ABSTRACT

VASTRAD, ADITH SAGAR. Verilog Design and Verification of an Application Specific Branch Predictor for astar Benchmark. (Under the direction of Dr. Eric Rotenberg).

This thesis effort builds upon the work previously done by our research group on *Custom EXACT Branch Predictor for astar Benchmark*[1]. It had demonstrated the potential gains of designing a custom predictor for astar benchmark, with the help of a C++ simulator. In this thesis, we design the predictor in Verilog with well defined interfaces, verify its functionality and integrate it with the AnyCore[2] processor.

This work comes as the next steps in realizing a real hardware design that fits into the Post silicon microarchitecture (PSM)[3] paradigm. PSM leverages a re-configurable FPGA fabric to dynamically instantiate microarchitecture modules to help improve the application performance. It interacts with the processor core through a well defined interface to observe and intervene as necessary. This predictor, with its well defined interfaces fits perfectly into the PSM design paradigm. It can be used to improve the branch prediction accuracy and performance of the astar benchmark.

Detailed analyses of the data path, control signals and structures used in the predictor design are presented. We also look at the verification methodologies used to verify the functional correctness of the predictor. Changes made in the AnyCore RTL to integrate the predictor have also been presented. The branch prediction statistics and performance statistics obtained from running the astar benchmark checkpoint on the AnyCore processor are also presented. Finally, timing and area results obtained from synthesizing the design are presented.

Verilog Design and Verification of an Application Specific Branch Predictor for astar Benchmark

by
Adith Sagar Vastrad

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina
2019

APPROVED BY:

Dr. Huiyang Zhou	Dr. James Tuck

Dr. Eric Rotenberg
Chair of Advisory Committee

BIOGRAPHY

Adith was born in the year 1993, in the state of Karnataka in India. He earned his Bachelor's degree in Electronics and Instrumentation Engineering from Birla Institute of Technology and Science, Pilani - K.K. Birla Goa Campus, India, in 2015. He then worked as an ASIC Engineer at Nvidia Bangalore, India for close to two years, before joining NC State University in Fall 2017. With the defense of this thesis, Adith will receive his Master of Science degree in Computer Engineering.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my parents Jayashree and Shankar Vastrad for their continuous support and encouragement. Their valuable life lessons have had an immense impact on my life.

I would like to thank my advisor, Dr. Eric Rotenberg for supporting and guiding me throughout the duration of this thesis. I have learned a great deal from the discussions I have had with him and also from his graduate courses. I would also like to acknowledge Dr. Huiyang Zhou and Dr. James Tuck for agreeing to be on my committee and supporting me with my work.

I would like to thank my peers and friends in the research group: Aayush Chaudhary, Mohit Karandikar, Saransh Jain, Utkarsh Mathur for always willing to discuss and help me when in need. A special mention to Aayush for his help in debugging issues with the predictor design. I am also grateful to my friends Mansi Jain and Vijayalakshmi Sundar for being part of my Master's journey.

This thesis was supported in part by NSF grant no. CCF-1823517, and Intel. Any opinions, findings and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation or Intel.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Modern processors boost application performance by exploiting instruction level parallelism (ILP). They use large buffers (reorder buffer, issue queue, etc.) to store instructions and find independent instructions that may be executed out of order. Large window sizes increase the chance of finding more independent instructions ready to be executed. Thus, the front-end (fetch stage) needs to feed these hungry buffers with as many *good instructions* as possible. Instructions such as conditional branch, call, return, etc. change the control flow of the program execution. Branch predictors are used to predict the branch outcome at the fetch stage and continue fetching instructions from the predicted target. Thus, highly accurate predictors play an important role in feeding *good instructions* to the back-end stages.

1.1 Need for Application Specific Predictors

State of the art branch predictors such as L-TAGE[4], that use large number of history bits, are shown to be accurate predictors for general purpose applications. The primary role of history bits is to distinguish dynamic instances of branch instructions. However, even large number of history bits fail in this regard when it comes to certain types of branches or applications.

One such example is a load dependent branch. Each dynamic instance of the load instruction feeding the branch may be loading the value from a different address. The branch outcome may change depending upon the loaded value. History bits alone can not distinguish these load instances. Additionally, stores to these addresses may change

the outcome of future branches. Since most predictors are passively trained, i.e., trained only after the next occurrence of the branch, such intervening stores may result in higher misprediction rates. These problems are overcome by the EXACT[5][6] predictor. It is an auxiliary predictor that specifically targets load dependent branches.

1.2 Post Silicon Microarchitecture

Many proposed novel microarchitectures such as CLEAR[7], EXACT, Slipstream processor[8], perform extremely well for certain applications and do not add any value for the rest of the applications. Many commercial processor designers see this as a big cost to pay in silicon for a small gain, that is restricted to specific applications.

Post Silicon Microarchitecture[3] (PSM) is a design paradigm in which such novel microarchitectures can be instantiated on a need or application specific basis. It uses an FPGA fabric and a well defined interface with the processor core to achieve this. Thus, it not only overcomes the cost of hardening the design in silicon but also leverages the advantages of application specific microarchitectures.

This thesis describes the design of a branch predictor that is highly specific to a particular program region of the astar benchmark. Thus, it is a perfect candidate that can be mapped to the PSM fabric.

1.3 Background

This thesis builds upon the work done on *Custom EXACT Branch Predictor for astar Benchmark*[1] by Chaudhary. They propose a highly customized branch predictor that closely mimics the astar program execution pattern. They demonstrate the potential performance gain by implementing the predictor in an in-house C++ simulator. The prediction techniques, structures, predictor repair and update mechanisms proposed by them are refined to suit the Verilog implementation.

1.4 Contributions

This thesis describes the Verilog design of the predictor mentioned in Section 1.3 and integrates the predictor with the AnyCore[2] processor. It also demonstrates that the design is synthesizable and thus can be mapped to an FPGA fabric. The overall contributions of this thesis include:

1. Verilog design of branch predictor for astar benchmark.
2. Functional verification of the branch predictor.
3. Define the interface of the predictor module.
4. Use the well defined interface to integrate the predictor with the AnyCore processor and demonstrate its functionality.
5. Synthesize the design and present timing and area reports.
6. Demonstrate performance improvement achieved through AnyCore RTL simulations.

Chapter 2

RELATED WORK

2.1 Post Silicon Microarchitecture

The Verilog design presented in this thesis will eventually be mapped onto the PSM[3] fabric. The reconfigurable fabric proposed in ReMAP[9] is closely related to PSM. The reconfigurable fabric is coupled with a cluster of cores. Specific portions of program are parallelized and accelerated in the fabric. However, this also involves making changes at the assembly code level. PSM on the other hand, dynamically observes and intervenes the core to enable parallelization or any other application specific novel microarchitectures.

2.2 EXACT and Custom astar Predictors

The EXACT[5][6] predictor is specifically designed to predict load dependent branches. It uses load address information to distinguish among dynamic instances of a load dependent branch. This distinction helps in reducing aliasing affects caused while indexing the predictor tables and improves prediction accuracy. The EXACT predictor also monitors stores at the retire stage and actively updates the branch outcome in the predictor tables using the store address and store value.

The custom astar branch predictor proposed by Chaudhary[1], is the basis for this thesis. It also leverages the techniques proposed in EXACT. It uses two predictor tables to predict two different branch program counters (PCs). It also performs active updates by monitoring stores at the retire stage. The detailed explanation of this predictor is presented in Section 3.5.

Chapter 3

astar CHARACTERIZATION

In this section we will look at the astar benchmark in detail and the high level working of the predictor as proposed by Chaudhary[1]. astar is part of the SPEC CPU 2006[10] benchmark suite. It is a path-finding algorithm that looks at neighboring elements in a graph to determine the path. In the following sections we will look at code snippets and the branches of interest that are predicted by this design.

3.1 makebound2() Function

The astar benchmark spends a significant part of its execution time in the *makebound2()* function, as measured by running gprof on a linux server[11]. Figure 3.1 shows the code snippet of the *makebound2()* function. The function uses *bound1p* as the input worklist and *bound2p* as the output worklist. For each iteration of the loop in Figure 3.1 line 13, the *index* represents an element in a 2D array. The eight instances of *index1* represent the eight neighboring elements surrounding the *index* element.

3.2 Branches of Interest

The predictor is used to predict the *fillnum* branch on Figure 3.1 line 18, *maparp* branch on Figure 3.1 line 19, and *endindex* branch on Figure 3.1 line 27. The seven other similar instances of these branches are also predicted.

```
 1: i32 wayobj::makebound2(i32pt bound1p, i32 bound1l, i32pt bound2p)
 2: {
 3:   i32 bound2l;
 4:   i32 index,index1;
 5:   i32 yoffset;
 6:   waymapcellpt waymap;
 7:   i32 i;
 8:
 9:   yoffset=maply;
10:   waymap=wayobj::waymap;
11:
12:   bound2l=0;
13:   for (i=0; i<bound1l; i++)
14:   {
15:     index=bound1p[i];
16:
17:     index1=index-yoffset-1;
18:     if (waymap[index1].fillnum!=fillnum)
19:       if (maparp[index1]==0)
20:       {
21:         bound2p[bound2l]=index1;
22:         bound2l++;
23:
24:         waymap[index1].fillnum=fillnum;
25:         waymap[index1].num=step;
26:
27:         if (index1==endindex)
28:         {
29:           flend=true;
30:           return bound2l;
31:         }
32:       }
33:
34:     index1=index-yoffset;
35-49:  <Nested if>
50:
51:     index1=index-yoffset+1;
52-66:  <Nested if>
67:
68:     index1=index-1;
69-83:  <Nested if>
84:
85:     index1=index+1;
86-100: <Nested if>
```

Figure 3.1: makebound2() Source Code

3.3 Critical Store

It is possible that multiple *index* elements may have overlapping neighboring *index1* elements. Elements once visited will not be visited again. This is ensured by the critical store on Figure 3.1 line 24 and the *fillnum* branches.

3.4 Worklist Population

On closely observing lines 27-36 in Figure 3.2, we can see that *makebound2()* function is called in a ping-pong fashion. The output worklist generated by the current *makebound2()* call becomes the input worklist for the next call. The store instruction on line 21 in Figure 3.1 populates the output worklist. These stores are essentially *index1* elements that have been visited in the current *makebound2()* call.

3.5 Predictor Overview

In this section we present the high level overview of the custom predictor proposed by Chaudhary[1]. The predictor hardware essentially mimics the program execution pattern. As seen in Figure 3.3 it uses the following structures:

- Worklist0 and Worklist1 store the list of bound1p and bound2p *index* values
- Index1 array stores the eight *index1* values generated from the current *index* of the current worklist
- Fillnum table holds the predictions for fillnum branches
- Maparp table holds the predictions for maparp branches
- Active update queue holds the predictions corresponding to *index1* stores that are yet to be retired
- Active update unit from the retire stage populates output worklist and updates predictions in the fillnum table

Fillnum table and maparp table are direct mapped structures indexed by current *index1* values. Active update queue is a fully associative structure that has the ability to match a given *index1* value. A hit in the active update queue takes precedence over the prediction

```
 1: bool wayobj::fill(i32 startx, i32 starty, i32 endx, i32 endy)
 2: {
 3:   i32 boundl;
 4:   bool flodd;
 5:
 6:
 7:   if (fillnum==65535)
 8:   {
 9:     memset(waymap,0,((1<<mapxshift)<<mapyshift)*sizeof(waymap[0]));
10:     fillnum=0;
11:   }
12:   fillnum++;
13:
14:   bound1p[0]=index(startx,starty);
15:   waymap[starty<<shift|startx].fillnum=fillnum;
16:   waymap[starty<<shift|startx].num=0;
17:
18:   flodd=false;
19:   boundl=1;
20:
21:   flend=false;
22:   endindex=index(endx,endy);
23:   step=1;
24:
25:   while ((boundl!=0)&&(flend==false))
26:   {
27:     if (flodd==false)
28:     {
29:       boundl=makebound2(bound1p,boundl,bound2p);
30:       flodd=true;
31:     }
32:     else
33:     {
34:       boundl=makebound2(bound2p,boundl,bound1p);
35:       flodd=false;
36:     }
37:
38:     step++;
39:   }
40:
41:   return flend;
42: }
```

Figure 3.2: fill() Source Code

made by the fillnum table for a *fillnum* branch. As the predictor make predictions, *index1* values are popped from the array. Once all the *index1* values are exhausted, a new set is generated by popping the next *index* value in the current worklist. Once all *index* values are exhausted, it is time to switch out the current worklist and use the the other worklist as the new current worklist. A detailed analyses of the predictor datapath and control signals are presented in Chapter 4.

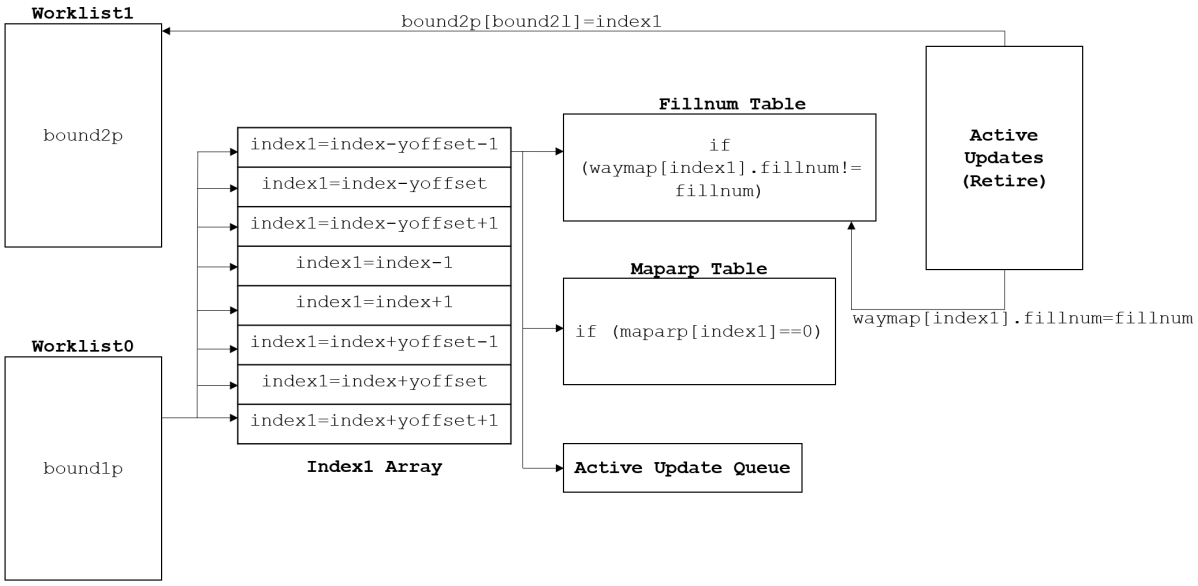

Figure 3.3: Custom Predictor Overview

Chapter 4

RTL IMPLEMENTATION

In this chapter we will take a deep dive into the predictor datapath and its control signals.

4.1 Module Interface

Table 4.1 describes all the input and output signals used in the predictor design.

Table 4.1: Predictor Module Interface Signals

Direction	**Signal Name**	**Width** (bits)
input	clk	1
input	reset	1
input	bound1p0_fet_vld	1
input	bound1p0_ret_vld	1
input	bound1p0_value	log2(BP_TABLE_SIZE)
input	wl_switch	1
input	cp_fix_pred_vld	1
input	active_update_vld	1
input	active_update_index	log2(BP_TABLE_SIZE)
input	inst_fet_vld	1
input	br_type	2
input	commit_cpq	1
output	stall	1
output	pred	3

Table 4.2: Predictor Sizes Description

Name	Value	Description
AQ_SIZE	16	Number of entries in active update queue
BP_TABLE_SIZE	2^{18}	Number of sets in *fillnum* and *maparp* tables
NUM_INDICES	8	Number of sets in index1 array
WL_SIZE	512	Number of sets in worklist fifos

Here is a brief explanation of all the predictor interface signals:

- *bound1p0_fet_vld* comes from the fetch stage and it being high indicates that the store responsible for populating the initial *bound1p[0]* value (Figure 3.2 line 14) is being fetched.
- *bound1p0_ret_vld* comes from the retire stage and it being high indicates that *bound1p[0]* store is being retired.
- *bound1p0_value* comes from the head of the store queue and contains the store value of *bound1p[0]* . This value is considered valid only when bound1p0_ret_vld is high.
- *wl_switch* comes from the retire stage and it being high indicates that the instruction responsible for switching out the current worklist is being retired. This indicates that the call to the *makebound2()* function is being retired.
- *cp_fix_pred_vld* comes from the retire stage and it being high indicates that the pipeline is being flushed and the predictor state needs to be repaired. The predictor state is obtained from the checkpoint queue maintained by the predictor.
- *active_update_vld* comes from the retire stage and it being high indicates that a critical store instruction is being retired.
- *active_update_index* is the *index1* value that is being stored. It comes from the head of the store queue and is valid only when *active_update_vld* is high.
- *inst_fet_vld* comes from the fetch stage and it being high indicates that an instruction is being fetched. When it is high, the current state of predictor is pushed into the checkpoint queue.

- *br_type* comes from the fetch stage and indicates the type of branch to be predicted. 0=*fillnum* branch, 1=*maparp* branch, 2=*endindex* branch, 3=others. It is valid only when *inst_fet_vld* is high.

- *commit_cpq* comes from the retire stage and it being high indicates that an instruction is being retired. When it is high, the head element of the checkpoint queue is popped.

- *stall* output signal when high indicates to the fetch stage of the core that it may need to stall fetching instructions. It goes high when the input worklist is empty and the last *index1* value has been used to make a prediction.

- *pred* encodes three predictions in one signal. *pred[2]* is always not taken and corresponds to *endindex* branch. *pred[1]* is obtained from indexing into the maparp table and corresponds to *maparp* branch. *pred[0]* is obtained from indexing into the fillnum table and corresponds to *fillnum* branch. As the predictor always provides a three bit prediction, the core must choose the appropriate prediction.

Figure 4.1, shows the interaction between the processor core stages and the predictor. A detailed description of it is presented in Section 4.2.

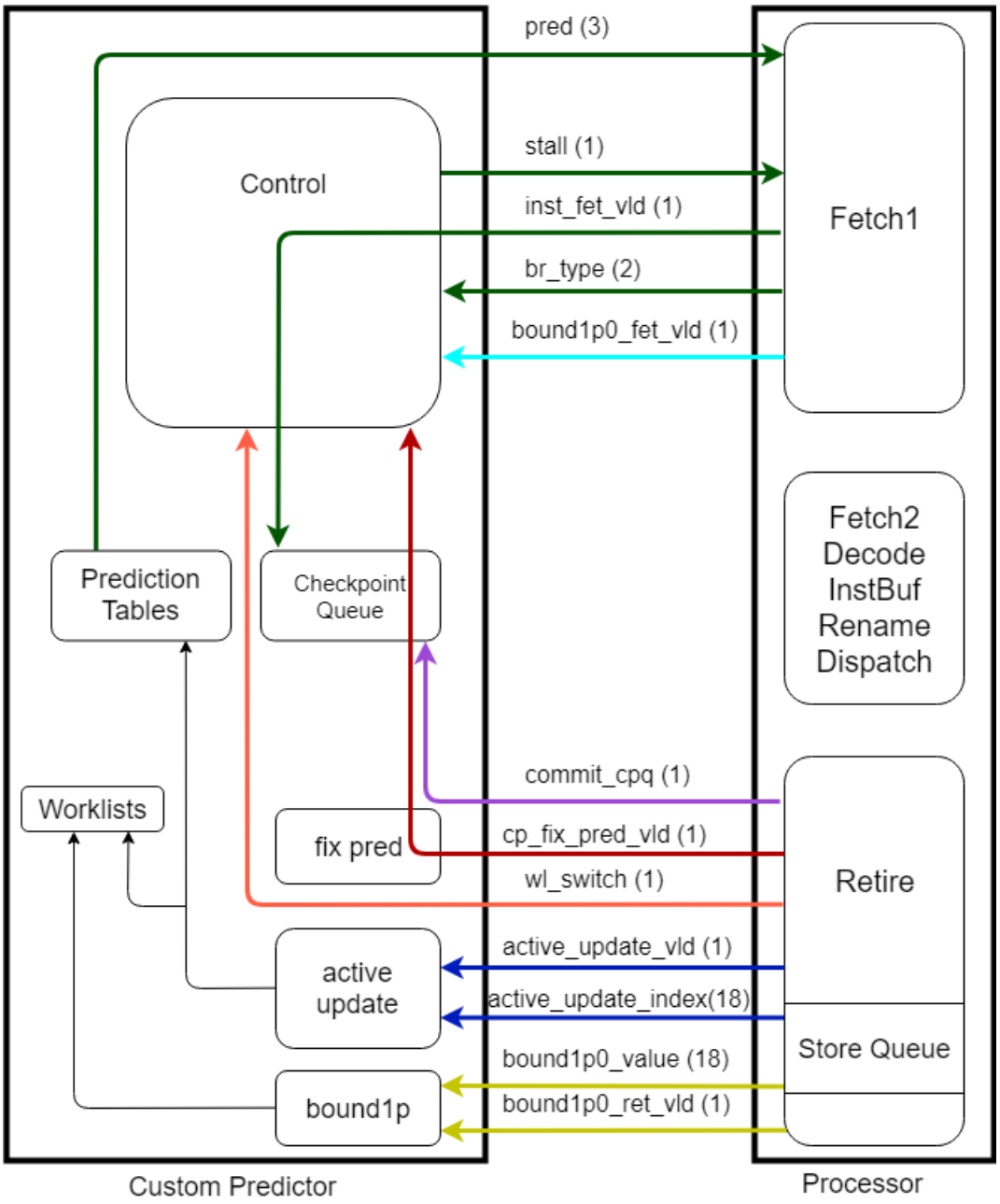

Figure 4.1: Predictor and Processor Core Interface

4.2 Datapath and Control Signals

This section describes datapath and control signals in detail.

4.2.1 Checkpoint Queue

The predictor maintains a queue of checkpoints that store the predictor state. The current state of the predictor is pushed into the queue when *inst_fet_vld* is high. The head element of the queue is popped when *commit_cpq* signal is high. If *cp_fix_pred_vld* is high, the head element of the queue is used to repair the state of the predictor. The contents of each checkpoint element in the queue is described in Table 4.3.

Table 4.3: Checkpoint Payload Description

Name	**Width**(bits)	**Description**
cp_pred	1	Prediction made by the predictor
cp_br_type	2	Type of branch predicted
cp_wl0_head	log2(WL_SIZE)+1	Current head pointer of worklist 0
cp_wl1_head	log2(WL_SIZE)+1	Current head pointer of worklist 1
cp_wl_select	1	Current input worklist
cp_index	log2(BP_TABLE_SIZE)	Current *index* value from the input worklist
cp_index1	log2(BP_TABLE_SIZE)	Current *index1* value among the 8 iterations
cp_index_iter	log2(NUM_INDICES)+1	Current iteration count among the 8 iterations
cp_aq_tail	log2(AQ_SIZE)+1	Current tail pointer of the active update queue

4.2.2 bound1p[0]

The *bound1p0_vld* fifo is reset to 0 when bound1p0_fet_vld is high. It is set to 1 when bound1p0_ret_vld is high. This fifo indicates that a valid bound1p[0] index value has been committed and the predictor is ready to make predictions.

4.2.3 curr_wl

The *curr_wl* fifo is 1 bit wide and maintains the current worklist from which predictions are to be made. If *cp_fix_pred_vld* signal is high, then the *curr_wl* is set to the checkpointed

worklist *cp_wl_select*. Else if *wl_switch* signal is high, the flip flop is inverted. Thus, the core needs to make sure this signal is high only for one cycle when it intends to switch worklists.

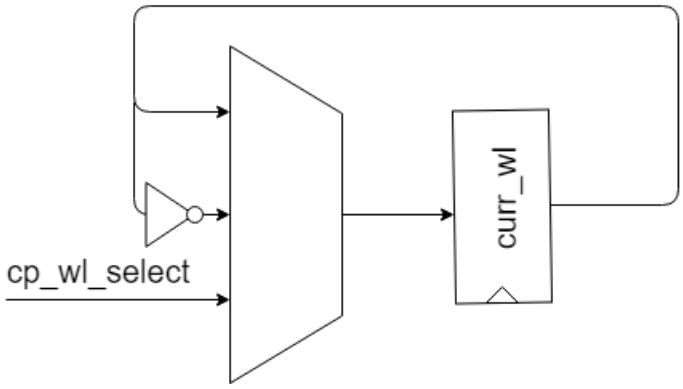

Figure 4.2: Current Worklist Datapath

4.2.4 index1_head

index1_head fifo represents the position of the *index1* value in the index1 array to be used for making predictions. It is four bits wide. The last three bits are used to address the index1_array. If *bound1p0_ret_vld* is high, *index1_head* is set to 8, indicating that all the eight neighboring elements have been exhausted. At the time of making a prediction, the index1_head is rolled over and increments till 8.

It needs to be repaired if *cp_fix_pred_vld* signal is high. If the mispredicted branch is *fillnum* or *maparp* and the true outcome is "taken", then the index1_head is updated to *cp_index_iter + 1*, else to *cp_index_iter*. This essentially means that none of the three branches will use this *index1* anymore and we need to move on to the next neighboring element.

Else if *wl_switch* signal is high, then the flip flop is set to 0, indicating the beginning of the first of the eight neighboring elements of the new worklist.

Else if the wire *inc_index1_head* is high, then the flip flop is incremented by 1. This

wire is high only when the current predictions are "taken" for either *fillnum* or *maparp* branches, or if it encounters an *endindex* branch. This essentially means that the current *index1* value has been used completely and we need to move on to the next neighboring element. This mechanism closely follows the program behavior seen in Figure 3.1 lines 18-32 and seven other replicas.

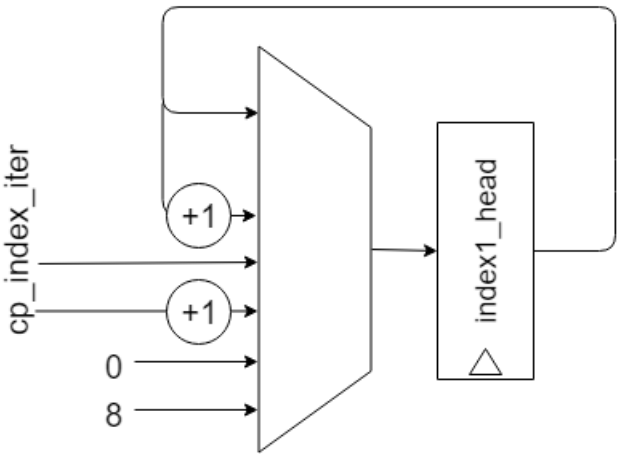

Figure 4.3: Current Index1 Array Head Datapath

4.2.5 index1 Array

This array has eight entries and is indexed by *index1_head*. Populating these entries correctly is extremely important for the predictor to do a good job. As seen in Figure 3.3, this array is populated by adding some constants (yoffset and 1) to *index* value. This value changes depending on the circumstances. Thus, we can think of *index* being driven by a MUX.

If *bound1p0_ret_vld* is high, then the MUX selects *bound1p0_value*. This happens at the beginning of the call to *fill()* function.

Another interesting case is when *cp_fix_pred_vld* is high, the index1 array needs to be repaired. If the true outcome of the checkpointed *fillnum* or *maparp* branch is taken and *cp_index_iter* was the last element, then the MUX selects the *index* pointed to by *cp_wl0/1_head* (shown as the unified signal *cp_wl_head_element* in Figure 4.4) from

the worklist, essentially popping the worklist element. This worklist is also obtained from the checkpoint signal *cp_wl_select*. This essentially means that we have checked all eight neighbors and need to move on the the next element in the input worklist. If this is not the case, then the MUX selects the checkpointed *cp_index*.

Else if at the time of switching worklists (*wl_switch* is high), the MUX selects the head element of the current output worklist.

Else if *index1_head* is the last element and the wire *inc_index1_head* is high, then MUX selects the head element of the *curr_wl*.

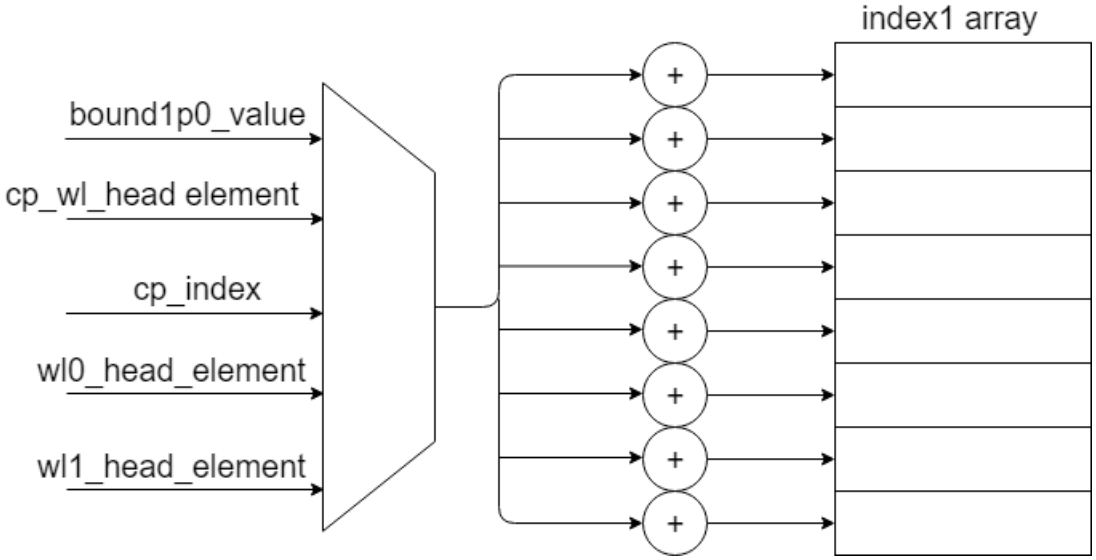

Figure 4.4: Index1 Array Datapath

4.2.6 curr_index

This flip flop maintains the value of the current *index* used to make the predictions. Basically, this flip flop is updated similarly to how index1 array is updated. If *index1_head* is the last element and the wire *inc_index1_head* is high, then the flip flop is updated with the head element of the *curr_wl*.

Else if *wl_switch* is high, the flip flop is updated with the head element of the current output worklist.

Else if *cp_fix_pred_vld* is high, the flip flop is updated with MUX output described in section 4.2.5.

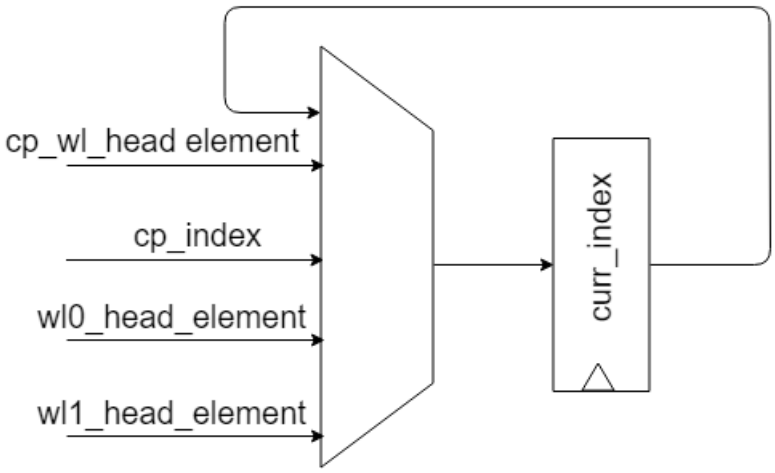

Figure 4.5: Current Index Datapath

4.2.7 Worklist FIFO Interface Signals

The worklist fifos work like a normal fifo for most part with one main difference. Along with maintaining a head and tail value, they also maintain the commit_head value. A push to the fifo adds an element at the tail. A pop from the fifo only moves the head, while the commit_head stays intact. It is moved only when the commit signal is high. Thus even though an element is "popped" from the fifo, it does not free the space. Only commits can free up space in the fifo. This ensures that the worklists do not get overwritten when they are popped, as they may need to be rolled back in case of predictor repairs. The fifo RAM can also be used to get an element by providing an address as input. Table 4.4 shows the signals used by the fifo design.

Here is a brief explanation of all the fifo interface signals:

- *clear*, when high, resets the commit_head=head=tail=0, emptying the worklist.

Table 4.4: Worklist FIFO Interface Signals

Direction	**Signal Name**	**Width** (bits)
input	clk	1
input	reset	1
input	clear	1
input	pop	1
input	commit	1
input	push	1
input	push_element	log2(BP_TABLE_SIZE)
input	set_head	1
input	head_val	log2(WL_SIZE)+1
input	set_tail	1
input	tail_val	log2(WL_SIZE)+1
input	rd_addr	log2(WL_SIZE)+1
output	head_element	log2(BP_TABLE_SIZE)
output	rd_element	log2(BP_TABLE_SIZE)
output	curr_head	log2(WL_SIZE)+1
output	curr_tail	log2(WL_SIZE)+1
output	full	1
output	logical_empty	1
output	empty	1

- *pop*, when high, moves the current head by 1.
- *commit*, when high, moves the commit_head by 1.
- *push*, when high, stores the value on *push_element* at the current tail and moves the tail by 1.
- *set_head*, when high, sets the current head value to *head_val*. Similarly, *set_tail*, when high, sets the current tail value to *tail_val*.
- The address on *rd_addr* is used to lookup the fifo RAM and drive the *rd_element* output.
- *head_element* always has the value at the head index.
- *curr_head* outputs the current head index. Similarly, *curr_tail* outputs the current tail index.

- *full* output is high when commit_head=tail and their MSBs are different.
- *logical_empty* output is high when head=tail and their MSBs are also equal.
- *empty* output is high when commit_head=tail and their MSBs are also equal. This indicates that all fifo elements have been committed.

An element from the current worklist is popped when *index1_head* is the last element and the wire *inc_index1_head* is high. An element from the current output worklist is popped when *wl_switch* is high, as this becomes the new input worklist.

The *active_update_index* element is pushed into the output worklist when *active_update_vld* is high.

The worklist commit signals are generated by the predictor. It monitors the retiring branch of interest instruction's checkpointed *cp_index_iter* (from checkpoint queue head). If it transitions from 7 to 0, a commit signal is sent to the current input worklist.

The worklist head needs to repaired when *cp_fix_pred_vld* is high. This is achieved by making the *wl0/1_set_head* high. The head is set to the checkpointed *cp_wl0/1_head*. It is set to *cp_wl0/1_head + 1* if an element is being popped during the repair, as described in Section 4.2.5.

4.2.8 Active Update Queue

This structure works as a regular fifo. Additionally, it is a fully associative structure that searches for a given *index1* element between its head and tail. A hit (predict taken) in this queue takes precedence over the fillnum table prediction. The *curr_index1* element is pushed in this queue, when a *maparp* branch is predicted "not taken". The head element is popped when *active_update_vld* signal is high.

4.2.9 Prediction Tables

The predictor has two direct mapped tables, one for *fillnum* branches and another for *maparp* branches. These tables are indexed by *index1* values to make a prediction.

The fillnum table is cleared when *bound1p0_ret_vld* is high. At the time of active update, the *active_update_index* entry in the fillnum table is updated to 1, indicating "taken" prediction. The maparp table on the other hand is trained when a *maparp* branch is mispredicted. The prediction outcome is updated at the *cp_index1* entry.

4.2.10 AnyCore Changes

The fetch stage has to choose the prediction from the astar predictor for branches of interest. These branches are identified by their program counters. Since the predictor outputs a 3-bit prediction, the core must also choose the prediction appropriately. *pred[2]* corresponds to *endindex* branch, *pred[1]* corresponds to *maparp* branch, *pred[0]* corresponds to *fillnum* branch. The fetch stage is also responsible for generating *inst_fet_vld* whenever an instruction is fetched. Additionally, it should also stall fetching instructions when it sees the *stall* signal asserted by the predictor and the call to *makebound()* function has been made. When this call instruction retires, the predictor will deassert the *stall* signal and the fetch stage may continue fetching instructions.

The retire stage has the responsibility to fix the predictor state whenever the processor pipeline needs to be flushed (at branch mispredictions, exceptions, load-store violations, etc.) by making the *cp_fix_pred_vld* high. The retire stage is also responsible for generating the *active_update_vld* signal when stores of interest retire. These stores are identified by their program counters. Additionally, it is also responsible for generating *commit_cpq* whenever an instruction is being retired.

4.3 Corner Case

While debugging issues in the predictor design we identified a corner case that was not identified by the C++ simulator. The predictor proposed by Chaudhary[1], makes the *wl_switch* signal high and switches the current input worklist upon fetching the *makebound()* call instruction. It is possible that the input worklist may switch well in advance and underflow since active updates (stores) may still be pending. Additionally, If this instruction is in the wrong control flow path, it will be squashed in the future. The switched worklist may be corrupted by predicting branches of interest that follow it. Both these issues have been fixed by generating the *wl_switch* signal at the retire stage instead of the fetch stage, thereby removing the worklist switch speculation.

Chapter 5

DESIGN VERIFICATION

This chapter describes the verification methodologies used to verify the correctness of the predictor.

5.1 Verification by Trace File

The standalone predictor design is verified using a custom testbench and a trace file of instructions. This trace files contains the instructions of interest that have an affect on the predictor. For example, *fillnum* branch, *maparp* branch, *endindex* branch, critical stores, instruction that induces worklist switch, etc. The trace files is generated by running the astar benchmark on a C++ simulator that incorporates the predictor designed by Chaudhary[1].

```
1, 3450380, 3450390, 17a58, 0, 0
1, 3450385, 3450396, 17a6c, 0, 1
0, 3450395, 3450396, 17a94, 0, 0
1, 3450399, 3450409, 17aa8, 0, 0
1, 3450404, 3450415, 17abc, 0, 1
```

Figure 5.1: Trace File Sample

Figure 5.1, shows the sample trace file. The first column indicates if the instruction is in the correct(1) control flow path or the wrong(0) control flow path. Second column indicates

the cycle at which the instruction is fetched. Third column indicates the cycle at which the instruction is retired. Fourth column indicates the instruction program counter. Fifth column indicates the original prediction if the instruction is a branch of interest or the *index1* value if it is a critical store. Sixth column indicates if the branch is mispredicted(1).

The testbench maintains a fifo to hold instructions. It reads the trace file line by line and pushes the instruction into the fifo's tail when the instruction's fetch cycle matches the testbench's cycle counter. At this time, if it is a branch of interest, the astar predictor is invoked to make a prediction. The predictor state is also checkpointed and stored in the fifo entry. The instruction at the head of the fifo is popped when the its retire cycle matches the testbench's cycle counter. At this time the testbench is responsible for making active updates to the predictor and also use the checkpoint payload to fix the predictor in case of a misprediction.

The testbench essentially mimics the fetch and retire stages of a processor core. The astar predictor is deemed to be functionally correct if the prediction made by it matches the original prediction in the trace file. This check is done when the instruction is popped from the fifo.

5.2 Verification by AnyCore Integration

The astar predictor is integrated with the AnyCore processor as described in Section 4.2.10. The RTL simulation also produces an output trace file with a structure similar to the one described in Figure 5.1. This trace file is compared with the trace file generated by the C++ simulator. The design is deemed to be functionally correct, if the predictions in both the trace files match.

Chapter 6

RESULTS

This chapter describes the branch predictor statistics and performance statistics obtained through the simulation of AnyCore RTL.

6.1 Simulation Methodology

The AnyCore processor baseline configuration is described in Table 6.1.

Table 6.1: AnyCore Baseline Configuration

Fetch/Issue/Retire Width	1/3/1
Active List	96
Issue Queue	16
LSQ	32

The AnyCore RISC-V port had some bugs at the time of the predictor design and experimental phase. These limited the core width and also required benchmark phases free of some unimplemented instructions. Thus, we ran the astar benchmark with rivers reference input for about 6.8 million instructions on a 1-wide core. This region of instructions hit the *fill()* and *makebound()* functions. Since the predictor design is compact, it is able to provide only one prediction at a time, which is fine for a 1-wide fetch machine. It can also be integrated with a wider machine by truncating the fetch bundle at a branch of interest.

6.2 Branch Misprediction Rate

Figure 6.1, shows the branch misprediction rates normalized to the baseline. We can see that the misprediction rate has reduced by 85%.

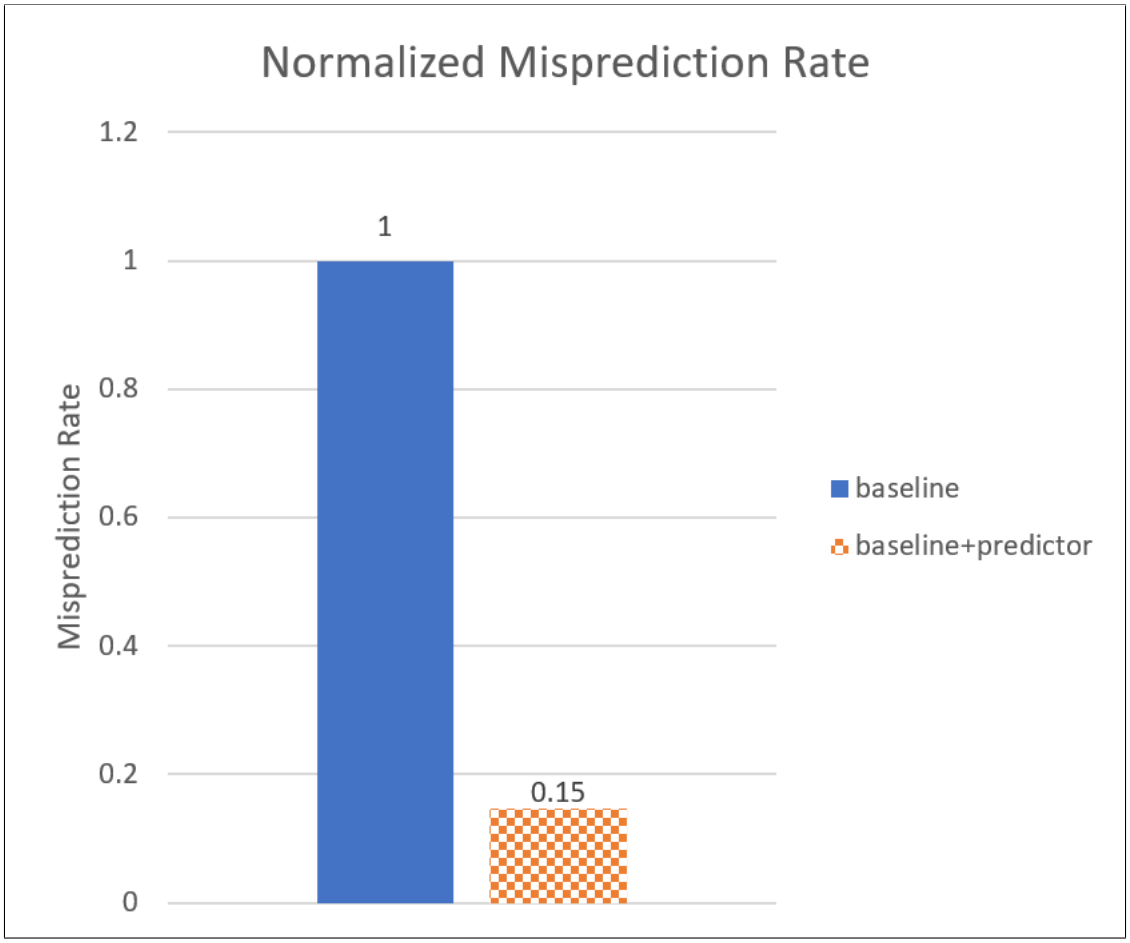

Figure 6.1: Normalized Misprediction Rate Comparision

6.3 IPC

Figure 6.2, shows the IPC normalized to the baseline. We can see that the IPC has improved by almost 2.79 times.

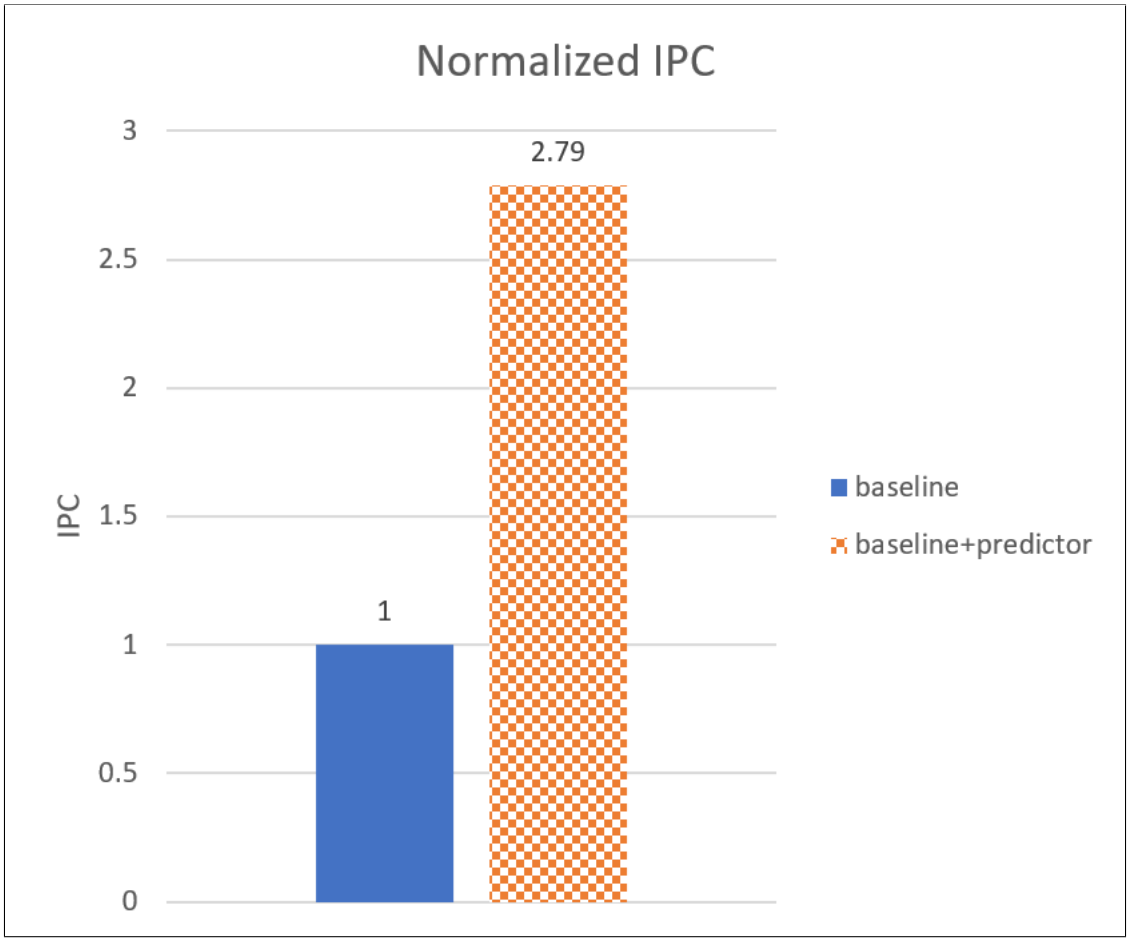

Figure 6.2: Normalized IPC

The improvements seen in misprediction rates and IPC follows a similar trend as seen in the C++ simulations performed by Chaudhary[1].

6.4 Synthesis

The design is synthesized using the 45nm FreePDK library, a part of the NANGATE Open Cell Library (NangateOpenCellLibrary_PDKv1_2_v2008_10). Synopsys Design Vision is used to synthesize the design. The implied SRAMS used for the large *fillnum* and *maparp* prediction tables, worklist fifo, and active update queue, are excluded from the synthesis process. Only the design state and control signals are synthesized. Table 6.2 shows the synthesis results obtained. A clock speed of *125 MHz* is achieved. To get a better idea about the area results, we look at the area of a 2-input NAND gate in the library. It measures 0.798 units. Thus, the overall cell area of the predictor design is equivalent to the area of *6305* 2-input NAND gates.

Table 6.2: Synthesis Results

Clock Speed	125 MHz
Cell Area	5031.656 units

Chapter 7

CONCLUSION

7.1 Summary

This thesis is an effort towards designing an application specific microarchitecture that can be mapped to the PSM fabric. The design clearly defines the interface that the PSM can use. The thesis successfully demonstrates that the predictor works correctly when integrated with the AnyCore processor. The Verilog design is also able to replicate performance improvements seen in the C++ simulator. A corner case missed by the C++ simulations has been identified by the RTL simulations and is fixed in the Verilog design.

7.2 Future Work

7.2.1 Pipelining

As seen in Table 4.2, the predictor tables are large: 2^{18} 1-bit entries each, i.e., 32 kilobytes (32KB), which is the size of a typical L1 instruction cache minus its metadata (tags, LRU, etc.) overhead. This is necessary to reduce aliasing among the *index1* values. In order to meet timing requirements, it may be necessary to pipeline the predictor. Mechanisms to deal with delayed predictions due to pipelining are to be explored.

7.2.2 Wide AnyCore

Usually multiple branches in a fetch bundle are predicted in parallel and the bundle is broken at the first predicted taken branch. Although the predictor design has been demonstrated on a 1-wide AnyCore configuration, it can be easily extended to a wider

core by breaking the fetch bundle at the first branch of interest. Thus, the predictor can still make one prediction and also support a wider core. However, if the predictor needs to make multiple predictions in parallel, additional changes are required to be explored.

7.2.3 High Level Synthesis

Although this work is an ASIC design, the feasibility of synthesizing RTL with the help of HLS tools from C/C++ code needs to be explored. This may be advantageous, as microarchitecture definitions in C/C++ have a fast turn around. This opens the door for exploring more microarchitectures suitable for PSM.

7.2.4 Post Silicon Microarchitecture

The end goal is to map the design on the PSM fabric and instantiate it on a need basis. The instructions of interest are currently identified by their hard-coded program counters. A more automatic approach needs to be explored for a fully functioning PSM fabric.

7.2.5 Interface Delays

In the current design, the communication between the predictor and the processor core happens in one cycle. However, this may not be practically achievable since interacting with PSM fabric will be a long latency event, similar to accessing memory. One possible solution is to have multiple queues for instructions of interest. The predictor may then run ahead, generate many predictions and push them onto the queues. The fetch stage will now look for predictions in these queues. Similarly, the retire stage may push active update stores onto the queues. The predictor will now look for values in these queues to update the prediction tables.

REFERENCES

[1] A. Chaudhary, "Custom exact branch predictor for astar benchmark," Master's thesis, North Carolina State University, 2019.

[2] R. CBasu Roy Chowdhury, *AnyCore: Design, Fabrication, and Evaluation of Comprehensively Adaptive Superscalar Processors*. PhD thesis, North Carolina State University, 2016.

[3] E. Rotenberg, "FoMR: Post silicon micro-architecture, NSF grant no. CCF-1823517," 2018.

[4] A. Seznec, "A 256 kbits l-tage branch predictor," *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, vol. 9, pp. 1–6, 2007.

[5] M. Al-Otoom, E. Forbes, and E. Rotenberg, "Exact: Explicit dynamic-branch prediction with active updates," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, (New York, NY, USA), pp. 165–176, ACM, 2010.

[6] M. Al-Otoom, R. Sheikh, and E. Rotenberg, "A case for a software-managed reconfigurable branch predictor," 2012.

[7] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez, "Checkpointed early load retirement," in *11th International Symposium on High-Performance Computer Architecture*, pp. 16–27, Feb 2005.

[8] K. Sundaramoorthy, Z. Purser, and E. Rotenburg, "Slipstream processors: Improving both performance and fault tolerance," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, (New York, NY, USA), pp. 257–268, ACM, 2000.

[9] M. A. Watkins and D. H. Albonesi, "Remap: A reconfigurable heterogeneous multicore architecture," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 497–508, IEEE Computer Society, 2010.

[10] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[11] R. Sheikh, J. Tuck, and E. Rotenberg, "Control-flow decoupling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 329–340, Dec 2012.