CSE470 Final Project: Loop Termination Buffer

Milin Kodnongbua

May 2021

1 Introduction

Accurate branch prediction is important in pipelined processor as it reduces the downtime when miss-speculation occurs. This project focuses on predicting the termination of inner loops with a constant number of iterations. The code sample below shows the inner loop `j` of size 3 which will be repeated 10 times. Typical 2-bit BHT predictor will always predict the branch will be taken (the loop doesn't end) which makes this predictor mispredicts every time this inner loop terminates.

```
int test_loop() {
  int x = 0;
  for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 3; j++) {
      if (j < 100) {
        x++;
      }
    }
  }
  return x;
}
```

Listing 1: Inner loop example

We implemented the branch predictor that predicts the branch pattern $(1^N0)^M$ where N is the inner loop size and M is the outer loop size, and successfully integrated to the BlackParrot processor. The work is heavily based on a paper *Loop Termination Prediction* by Timothy Sherwood and Brad Calder [1].

2 Structure

Figure 2.1 shows a brief structure of the loop termination buffer (LTB). It contains multiple entries containing the loop information and current counter of the loop. The tag contains the hash of the address of the branching instruction. Loop trip count (Trip Cnt) stores the size of the loop e.g., the pattern 1110^N would have the trip count set to 3. The confidence

bit (Conf) signifies if the trip count is valid. The loop iteration counters store the number of times the branch has been taken since it was last non-taken. The speculative counter (Spec Cnt) counts as we make the prediction. The non-speculative counter (Non-Spec) counts as the branch has been resolved.

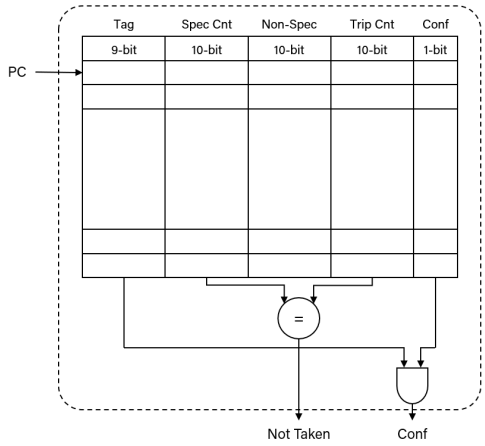

Figure 2.1: Loop Termination Buffer

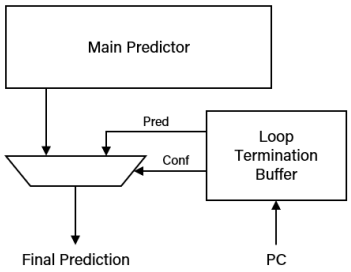

Figure 2.2: Integration to the main predictor

Figure 2.2 shows the loop termination buffer being integrated with the main predictor to make a final prediction. The LTB overrides the prediction if it is confident. (Also see Listing 12)

3 Algorithm

In this section, we describe how the LTB updates itself and how it makes predictions. The corresponding SystemVerilog code can be found in Listings 6 and 8.

3.1 Branch Resolution

We now describe how the LTB updates when it gets the prediction results (i.e., whether it made the correction prediction).

If a branch is mispredicted and it should not be taken, we insert it to the LTB with all the fields being zero.

If the branch is already in the LTB, we consider the following. If the branch is taken, we increse the non-speculative counter by 1. If in addition that branch is mispredicted, we set the speculative counter to the non-speculative counter. If the branch is not taken, we set the confidence bit if the non-speculative counter equals to the trip counter, we set the trip counter to be the non-speculative counter, we decrease the speculative counter by the non-speculative counter, and we set the non-speculative counter to 0.

3.2 Branch Prediction

During a branch prediction, we do the following. We check if the branch is in the LTB. If not, we ignore it. Otherwise, we increase the speculative counter by 1 and modulo by the trip count. We predict the branch is not-taken (signifying the end of loop) if the speculative counter is 0, and the branch is taken otherwise. The LTB is confident when the address matches the tag, and the confidence bit is set.

4 BlackParrot Integration

4.1 Design Choice

In BlackParrot, there are two fetching stages. However, the branch prediction is limited to only one stage. The speculative counter is implemented using flip-flops because two entries can be changed during one cycle. It is incremented during branch prediction and can be updated/reset during branch resolution. Other fields are implemented using an asynchronous memory with 2 read ports (see Listing 9). We use asynchronous memory because we require previous counter values to add one to it in one cycle, and we require 2 read ports because the branch prediction also requires the trip count information to predict the branch direction.

4.2 Design Complications

Since there are two fetching stages in BlackParrot, when there is a rollback due to misprediction or queue not being ready, we fetch the last two addresses again.

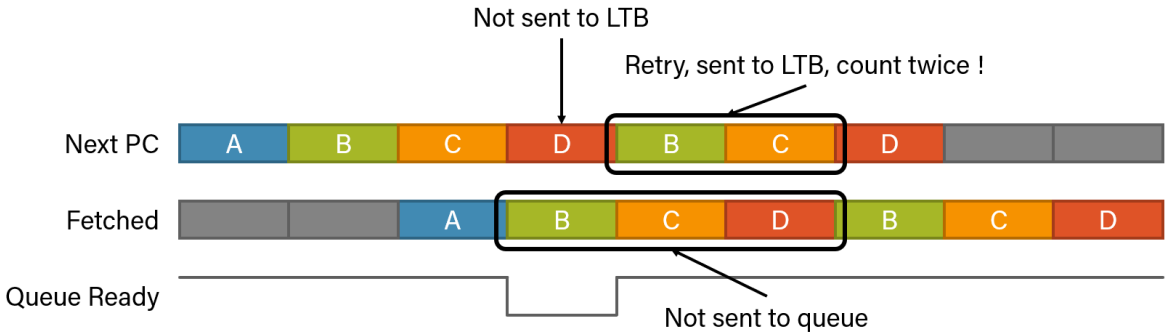

Figure 4.1: Duplicate requests to LTB

In Figure 4.1, the queue is not ready when we are fetching address D. We have to rollback and retry fetching address B and C. This causes the speculation counter of B and C be counted twice. To prevent that, there is a flag to the LTB signifying that this is a retrial, so it doesn't update the counter and gives the same prediction result.

5 Experiments and Results

We tested our implementation on the three sample codes below, the inner loop example (Listing 1), and CoreMark.

```
int test_loop() {
  int x = 0;
  int y = 0;
  for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 4; j++) {
      x++;
    }
    y++;
    for (int j = 0; j < 7; j++) {
      if (j < 100) {
        y++;
      }
    }
  }
  return x + y;
}
```

Listing 2: Two Inner Loop (Inner2)

```
int test_loop() {
  int x = 0;
  for (int k = 0; k < 12; k++) {
    for (int i = 0; i < 13; i++) {
      for (int j = 0; j < 5; j++) {
        x += j;
      }
    }
  }
  return x;
}
```

Listing 3: Three Nested Loop (Nested3)

```
int test_loop() {
  int acc;
  for (int i = 0; i < DIM_SIZE; i++) {
    for (int j = 0; j < DIM_SIZE; j++) {
      acc = 0;
      for (int k = 0; k < DIM_SIZE; k++) {
        acc += input1_data[i * DIM_SIZE + k]
            * input2_data[k * DIM_SIZE + j];
      }
      out_data[i * DIM_SIZE + j] = acc;
    }
  }
}
```

Listing 4: Matrix Multiply (MatMul)

Figure 5.1 shows the branch mispredictions of with and without LTB. The 'Improve by LTB' counts branches that are correctly predicted because of LTB and incorrectly in the baseline. Likewise, the 'Worsen by LTB' counts branches that are incorrectly predicted because of LTB but are correctly predicted before. The result shows the LTB works very well under the sample codes that were expected to be improved.

Further investigation shows that the LTB takes 3 outer loop iterations to learn the length of the inner loop. It then correctly predicts the loop termination afterwards. For example, Listing 1 has 10 outer loop iterations and 3 outer loop iterations. The LTB is not confident during the first 3 outer loop iterations, but is confident and correct in the last 7 iterations.

Hence, the 7 misprediction reduction by the LTB. Similarly, the improvement for Inner2 is $2 \cdot (10 - 3) = 7$. For Nested3 and MatMul, the LTB correctly predicts both the two inner loops. Hence, the improvement for Nested3 is $(12 - 3) + (12 \cdot 13) - 3 = 162$; and for MatMul is $(16 - 3) + (16 \cdot 16) - 3 = 266$. The small discrepancy between this calculation and the actual result shown in the table is yet to be investigated, but our hypothesis that it is due to flow changes similar to what is described in Section 4.2 that we didn't address.

The LTB provides a slight improvement to the branch prediction in CoreMark. We didn't inspect the proportion of the branches that are inner loops, but we suppose the LTB mostly identify the inner loops in the 3,437 reduction. However, we are still unsure about the 464 mispredictions attributed to the LTB.

	Listing 1	**Inner2**	**Nested3**	**MatMul**	**CoreMark**
# Branch	87	217	1,123	5,160	265,996
Misprediction					
Baseline	19	31	179	290	34,838
LTB	12	17	20	23	31,285
Improve by LTB	7	14	162	267	3,437
Worsen by LTB	0	0	3	0	464
% of Branch					
Baseline	21.8	14.3	15.9	5.6	13.1
LTB	13.8	7.8	1.8	0.4	11.8
Improve by LTB	8.0	6.5	14.4	5.2	1.3
Worsen by LTB	0.0	0.0	0.3	0.0	0.2

Figure 5.1: Branch misprediction with and without LTB

Bottom half of the table shows the percentage over the number of branches in each program. However, using percentange is inaccurate because the number of mispredictions the LTB can identify depends on the number of iterations of the outer loops. For example, if the outer loops of the Listing 1 runs 1,000 times. The LTB would correcly predict the inner loop termination 997 times.

6 Discussion

Since we didn't explore what the proportion of particular branches is in the the typical software, it is unclear how useful it will be in the real world.

The current implementation of LTB has 8 entries without associativity, so it is prone to address collisions which would waste the potential improvement. The example code was manually adjusted so that no two loops map to the same entry in the LTB just to illustrate the effect of the LTB not worrying about the collisions. A typical software wouldn't have

too many outstanding loops, so LTB will greatly benefited from having more associativity rather than more entries.

Lastly, using an asynchronous memory could mean a slower pipeline, and using flip-flops could require more chip area than the potential benefit it generates.

Nevertheless, the loop termination buffer (LTB) is a great and accurate tool to predict the branch of pattern $(1^N0)^M$.

7 Remark

Our fork of BlackParrot repository is located at `https://github.com/milmillin/black-parrot`. The appendix also contains some relevant part of the code.

References

[1] Timothy Sherwood and Brad Calder. Loop termination prediction. In Mateo Valero, Kazuki Joe, Masaru Kitsuregawa, and Hidehiko Tanaka, editors, *High Performance Computing*, pages 73–87, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

8 Appendix

8.1 bp_fe_ltb.sv Excerpt

```
module bp_fe_ltb
 import bp_common_pkg::*;
 import bp_fe_pkg::*;
 #(parameter bp_params_e bp_params_p = e_bp_default_cfg
   `declare_bp_proc_params(bp_params_p)
   )
  (input                                clk_i
   , input                              reset_i

   , output logic                       init_done_o

   // Synchronous read
   , input                              r_v_i
   , input [vaddr_width_p-1:0]          r_addr_i
   , input                              r_retry_i      // won't write
   , output logic                       pred_v_o
   , output logic                       pred_conf_o
   , output logic                       pred_taken_o

   // Synchronous write
   , input                              w_v_i              // branch
   , input                              br_mispredict_i
   , input                              br_taken_i
   , input [vaddr_width_p-1:0]          br_src_addr_i
   , output logic                       w_yumi_o
   );
```

Listing 5: LTB module definition

```
  for (genvar i = 0; i < ltb_els_lp; i++)
    begin : spec_counters
      logic mispred_not_taken = w_v_i & ~rw_same_addr & br_mispredict_i &
                                ~br_taken_i & w_idx_one_hot[i];
      logic mispred_taken     = w_v_i & ~rw_same_addr & br_mispredict_i &
                                 br_taken_i & w_idx_one_hot[i];

      logic [ltb_cnt_width_p-1:0] spec_cnt_n;
      always_comb begin
        if (reset_i | mispred_not_taken)
          spec_cnt_n = '0;
        else if (mispred_taken)
          spec_cnt_n = non_spec_cnt_plus1;
        else if (w_v_i & ~rw_same_addr & ~br_taken_i & w_idx_one_hot[i])
          spec_cnt_n = spec_cnt_r[i] - non_spec_cnt_plus1;
        else if (r_v_i & ~r_retry_i & r_idx_one_hot[i] & r_tag_match)
   begin
          if (tag_mem_r0_data_lo.conf & (spec_cnt_r[i] ==
   tag_mem_r0_data_lo.trip_cnt))
            spec_cnt_n = '0;
```

```
          else
            spec_cnt_n = spec_cnt_r[i] + 1;
        end else
          spec_cnt_n = spec_cnt_r[i];
      end

      bsg_dff #(.width_p(ltb_cnt_width_p))
        spec_counter
        (.clk_i(clk_i)
         ,.data_i(spec_cnt_n)
         ,.data_o(spec_cnt_r[i])
         );
    end
```

Listing 6: Generation of speculative counters

```
typedef struct packed
{
  logic [ltb_tag_width_p-1:0] tag;
  logic [ltb_cnt_width_p-1:0] non_spec_cnt;
  logic [ltb_cnt_width_p-1:0] trip_cnt;
  logic                       conf;
  logic                       overflow;
}  bp_ltb_entry_s;
```

Listing 7: Entry struct in the memory

```
always_comb begin : w_data_li
  if (is_clear)
    tag_mem_data_li = '0;
  else begin
    if (br_taken_i)
      // Increase non_spec_cnt
      tag_mem_data_li = '{
        tag: w_tag_li
        ,non_spec_cnt: non_spec_cnt_plus1
        ,trip_cnt: tag_mem_r1_data_lo.trip_cnt
        ,conf: tag_mem_r1_data_lo.conf
        ,overflow: tag_mem_r1_data_lo.overflow | non_spec_cnt_ovf
      };
    else begin
      if (br_mispredict_i & ~w_tag_match)
        // Insert to LTB
        tag_mem_data_li = '{
          tag: w_tag_li
          ,non_spec_cnt: '0
          ,trip_cnt: '0
          ,conf: '0
          ,overflow: '0
        };
      else
        // Reset non_spec_cnt
        tag_mem_data_li = '{
          tag: w_tag_li
```

```
          ,non_spec_cnt: '0
          ,trip_cnt: tag_mem_r1_data_lo.non_spec_cnt
          ,conf: (tag_mem_r1_data_lo.non_spec_cnt != 0)
                  & (tag_mem_r1_data_lo.non_spec_cnt ==
 tag_mem_r1_data_lo.trip_cnt)
                  & ~tag_mem_r1_data_lo.overflow
          ,overflow: '0
        };
    end
  end
end
```

Listing 8: Combinational logic for non-speculative fields

```
bsg_mem_2r1w
 #(.width_p($bits(bp_ltb_entry_s)), .els_p(ltb_els_lp))
 tag_mem
  (.w_clk_i(clk_i)
   ,.w_reset_i(reset_i)

   ,.w_v_i(tag_mem_w_v_li)
   ,.w_addr_i(tag_mem_w_addr_li)
   ,.w_data_i(tag_mem_data_li)

   ,.r0_v_i(tag_mem_r0_v_li)
   ,.r0_addr_i(tag_mem_r0_addr_li)
   ,.r0_data_o(tag_mem_r0_data_lo)

   ,.r1_v_i(tag_mem_r1_v_li)
   ,.r1_addr_i(tag_mem_r1_addr_li)
   ,.r1_data_o(tag_mem_r1_data_lo)
   );
```

Listing 9: Asynchronous memory declaration

```
assign w_yumi_o = is_run & w_v_i & ~rw_same_addr;

// r_v_i reg
logic                        r_v_r;
logic                        r_tag_match_r;
logic                        r_conf_r;
logic [ltb_idx_width_p-1:0] r_idx_r;
bsg_dff_reset
 #(.width_p(3+ltb_idx_width_p))
 r_v_reg
  (.clk_i(clk_i)
   ,.reset_i(reset_i)
   ,.data_i({r_v_i, r_tag_match, r_idx_li, tag_mem_r0_data_lo.conf})
   ,.data_o({r_v_r, r_tag_match_r, r_idx_r, r_conf_r})
  );

assign pred_v_o            = r_v_r & r_tag_match_r;
assign pred_conf_o         = r_conf_r;
assign pred_taken_o        = ~(spec_cnt_r[r_idx_r] == 0);
```

Listing 10: Output port assigmnents

8.2 bp_fe_pc_gen.sv Excerpt

```
// LTB
wire ltb_r_v_li = next_pc_yumi_i & ~ovr_taken & ~ovr_ret;
wire ltb_r_retry_li = redirect_v_i | redirect_v_r;

wire ltb_w_v_li = next_pc_yumi_i
    & ((redirect_br_v_i & ~redirect_br_nonbr_i)
      | (attaboy_v_i));
wire ltb_mispredict_li = redirect_br_v_i & ~redirect_br_nonbr_i;
wire ltb_taken_li =
  (redirect_br_v_i & ~redirect_br_nonbr_i & redirect_br_taken_i)
  | (attaboy_v_i & attaboy_taken_i);
wire [vaddr_width_p-1:0]   ltb_src_addr_li     = br_metadata_fwd.
 src_vaddr;

wire ltb_init_done_lo;
wire ltb_v_lo;
wire ltb_conf_lo;
wire ltb_taken_lo;
wire ltb_w_yumi_lo;

if (ltb_enabled_p) begin
  bp_fe_ltb
  #(.bp_params_p(bp_params_p))
  ltb
    (.clk_i(clk_i)
    ,.reset_i(reset_i)
    ,.init_done_o(ltb_init_done_lo)

    ,.r_v_i(ltb_r_v_li)
    ,.r_retry_i(ltb_r_retry_li)
    ,.r_addr_i(next_pc_o)
    ,.pred_v_o(ltb_v_lo)
    ,.pred_conf_o(ltb_conf_lo)
    ,.pred_taken_o(ltb_taken_lo)

    ,.w_v_i(ltb_w_v_li)
    ,.br_mispredict_i(ltb_mispredict_li)
    ,.br_taken_i(ltb_taken_li)
    ,.br_src_addr_i(ltb_src_addr_li)
    ,.w_yumi_o(ltb_w_yumi_lo)
    );
end else begin
  assign ltb_init_done_lo = '0;
  assign ltb_v_lo = '0;
  assign ltb_conf_lo = '0;
  assign ltb_taken_lo = '0;
  assign ltb_w_yumi_lo = '0;
end
```

Listing 11: LTB module declaration

```
if (ltb_enabled_p) begin
```

```
  // override prediction
  assign btb_taken = (ltb_v_lo & ltb_conf_lo) ?
    (ltb_taken_lo ? btb_br_tgt_v_lo : 0) :
    (btb_br_tgt_v_lo & (bht_val_lo[1] | btb_br_tgt_jmp_lo));
end else
  assign btb_taken = btb_br_tgt_v_lo & (bht_val_lo[1] |
 btb_br_tgt_jmp_lo);
```

Listing 12: Final prediction integration