Why does the Alpha AXP predict a coroutine transfer the way it does?

devblogs.microsoft.com/oldnewthing/20191121-00

November 21, 2019

Raymond Chen

I noted some time ago that the Alpha AXP has a dedicated *branch to coroutine* instruction. The behavior of this instruction is to branch to the address held in the specified register, but the interesting part is how the instruction is predicted: The processor predicts a branch to the address at the top of the return address predictor stack, and the value at the top of the return address predictor stack is replaced by the address of the instruction that follows the `JSR_CO`.

Why does the processor use this algorithm? Why doesn't it just push the address of the following instruction onto the return address predictor stack?

Let's illustrate with an example. Note that the entire issue of `JSR_CO` applies only to stackful coroutines. Stackless coroutines use normal `JSR` subroutine calls.

Consider two coroutines that call each other. On entry to the first coroutine, the return address predictor stack contains the return address predictions for the code that will run once the first coroutine returns.

… X2 X1

The first coroutine does some stuff and then calls the second coroutine. The processor incorrectly predicts a transfer to `X1`. This initial misprediction is unavoidable because there was no opportunity to put it on the predictor stack in the first place. The processor then puts the first coroutine's resumption address onto the top of the predictor stack, replacing `X1`.

… X2 C1

The second coroutine runs for a while, and then transfers control back to the first coroutine. This time, the processor correctly predicts a transfer to `C1`. It then puts the second coroutine's resumption address onto the return address predictor stack, replacing `C1`.

… X2 C2

The first coroutine does some stuff, say it calls a helper function that returns. The call is a normal `JSR`, so it pushes `C1b` onto the return address predictor stack, and the helper function returns with a normal `RET`, which pops the correctly-predicted address off the stack. In general, normal non-coroutine calls are predicted in the usual way, and when control is in the first coroutine, the pushes and pops on the predictor stack cancel out, and the return address predictor stack still looks like this:

… X2 C2

The first coroutine now calls back into the second coroutine, which resumes where it left off, namely at `C2`, which also matches the predicted return address. The resumption location in the first coroutine replaces `C2`, leaving

… X2 C1c

And so on. Control transfers back and forth between the two coroutines, and their resumption locations take turns at the top of the predictor stack. Let's say that the second coroutine is now finished, so it transfers control to the first coroutine for the last time.

… X2 C2z

Here, `C2z` is a resumption address for the second coroutine, but it will never be used since the coroutine is finished. (The processor doesn't know that, so it puts it onto the return address predictor stack anyway.)

The first coroutine now returns to its caller, which was `X1`, but that return address was lost to the predictor stack ages ago. The return is mispredicted.

… X2

But the bad state lasts for only one frame. When the caller returns to its own caller, the return address `X2` will be right there at the top of the return address predictor stack, and things are back to normal.

Now, since these are stackful coroutines, there's no requirement that the transfer back to the first coroutine happen at the top level of the second coroutine. In the case where the second coroutine transferred back to the first coroutine from a subroutine, the return address predictor will not only mispredict the transfer back to the first coroutine, but it'll also have the other return addresses, too, corresponding to activation frames still active in the second coroutine.

… X2 C2c C3 C4

This prediction stack will pay off if the first coroutine transfers back to the second coroutine, since it will resume with a stack whose `C2` through `C4` entries exactly match the activation frames.

In general, the algorithm for managing the return address predictor stack works well if a coroutine transfers out of its stack at the same frame that it transferred in. If it transfers out in a nested call, then the predictor won't work, but there wasn't much you could have done about them anyway.

You're also out of luck if control cycles among three or more coroutines. Again, there wasn't much you could have done about this either.

Raymond Chen

Follow