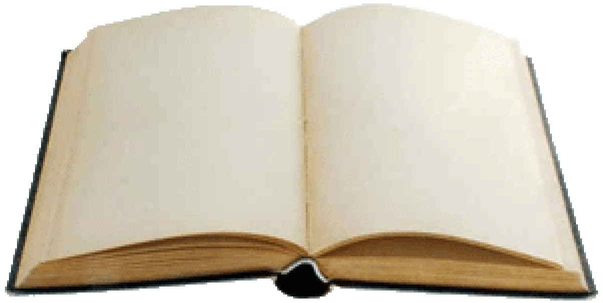
Instruction Fetching based on Branch Predictor Confidence

Kai Da Zhao
Younggyun Cho
Han Lin

2014 Fall, CS752 Final Project

Outline

Motivation

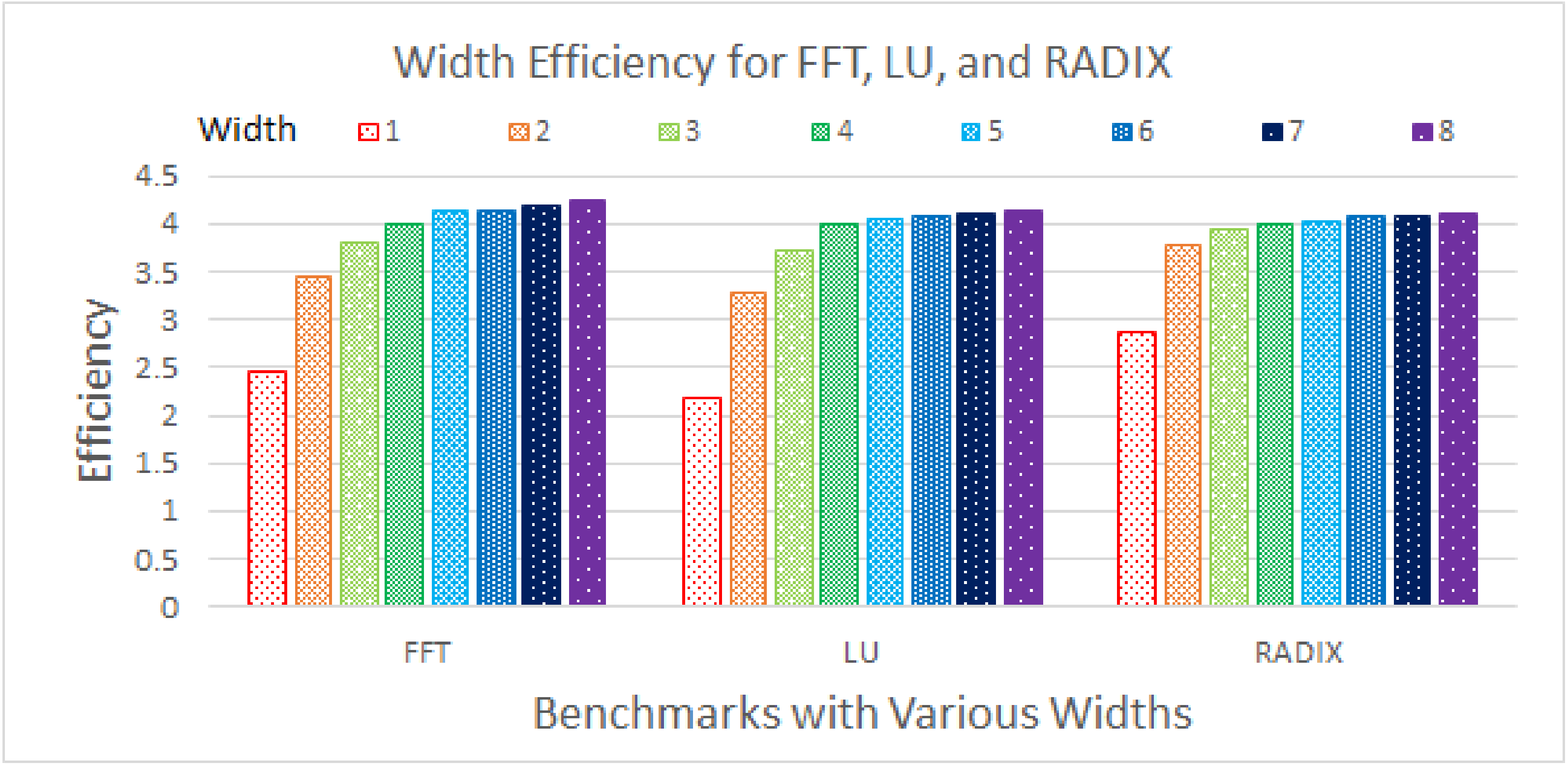


- Branches are hard to predict when there is no history or no pattern.

- High misprediction penalties for wide superscalar and deep pipelines.

- Direction prediction is difficult while target prediction is easy. Branches are primed for fetching from both paths.

Motivation

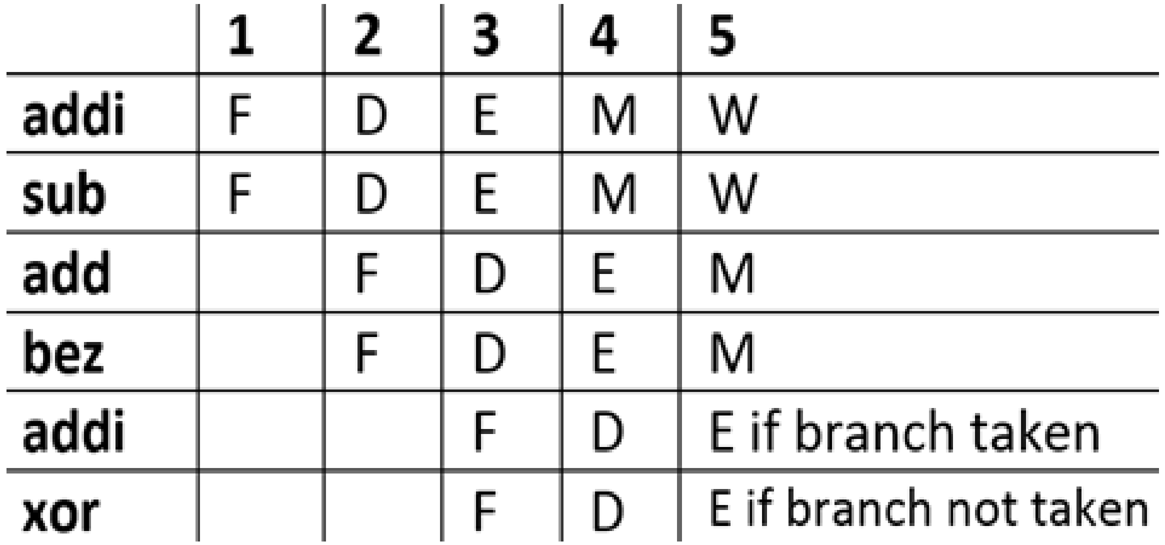

Width Efficiency for FFT, LU, and RADIX
Width
1
2
3
4
5
6
7
8
Efficiency
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
FFT
LU
RADIX
Benchmarks with Various Widths

Project Goals

- Reduce misprediction penalties
  - Use branch predictor confidence to selectively fetch instructions from branch paths
- Find optimal weight for each bit in branch history

Superscalar Dual Path Execution Example

```
loop: addi   r1, r1, 1
      sub    r2, r2, r3
      add    r4, r4, 1
      bez    r1, loop
      xor    r5, r5, r2
```



	1	2	3	4	5
addi	F	D	E	M	W
sub	F	D	E	M	W
add		F	D	E	M
bez		F	D	E	M
addi			F	D	E if branch taken
xor			F	D	E if branch not taken

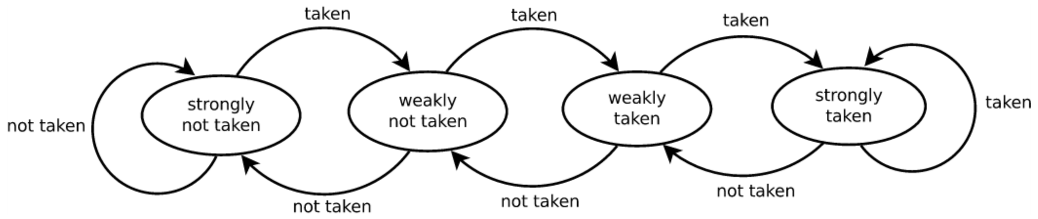
Related Works

- Dynamic hammock prediction[1] for multi-path execution

[1] A. Klauser, T. Austin, D. Grunwald, and B. Calder. “Dynamic Hammock Predication for Non-predicated Instruction Set Architectures”. *International Conference on Parallel Architectures and Compilation Techniques,* 1998.

Related Works

- Assigning confidence to conditional branch predictions[2]
    - Ones Counting
        - 01011101 gives 5/8, predict taken with low confidence
    - Saturating Counters
    - Resetting Counters



[2] E. Jacobsen, E. Rotenburg, and J. E. Smite. “Assigning Confidence to Conditional Branch Predictions”. *25th International Symposium on Computer Architecture,* 1996.

Proposed Confidence Assignment

- Exponent (for 8 bits)

$$\sum = \sum_{i=1}^{8} i^{exponent}$$

Weight		**Position**							
Exponent	∑	**7 (Least Recent)**	**6**	**5**	**4**	**3**	**2**	**1**	**0 (Most Recent)**
n	∑	1^n / ∑	2^n / ∑	3^n / ∑	4^n / ∑	5^n / ∑	6^n / ∑	7^n / ∑	8^n / ∑
0	8	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
1	36	2.8%	5.6%	8.3%	11.1%	13.9%	16.7%	19.4%	22.2%
2	204	0.5%	2.0%	4.4%	7.8%	12.3%	17.6%	24.0%	31.4%
4	8772	0.0%	0.2%	0.9%	2.9%	7.1%	14.8%	27.4%	46.7%

Proposed Confidence Assignment

- Weighted Ones Counting
  - combines ideas of all 3 confidence assignments
  - ones counting
    - use bits to represent taken or not taken
  - saturating counter
    - prediction sway towards recent history
  - resetting counter by
    - give more weight to recent branches

Proposed Fetcher for Various Widths

Since we vary the bit weights, we can set the fetching action for simplicity

	Confidence when to Fetch Number of Instructions Taken Branch								
Width	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
2	00.0%-33.3%	33.3%-66.7%	66.7%-100%	N/A	N/A	N/A	N/A	N/A	N/A
4	00.0%-20.0%	20.0%-40.0%	40.0%-60.0%	60%-80%	80%-100%	N/A	N/A	N/A	N/A
8	00.0%-11.1%	11.1%-22.2%	22.2%-33.3%	33.3%-44.4%	44.4%-55.6%	55.6%-66.7%	66.7%-77.8%	77.8%-88.9%	88.9%-100%
n	00.0%-1/(n+1)%	1/(n+1)%-2/(n+1)%	2/(n+1)%-3/(n+1)%	3/(n+1)%-4/(n+1)%	4/(n+1)%-5/(n+1)%	5/(n+1)%-6/(n+1)%	6/(n+1)%-7/(n+1)%	7/(n+1)%-8/(n+1)%	8/(n+1)%-9/(n+1)%

Modifying GEM5 (Part 1)

gem5/cpu/src/pred

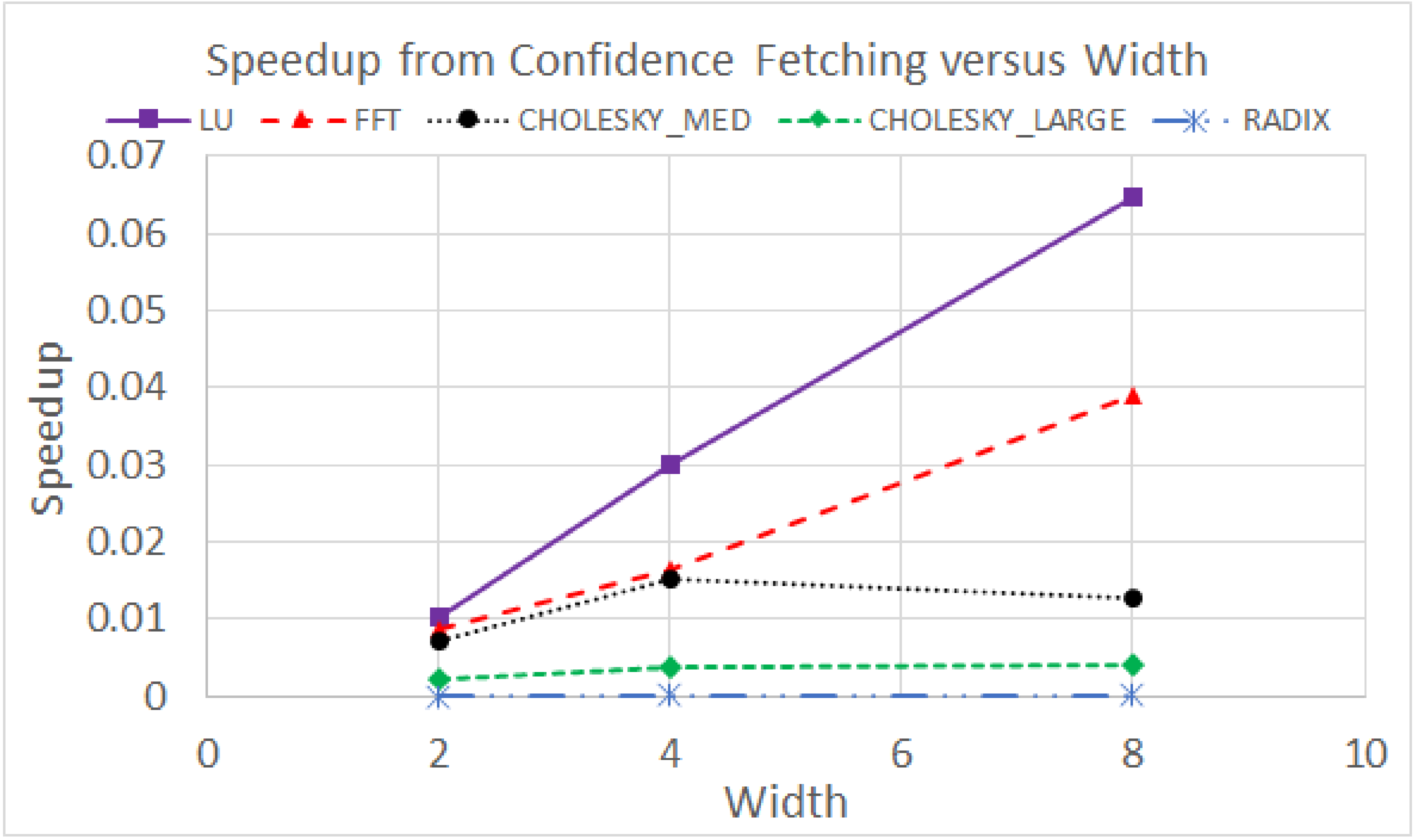
- changed predictTaken from bool to double
  - returns confidence instead of MSB
- changed from 2bit_local to 8bit_local
- changed from saturating counter to weighted ones counting
  - shift left, then increment if taken
- changed bpred_unit_impl to display branch instruction address and whether the branch was taken

Modifying GEM5 (Part 2)

gem5/cpu/src/inorder/resources

- changed branch_predictor to display branch instruction address and whether the branch was taken
- changed execution_unit to display prediction confidence

GEM5 Simulation Results

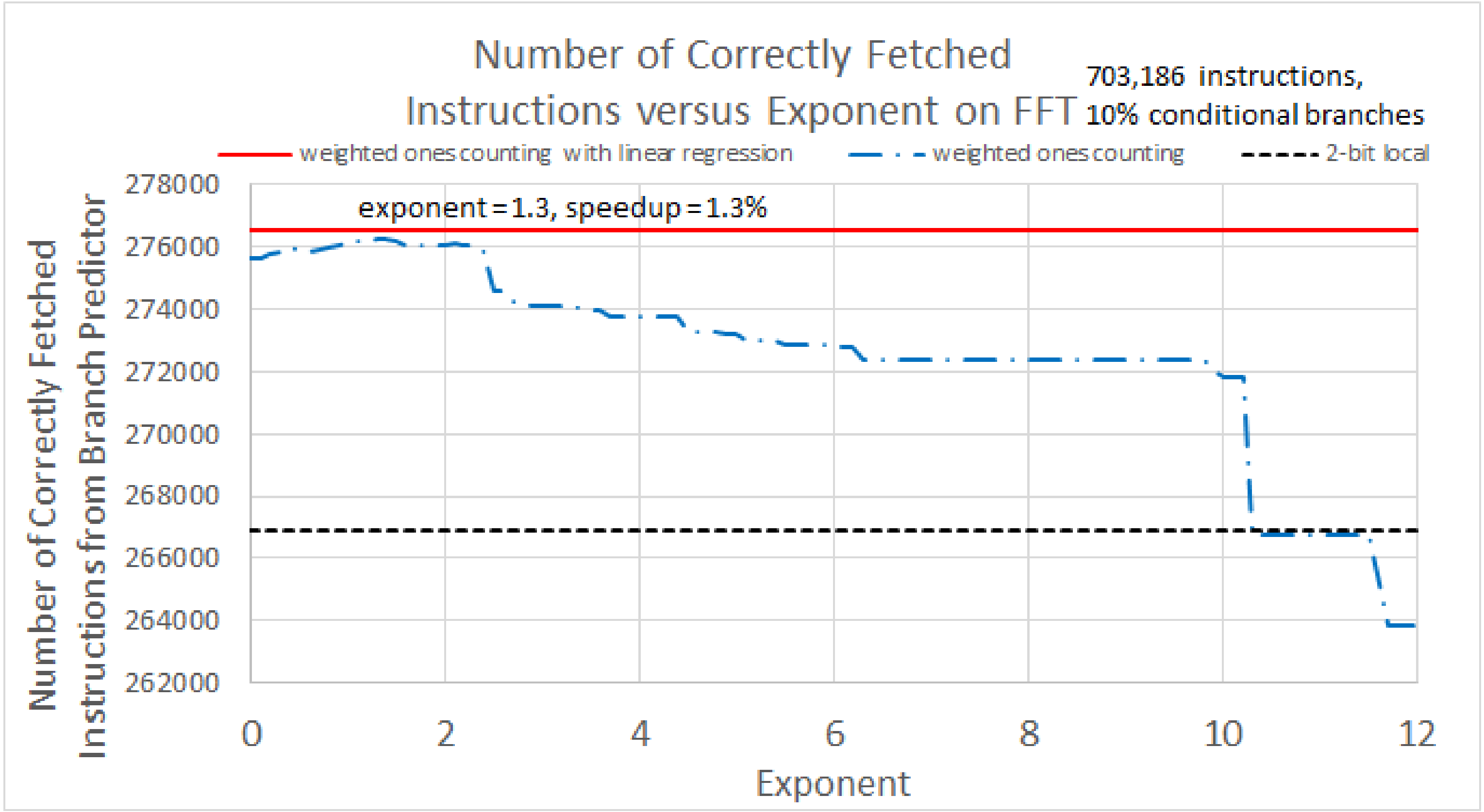


Optimal Exponent

- We implemented exponents = 2 in gem5. We want to find the optimal exponent
- Extracted program trace of branch instruction addresses from GEM5 and whether they were taken
- Used Multiple Linear Regression to find minimal sum of errors between sum of coefficents of bits and actual branch result

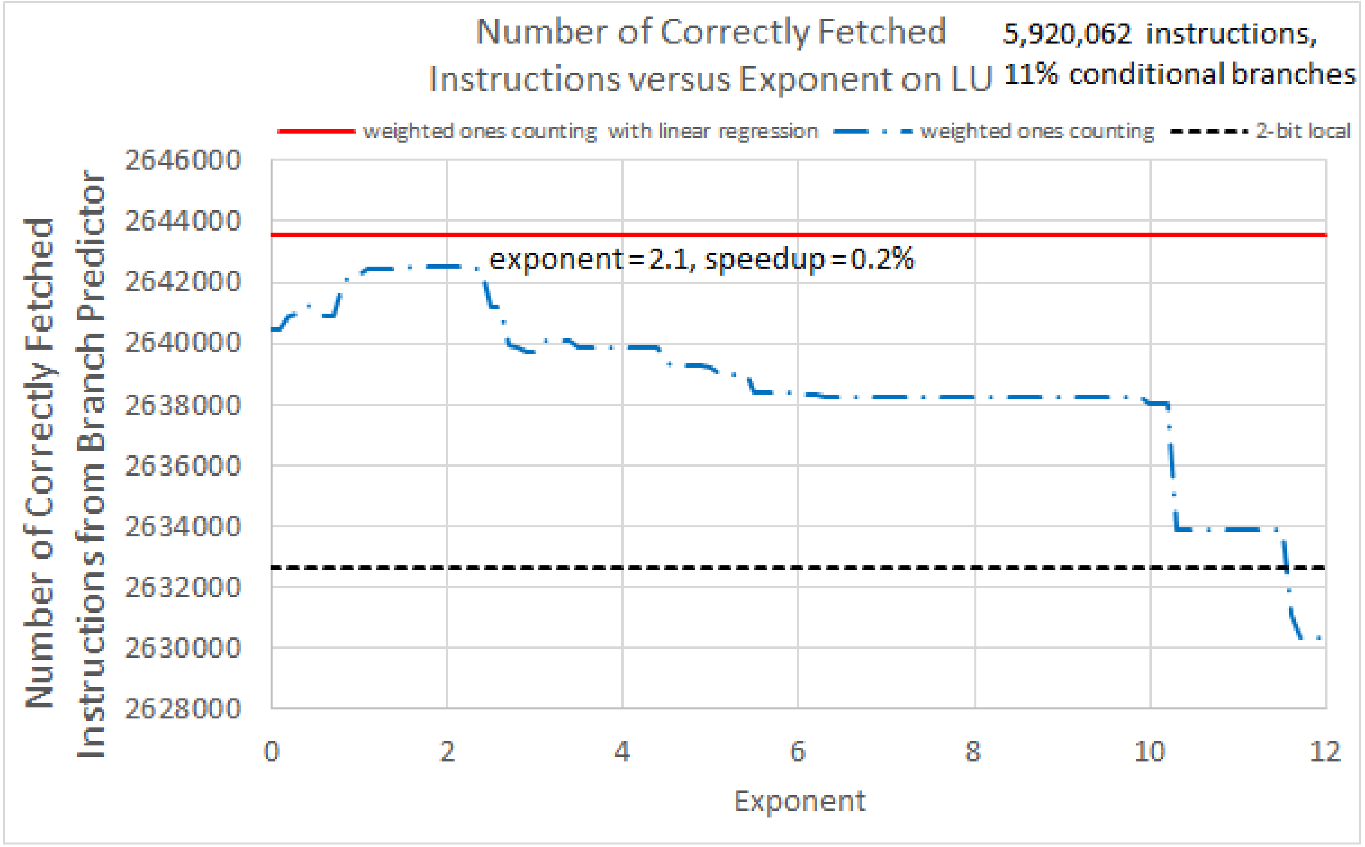
Regression Results

confidence = 0.0952bit[7] - 0.0794bit[6] + 0.0935bit[5] - 0.0567bit[4] + 0.2506bit[3] + 0.0465bit[2] + 0.3569bit[1] + 0.2546bit[0] + 0.0024



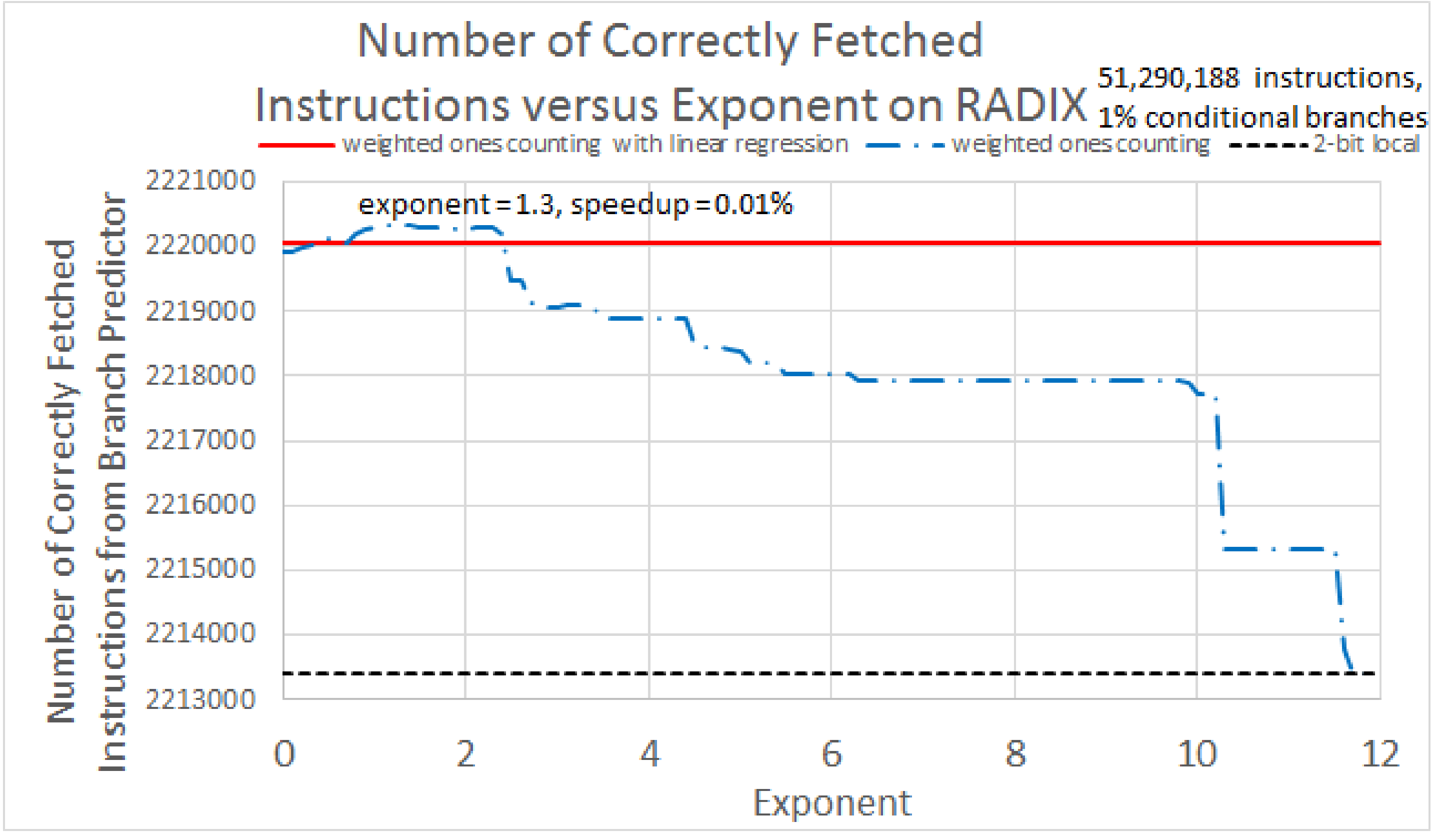
Regression Results

confidence = 0.1218bit[7] - 0.2389bit[6] + 0.1938bit[5] - 0.1708bit[4] + 0.1948bit[3] + 0.2301bit[2] + 0.2649bit[1] + 0.3622bit[0] + 0.007



Regression Results

confidence = 0.0728bit[7] - 0.1183bit[6] + 0.1268bit[5] - 0.0750bit[4] + 0.2580bit[3] + 0.1137bit[2] + 0.2975bit[1] + 0.3010bit[0] + 0.001



Conventional vs Proposed Fetcher

- Conventional Fetcher
  - Fetching from a single path
  - Harsh penalties for mispredicts
- Proposed Fetcher
  - Insurance policy for low confidence predictions
  - Performs relatively better on mispredictions, relatively worse to correct predictions

Conclusion

- up to 7% performance gain
  - Depends on percentage of branches
  - Low implementation cost
  - More power efficient
  - Less mis-fetched and mis-issued instructions
- Optimal exponent is 1.3-2.1 for width = 4

References (Questions?)

[1] A. Klauser, T. Austin, D. Grunwald, and B. Calder. “Dynamic Hammock Predication for Non-predicated Instruction Set Architectures”. *International Conference on Parallel Architectures and Compilation Techniques,* 1998.

[2] E. Jacobsen, E. Rotenburg, and J. E. Smite. “Assigning Confidence to Conditional Branch Predictions”. *25th International Symposium on Computer Architecture,* 1996.

[3] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. “Profile-assisted Compiler Support for Dynamic Predication in Diverge-Merge Processors”. *International Symposium on Code Generation and Optimization,* 2007.

[4] A. S. Al-Zawawi, V. K. Reddy, E. Rotenbberg, and H. H. Akkary. “Transparent Control Independence (TCI)”. *International Symposium on Computer Architecture,* 2007.

[5] A. Hilton, and A. Roth. “Ginger: Control Independence Using Tag Rewriting”. *International Conference on Parallel Architectures and Compilation Techniques,* 2007.

Instruction Fetching Based on Branch Predictor Confidence

Younggyun Cho, Kai Da Zhao, and Han Lin
{ycho52, kzhao32, hlin96}@wisc.edu

Department of Computer Sciences
University of Wisconsin-Madison
1210 W. Dayton St.
Madison, Wisconsin 53706-1685 USA

Abstract

Branch prediction evaluates the most likely execution path for the running program. Branches are often difficult to predict, especially when there is no history or no pattern. Furthermore, mispredicting branches will slow performance by requiring flushing the pipeline and squashing mispredicted instructions. One solution to alleviate this problem is to selectively fetch and execute instructions from both branch paths based on a confidence level. If the branch predictor is correct, then fetching instruction from both paths will have some misfetched instructions. However, if the branch predictor is wrong, then fetching instruction from both paths guarantees that at least one instruction was correctly fetched and the program can continue. Therefore, fetching instruction from both paths is only beneficial for cold branch predictors with low confidence.

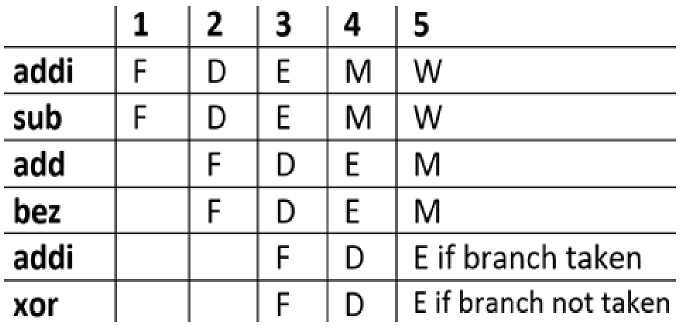
We propose a new method of extracting confidence to better determine how many instructions to fetch from each branch path. Then, we will demonstrate how to implement and simulate fetching based on confidence. Lastly, we will optimize our proposed work using linear regression to find the optimal weight for each coefficient. Our results show that fetching based on confidence can accrue up to 7% speedup with very low implementation costs.

1 Introduction

Cold branches predictors without local history or pattern will have low confidence with a high rate of misprediction. Moreover, many branch predictions are made with low confidence, costing steep penalties for wide superscalar and deep pipelines. This problem could be alleviated by the fetching instructions from both branch paths upon a low confidence prediction. Then, as the branch prediction gains more confidence, the fetcher can fetch more instructions from the predicted path.

The design of branch predictors provides necessary data and is primed for fetching from both paths. Branch prediction has two parts: the target predictor and the direction predictor. Direction prediction occurs at the decode stage and is difficult to predict / verify until execution. The branch PC target prediction occurs at fetch stage and is easy to predict / compute. Waiting until after the direction prediction is done at the execution stage will waste a cycle computation, which can heavily degrade performance depending on the percentage of conditional branches in the program. Therefore, we want to use a branch predictor with limited but sufficient information to improve performance.

```
loop: addi   r1, r1, 1
      sub    r2, r2, r3
      add    r4, r4, 1
      bez    r1, loop
      xor    r5, r5, r2
```



	1	2	3	4	5
addi	F	D	E	M	W
sub	F	D	E	M	W
add		F	D	E	M
bez		F	D	E	M
addi			F	D	E if branch taken
xor			F	D	E if branch not taken

Figure 1: This execution trace shows an example of dual path fetching on a superscalar computer. The bez instruction will fetch an instruction from both, the taken path (which branches to loop and fetch the addi instruction), and the not taken path (which falls through and fetch the xor instruction). Upon speculative fetching with low confidence, our proposed work will fetch instructions from both paths, which guarantees not to stall the critical path and improve the worst-case execution times.

Our proposed work will act like an autonomously adjustable insurance policy, costing high premiums for good coverage during low confidence, and costing nothing for no coverage during high confidence. Therefore, the insurance policy will automatically adjust with the branch confidence, allowing dual path execution to always perform at least comparable to single path execution.

This paper focuses implementing and optimizing the instruction fetching unit based on confidence. Section 2 presents the implementation, which is composed of how to assign confidence, generalize it for various widths, and implement it in gem5. Section 3 presents the experimental setup used to evaluate our hypothesis. Section 4 presents the performance of fetching from both paths versus fetching from one path. Section 5 describes related works and section 6 concludes the paper.

2 Implementation

Implementing dual-path execution has four portions: assigning confidence, optimizing coefficients, generalizing for various widths, and finally implementing in gem5 simulator. Assigning confidence is required to determine how many instructions to fetch from each path. Multiple linear regression finds optimal weights for bit to help determine the optimal performance of dual-path execution. Then, we demonstrate how to generalize our implementation for various widths so that it can be implemented on a generic system. Last, we show how to implement our results in gem5 for an accurate performance simulation.

2.1 Assigning Confidence

In related works, Klauser et al. [1] has already proposed dynamic hammock prediction for multi-path execution. However, their work references Jacobsen et al. [2], which proposed three methods of assigning confidence: ones counting, saturating counters, and resetting counters. Confidence can be extracted from ones counting by counting the number of times the branch predictor is recently correct. Confidence increases the more times the branch predictor is recently correct. Confidence can be extracted from saturating counters by gaining more confidence when the counter is highly or lowly saturated. Although saturating counters are not expected to perform better than ones counting, it can logarithmically reduce branch history table space. Resetting counters keep track of the number of good predictions since the last misprediction. Therefore, confidence increases as the counter increases, and resets to low confidence upon a misprediction.

Our proposed method of assigning confidence is to use weighted ones counting and give more confidence weight on more recent branches. If we want a monotonous method of extracting confidence, then we can give a weight to each bit position by multiplying the position by some exponent. Lastly, we divide the weights by the sum to keep the confidence between 0 and 1, so that the confidence is comparable. If the exponent is 0, then all bits are weighted equally and it will be exactly the same as ones counting. If the exponent is 1, then more recent bits will have linearly more weight than older bits. And likewise, if the exponent is 2, then recent bits will have quadratically more weight than older bits.

Weight		**Position**							
Exponent	∑	**7 (Least Recent)**	**6**	**5**	**4**	**3**	**2**	**1**	**0 (Most Recent)**
n	∑	1^n / ∑	2^n / ∑	3^n / ∑	4^n / ∑	5^n / ∑	6^n / ∑	7^n / ∑	8^n / ∑
0	8	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
1	36	2.8%	5.6%	8.3%	11.1%	13.9%	16.7%	19.4%	22.2%
2	204	0.5%	2.0%	4.4%	7.8%	12.3%	17.6%	24.0%	31.4%
4	8772	0.0%	0.2%	0.9%	2.9%	7.1%	14.8%	27.4%	46.7%

Table 1: Proposed Confidence Assignment. This figure demonstrates a method to extract confidence with monotonous weights based on the bit position.

2.2 Optimizing Coefficients with Multiple Linear Regression

A monotonous method of assigning confidence forces recent bits to have more weight than older bits. This may not be optimal because certain bits may be more weights to capture certain branching patterns. Therefore, we use multivariate regression to find the optimal weight of each bit.

We used gem5 to print the address of every conditional branch instruction and whether it was actually taken. Then we wrote a program to simulate ones counting to obtain the actual value of the counter at each branching instruction. We fed the counter and whether the branch was taken into MATLAB and used the `mvregress()` (multivariate regression) command to obtain the coefficient estimates for each bit. Linear regression analysis is widely used to find relationships between variables under the assumption of independently and identically distributed errors. In a simple linear regression model, a single response measurement is related to a single predictor, but our model is a regression with more than two variables called a multiple regression. Therefore, we used multiple linear regression to find the minimum sum of errors between each coefficient bit and the actual branch result.

2.3 Generalizing for Various Widths

Since we vary the confidence weight of each bit, we can fix the intervals between how many instructions to fetch. Two wide issues are divided into three intervals: two instructions are fetched from the not taken path if the confidence is less than 33%, one instruction is fetched from both paths if the confidence is between 33% and 67%, and two instructions are fetched from the taken if the confidence is greater than 67%. Four wide issues are divided into five intervals: four instructions are fetched from the not taken path if the confidence is less than 20%, three instructions are fetched from the not taken path and one instruction is fetched from the taken path if the confidence is between 20% and 40%, two instructions are fetched from each path if the confidence is between 40% and 60%, and so on. This pattern can be generalized for various widths by dividing the confidence levels equally by one plus the number of width. As the width increases, more confidence levels are required because the branch predictor should have more confidence before fetching many instructions from a particular path. The generic equations for various wide issues are shown the last line of table 2 below.

	Confidence when to Fetch Number of Instructions Taken Branch								
Width	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
2	00.0%-33.3%	33.3%-66.7%	66.7%-100%	N/A	N/A	N/A	N/A	N/A	N/A
4	00.0%-20.0%	20.0%-40.0%	40.0%-60.0%	60%-80%	80%-100%	N/A	N/A	N/A	N/A
8	00.0%-11.1%	11.1%-22.2%	22.2%-33.3%	33.3%-44.4%	44.4%-55.6%	55.6%-66.7%	66.7%-77.8%	77.8%-88.9%	88.9%-100%
n	00.0%-1/(n+1)%	1/(n+1)%-2/(n+1)%	2/(n+1)%-3/(n+1)%	3/(n+1)%-4/(n+1)%	4/(n+1)%-5/(n+1)%	5/(n+1)%-6/(n+1)%	6/(n+1)%-7/(n+1)%	7/(n+1)%-8/(n+1)%	8/(n+1)%-9/(n+1)%

Table 2: Proposed Fetcher for Various Widths. This figure demonstrates how to determine how many instructions to fetch from each path based on the confidence.

In order to fairly evaluate the performance of fetching few instructions from each path, we need to compare simulation time for running the program on various widths. Due to data dependencies, we expect diminishing returns as the width increases.

2.4 gem5 Implementation

Our proposed work is implemented in gem5, a simulator that integrates GEMS and M5. The gem5 simulator provides high accuracy and flexibility. We need to analyze code relevant to branch operations before starting to modify code to implement our proposed work in gem5. 'src/cpu/pred' is the main directory of the branch predictor. From the three different types of branch prediction strategies (2bit_local, bi_mode, and tournament), we implemented our proposed prediction with confidence modifying the local predictor. 'bpred_unit' linked to any types of CPUs can choose one of branch prediction strategies. 'bpred_unit_impl' provides the 'predict' function for OOO and the 'predictInOrder' function for inorder cpu based on the 'lookup' function generated from '2bit-local'. To complete our proposed work, we have to modify source codes to extract confidence levels from branch histories and to have controllability to issue the number of instructions to the branch taken path and the branch not taken path simultaneously or one of paths based on confidence levels. To vary the stage width, we modified 'InOrderCPU.py' and then, changed the predictTaken function from bool to double to return confidence levels instead of the most significant bit, which was implemented for a 2-bit local branch predictor. Due to the fact that we use an 8-bit shifting register to determine whether branch local branch predictor, we need to change the size of the branch predictor counter in 'BranchPredictor.py' from 2 bits to 8 bits and modify the increment and the decrement functions in 'sat_counter.hh' from saturating counter to weighted ones counting. The most recent branch is the least significant bit while the oldest branch is the most significant bit. Therefore, we shift the register to the left on every branch. Then if the branch is taken, we increment the value in the register. Branch predict implementation unit file is modified to display branch instruction addresses and whether actual branch was taken. We modify 'predictInOrder' function to execute based on confidence levels because 'predicInOrder' function receives its target address either from branch target buffer (BTB) or return address stack (RAS). The predictor will pop the return address from the RAS upon using it. In contrast, the predictor will set the PC to instruction's predicted target upon using the BTB. ‘bpred_unit.hh’ includes the param object that contains information of the size of the BP and BTB. ‘uncondBranch’ function tells the branch predictor to commit any updates and ‘squash’ function corrects the proper address for BTB and taken or not taken information for BP. If a branch is not taken due to the fact that the BTB address is not available, ‘lookup’ function will set the counter in the local predictors to not taken.

Several codes in ‘cpu/inorder’ directory are also required to modify for program trace. ‘branch_redictor.cc’ file has ‘execute’ function that requests resources based on available slot_num. If the requested command is a ‘PredictBranch’, it will forward the predicted PC to the advance PC. When the presently executing instruction is one of control instructions, ‘predictInOrder’ function in ‘bpred_unit_imp.hh’ file is called to receive the confidence level and the target PC. The confidence level determines the number of instructions to fetch from one particular path or multiple paths. As soon as the mispredicted branch is revealed, ‘squash’ function is performed to flush incorrect instructions. In ‘execution_unit.cc’ file, ‘execution’ function manages mispredicted instructions. It will set up the squash flag by calling ‘setPredTarg’ function and ‘setSquashInfo’ function.

3 Experimental Setup

SPLASH-2 kernel benchmarks include FFT, LU, RADIX, and CHOLESKY. We used SPLASH-2 kernel benchmarks to evaluate our proposed work in SE mode in the ALPHA architecture. The SPLASH-2 suite contains parallel applications that will be a good fit of our superscalar computer. We used SE to focus on a single thread with multiple issue widths.

We modifying gem5 to print the number of instructions predicted correctly and incorrectly with high, medium, and low confidence. The following two equations generalize performance gain and performance lost:

1) performanceGain = numberOfMispredictionsWithMediumConfidence + 2 * numberOfMispredictionsWithLowConfidence

2) performanceLost = numberOfCorrectPredictionsWithMediumConfidence + 2 * numberOfCorrectPredictionsWithLowConfidence

These two equations bring three important points into consideration when evaluating the performance of our proposed work. First, compared to the conventional fetcher, our proposed fetcher gains performance on mispredictions because fetching instructions from both paths allows the processor to continue upon a misprediction. Likewise, our proposed fetcher loses performance on correct predictions because of fetched instructions from wrong path. Second, predictions with high confidence have no impact on performance because our proposed fetcher will do the same thing as the conventional fetcher. Third, performance gains and lost is greater when confidence is lower. When predicting with low confidence, our proposed fetcher will likely going to fetch half of the issue width from both paths. Therefore, if a misprediction occurs, then the processor with our proposed fetcher can execute half of the issue width. On the other hand, if the prediction is correct, then the processor with our proposed fetcher will waste half of the issue width.

For evaluated how the exponent affects performance by using a constant width of four. Then, we varied the exponent from 0 to 12. Using an exponent of 0 weights all bits equally while using an exponent of 12 or greater will fetch all 4-wide instructions based on only the most recent branch. We hypothesize that there will be an optimal somewhere in-between. An exponent of 0 does not give enough weight to the most recent branch and causes multiple rounds of mispredicts before changing direction. On the other hand, an exponent of 12 will fetch all instructions based on the most recent branch, which will completely mispredict every time a branch changes.

4 Results

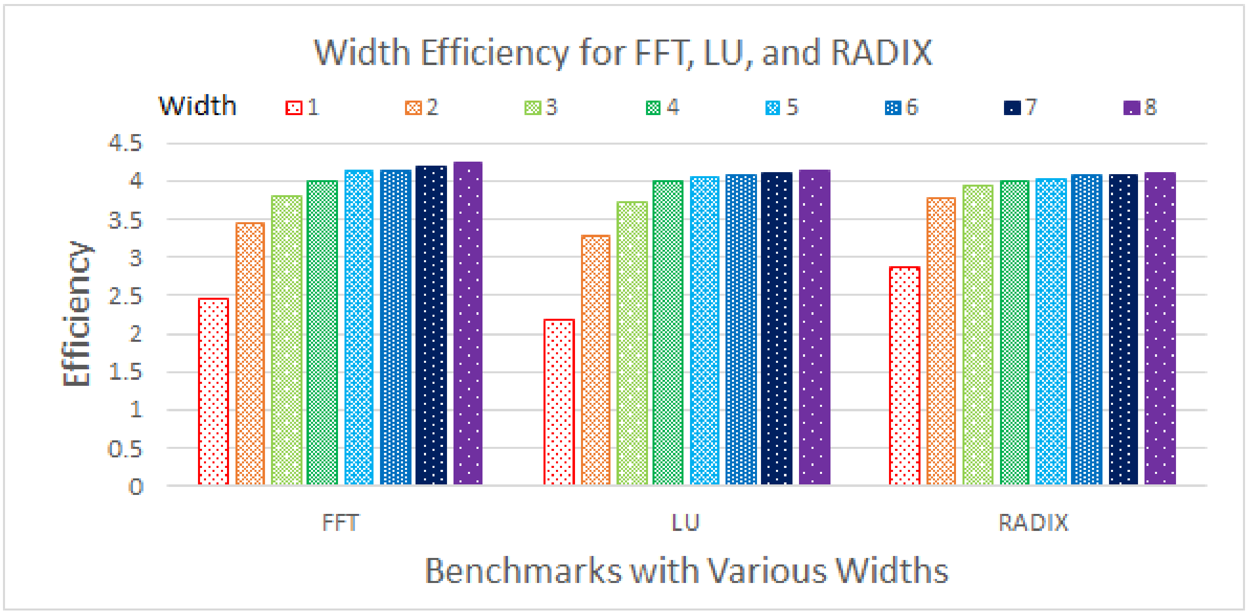


Figure 2: Width Efficiency for FFT, LU and RADIX.

We evaluated the efficiency of various widths by comparing the gem5 simulation time for various widths. Then we normalized the simulation time to 4 widths. The results in figure 2 shows width efficiency relative to 4 wide issue for our benchmarks. The figure shows that running FFT with one issue width will have more than half of the performance of eight issue widths. This result confirms that there are diminishing returns from fetching more instructions from a single path. Fetching additional instructions from a single path will only offer marginal performance improvement, and that it is very beneficial to fetch instructions from both paths.

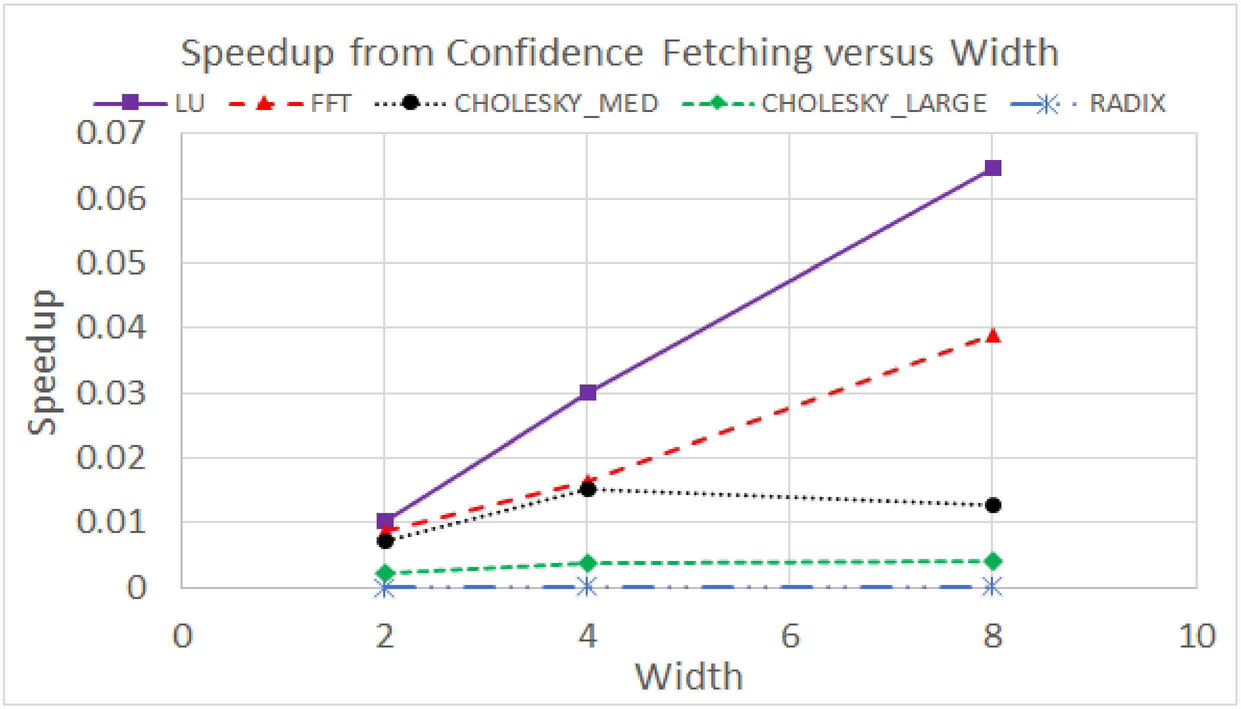


Figure 3: Speedup from Confidence Fetching versus Width.

The results in figure 3 show that speedup increases as issue width increases. FFT and LU have high speedups because branch predictor is often wrong with low confidence. CHOLESKY_LARGE and RADIX have low performance gain because both benchmarks have a relatively few number of branches and even fewer mispredictions for a large number of instructions.

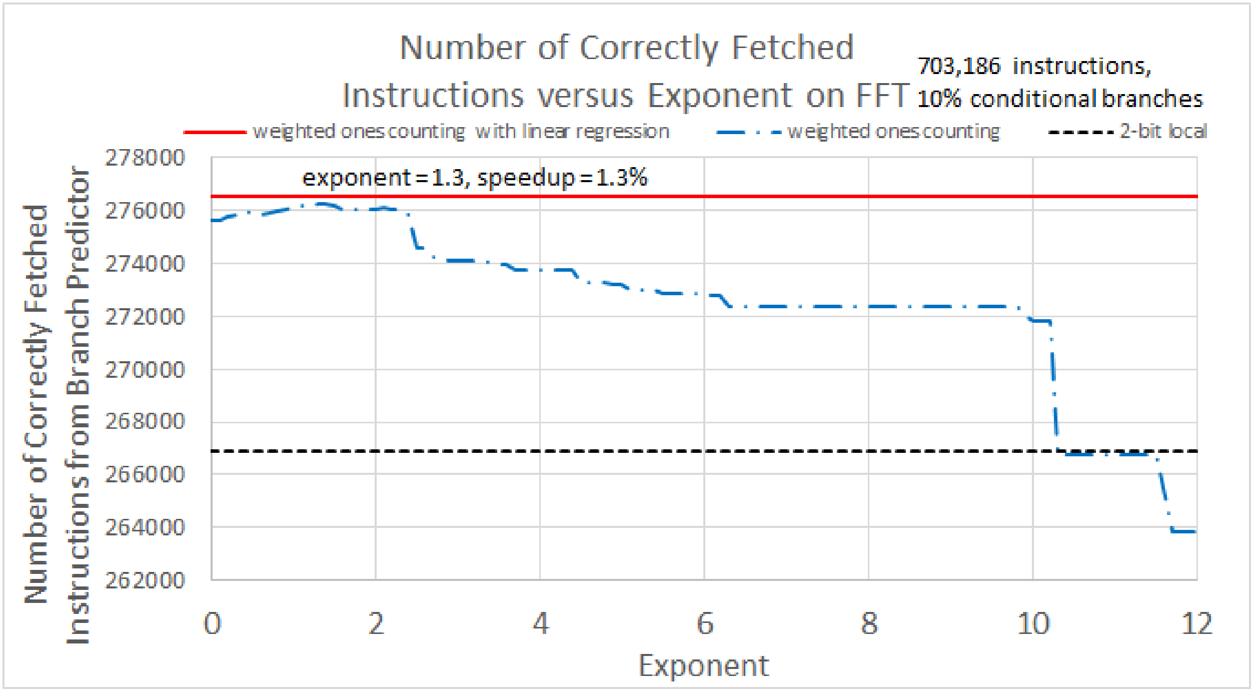


Figure 4: Number of Correctly Fetched Instructions versus Exponent on FFT.

FFT has about 10% conditional branches, and we can assume that any decent branch predictor is correct at least 50% of the time. Therefore, using Amdahl's law, we can accrue a speedup of up of 5% with an optimal branch predictor. We accomplished a speedup of 1.3%, which is relatively good compared to its maximum of 5%. For FFT, multiple linear regression gives confidence = 0.0952bit[7] - 0.0794bit[6] + 0.0935bit[5] - 0.0567bit[4] + 0.2506bit[3] + 0.0465bit[2] + 0.3569bit[1] + 0.2546bit[0] + 0.0024. Therefore, a lot of confidence is placed most recent, second most recent, and fourth most recent bits, meaning that the third most recent bit fails to capture the branching patterns of conditional branches in nested loops.

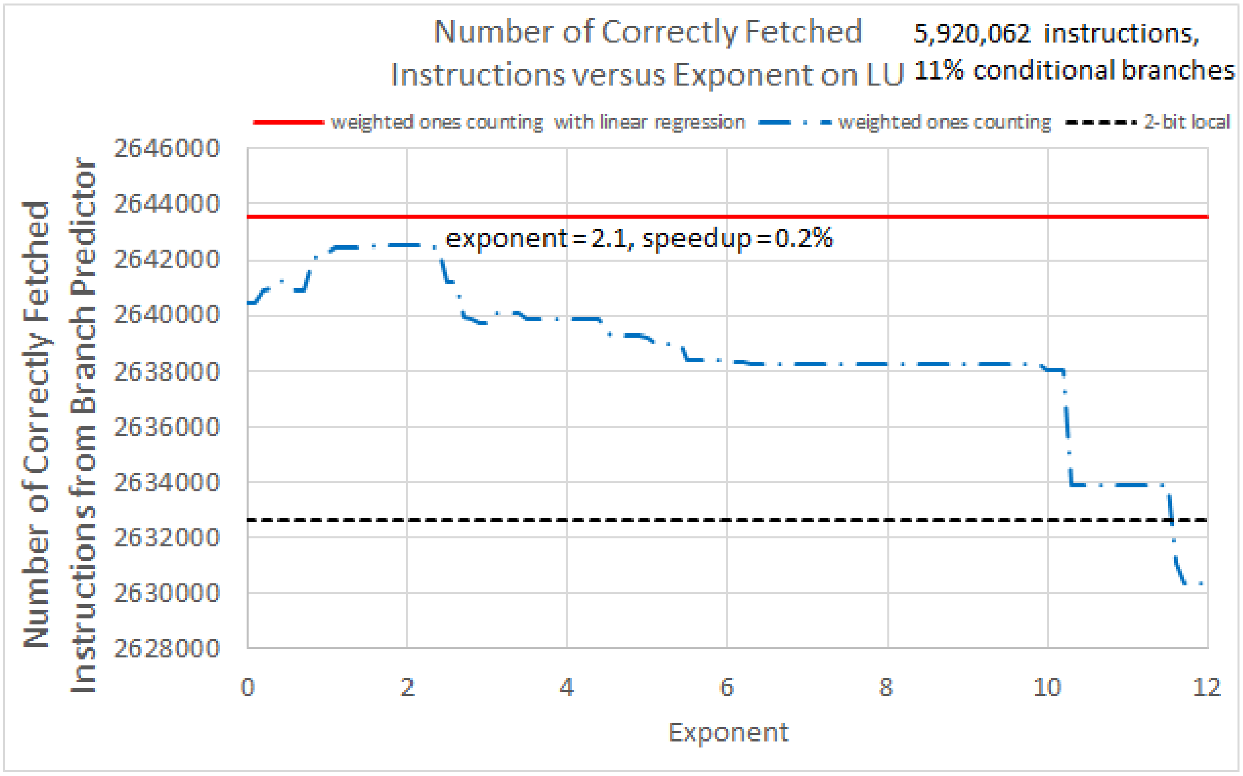


Figure 5: Number of Correctly Fetched Instructions versus Exponent on LU.

Again, the results show that fetching instructions from both paths based on weighted ones counting outperforms fetching all instructions from a single path. For LU, multiple linear regression gives: confidence = 0.1218bit[7] - 0.2389bit[6] + 0.1938bit[5] - 0.1708bit[4] + 0.1948bit[3] + 0.2301bit[2] + 0.2649bit[1] + 0.3622bit[0] + 0.007. Except for negative weights, the weight of each coefficient for the LU program does resemble that of the monotonous method.

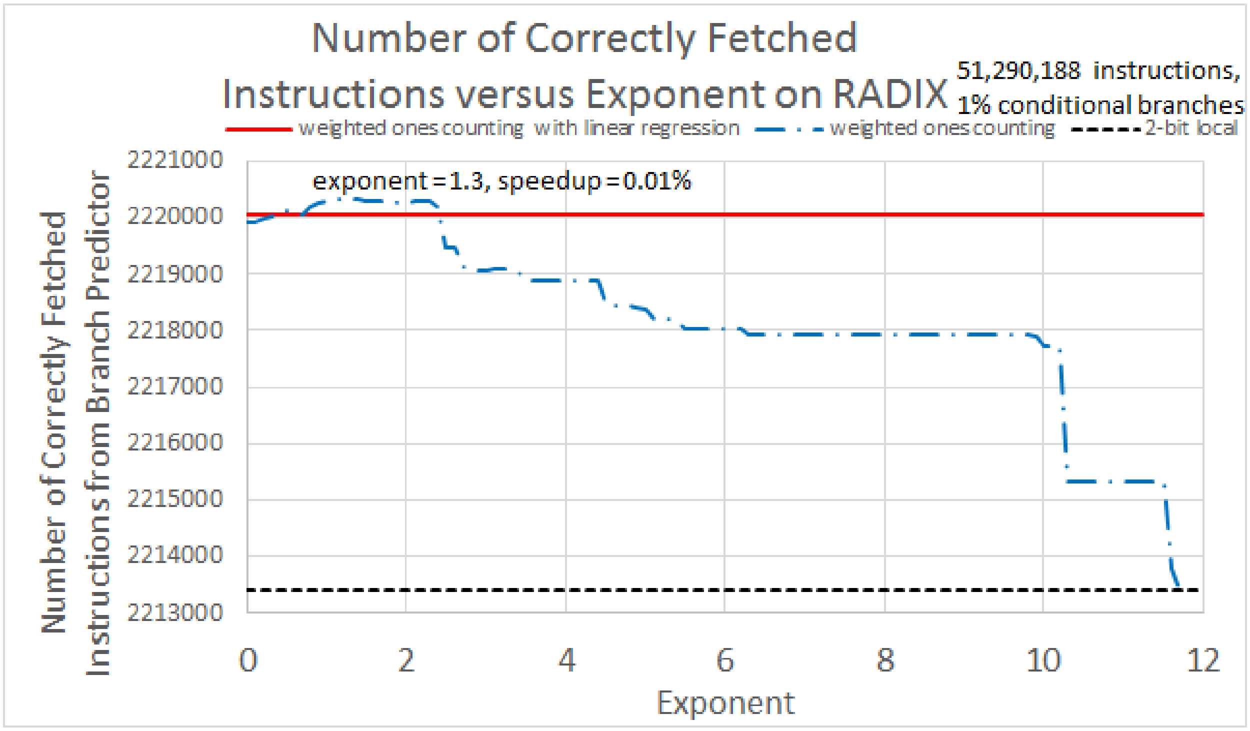


Figure 6: Number of Correctly Fetched Instructions versus Exponent on RADIX.

For RADIX, multiple linear regression gives: confidence = 0.0728bit[7] - 0.1183bit[6] + 0.1268bit[5] - 0.0750bit[4] + 0.2580bit[3] + 0.1137bit[2] + 0.2975bit[1] + 0.3010bit[0] + 0.001. Like FFT, RADIX also gives relatively low weight to the third most recent bit, which probably means that RADIX also have similar branching patterns of conditional branches in nested loops. Unlike the two previous benchmarks, the weighted linear regression actually performs worse for RADIX. This is because `mvregress()` attempts to minimize the square of the errors, without any knowledge of how the simulation will use the confidence value for variable fetching. In other words, multiple linear regression attempts to maximize the number of correct fetches. However, partially correct fetches are important as well, but was not considered.

For all three benchmarks, the results show that fetching instructions from both paths based on weighted ones counting outperforms fetching all instructions from a single path based on the 2-bit saturating counter. As predicted, weighted ones counting performs poorly when there is either not enough weight or too much weight assigned to recent bits. Giving too much weight to recent bits performs worse than giving too little weight. When recent bits have too little weight, older bits can steer the fetches to the correct path. However, if we put too much weight on recent bits, then mispredicts will occur when the branch predictor is cold and when the branch changes result, causing all widths to fetch incorrect instructions.

5 Related Works

The paper by Klauser [1] proposes dynamic predication for simple branch hammocks. Extra hardware is required to implement dynamic hammock predication, renamer, scheduler, and pipeline recovery support. Our proposed algorithm is very similar compared to dynamic hammock predication, because both work concurrently executes both paths of the branch. However, our proposed work has lower design complexities. This is because we plan on squashing instructions to avoid the need for renaming and checkpoints. Furthermore, we plan on focusing on various methods to classify branch predictor confidence and various issue widths.

The paper by Jacobsen [2] proposes three reduction functions for extracting confidence, which are 1) ones counting, 2) saturating counters, and 3) resetting counters. We will implement their three reduction function as well as one of our own, which combines ideas of all three. Furthermore, they used confidence to predict the percent of mispredictions. We will use confidence to determine how many instructions of each path to fetch.

The paper by Kim [3] proposes dynamic predication to reduce the branch misprediction penalty caused by hard-to-predict branch instructions. This paper mainly evaluates new code generation algorithms for dynamic predication. For high performance in the diverge-merge processor, compiler support is essential. Our proposed work is similar to dynamic hammock predication, because both works may fetch useless instructions. However, our proposed work can execute if-then-else statements without compiler supports and without ISA modifications. Our branch misprediction penalty is dependent on the branch predictor confidence.

The paper by Al-Zawaai [4] decouples misprediction dependent and independent instructions. The paper attempts transparent control independence by executing the misprediction-independent instructions out-of-order while waiting for the branch direction predictor. This is different from our proposed work, which focuses on the critical path in-order while waiting for the branch to resolve, because misprediction-independent instructions can be executed concurrently on the superscalar processor.

The paper by Hilton [5] also attempts control independence, but with tag rewriting. The paper's proposed architecture (Ginger) is used to find and replace the pipeline upon misspeculation instead of squashing. This is also different from our work. Our proposed work is content with squashing misspeculated instructions because of the correctly speculated instructions.

6 Conclusion

Regardless of a confidence level of a branch predictor, the conventional instruction fetcher simply fetches all instructions from a single path. Therefore, the conventional fetcher will incur large penalties upon a branch misprediction. Furthermore, performance penalties will be even greater for wider widths and deeper pipelines because processor will need to flush more mispredicted instructions. Our proposed work is like an insurance policy. It can fetch instructions from multiple paths based on confidence of a branch predictor. With low confidence, it will fetch instructions from both paths. Several instructions from a particular path are always correct. This can give relatively better performance on mispredictions. However, there exists a downside. Our proposed work also needs to squash several instructions fetched from the other path on correct predictions. Although this will decrease performance, our proposed fetcher has a higher performance gain over the conventional fetcher.

Speedup of our proposed fetcher is highly dependent on the percentage of conditional branches and the percentage of mispredictions. Based on our results, our proposed fetcher can achieves better performance in general and can accrue performance gains of up to 7%. Furthermore, with fewer misfetched and misissued instructions that needs to be squashed, our proposed work is more power efficient.

From our simulation results, we can determine the optimal weight of each bit of the local counter. In general, although using multiple linear regression does provide near optimal fetching based on confidence, each benchmark has its own set of weight for each bit. Multiple linear regression requires running the program at least once beforehand to extract the weight of each bit from the program trace. Therefore, using optimal fetching based on linear regression is only helpful for iterative tasks. On the other hand, using the monotonous method of extracting confidence with exponent = 2 will also provide a near optimal fetching. Furthermore, using exponent = 2 offers low implementation costs because computers are built using base 2. Therefore, the overall near optimal design of fetching based on confidence level is lightweight that can easily be integrated in many processors for faster performance.

Branch target buffers are fixed-size hash tables that map PC to branch target address. Perleberg [6] suggests that BTB can reduce the performance penalty of branches. If the branch predictor predicts taken, we cannot fetch any instructions from a branch taken path unless BTB has a target address. To obtain better performance, we need to consider BTB performance with our proposed work because the type and amount of information stored in a BTB has a significant impact on the performance of branch instructions. There are already several research papers that evaluate BTB designs. In future works, we will examine our proposed work with various BTB designs by changing number of the BTB entries and size of BTB tags in gem5 simulator to further improve performance.

Acknowledgments

We would like to thank Professor David Wood for providing us with some related works, suggesting to generalize our implementation for various widths, and suggesting to do linear regression to find the optimal weight of each bit. We would also like to thank Nilay Vaish for teaching us how to run benchmarks on gem5.

References

[1] A. Klauser, T. Austin, D. Grunwald, and B. Calder. "Dynamic Hammock Predication for Non-predicated Instruction Set Architectures". *International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[2] E. Jacobsen, E. Rotenburg, and J. E. Smite. "Assigning Confidence to Conditional Branch Predictions". *25th International Symposium on Computer Architecture*, 1996.

[3] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. "Profile-assisted Compiler Support for Dynamic Predication in Diverge-Merge Processors". *International Symposium on Code Generation and Optimization,* 2007.

[4] A. S. Al-Zawawi, V. K. Reddy, E. Rotenbberg, and H. H. Akkary. "Transparent Control Independence (TCI)". *International Symposium on Computer Architecture*, 2007.

[5] A. Hilton, and A. Roth. “Ginger: Control Independence Using Tag Rewriting”. *International Conference on Parallel Architectures and Compilation Techniqu*es, 2007.

[6] C. H. Perleberg. “Branch Target Buffer Design and Optimization”. *Technical Report COMP TR*, Rice University, May 1989.