A proposal for solving the assignment of branch predicates

In the original proposal, a branch tracking buffer (BTRB) was used to assign the predicate addresses to each instruction. The predicate values are dynamically generated and forwarded through the predicate forwarding bus. The main drawback of BTRB is its complexity that makes it a non-scaleable component. The following proposal suggests a rather different solution.

The main idea is that each branch would generate a predicate value based on whether it is predicted taken or not. Since a conditional branch is a relative jump, by having the time tag of the branch, we can compute the time tag of the target instruction. An active station with a branch instruction will generate a predicate value and will forward it on the predicate forwarding bus (PFWB). It also has to forward the time tag of the branch and its target instruction. All Active Stations will snoop the PFWB. If their time tag is less than the branch target time tag and greater than the time tag of the branch, they will snarf the predicate value. In other words, if an instruction is inside the domain of a taken branch, it would assume that it doesn't have to be executed. Otherwise if the branch is not taken, the instruction will execute.
Note that an instruction could still depend on the outcome of a branch even though it is out of domain of that branch. But this could only happen if there is another branch inside the domain of the first one that could affect this instruction. This second branch has to forward its predicate in case it is on the not taken path of the first branch. However, If a branch is inside the domain of another branch and the first one is predicted taken, the second branch does not need to send a predicate on the bus. This is different from the older solution where each branch had to be executed regardless of whether it was on the predicted path or not.

It has to be mentioned that every time the input predicate of a branch make a transition from 0 to 1, the branch needs to re-evaluate itself and forward the predicate. This is to ensure that the instructions later in time get the right order. In the other words, the whole idea is that each instruction should only look at the last predicate value that could affect it. If the instruction is out of domain of a branch, the predicate of that branch will be ignored!

One major advantage of this method is that the AS needs to only remember the last relevant predicate value. It is the responsibility of branch instructions to send their correct predicate values. We no longer need to keep track of canceling predicates and multiple branches could jump to the same location – note that in reality only one of them would make the jump!

One other issue on the MIPS architecture is the branch delay slot. This can be easily solved by comparing the time tags of the current instruction and the branch instruction. If the instruction that snoops the bus is the immediate instruction after the branch, it will execute regardless of the outcome of the branch.

The hardware overhead of this method is the need for an adder and comparators. The PE could be used to compute the time tag of the branch target instruction. The comparators are however needed.

To better describe how this skim works, I will use a simple example. Figure 1 shows a column of Active stations with two nested branches. Suppose that Br1 changes its predicate from taken to not taken. It will send a predicate value, announcing that the branch is not taken, along with the time tags of the target instructions. All other instructions after the branch will see the predicate and figure out that they have to be executed. Now suppose that Br2 is predicted taken. It will send its predicated value along with the its time tag and target time tag. The instruction in the delay slot of Br2 would execute regardless of the predicate value. The rest of the instructions in region 2 would figure out that they fall in the domain of Br2. They would decide, therefore, that they do not have to be executed. The instructions in region 3 and 4 have a time tag larger than the target of Br2 and hence will execute.

Now suppose that Br1 is taken. All instruction in Region 1,2 and 3 will figure out that they are on the non-predicted path and do not have to be executed. Note that Br2 doesn't even have to send any predicate on the forwarding bus. In the case that Br2 had already sent a predicate, it would be overwritten by what Br1 sends. This effectively shows the memoryless nature of this algorithm.

Note that if both Br1 and Br2 were predicted taken and Br2 had sent its prediction before Br1, it has to send a new predicate once it observes the taken predicate of Br1. This might not be important for this example but would matter for a case like the second example. In general, if a branch has sent out a wrong predicate value, it has to send the correct value.

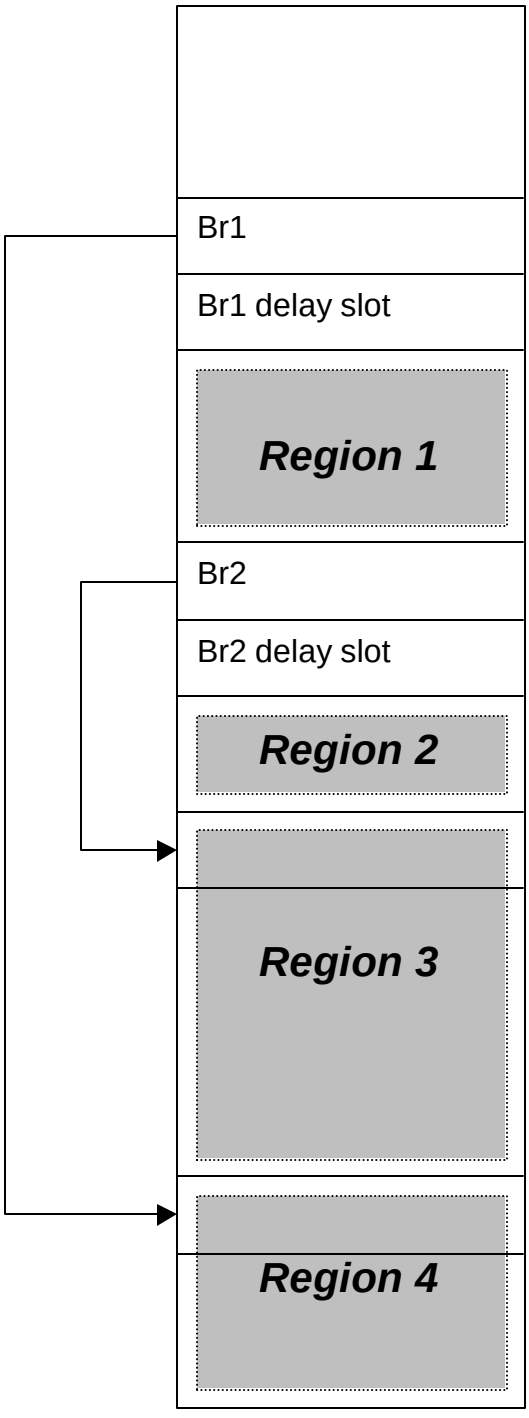
Figure 2 shows a rather more complicated case. The following table summarizes different valid combination of branch outcomes.

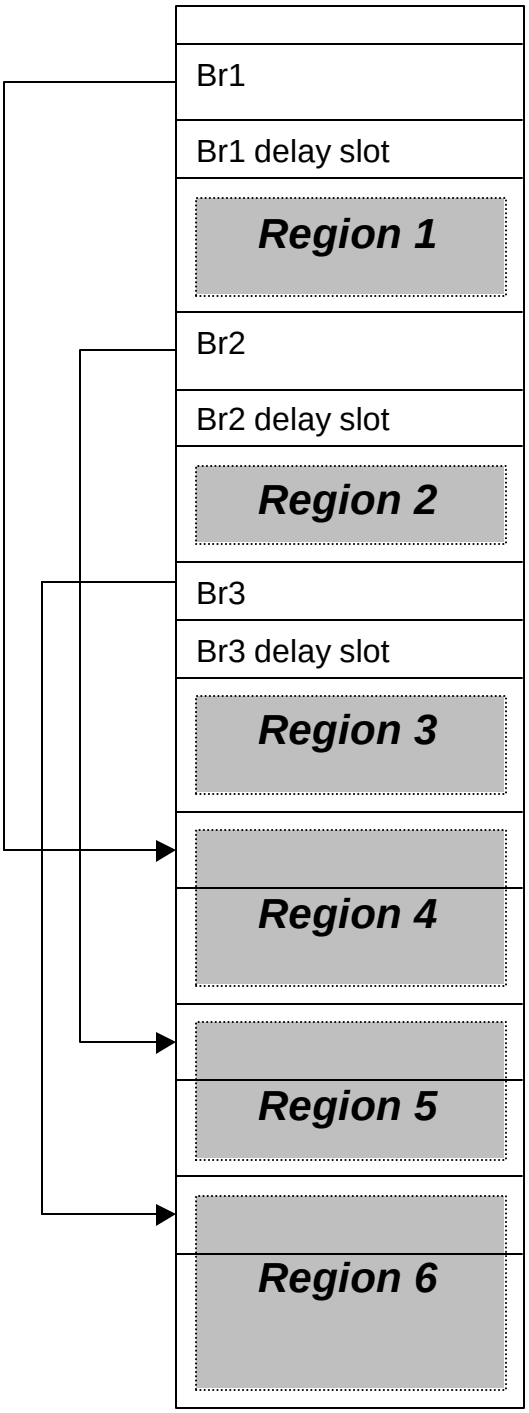
Br1	Br2	Br3	R1	R2	R3	R4	R5	R6
T	X	X	I	I	I	E	E	E
NT	T	X	E	I	I	I	E	E
NT	NT	T	E	E	I	I	E	E
NT	NT	NT	E	E	E	E	E	E

T: Branch predicted Taken
NT: Branch predicted Not Taken
I: AS is Idle
E: AS is executing

What is important is that all the transition from one row to another is correctly reflected in the input predicate values of the instructions in each region. If me make sure that the order in which instructions see the output predicates of each branch is kept as same as the branches appear in the instruction stream, then I have selected two different scenarios that show how the proposed algorithm works.

1. The predicate value of Br1 changes from taken to not taken.
2. Instructions in all regions assume to be executed.
3. Suppose both Br2 and Br3 predicted taken.
4. Suppose Br3 puts is the first one which forwards its predicate
5. Instructions in R3, R4 and R5 change their status to Idle.
6. Br2 puts its predicate on the bus.
7. R2, R3 and R4 change their status to Idle. R5 changes its status to Executing.
8. Br3 doesn't need to do anything at this time. It is not on the predicted path.


Br1
Br1 delay slot
Region 1
Br2
Br2 delay slot
Region 2
Region 3
Region 4


Br1
Br1 delay slot
Region 1
Br2
Br2 delay slot
Region 2
Br3
Br3 delay slot
Region 3
Region 4
Region 5
Region 6