ECE 511: Computer Architecture 27. Oct. 2004

Lecture 17: Dynamic Optimization

Lecturer: Matthew Frank *Scribe: Sebastian Vogel*

Last time: Historical perspective on ILP
Trace caching (historical motivation)

- Optimizations in the front end
- Make fetch unit more efficient
- CISC: reuse previously decoded instructions

This time: take traces and optimize them

“Optimize” means reducing the path length to improve performance. We have to avoid any change making the code worse.

Example: optimization of a loop by compilers

```
1    Y = val
2    For (i=1 to n) {
3         X = sqrt(Y)
4         Do something with X,i
5    }
```

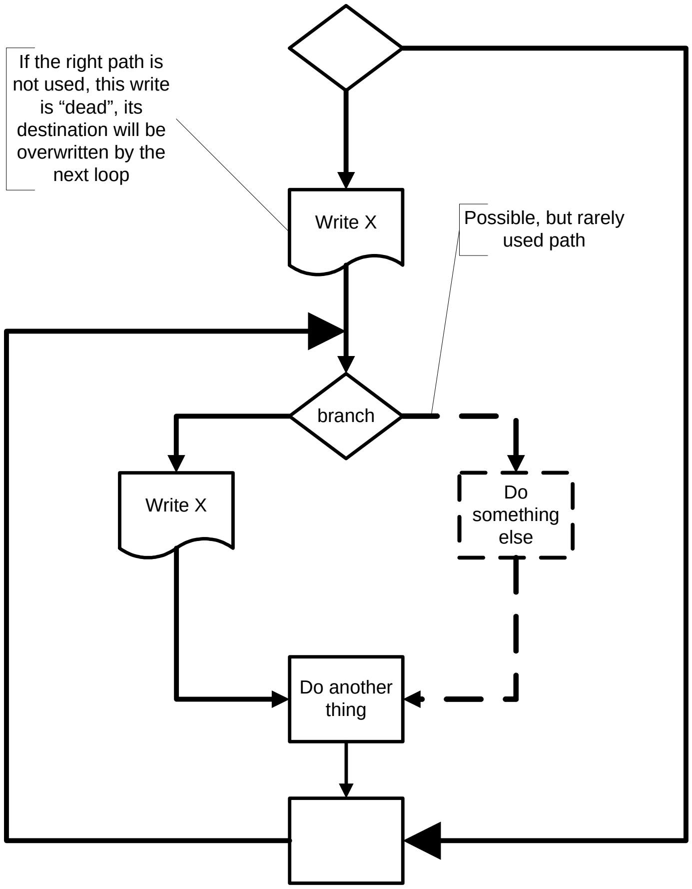
<- this operation is the same in every iteration of the loop. However, can’t place it after line 1 because loop might never run if n=0

gcc and other compilers do the following:

```
1    Y = val
2    if(n>0) {
3         Do {
4            X = sqrt(Y)
5            Do something with X,i
6            i = i + 1
7         } while (i<n)
8    }
```

<- now we can place this line after line 2

Example: WAW optimization of a trace



The compiler does not know that the right path is rarely used, so it cannot remove the first Write X instruction. The typical path, however, can be recorded from the ROB, stored and be used if the PC of the beginning of this trace is fetched again.

Therefore, the fetch unit can get the instruction stream from two different places: from the ICache or from the trace cache

When to chop a trace:

- Branch misprediction
- Trace length exceeds cache line size
- At branches that are hard to predict

The trace goes into an optimizing state machine (which we will call "optimizer") and will finally be saved in the trace cache.
Predictable branches will be turned into assertions: Like a branch, on execution, the condition of an assertion will be tested, but in case of a misprediction the fetch unit will not start fetching at the PC of the branch/assertion but at the beginning of that trace which contained the misprediction. This means we maintain a valid RAT representing the state of the machine at the beginning of a trace, and stores within that trace are not allowed to retire until the end of the trace is reached without any mispredictions.

Even without having an improver, this concept still boosts performance: In a CISC-like ISAs like x86, the fetch unit decodes the instructions into several micro-ISA RISC-like instructions. The trace cache of the P4, for example, only stores the decoded instructions (without replacing branches with assertions), which still offers a performance improvement.

The longer a trace remains in the improver, the more improvement can be done with the code. However, if you allow only several 100 cycles for optimization instead of several 10,000, the decoded/optimized trace is available at an earlier point of time. As long as a trace remains in the improver, the execute unit is still using the not yet optimized code from the ICache.

What can an improver do?

A “lifeness analysis” can remove “dead code” (e.g., the first write that causes a WAW).

- Move backwards over the trace.
- At start, all registers are “life”
- For each instruction {

```
    If you find that its destination register is “dead”: {
        mark it for removal
        }
    Else {
        Mark destination register as “dead”
        Mark source registers as “live”
        }
}
```

Example:
Use one bit for each register and apply the above rules on every instruction of the trace, starting at the end of a trace (1=life, 0=dead)

ADD R15 <-R15, R03

MUL R14 <-R15, R06