ABSTRACT

CHAUDHARY, AAYUSH. Custom EXACT Branch Predictor for astar Benchmark. (Under the direction of Dr. Eric Rotenberg).

Computer architects have explored techniques for increasing IPC for general applications. There have been several proposals targeting branch prediction in academia. We intend to examine the impact of customizing the branch predictor to a certain benchmark (astar) and study the feasibility of implementing the custom branch predictor on a post silicon microarchitecture (PSM) fabric.

Post silicon microarchitecture is a recently proposed microarchitecture, where a superscalar core and a reconfigurable fabric interact via a well defined interface. In this thesis, we exploit this interface to design our custom predictor for astar.

Load dependent branches are the leading cause of mispredictions. astar is one of the highest mispredicted benchmarks with a MPKI of 28. A lot of these mispredictions are from load dependent branches. With custom EXACT branch predictor, we target these branches. EXACT (EXplicit dynamic-branch prediction with ACTive updates) indexes into the branch predictor table using an ID based on predicted load address. For our custom predictor, we use a specialized address indexing mechanism approach imitating the way astar references the data structure. If there is a store to the same address, we need to actively update our branch predictor tables and alter the prediction for future instances of the dynamic branch. We use this concept of active update by snooping on stores at retirement.

We observe a max increase of 4.9x in IPC of astar benchmark with custom predictor and perfect caches. Even for real caches, we see a geometric mean speedup of 3x. The misprediction rate of astar drops from 16% to less than 2% with the custom EXACT predictor.
Custom EXACT Branch Predictor for astar Benchmark

by
Aayush Chaudhary

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

Aayush Chaudhary was born in Ghaziabad, Uttar Pradesh, India. He did his schooling in various cities in India. After doing his high school from Delhi Public School, Vasant Kunj, New Delhi, he decided to pursue Electrical Engineering from Indian Institute of Technology Bombay (IITB), Mumbai in 2012. During his undergraduate studies, he pursued several projects in the field of computer architecture and ASIC design. He worked on dynamic core architecture during his undergraduate studies. He graduated in 2016 with honors in Electrical Engineering from IITB.

After his undergraduate studies, Aayush joined Indian Space Research Organization, Ahmedabad, India where he worked as an engineer to design the SDRAM controller for Chandrayaan-2 (Moon Mission-2) project. However, he was more enthusiastic to dive deeper into computer architecture and was accepted to join Computer Engineering at NC State in 2017. It was not long after that he decided to pursue his master’s thesis under the guidance of Dr. Eric Rotenberg.

He worked on hiding load latency architectures and value prediction initially during summer 2018 and switched to custom branch predictor soon after. His primary research interest is in high performance CPU architectures, control flow techniques and GPU/GPGPU architecture. Upon completion of master’s degree from NC State University, Aayush will be joining Samsung ACL, San Jose as a GPU architect.
ACKNOWLEDGEMENTS

This thesis is the result of constant support from several individuals who I wish to acknowledge. First and foremost, I would like to thank Dr. Eric Rotenberg for his constant support and guidance since the beginning of my master’s thesis. During my master’s studies, I learned a lot from him. His passion for computer architecture is tremendous. I would also like to thank him for being available at all times for his students, and guiding them especially during tough times. I would also like to thank my committee members, Dr. Huiyang Zhou and Dr. James Tuck for supporting me in this work.

A special thanks to my parents, Vijendra Kumar Chaudhary and Raj Chaudhary; and my brothers, Tanuj Chaudhary and Anshul Chaudhary. I would like to thank them for all their constant support throughout my career and sacrificing many things for my education. Their constant love and support guided me through the difficult phases of my life.

I would also like to acknowledge all the members of our research group for the great discussions we had during our meetings. In particular, Utkarsh Mathur, Saransh Jain, Adith Vastrad, Vinesh Srinivasan and Chanchal Kumar.

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 ........................................................................ vi

List of Figures ........................................................................ vii

Chapter 1 Introduction and Motivation ................................. 1
  1.1 Post silicon microarchitecture .................................... 1
  1.2 Sources of mispredictions in astar benchmark .......... 3
  1.3 Custom EXACT predictor .......................................... 4
  1.4 Contributions ............................................................ 4
  1.5 Thesis outline ............................................................ 5

Chapter 2 Related Work .................................................... 6
  2.1 Post silicon microarchitecture .................................... 6
  2.2 Load based branch prediction techniques ................ 7
  2.3 EXACT: EXplicit dynamic branch prediction with ACTive updates ....... 7

Chapter 3 astar Benchmark .................................................. 10
  3.1 makebound2() function ............................................. 10
  3.2 Analysis of one instance of makebound2 ................. 10
  3.3 Call to makebound2 - fill() function ....................... 13

Chapter 4 Microarchitecture for Custom EXACT Predictor .... 14
  4.1 Prediction algorithm overview .................................. 14
  4.2 Worklists ................................................................. 15
    4.2.1 Removing an index from the worklist .............. 16
  4.3 Index1 array .............................................................. 16
  4.4 Branch predictor tables: waymap_fillnum and maparp 17
  4.5 Active update queue .................................................. 17
  4.6 Active update unit ...................................................... 18
  4.7 Prediction state machine .......................................... 18
  4.8 Illustration using an example .................................... 20
  4.9 Checkpoints ............................................................. 23
  4.10 Recovery ................................................................. 23
  4.11 Handling limited resources ..................................... 24
    4.11.1 Branch predictor table size ............................. 24
    4.11.2 Worklist size .................................................... 24
    4.11.3 Active update queue size ............................... 25
  4.12 PSM interface .......................................................... 25
  4.13 Pipelined predictor .................................................... 25

Chapter 5 Results .............................................................. 26
LIST OF TABLES

Table 5.1 Baseline Configuration .............................................. 27
Table 5.2 Custom Branch Predictor Configuration ....................... 27
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview of Post Silicon microarchitecture</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Motivation for custom predictor</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>High level overview of EXACT</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>makebound2 function in astar source code</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Visualization of makebound2</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Instance of makebound2</td>
<td>12</td>
</tr>
<tr>
<td>3.4</td>
<td>fill() function</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>microarchitecture</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>Prediction State Machine</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>Example - state 1</td>
<td>21</td>
</tr>
<tr>
<td>4.4</td>
<td>Example - state 2</td>
<td>21</td>
</tr>
<tr>
<td>4.5</td>
<td>Example - state 3</td>
<td>21</td>
</tr>
<tr>
<td>4.6</td>
<td>Example - state 4</td>
<td>21</td>
</tr>
<tr>
<td>4.7</td>
<td>Example - state 5</td>
<td>21</td>
</tr>
<tr>
<td>4.8</td>
<td>Example - state 6</td>
<td>21</td>
</tr>
<tr>
<td>4.9</td>
<td>Illustration via an example</td>
<td>21</td>
</tr>
<tr>
<td>4.10</td>
<td>Example - state 7</td>
<td>22</td>
</tr>
<tr>
<td>4.11</td>
<td>Example - state 8</td>
<td>22</td>
</tr>
<tr>
<td>4.12</td>
<td>Illustration via an example</td>
<td>22</td>
</tr>
<tr>
<td>5.1</td>
<td>Results for different configuration for 16 wide processor</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Results for different configuration for 8 wide processor</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Results for different configuration for 4 wide processor</td>
<td>29</td>
</tr>
<tr>
<td>5.4</td>
<td>Configuration = 0,1,1,1</td>
<td>30</td>
</tr>
<tr>
<td>5.5</td>
<td>Configuration = 0,0,0,0</td>
<td>30</td>
</tr>
<tr>
<td>5.6</td>
<td>Aliasing due to sizing waymap_fillnum</td>
<td>30</td>
</tr>
<tr>
<td>5.7</td>
<td>Configuration = 0,1,1,1</td>
<td>31</td>
</tr>
<tr>
<td>5.8</td>
<td>Configuration = 0,0,0,0</td>
<td>31</td>
</tr>
<tr>
<td>5.9</td>
<td>Aliasing due to sizing maparp</td>
<td>31</td>
</tr>
<tr>
<td>5.10</td>
<td>Reducing both waymap_fillnum and maparp size (config = 0,1,1,1)</td>
<td>31</td>
</tr>
<tr>
<td>5.11</td>
<td>Configuration = 0,1,1,1</td>
<td>32</td>
</tr>
<tr>
<td>5.12</td>
<td>Configuration = 0,0,0,0</td>
<td>32</td>
</tr>
<tr>
<td>5.13</td>
<td>IPC vs worklist size</td>
<td>32</td>
</tr>
<tr>
<td>5.14</td>
<td>IPC for different pipeline depth</td>
<td>33</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction and Motivation

In the past, computer architects have developed techniques to speedup the performance of applications in general. Architects have explored several techniques including better control flow handling (branch prediction, multipath execution, predication, control independence, etc.), better dataflow handling (value prediction, etc.), better approaches for high fetch bandwidth (trace cache, etc.), etc. to speedup the performance of out of order core.

However, its getting harder to get performance in general ways. In this work, we explore customized solution for benchmarks to improve their performance. The customized microarchitecture is implemented on a reconfigurable fabric. Post Silicon Microarchitecture (PSM) [1] is a novel microarchitecture with a superscalar core and reconfigurable fabric. The custom microarchitecture implemented on reconfigurable fabric observes and intervenes into the superscalar core to control the execution of an application.

In particular, we explore astar benchmark and propose a novel custom EXACT [2] branch predictor. The predictor efficiently mimics the software execution in hardware and predict branches accurately.

1.1 Post silicon microarchitecture

Post silicon microarchitecture (PSM) instantiates new microarchitectures on top of general purpose instruction-level parallel (ILP) cores. PSM refers to dynamic instance of a certain microarchitecture on reconfigurable fabric, which interacts with general purpose core to
predict branches, control the instruction flow, parallelize the loops etc.

Figure 1.1 describes the overall idea of post silicon microarchitecture. A microarchitecture is constructed on the fly in reconfigurable hardware. This new microarchitecture snoops on certain signals in superscalar core via the PSM interface. The reconfigurable logic intervenes superscalar core to either help in execution or override certain signals during execution (e.g.: branch outcome).

![Figure 1.1: Overview of Post Silicon microarchitecture](image)

As described in the PSM project proposal [1], PSM has two major use cases: application-specific parallelization and novel microarchitecture on superscalar processors. `makebound2` function of class `wayobj` in astar[3] benchmark presents an interesting case for parallelization of the loop iteration. The function `makebound2` can be vectorized by exploiting control flow independence across loop iterations. The loop can be vectorized by ignoring the store-load synchronization and implementing a custom reduction operation on sub-worklists to generate the final output worklist.

Novel architectures such as control flow decoupling [4], EXACT-S [2][5], CLEAR
[6], Slipstream [7] etc. can be implemented alongside the superscalar processor. Branch predictors are a possible class of architecture which is explored in this thesis. A particular application which benefits from a certain microarchitecture configures the PSM fabric with that microarchitecture and programs the interface between the PSM fabric and the superscalar core for necessary snooping.

Custom EXACT predictor is one such microarchitecture which targets astar benchmark. The predictor is highly customized to astar benchmark and essentially mimics the software execution.

1.2 Sources of mispredictions in astar benchmark

astar benchmark with a misprediction per kilo instructions (MPKI) of 28 and misprediction rate of 16% (gshare branch predictor with 64K entries), is one of the highest mispredicted benchmarks. Most of these mispredictions are from makebound2 function. The two load dependent branches in makebound2 function described in chapter 3 are highly mispredicted. Since, there is a store that happens in the control dependent region of theses branches, the branch direction alters for future iterations of this loop. General predictors like gshare [8] are not able to account for the store and update the predictor tables, leading to mispredictions. Another source of misprediction arise due to the way astar references its data structures.

Figure 1.2 shows IPC (instructions per cycle) for:

- Perfect prediction for all branches (perf)
- Perfect prediction for all the branches in makebound2 function (perf_makebound)
- Real prediction with gshare (real_gshare)

The results obtained are for a 16 wide architecture with perfect data cache and perfect fetch. We can see from figure 1.2 that there is a significant boost in IPC (5.6x) by predicting the branches in makebound2 accurately. Since the default predictor is not able to predict the branches in makebound2 correctly, we can implement a custom predictor to improve prediction accuracy for these branches.
1.3 Custom EXACT predictor

Custom EXACT predictor mimics the execution of \texttt{makebound2} function. It overrides the prediction of default predictor for the branches in \texttt{makebound2}.

EXACT [2] predictor uses predicted load address to generate an ID to index into the branch predictor. Instead, the custom EXACT predictor for \texttt{astar} uses the index which is available from the worklist as an ID for indexing into the predictor. Similar to EXACT, we do active updates by snooping stores at retire. The active updates by the stores account for the stores that alters the branch direction for the branches in \texttt{makebound2} function. The microarchitecture for the custom predictor will be discussed in chapter 4.

1.4 Contributions

The thesis makes following contributions:

1. Proposed a new paradigm: FPGA accelerator as a predictor
2. Study of astar benchmark (makebound2 function)

3. Proposed, implemented and evaluated a novel customized branch predictor for astar benchmark

4. Demonstrated a use-case of active updates, and the performance gain by predicting load dependent branches correctly

5. Demonstrated that the application specific microarchitectures can significantly boost the performance of a workload and hence PSM has great potential for certain applications

1.5 Thesis outline

Further chapters provide a deep dive into the details of custom EXACT predictor. Chapter 2 briefs about the related work in past and gives an overview of EXACT predictor. Chapter 3 talks about the astar source code and explains the makebound2 function. Chapter 4 explains the microarchitecture of the custom predictor. In chapter 5, we discuss the results obtained by implementing the custom EXACT predictor in 721sim and chapter 6 concludes the thesis.
Chapter 2

Related Work

2.1 Post silicon microarchitecture

Watkins et al. proposed ReMAP [9] architecture, which closely matches the goal of post silicon microarchitecture. ReMAP is a reconfigurable architecture targeting acceleration and parallelization of application in a heterogeneous chip multiprocessor. The threads share a reconfigurable fabric. The threads can configure the fabric for individual computation or fine-grained communication with integrated computation. The key functions in an application is explicitly parallelized. The portions of communication and computation are accelerated in reconfigurable fabric. The PSM approach differs from ReMAP because the application is not explicitly parallelized in case of PSM. Instead, the reconfigurable fabric intervenes within the microarchitecture for parallelization.

Smart Memories [10] by Mai et al. presents a modular reconfigurable architecture comprised of many processing tiles, each containing local memory, local interconnect, and a processor core. One can reconfigure the processing elements, mats of memories and interconnection network in the smart memory architecture. This is different from PSM since PSM intervenes the pipeline stages of the superscalar processor.

Carrillo et al. proposed OneChip [11], which integrates reconfigurable functional units into the superscalar processor pipeline. RFU stores multiple configurations and FPGA switches between multiple configurations. This approach requires new instructions created for RFU unit and a new programming model. In contrast, PSM observes and intervenes the superscalar processor via a PSM fabric and does not require any change in ISA.
2.2 Load based branch prediction techniques

Al-Otoom et al. proposed EXplicit dynamic branch prediction with ACTive updates (EXACT) [2] predictor to avoid mispredictions of load dependent branches. We use the concept of active updates similar to the one described in this paper. Section 2.3 particularly talks about EXACT predictor and the similarities with the custom EXACT predictor.

Gonzales et al. in [12] proposed to explicitly value predict the source operand to find the direction of control flow at fetch. Chen et al. proposed ARVI [13] predictor that uses dynamic data dependence information. The ARVI predictor uses live-in register values of dynamic branch’s backward slice to predict load dependent branches.

ABC (address branch correlation) [14] by Gao et al. proposes to predict branches that miss in L2 cache based on the address. The address is available since the request has been sent to memory. If the branch depends on a stable data, the paper observes that address of the load (rather than the value) is sufficient to predict the branch outcome.

2.3 EXACT: EXplicit dynamic branch prediction with ACTive updates

EXACT predictor targets branches that are directly or indirectly dependent on load instruction. Global branch history is unable to distinguish different dynamic instances of a branch, and hence leads to mispredictions. Therefore, instead of waiting for the load value or relying on global branch history, the address of the load can be used to predict the load dependent branches. The high level overview of EXACT [2] is shown in figure 2.1. EXACT has three major components: the ID generation unit, active update unit and the predictor itself.

The ID generation unit is responsible for generation of ID for the dynamic branch by hashing PC with the load address. The EXACT-H generates index for a branch based on an ID ‘d’ distance away and accesses the branch predictor tables using that index.

Since EXACT uses load address for generation of ID, the store to the same address can alter the prediction for future iterations. Therefore, Al-Otoom et al. proposed the concept
of active updates, where a store updates the branch predictor table entry. The active update unit observes the committed store and infers where and how to flip predictions in the explicit predictor.

They also proposed the software version of EXACT, EXACT-S [5]. In this case, the address used to index into the predictor is the actual address (not ‘d’ distance away). Therefore, this leads to better accuracy. Using the precise load address simplifies the cost of predictor, by avoiding the huge conversion tables required by EXACT-H. EXACT-S requires shadow code in the data segment of the program to generate address based indices to access the explicit predictor.

Custom EXACT predictor for astar derives from EXACT predictor. The ID generation unit for custom EXACT predictor includes the worklists and the index1 array. The index is popped from the worklist to generate 8 other indices, which are used to index into the predictor. In contrast to EXACT, we don’t use load addresses to access the predictor. We instead access the predictor using the generated indices from the worklist index.

The active update unit for custom predictor monitors stores at retire. But, instead of
using the store address, we use the source operand of the store to index into our predictor. The active update is essential for the custom predictor, since the store in astar benchmark alters the prediction for the load dependent branch for future iterations.

Similar to EXACT, we have active updates and passive updates to the branch predictor table.

Therefore, the custom EXACT branch predictor for astar benchmark proposed in this thesis derives and builds upon the concepts from EXACT predictor.
Chapter 3

astar Benchmark

astar is a SPEC CPU 2006 [3] integer benchmark. A* (astar) is a path finding algorithm for maps and graphs. It uses neighbourhood relationship for finding the path.

3.1 makebound2() function

Figure 3.1 shows the snippet of makebound2 function. The branches of interest are shown in red boxes. There are eight instances of the code shown in blue box, with different values of index1. One can imagine the eight instances as a grid with index as the centre element and the surrounding eight boxes represented by different values of index1. This representation is shown in figure 3.2.

3.2 Analysis of one instance of makebound2

Figure 3.3 shows the zoomed in view of instance 1 from figure 3.1. In line 70 of code snippet, a new index1 is generated using the index and yoffset. yoffset is constant for a particular benchmark and index is picked from bound1p array.

The two branches follow next: waymap_fillnum and maparp. If the branches are {NT, NT}, we go inside control dependent region for both the branches. In line 74, the generated index1 is stored in another array (bound2p) which will be used in another iteration of makebound2 function.

The store which alters the prediction of waymap_fillnum branch for future iterations
Figure 3.1: makebound2 function in astar source code
is shown in line 77. Since the store to *waymap.fillnum* is the value of fillnum itself, the outcome of branch in line 71 has to be “taken” for future iterations. This can be visualized as marking a particular block in a grid as visited to ensure that the block is not processed again. The branch on line 80 checks if this is the last index. It is a predictable branch and is usually not taken.
3.3 Call to makebound2 - fill() function

Figure 3.4 shows the fill function which calls makebound2 function. Note that makebound2 is called with different arguments alternatively. bound1p and bound2p arrays are passed alternatively as working array. This ensures the working array is the one that has indices that were pushed during the previous iteration of makebound2. The other array is empty and new indices are stored in this empty array.

```cpp
bool wayobj::fill(i32 startx, i32 starty, i32 endx, i32 endy)
{
    i32 bound1;
    bool flodd;

    if (fillnum==65535)
    {
        memset(waymap,0,((3<<mapxshift)<<mapyshift)*sizeof(waymap[0]));
        fillnum=0;
    }
    fillnum++;

    bound1p[0]=index(startx,starty);
    waymap[starty<<shift|startx].fillnum=fillnum;
    waymap[starty<<shift|startx].num=0;

    flodd=false;
    bound1=1;

    flend=false;
    endindex=index(endx,endy);
    step=1;

    while ((bound1!=0)&&(flend==false))
    {
        if (flodd==false)
        {
            bound1=makebound2(bound1p,bound1,bound2p);
            flodd=true;
        }
        else
        {
            bound1=makebound2(bound2p,bound1,bound1p);
            flodd=false;
        }

        step++;
    }
    return flend;
}
```

Figure 3.4: fill() function

The figure also shows bound1p[0] store. This store acts as the trigger to the custom predictor and provides the starting index for the custom predictor.
Chapter 4

Microarchitecture for Custom EXACT Predictor

The custom EXACT predictor is a custom ASIC-like hardware on PSM fabric, which mimics the execution, timely and accurately but permitted to be wrong. The custom predictor executes in parallel with the default predictor and a mux selects the prediction either from the default predictor or from the custom predictor.

Figure 4.1 shows the major components of custom EXACT predictor for astar. The two worklists mimic $bound1p$ and $bound2p$ arrays. The index1 array stores the eight indices generated using the index from the worklist. The waymap_fillnum and maparp tables are the branch predictor tables which are indexed using the indices from the index1 array. The active update unit at retire snoops on stores and updates entries in waymap_fillnum table.

4.1 Prediction algorithm overview

Initially, $bound1p[0]$ store is snooped for starting index. If the branch in makebound2 function is fetched and the $bound1p[0]$ store has not yet retired, the fetch unit is stalled until the store retires.

Once the starting index is available, the index1 array is generated and we traverse this array for all eight indices. With an index from index1 array, the waymap_fillnum table is accessed for the prediction. If the entry in waymap_fillnum table corresponding
to that index is 0, we predict \{NT\} and move to maparp branch.

If the entry in waymap_fillnum table is 1, predict \{T\} and move to next index in index1 array. This implies going to next instance in *makebound2* function.

After the waymap_fillnum is predicted \{NT\}, the maparp table is accessed for generating a prediction for maparp branch. If the entry corresponding to index is 0, we predict \{NT\} and move to endindex branch. If the entry in maparp table is 1, predict \{T\} and move to next index in index1 array.

For the endindex branch, predict \{NT\} and move to next index in index1 array.

Once all the entries in input worklist are popped, the *makebound2* function should return. When the *makebound2* function is called again the input and output worklists are swapped and the prediction continues.

### 4.2 Worklists

The worklist1 and worklist2 correspond to *bound1p* and *bound2p* arrays in the *astar* source code. One of the worklist is the “input worklist”. We pop an index from the input worklist
to generate eight other indices. The second worklist serves as an “output worklist” and stores the indices for which the prediction of waymap_fillnum and maparp branches was \{NT, NT\}. This update to output worklist is done at retire time by snooping the store in line 74 of code snippet in figure 3.2.

For the next iteration of `makebound2` function, the input and output worklists are swapped, similar to swapping of `bound1p` and `bound2p` array in figure 3.4.

If the input worklist underflows, we switch to default predictor, but continue to train our predictor by snooping the stores and updating the waymap_fillnum table and the output worklist.

### 4.2.1 Removing an index from the worklist

Since, there is a chance of rollback in case of mispredict, we can’t actually pop the index from the worklist when we pick a new index from input worklist. We instead move the head pointer in the worklist to give an illusion of pop.

One can safely pop the index from the input worklist if there are no chances of rolling back to that index. Therefore, there is a delay in actual pop of the index from the input worklist. Let’s assume there are indices ‘A’ and ‘B’ in worklist1. Index ‘A’ generates eight indices and generates predictions based on state machine. It might happen that the next iteration of `makebound2` reaches fetch stage and picks index ‘B’ for next iteration. It is still not safe to remove index ‘A’ from worklist since all eight instances corresponding to index ‘A’ have not yet retired. If misprediction occurs for some instance of index ‘A’, index ‘A’ is required again to restore the index1 array.

Therefore, it is safe to actually pop the worklist element only when all the eight instances of `makebound2` corresponding to that element have retired.

### 4.3 Index1 array

Index1 array stores the indices generated using the index from the input worklist. The array is updated when a new index is popped from the worklist. The following values are stored in the array:
1. index - yoffset - 1
2. index - yoffset
3. index - yoffset + 1
4. index - 1
5. index + 1
6. index + yoffset - 1
7. index + yoffset
8. index + yoffset + 1

yoffset is fixed for a particular input set to astar. It is 512 for astar_rivers and 2048 for astar_bigLakes. Therefore, the computation of these eight indices can be done when an index is available from the worklist.

Using the indices in index1 array, we access waymap_fillnum and maparp table to generate the predictions for waymap_fillnum branch and maparp branch respectively.

4.4 Branch predictor tables: waymap_fillnum and maparp

The two tables are direct mapped structures, which are indexed using the lower bits of indices in the index1 array. Each entry is a single bit, 0 denoting “Not Taken” and 1 denoting “Taken”. The waymap_fillnum table is trained using active updates by the active update unit. The maparp table is passively trained after a mispredict is detected. Also, the fillnum table is cleared when the bound1p[0] store is fetched again, i.e. when fill function is called.

4.5 Active update queue

This queue is required since we are delaying the active updates to retire stage. The index is placed in active update queue if we predict {NT, NT} for waymap_fillnum and maparp branches. Thus, the queue stores potential indices which should receive an active update,
unless a mispredict rolls back the tail of active update queue. The index that receives active update, updates the entry in branch predictor table and pushes itself in output worklist for the next iteration.

Since the active updates are delayed, the branch predictor tables will have stale entries for the instructions in window. Hence, active update queue is searched for the matching index for the prediction of waymap_fillnum branch. If a match is found, we predict \{T\} where the original prediction would have been \{NT\}.

Another alternative is to do active update as soon as you predict \{NT\} for maparp branch. In this case, there is no simple way to recover the branch predictor tables if a mispredict occurs. Shifting the active updates to retire has one additional benefit. The branch predictor tables can be trained even if our input worklist underflows and the prediction is made using the default predictor.

4.6 Active update unit

The active updates to waymap_fillnum table are done at retire. We snoop on the store at line 74 in figure 3.3 and similar stores of other instances of makebound2. One of the source operand of the store instruction is the index that has to be actively updated. Therefore, we can directly snoop on the index from the register file. The active update signal is also sent to active update queue since the active update queue holds indices with pending active update. The index is popped from the active update queue and put in output worklist for next iteration.

If the input worklist underflows, the waymap_fillnum table is still actively updated by snooping the stores mentioned above. Even if the default predictor was used to predict the branches, and we see this store retiring, we can be sure that there will be a store to fillnum data structure. Thus, there has to be an active update for this index.

4.7 Prediction state machine

Since there are three branches in one instance, a state machine is required to keep track of which type of branch to predict next - fillnum branch, a maparp branch or an endindex
branch. The state machine is shown in figure 4.2.

![Prediction State Machine](image)

**Figure 4.2: Prediction State Machine**

The initial state is entered after the `makebound2` function is called. We stay in this state until `bound1p` is not valid, i.e. `bound1p[0]` store shown in figure 3.4 has not yet retired. The fetch stage is stalled until the value of `bound1p[0]` is available. This serves as initial trigger for the custom predictor, since an initial index is required to start with.
The first branch is “waymap_fillnum”. Using $i^{th}$ index from the index1 array, the waymap_fillnum table is accessed to make prediction. The output of the state denotes the prediction of that branch. If we predict not taken, we go inside control dependent region, and hence the next state “maparp”. On the contrary, if prediction was taken, we directly jump to next instance.

The state “index1_iter++” accesses next element of index1 array. If index1_iter reaches 8, i.e. it has accessed all eight instances in the makebound2 function, we pop another index from the input worklist.

Similarly, the states “maparp” and “endindex” are for predicting the maparp and endindex branch respectively. The prediction of maparp branch is done using maparp table. We simply predict “Not Taken” for the endindex branch.

4.8 Illustration using an example

Let’s assume after certain predictions, the state of our worklists and branch predictor tables are as shown in figure 4.3. The index ‘x’ in worklist 1 is used to generate eight other indices in index1 array. Now, using index ‘a’ the waymap_fillnum table is accessed for prediction of waymap_fillnum branch as shown in figure. Since the entry corresponding to index ‘a’ is ‘1’, we predict {T}, and move to next index.

Since the prediction was {T} for previous branch, we jump to next index in index1 array and again encounter waymap_fillnum branch. We access waymap_fillnum table using index ‘b’ and see a ‘0’. We search active update queue for a pending update to ‘b’. Since the queue is empty, we predict {NT}.

In figure 4.5, we access maparp branch since the prediction of waymap_fillnum branch was {NT}. Since the entry in maparp branch is also ‘0’ we predict {NT} and push index ‘b’ in active update queue indicating there is a pending update to index ‘b’.

We continue prediction for other indices in index1 array similarly. Next, we increment head of worklist1 to index ‘y’. We again generate eight indices and start accessing branch predictor tables using these as shown in figure 4.6.
Now, the first index is ‘b’ again. We access the same entry in waymap_fillnum table and find a ‘0’ since the store with index ‘b’ has still not retired (figure 4.7). Irrespective of a ‘0’ in waymap_fillnum table, we predict \{T\} in this case, since we find a matching index in active update queue.

By this time, the store with index ‘b’ retires (figure 4.8). The index ‘b’ should now be pushed into worklist2 for next iteration. Also, the waymap_fillnum should receive an active update to index ‘b’. Since the waymap_fillnum table is now updated, we can remove the index from active update queue.

We again continue prediction using all the indices in worklist1, until the worklist1 is empty. At this point, the \texttt{makebound2} function should return (provided the worklist is sufficiently large). When the \texttt{makebound2} function is called again, we swap the worklists (figure 4.10) and again follow the same procedure.

When the \texttt{fill} function is called again, we clear the worklists, index1 array, active update queue and the waymap_fillnum table. We don’t clear the maparp table, since it is not cleared in the astar source code. We suffer a huge penalty in performance by clearing the maparp table.
4.9 Checkpoints

All the branches are checkpointed. For custom EXACT predictor, the following also needs to be checkpointed:

- Current input worklist, to correctly restore the input worklist as worklist1 or worklist2 after misprediction.
- Head of the input worklist, to restore the head to point to correct index element.
- Branch type: whether it is a fillnum branch, a maparp branch or an endindex branch.
- Prediction made by the custom predictor which turned out to be incorrect. Based on this, the state machine is restored to correct state.
- Index to restore the index1 array after misprediction.
- Index1_iter to point to the correct instance in makebound2 function after recovery.
- Active update queue tail, since bad instructions will push entries in the active update queue which leads to misprediction later if not restored.

4.10 Recovery

The recovery for default predictor is augmented with recovery for the state machine if the branch is predicted using custom predictor. The input worklist is restored using the checkpointed worklist head. Since active updates are done at retire, the output worklist is not corrupted. Active update queue is also restored using the checkpointed active update queue tail.

The following scenarios can occur and the state machine has to be recovered:

1. waymap_fillnum branch gets mispredicted as “Taken” branch:
   In this case, the next state points to index1_iter++. Since the branch was actually not taken, the next state is set as maparp.

2. waymap_fillnum branch gets mispredicted as “Not Taken” branch:
   In this case, the next state points to maparp. Since the branch was actually taken, the next state is set as index1_iter++. We move to next index in index1 array.
3. maparp branch gets mispredicted as “Taken” branch:
   In this scenario, the fillnum branch is already resolved as \{NT\}. The next state points to index1_iter++ instead of endindex, which needs to be recovered. Also, the index is pushed in front into the active update queue since this index has a pending active update (since the actual prediction is \{NT, NT\} for fillnum and maparp).

4. maparp branch gets mispredicted as “Not Taken” branch:
   In this case, the next state points to endindex. Since the actual prediction is taken, the next state is set to point to index1_iter++.

   The maparp table is passively trained. Therefore, if a mispredict is detected, the maparp table is updated to reflect the actual prediction for that index, and avoid mispredicts in future.

   Another case which requires recovery to the state machine occurs when any of the branches after the eight instances mispredicts (PC = 0x17cf8 and PC = 0x17d04) and jumps to the beginning of `makebound2`. By the time mispredict is detected, the custom predictor pops multiple indices from input worklist. The input worklist and the state machine has to be restored to correct state before making predictions. Therefore, checkpoints for these branches are also required.

4.11 Handling limited resources

4.11.1 Branch predictor table size

   By simulating the custom predictor with different branch predictor sizes, we found that we require \(2^{18}\) elements in our branch predictor tables to ensure no aliasing. If the size of branch predictor table is reduced, we need to mask the upper bits of index, and access the branch predictor tables using the lower bits. This leads to aliasing of multiple indices mapping to same element, leading to huge loss in performance.

4.11.2 Worklist size

   By experimenting with different worklist sizes, we concluded that worklist size of 512 is required to ensure no overflow or underflow. If the size is reduced, the output worklist can overflow. In this case, we continue to active update our table but stop pushing the
index into the output worklist.

The input worklist can also underflow. In this case, we switch to default predictor for generating predictions. We continue to active update our tables using the stores at retire, so that there is no loss in accuracy for future iterations of \textit{makebound2}.

### 4.11.3 Active update queue size

A size of 16 elements is sufficient to hold indices with pending active update. Thus, active update queue never overflows.

### 4.12 PSM interface

Fetch stage requires interface with custom predictor to snoop on \textit{bound1p} store and the branches in \textit{makebound2} function which are to be predicted by custom predictor. Writeback stage needs to send a mispredict signal for recovery of state machine. Retire stage needs to snoop on stores to \textit{bound2p} array (figure 3.3) for active updates and update the output worklist and branch predictor tables.

### 4.13 Pipelined predictor

We observed a significant decrease in performance if we reduce the size of worklists and branch predictor tables due to aliasing. Therefore, to achieve good performance using the custom predictor, we require large branch predictor tables and worklists. To provide a single cycle prediction in fetch stage, accessing a big branch predictor table can be a cycle time bottleneck.

Therefore, we decided to pipeline the predictor and modeled the loss in performance due to pipeline latency. A queue of size equal to pipeline depth is required to maintain the predictions. A prediction is pushed into the queue by the custom predictor but is available ‘n’ (pipeline depth) cycles later. Therefore, we need to stall the fetch unit for the very first branch and at every mispredict. We don’t stall for every branch, since the custom predictor can continue to generate predictions after the first index is available.
Chapter 5

Results

5.1 Methodology

The custom predictor was implemented on 721sim alongside the default gshare predictor. 721sim is a detailed cycle-level execute-at-execute execution-driven C++ superscalar processor simulator based on RISC-V ISA [15] used in Prof. Rotenberg’s research lab and ECE 721 Advanced Microarchitecture course. All the results presented in this thesis are derived from simulations from 721sim with RISC-V binary compiled at O3 optimization level, and ref input set (473.astar_rivers_ref.322.0.20.gz). This simpoint makes calls to makebound2 repeatedly, and hence is used in our evaluation.

Baseline configuration is listed in table 5.1. 16 wide architecture is used unless otherwise mentioned for the evaluation of custom predictor.

The additional structures are required for custom EXACT branch predictor. The default configuration related to custom branch predictor is shown in table 5.2.

5.2 Custom predictor on different configurations

We evaluated our custom predictor with different configurations (perf BP, perf D$, perf I$, perf fetch) and for different fetch/retire width.

The results obtained are shown in figure 5.1. We evaluate the performance of baseline with the gshare branch predictor table of size 64K entries and global BHR length of 16.
Table 5.1: Baseline Configuration

<table>
<thead>
<tr>
<th>Fetch/Issue/Retire width</th>
<th>16/8/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active list</td>
<td>512</td>
</tr>
<tr>
<td>LSQ</td>
<td>256</td>
</tr>
<tr>
<td>Issue queue</td>
<td>128</td>
</tr>
<tr>
<td>D-cache</td>
<td>Assoc = 4, Block size = 64, Size = 64KB, MSHR = 32</td>
</tr>
<tr>
<td>I-cache</td>
<td>Assoc = 8, Block size = 64, Size = 64KB, MSHR = 32</td>
</tr>
<tr>
<td>L2-cache</td>
<td>Assoc = 8, Block size = 64, Size = 256KB, MSHR = 32</td>
</tr>
<tr>
<td>BP table size (gshare)</td>
<td>64K 2-bit entries (16KB)</td>
</tr>
<tr>
<td>Perfect I-cache</td>
<td>0/1</td>
</tr>
<tr>
<td>Perfect D-cache</td>
<td>0/1</td>
</tr>
<tr>
<td>Perfect fetch</td>
<td>0/1</td>
</tr>
</tbody>
</table>

Table 5.2: Custom Branch Predictor Configuration

<table>
<thead>
<tr>
<th>Worklist</th>
<th>512 18-bit entries (1.125KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>waymap_fillnum table</td>
<td>256K 1-bit entries (32KB)</td>
</tr>
<tr>
<td>maparp table</td>
<td>256K 1-bit entries (32KB)</td>
</tr>
<tr>
<td>Active update queue size</td>
<td>16 18-bit entries (36B)</td>
</tr>
</tbody>
</table>
Increasing the gshare table to 256K entries for baseline configuration also yields the same IPC across all configurations. This shows that increasing the size of branch predictor table has no effect on branch prediction accuracy.

For custom predictor evaluation, the default configuration in table 5.2 is used with gshare predictor table of size 64K entries (2-bit each) and custom predictor tables of size 256K entries (1-bit each). The x-axis shows the different configurations with 0 meaning real and 1 meaning perfect. Custom branch predictor improves performance across all configurations. With perfect caches and perfect fetch, the IPC increases by almost 5x.

![Figure 5.1: Results for different configuration for 16 wide processor](image)

On average across all configurations, we see a gain of 3.5x on 16 wide processor. Misprediction per kilo instructions (MPKI) decreases from 28 to 2.48. The misprediction rate for astar benchmark drops to less than 2% with custom EXACT predictor.

We observe a similar trend for 8 wide and 4 wide processor. The results are shown in figure 5.2 and figure 5.3 for 8 wide and 4 wide architectures respectively. On average, the IPC of 8 wide and 4 wide processor increases by 2.9x and 2.2x respectively across all configurations.
Figure 5.2: Results for different configuration for 8 wide processor

Figure 5.3: Results for different configuration for 4 wide processor
For a real processor (real caches and real fetch), we see a gain of 3x for 16 wide, 2.8x for 8 wide and 2.2x for 4 wide processor. We observe that the I-cache being perfect or real has no impact on performance. Having a perfect fetch increases performance significantly, since there are less mispredictions and hence the fetch unit is on correct path.

The performance of a 4-wide architecture with custom predictor is almost double the performance of a 16-wide architecture with just a default predictor. Therefore, we can save on power and area by reducing other resources and still achieve better performance.

5.3 Impact of aliasing in branch predictor table

There are two branch predictor tables which are accessed for making a prediction, waymap_fillnum and maparp. We study the effect of aliasing by reducing the size of a table, keeping the other sufficiently large such that no aliasing occurs for that table. The results are shown in figure 5.6 and figure 5.9. The results are shown for 16-wide architecture.

There is no aliasing for table size greater than 32KB, and hence no drop in IPC. The reduction in branch predictor table by a factor of 2, drops IPC significantly due to aliasing. Since, we actively update the waymap_fillnum table, the next index which maps to the same entry sees a 1 and predicts \{T\} when it should have predicted \{NT\}. As we reduce the size further, we see a decrease in IPC due to more indices mapping to actively...
updated entry.

There is a similar curve for IPC vs waymap_fillnum size for real caches and real fetch. Decreasing the size of table beyond 8KB drops IPC lower than the default predictor due to aliasing. One possible approach to reduce aliasing is to hash the lower bits of index with PC.

We next see the effect of reducing the sizes of waymap_fillnum and maparp table together. The results for 16-wide architecture is shown in figure 5.10.
As we can see from the results, there is a significant drop in IPC by reducing the sizes of branch predictor tables. Therefore, it is critical to maintain a table size, such that no aliasing occurs. Since this is a resource heavy predictor and branch predictor being on critical path in fetch unit, we considered pipelining the predictor, the results of which will be discussed later.

5.4 Impact of reducing the worklist size

Next, we study the effect of reducing the worklist size. We switch to default predictor when the input worklist becomes empty, and switch to custom predictor when `makebound2` is called for next iteration. Figure 5.13 shows the results obtained for 16-wide architecture.

![Figure 5.11: Configuration = 0,1,1,1](image1)

![Figure 5.12: Configuration = 0,0,0,0](image2)

![Figure 5.13: IPC vs worklist size](image3)

The IPC drop with worklist size is gradual and saturates to the IPC of default predictor beyond 128 elements, since it is the default predictor that is predicting more often than the custom predictor due to input worklist getting empty quite early.

5.5 Pipelining the predictor

Since reducing the branch predictor table sizes drops IPC significantly, we modeled the drop in IPC due to pipelining the predictor keeping the size of branch predictor table such that no aliasing occurs. Pipelining the predictor incurs penalty for the very first
branch and at every mispredict, since the queue of prediction is empty in these cases. The results for modeling the pipeline latency are shown in figure 5.14. The different bars show the IPC with different pipeline depth. The pipeline depth of 0 implies there is no pipelining in the custom predictor.

![IPC for different pipeline depth](image)

Figure 5.14: IPC for different pipeline depth

As we can see, for all configurations, the IPC drop is very minimal for the pipeline depth of less than 4. Even with the pipeline depth of 16, the IPC is not dropping drastically and there is a speedup of 2.9x compared to baseline. Therefore, it is significantly better to pipeline the predictor with large tables than to suffer huge penalty due to aliasing.
Chapter 6

Conclusion and Future Work

6.1 Summary

Load dependent branches are the leading cause of mispredictions. In astar benchmark, the two branches in `makebound2` function are highly mispredicted. Both these branches are load dependent and store alters the prediction of first branch leading to mispredictions by the default predictor.

For the PSM design, we proposed a highly customized branch predictor for the astar benchmark, which essentially mimics the execution of astar benchmark in hardware. The custom predictor has separate branch predictor tables which are actively updated by retiring stores, leading to accurate predictions for a branch in `makebound2` function.

With enough resources to ensure no aliasing, the predictor achieves an accuracy of more than 99% for branches in `makebound2` function. The custom predictor reduces MPKI for astar benchmark from 28 to 2.48, with a mis prediction rate of less than 2%.

Our results show that having application specific design alongside the default superscalar core targeted at improving certain characteristics of benchmark execution can significantly improve the performance of the benchmark.

The custom branch predictor is a new paradigm which can be implemented on PSM fabric and can be tuned to specific application, leading to better performance.
6.2 Future work

The above results show great performance improvement with custom predictor for astar benchmark. To implement on PSM fabric, we need to carefully analyze the interfaces required by custom EXACT predictor. We need to snoop at fetch stage to trigger the custom branch predictor. We also need to have an interface in writeback stage to detect a mispredict and in retire stage to snoop on stores for active updates.

Pipelining the predictor is a better alternative than reducing the sizes of structures as observed from above results. Therefore, it is worth designing a pipelined predictor to prevent this huge predictor from becoming cycle time bottleneck.

We then intend to implement the custom EXACT predictor in Verilog or do a high level synthesis (HLS) for cycle time and power analysis.

Also, other benchmarks can be explored for such opportunistic branches. A custom predictor can be designed to improve branch prediction accuracy for such benchmarks.
REFERENCES


