CIS501: Out-of-Order
(Dynamic Scheduling)

Shreyas S. Shivakumar

December 2, 2019

1 Lecture Notes

1. **Why do we need *code scheduling* and what are the *two ways* of doing it?**

   - We can't really avoid **load-to-use** delays in the designs that we have looked at so far. But what if we could **fill** those delay slots with instructions that are *independent* of it? This way we can *increase CPI* and *increase ILP*.
   - **Static Scheduling:** at *compile* time, done by software.
   - **Dynamic Scheduling:** at *run* time, done by hardware.
   - We will be primarily looking at *dynamic scheduling*; a process that creates more opportunities to schedule things in parallel by moving independent instructions around. This is very common today and exists on cellphones, laptops, etc.

2. **Static Scheduling**

   Some key characteristics / criteria to properly perform *static scheduling* are listed below. These must be taken care of at *compile time*.

   - **Scheduling Scope:** Ideally we want a larger scope because if the scope is too small there's nothing that the compiler can do to improve performance. It cannot rearrange and re-order instructions if it doesn't see

enough instructions. *Branch Instructions* specifically can cause problems with scheduling scope because it is impossible to move memory operations past branch instructions.

- **Registers:** We need a sufficient amount of registers to hold all *"live"* values. But we are limited by the ISA in how many registers are available to us.
- **Alias Analysis:** We need to be able to tell whether a LOAD/STORE instruction reference the *same memory locations*. This is often difficult for the compiler to do and prevents reordering of loads above stores.

However, there are situations where *Static Scheduling* works really well such as in **SAXPY** (Single-precision AX Plus Y). Since all loop iterations are independent - loop unrolling. Alias analysis is tractable. Limited registers are still a challenge.

3. Dynamic Scheduling

In this variant, the *hardware* is responsible for the re-scheduling of instructions. The key motivations and criteria are similar to the ones mentioned above. We need to increase *scheduling scope* and we can use *branch prediction* to unroll branches. The benefit of this approach is that everything happens under the hood and the software layers are not exposed to any of the inner mechanisms.

The **Out-of-Order Pipeline** can be divided into the following modules

- **In-Order** Front End - Fetch, Decode, Rename, Dispatch
- **Out-of-Order** Execution - Buffer of Instruction $\rightarrow$ { Issue, Register Read, Execute, Write-back }
- **In-Order** Commit - Commit

Programs will enter *in-order* and exit *in-order*, maintaining the illusion of sequential program execution. But between the *Dispatch* and *Commit* stages, the instructions can be processed out of order and hence we can reap the benefits of an out-of-order design while maintaining a final sequential ordering.

Dependency Types:

- **RAW** (Read After Write) - *True Dependence*
- **WAW** (Write After Write) - *False Dependence*
- **WAR** (Write After Read) - *False Dependence*

Both **WAW** and **WAR** are *False* dependencies and can be eliminated completely by *Register Renaming*. The *True* dependency **RAW** is enforced by the final *Dynamic Scheduling*.

4. Part 1: Register Renaming

To understand the *Register Renaming* algorithm, the following new ideas are required:

Q: What is the difference between architectural registers and physical registers?

In an ISA such as LC4, we have registers $R0...R8$. These are actually indirections / pointers to actual physical registers which are more than 8 in number. The pointers to the physical register locations are called *architectural registers* and the actual physical locations are *physical registers*.

$$\#\text{physical registers} > \#\text{architectural registers}$$

Q: What are the additional hardware structures required for register renaming?

The first is a **Map Table**, which maintains the mapping from *architectural registers* to the *physical registers*. The second is a **Free List**, which maintains the list of available registers (in a queue).

Q: What is the goal of the Register Renaming algorithm?

The goal is to *eliminate contention / eliminate false dependencies*.

5. Part 2: Dynamic Scheduling

The previous *register renaming* happened in-order, and the instructions then move onto the actual dynamic scheduling, i.e when the instructions no longer follow the program order. By this stage, all the *False Dependencies* have been removed and only *True Dependencies* remain.

The instructions from the previous stage are put into a **Instruction Buffer** or *Instruction Window / Instruction Scheduler*.

Additionally, there is another hardware structure called the **Ready Table** which maintains whether an instruction is *ready* or not, and then executes the oldest *ready* instruction. The ready-ness of the instruction is maintained by a single **Ready-Bit**.

Q: Why give preference to the oldest instruction?

Old instructions will produce things that more instructions may depend on. By processing older instructions first we also guarantee that *progress* will be made.

The other key pieces of hardware are the **Issue Queue**, which is the *central piece* of scheduling logic. It holds the instructions from the *Dispatch Stage* to the *Issue Stage*. It tracks Ready inputs as well as their *Birthday* to indicate how old each instruction is.

Q: Why maintain birthday instead of age?

Age changes over time but birthday remains consistent and is hence easier to maintain.

Note: The Issue Queue is not *read* in program order which means that the structure itself contains instructions not in program order. This is technically not a *queue* by the proper definition and works more like an Issue Table.

The next piece of hardware is the **Re-order Buffer (ROB)**, which holds the instructions from the *Fetch Stage* through to the *Commit Stage*.

In summary:

- The *Dispatch Stage* fills up the *Issue Queue* and the *Issue Stage* involves figuring out which instructions in the *Issue Queue* are ready and can leave the queue to go onto the next stage.
- This involves **Selecting** ready instructions that can leave and **Waking Up** dependent instructions. This *search* and *wake-up* is done using Content Addressable Memory (CAM).

6. Register Renaming - Continued

In the previous description of the *register renaming* algorithm, once a register was overwritten it was *discarded*. But in reality, there is a finite number of these registers and they need to be **reused**. We can do this by *tracking / logging* the overwritten registers in the **Re-Order Buffer** (ROB) and then we can **free** them after the **Commit** stage.

The only change to the algorithm is that we need to keep track of the overwritten register before updating the *MapTable* with the new register. Once the instruction is finished at the *Commit Stage*, we can **free** that register.

Re-Order Buffer (ROB):

- Holds all the information for *recovery* and *commit*.
- This information is not removed until the very last *commit*.
- Helps track instructions for an in-order *commit*.
- Useful in freeing up physical registers.
- Useful in recovering from mis-predictions.

Recovery:

Everything that leaves the *Commit Stage* **must be** correct. This implies that up-to the *Commit Stage*, we are free to use any method of speculation and guessing, but once an instruction leaves the *Commit Stage*, it affects the system permanently. This also gives us the opportunity to **recover** from misspeculations prior to the *Commit Stage*. This involves completely removing the wrong path instructions, i.e rewinding all the changed state in the system. This can be done the following ways:

- Log-based Reverse Renaming - *track old mapping; slow*
- Checkpoint-based Recovery - *checkpoint MapTable and free list; more state*
- Hybrid - *combination of both*

Note: At the *Commit Stage* the instruction becomes **Architected State**. This means that we've really made the changes and instruction has really happened.

Q: Why is it safe to free an overwritten register?

It is safe because any instruction that requires it's value should read the new updated value and not the value that was overwritten.

7. Dynamic Scheduling - Continued

To recap, the *Issue Queue* holds all waiting (un-executed) instructions. The *Issue Queue* also holds ready/not-ready status of all the source registers for each instruction.

Note: The **Di** stage is a combination of *Decode, Rename* and *Dispatch*.

The different hardware structures necessary to make this function smoothly are:

- Map Table
- Ready Table
- Re-Order Buffer
- Issue Queue

Once a register is overwritten, it is placed in the **Re-Order Buffer** to indicate that it can be released / free-d once the instruction is complete, i.e once the instruction leaves the *Commit Stage*.

Q: Where do instructions wait?

In **In-Order** execution, instructions usually waited in the *Decode Stage*. In **Out-of-Order** execution, they will need to wait too, but they can wait in more than one place such as the **Re-Order Buffer** and the **Issue Queue**. Instructions that are stalled after the *Di Stage* are usually *hanging out* in the **Issue Queue**.

8. Memory Operations

So far, we have primarily looked at *read* operations that use registers. What happens when we perform memory operations such as *load* and *store* where the actual contents of the memory are changing?

As before, the types of dependencies are the same - **WAW** and **WAR** are *False* dependencies and should be eliminated. **RAW** is a *True* dependency and has to be enforced.

Problem: The overall challenge is that we can't rename memory the same way that we did with registers because *memory addresses are* ***not known*** *at the Rename Stage.*

9. Store Instructions

- Let's look at ***only*** *store* instructions and ignore *load* instructions for now.
- Store instructions write to data cache and not registers so we cannot use the *register renaming* algorithm that we've seen previously.
- Writing to the data cache is unrecoverable so we must be careful about what we finally write to the data cache - do it only when certain; at the *Commit Stage.*
- *Thoughts:* Maybe put the *store* instructions somewhere else?

Consider the following sequence of instructions:

MUL P1 $\times$ P2 $\rightarrow$ P3
JNZ P3
STR P5 $\rightarrow$ [P3+4]
STR P4 $\rightarrow$ [P6+8]

Q: ***Out-of-Order Stores*** *- What if* $P3 + 4 == P6 + 8$*?*

The two store instructions will write to the same address (WAW dependency : structural hazard). And we won't know this until the *Execute Stage* because that is when the memory addresses are actually known. **Naïve Solution:** All stores execute in order.

Q: ***Branch Mis-prediction*** *- What if the JNZ instruction is mis-predicted?*

We require a degree of speculation to handle branch instructions efficiently and usually on mis-speculation we can go back and re-wind to the previous

steady state. But here we would have written to the *data cache*, and this cannot be *undone*. In this case, what if the 4^{th} instruction proceeded out of order (since it is independent) but then the 2^{nd} instruction was mis-predicted. We would need to undo the 4^{th} instruction.

10. Store Queue

We add another hardware structure called the *Store Queue* that can solve the above two problems.

- At the *Dispatch Stage*, each *store* instruction is given a slot in the *Store Queue* and it is a FIFO queue.
- At the *Commit Stage*, the value is read from the *Store Queue* and written into the data cache.
- To facilitate recovery from mis-speculation, we can remove entries from the *store queue*.
- By introducing the *Store Queue*, store instructions don't interact with the *Memory Stage* at all and go to the SQ and then it is eventually committed.

11. Memory Forwarding

*Q: What if **load** instructions were thrown into the mix?*

Consider the following sequence of instructions:

FDIV P1 P2 $\rightarrow$ P9
STR P4 $\rightarrow$ [P5+4]
STR P3 $\rightarrow$ [P6+8]
LDR [P7] $\rightarrow$ P8

Q: Can LDR [P7] $\rightarrow$ P8 issue and begin executing?

Since the *load* instruction could depend on the second *store* instruction above, the *store* instruction might not have written it's value to the data cache yet. But since the value is already ready, we should be able to pass the value to the *load* instruction such as in bypassing. This is **memory forwarding**.

For the *load* instruction to get the correct value, it must also search the *Store Queue* in parallel with the data cache. This is different from regular *register bypassing* because here we will not know the addresses until the *Execute Stage*.

12. YOMS: Youngest Older Matching Store Policy

Consider the following sequence of instructions:

MUL P1 × P2 → P3
JNZ P3
LDR [P3+4] → P5
STR P4 → [P6+8]

If the 4^{th} instruction runs before the 3^{rd} instruction and if $P3+4 == P6+8$ we need to ensure that the value that the *load* instruction reads is not the value that was written by the *store* instruction.

Solution: Add an *age / birthday* field to the *Store Queue* entries. *Load* instructions must then read from the **youngest older matching store**.

In essence, we are preventing instructions from seeing values from the *future*.

13. Load Scheduling

Consider the following sequence of instructions:

MUL P1 × P2 → P3
JNZ P3
LDR [P3+4] → P5
STR P4 → [P6+8]

If $P3 + 4 == P6 + 8$, the *load* instruction should get the value from the *store* instruction as we've seen in the YOMS example above. **But**, the value isn't put into the *Store Queue* until the store instruction has finished it's *Execute Stage*. This is a **problem**.

Naïve Solution: All loads must wait for all earlier stores. This is called **Conservative Load Scheduling**. This option is *always safe* but comes at the cost of performance.

Q: What is a better alternative to this?

Let the *load* instruction do the best it can by speculation. If it is wrong, we can figure out a way to fix things. But if it is correct, we have achieved performance gains. This is called **Optimistic Load Scheduling** or **Load Speculation**.

- *Q: When do we speculate?* On every load!
- *Q: How do we detect a mis-speculation?* The Store Queue can be used by the load instruction so we can check for any load instruction that ran too soon - this is done by a new piece of hardware called the **Load Queue**.
- *Q: How do we recover?* Squash offending load instructions and all newer instructions. The penalty here is high but it is still worth it!

Load Queue:

- Detects load ordering violations
- *Load Instructions:* when a load executes, write it's address into the *Load Queue*.
- *Store Instructions:* when a store executes, search the *Load Queue*.

Load Queue vs Store Queue:

- Store Queue handles forwarding and allows out-of-order stores.
- Load Queue handles the detection of load ordering violations.
- Together they allow for aggressive load scheduling.

When this works, it *increases performance* and *increases ILP*. When there is a conflict the performance is worse than just waiting. The next improvement is to **avoid squashes** by predictive load scheduling. Here a PC based predictor can be used or some algorithms make predictions over sets of instructions for such dependencies.

2 Textbook Notes

1. Register Renaming

- The goal is to eliminate dependencies that are not *true* data dependencies (also called *anti-dependencies*), but could either lead to potential hazards or prevent the compiler from flexibly scheduling code.
- Renaming registers during the unrolling process allows the compiler to move these independent instructions subsequently so as to better schedule code.

2. Dynamic Scheduling

Dynamic scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls.

Definition: Hardware support for reordering the order of instruction execution so as to avoid stalls.

Conceptually, this can be thought of as the process of analyzing the data flow of a program and then executing the instructions in some order that preserves the data flow order of the program while minimizing overall stall cycles.

Dynamic Scheduling is often extended by including hardware-based speculation, especially for branch outcomes. A speculation on load addresses allows load-store re-ordering as well, using the *Commit Stage* to avoid incorrect speculations from propagating to the programmer-visible registers and memory.

3. Commit Unit

The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory.

4. Re-order Buffer (ROB)

The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory of a register.

5. Power Efficiency

The downside to increasing exploitation of instruction-level parallelism via dynamic multiple issue and speculation is power efficiency.