ECE 511: Computer Architecture October 22, 2004

Lecture 15: Predicting Memory Dependencies

Lecturer: Matt Frank *Scribe: Brian Watson*

This lecture covers material discussed in the paper titled *Memory Dependence Prediction using Store Sets* by Chrysos and Emer. The paper makes the observation that is it desirable to issue `load` and `store` operations out of order with respect to one another without violating dependencies.

Consider the following example:

```
st M[R14] ← R13
ld R12 ← M[R15]
```

If R_{14} and R_{15} hold different values, there will be no dependency violation between the `load` and `store`. If they have the same value, it is desirable that the instructions execute in order. However, the scoreboard may issue out of order if:

- The `load` or `store` use the same register, and the earlier instruction is not ready
- The scoreboard issues randomly, like in most designs, and both instructions are ready

How do we tell the scoreboard to keep these instructions in order?

> We can add a tag for the implicit destination of the `store` and implicit source of the `load`. If we see a violation in the past, we will assign a special tag to that particular violation. When the `load` and `store` with a previous violation are fetched, they get assigned a new architectural tag that gets renamed to a certain physical tag.

There is an added complexity to this method, since the tags we use are not associated with physical tags. We must have some facility similar to a RAT for these tags. However, the tags do not need to be placed into the RRAT when an instruction retires or return to the free list.

The structure resembles a BTB; it's a small cache that records the PC's of the `load` and `store` when a dependency violation occurs between them. In addition, we need to store the PC's in the load queue when there is a violation. The bookkeeping of this procedure will occur in a new stage of the pipeline, the Store Set stage.

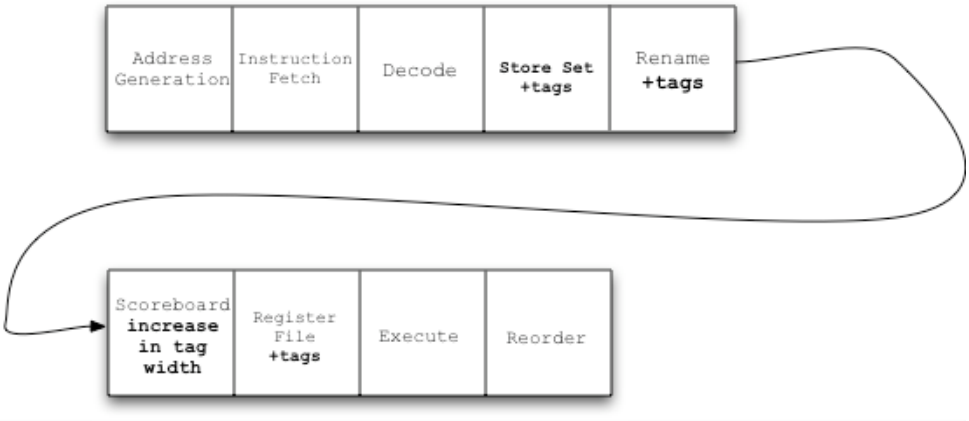

Figure 1: The new pipeline with the Store Set stage - changes in **bold**

We want to assign a `load` and `store` to the same set when a dependency violation occurs. There are four possible scenarios:

Case	Action
1. Neither `load` nor `store` in table	Assign the set a tag off of "free list"†
2. `load` in table but not `store`	Assign tag associated with `load` to `store`
3. `store` in table but not `load`	Assign tag associated with `store` to `load`
4. Both `load` and `store` in table	Assign smaller tag to both `load` and `store`

†The "free list" in most implementations is just a modulo counter that increments each time a new pair is assigned in the table.

Examples:

Case 1:

`PC 1000` and `PC 1012` are stored into the table, since their respective instructions have had a dependency violation for the first time:

PC	Instruction	Tag
1000	st	–
⋮	⋮	⋮
1012	ld	–

The proper action is to assign a tag from the "free list". For this example, we assume the counter is at 513, so the tag 513 will be assigned to both `PC 1000` and `PC 1012` in the table:

PC	Instruction	Tag
1000	st	513
⋮	⋮	⋮
1012	ld	513

Case 2:

Let's expand Case 1 to a situation where a `store` at `PC 1020` has a dependency violation with the `load` at `PC 1012` for the first time. We should assign the tag associated with the `load`, 513, to `PC 1020`. The table will now have an entry for `PC 1020`, associating it with tag 513:

PC	Instruction	Tag
1000	st	513
⋮	⋮	⋮
1012	ld	513
⋮	⋮	⋮
1020	st	513

Case 3:

Case 3 is the same method as Case 2 with the exception that the table is receiving a new entry to a load rather than a store.

Case 4:

Case 4 is probably the least intuitive of all of the cases. Let's assume the following entries are in the set table:

PC	Instruction	Tag
2000	ld	517
⋮	⋮	⋮
2016	st	517
⋮	⋮	⋮
2048	ld	524
⋮	⋮	⋮
2060	st	524

The state of the table would indicate that the `load` at `PC 2000` has had a dependency violation with the `store` at `PC 2016`, and the `load` at `PC 2048` has had a dependency violation with the `store` at `PC 2060`. If through some flow of a program a dependency violation were to occur between the `load` at `PC 2048` and the `store` at `PC 2016`, we should update the table to associate the `PC` of both instructions with the smaller of the two tags. The updated table would be:

PC	Instruction	Tag
2000	ld	517
⋮	⋮	⋮
2016	st	517
⋮	⋮	⋮
2048	ld	517
⋮	⋮	⋮
2060	st	524

Now, we know a dependency violation has occured between `PC 2048` and `PC 2060` in the past. By altering the table to this form, the violation could occur again. If it occurs again, the table would update to:

PC	Instruction	Tag
2000	ld	517
⋮	⋮	⋮
2016	st	517
⋮	⋮	⋮
2048	ld	517
⋮	⋮	⋮
2060	st	517

This update policy allows us to update the table when we find inter-dependencies without having to do a fully-associative lookup on the table or replace all of the tags simultaneously. The cost updating this way is the possibility of one additional dependency violation.

Conclusions:

One might argue that adding an additional stage to the pipeline will hinder performance. However, we need to look where there are dependent cycles in the pipeline. One such area of the pipeline is around the execution stages. These stages take a performance hit when there is a dependency violation, causing a flush of the pipeline. The Store Set stage significantly reduces the number of violations, yielding an overall performance increase in having the extra stage.

Unlike a branch predictor, the dependency table of store sets has no feedback. This means that the table may track dependencies that no longer occur. Over time, the table can get restrictive to a point where performance starts to decrease. The only solution is to clear the table after certain events or arbitrary intervals.