel, but in order. Commit means writing results to memory, and actual architectural registers.

2.3 Reorder Buffer (ROB)

The ROB holds instructions from allocation, and until retirement. They are held in the same order as in the program (program execution order). It provides a large physical register space for register renaming, one physical register per ROB entry.

- Physical register number = ROB entry number
- Each instruction has only 1 destination
- Buffers the execution results until retirement

The valid data is set after the instruction is executed, and the result is written to the physical register (i.e. to the ROB entry).

#entry	Entry Valid	Data Valid	Physical Register Data	Architectural Destination Register
0	1	1	12H	$1
1	1	1	33H	$4
2	1	0	xxx	$2
…				
39	0	0	xxx	XXX

Figure 4: ROB example

2.4 Architectural Register File

The ARF is also known as the Real Register File (RRF). It holds the architectural register file, where architectural registers are numbered. The value is an architectural register is the value written to it by the last instruction committed which wrote to this register:

#entry	Arch Reg Data
o($1)	9AH
1($2)	F34H

Table 3: RRF

2.5 Reservation Station / Issue Queue (RS / IQ)

This is the pool of all instructions that have not yet been executed. It holds an instruction's attributes, and source data, until it is executed, along with the instruction itself, sources, and destination (physical destination). When the instruction is allocated in RS, the operand values are updated.

Operand form	Data valid	Get value from
Architectural register	1	RRF (arch reg file)
Physical register	1	ROB (p#)
Physical register	0	Wait for value

Table 4:

3 Instruction Fetch and branches in OOOE

3.1 Fetch and branches

OOOE fetches multiple instructions each cycle. The simplest approach is to simply fetch in program order, and ignore branches (i.e. predicting branches as not taken). On average, 1 of 8 instructions is a branch. So what if a branch happens to be taken? The fetch may have fetched the wrong instruction, and so on execution, such branch is marked as 'mispredicted'.

3.2 Jump misprediction, flush at retire

When a mispredicted jump retires, we flush the pipeline. When the jump commits, all the instructions remaining in the pipe are younger than the jump, and are therefore from the **wrong** path. We need to reset the renaming map, so all the registers are mapped to the architectural registers (RRF). This is OK, since there are no consumer of physical registers, since the pipe is flushed. So we start fetching instructions from the correct path.

This has the disadvantage of a high misprediction penalty.

So OOOE requires an accurate branch predictor. It needs to predict branches at fetch/decode time, long before they are executed. Misprediction causes a few cycles stall, since we need to flush and restart the pipeline, e.g. 5 cycles. Since there is typically a branch roughly every 8 instructions, a 4 wide OOOE will fetch 8 instructions every 2 cycles. If it is always wrong, a flush will take place every 9 instructions, which is 5 cycles of flush after 2 cycles of work, resulting in only 29% work. If it is 50% then we get 45% work. For 90% correct this is 5 cycles every 20 cycles of work, bringing us up to 80% work.

3.3 Branch prediction

We predict using history, where the branch history is updated based on actual execution. The prediction is made before the instructions bytes are available. The hardware predicts the existence of branches in a group of instructions before they are even fetched. We need to predict taken branches location, and target, i.e. where a branch instruction exists in the fetched instructions, if the branch is taken or not taken, and the target of a predicted taken branch.

If the prediction rate is greater than 90%, then for an average of a branch per 5-10 instructions, we have >50-100 instructions between mispredictions. A high prediction rate is **crucial** for OOOE, speculative execution, since on misprediction everything is flushed, and the effective size of the window is determined by the prediction accuracy.

3.3.1 Predicting the target

An array called the **target array** is accessed using the branch address (branch PC). This can be implemented as an n way set associative cache. The target array predicts the following:

- Instruction is a branch
- Branch target
- Branch type (e.g. Conditional / unconditional)

Since tags are usually partial, there is a trade off between space and accuracy, and we can get false hits. Multiple branches can be aliased to the same entry. This is not a correctness issue, but rather only performance.

3.3.2 Predicting whether or not a branch is taken

We can predict whether a branch will or will not be taken based off heuristics in the program with either 1 or two bits. These are based off state machines, with 2 or 4 states, and work exactly as you think. Each have advantages / disadvantages, and certain series of branches that break them.

3.3.3 Putting it all together

For each instruction fetched, we need to predict if it is a taken branch, and predict its target. We can predict taken **only** if the target is known. Next cycle PC is the first predicted taken branch, or sequential address, if no taken branch. If possible, we can correct on decode branch existence, and target (if taken). On correction, we flush the wrong instructions fetched by the in order pipeline of fetch and decode, and restart fetching on the correct instruction. Execution then checks the branch prediction accuracy, and retirement updates the predictors, and corrects (flushes) the pipeline if needed.

3.4 Modern branch predictors

The above branch predictor is **very** basic. Modern branch predictors are **much** more complex. Branch prediction is an active research field, and CBP (championship branch prediction) workshops are held, where predictor algorithms code are submitted on a simulator framework, and compete by simulating predictors on real workloads that are not known in advance. Progress and improvements are being implemented in modern CPUs.

4 Spectre background

4.1 Background

This is a class of security vulnerabilities that affect modern CPUs that perform branch prediction, and other forms of speculation. Speculative execution resulting from a branch misprediction may leave observable side effects, that may reveal private data to attackers. There are lots of different variants, and we will probably only discuss variant 1 here. The public vulnerability name of variant one is the Bounds Check Bypass (BCB).

4.2 Cache side channel attacks

Secrets can be inferred by filling up a cache, and then getting the OS to load the secret into the cache. By establishing which of my blocks in the cache were emptied, I can infer information about the secret, based off the mapping function that mapped it into the cache..

4.3 BCB overview

Consider the OS function `foo`:

```
foo(int j) {
    if (j >= 0 && j < MAX_ALLOWED) {
        char x = os_table[j];
        int y = os_table2[x * 64];
        ...
    } else {
        return ERROR;
    }
}
```

The attacker trains the branch predictor by invoking a function `foo` with a valid input `j` many times. The attacker fools `foo` to access the secret by invoding it with `j = secret_address - os_table`. `j` is out of bounds, but the predictor still predicts based off the history that it is OK, and so `foo` speculatively accesses the secret. The attacker can then use a cache side channel attack to infer the secret value.