**Implementation of Tournament predictor in SimpleScalar.**

***Abstract:*** *In this project we have implemented the tournament predictor in simplescalar-3.0. Tournament predictor is a dynamic predictor which predicts the predictor. Tournament predictor takes advantage of choosing the right predictor for the right branches.*

**I.INTRODUCTION**

In computer architecture, control hazards arise from the pipelining of branches and other instructions that change the program counter. Basically, whenever we see a branch, jump, transfer, interrupt, or return in a loop you have the possibility of a control hazard. That is, if you haven’t optimized our code, we will have to face some stalls.

When a branch is executed, it may likely will receive some sort of stall or problem. The problems that arise from this include: When a branch is executed it may or may not change the Program Counter and thus may not change the instruction that is to be fetched. This means the next instruction have already been loaded into the pipeline, whether they are correct or not.

The program Counter is not changed until the end of MEM stage in the DLX pipeline. This will result in a three cycle stall.

If we use more hardware it is possible to determine the branch address and condition in the ID stage, resulting in a one cycle delay.

To solve these problems, we have to either stall the pipeline or predict the branch taken or not taken. If we stall the pipeline, the worst situations would arise if you always stall the pipeline no matter what happens. In this case we should always assume one of the following so as to atleast save some time in the pipeline.

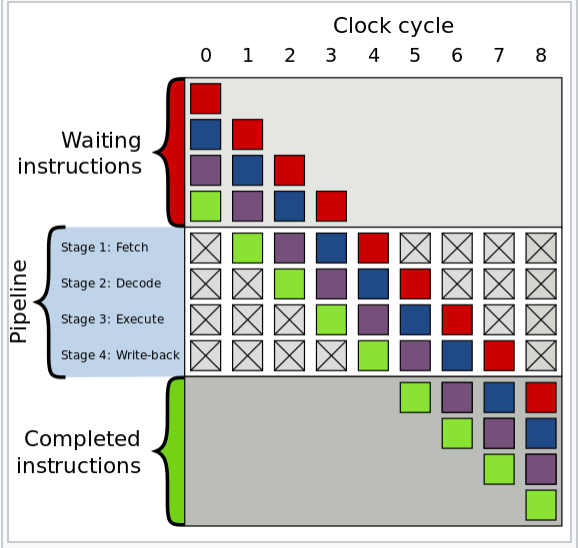
Control hazards can cause a great performance loss for the DLX pipeline than data hazards. When a branch is executed, it may or may not change the PC (program counter) to something other than its current value plus 4. If a branch changes PC to its target address, it is a **taken branc**h; if it falls through, it is **not taken**.

So as to reduce the control hazards, it is important to have branch prediction which is the most important aspect on whose basis the implementation is done.

In computer architecture, a branch predictor is a digital circuit that tries to guess which way a branch will go before that is known definitively. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving high effective performance in many modern pipelined microprocessor architectures such as x86.

In fig1, Each of the color boxes represent instructions independent of each other.

Two-way branching is usually implemented with a conditional jump instruction. A conditional jump can either be “not taken” and continue execution with the first branch of code which follows immediately after the conditional jump, or it can be "taken" and jump to a different place in program memory where the second branch of code is stored.



**Fig1:** Example of 4 stage pipeline.

It is not known for certain whether a conditional jump will be taken or not taken until the condition has been calculated and the conditional jump has passed the execution stage in the instruction pipeline.

Without branch prediction, the processor would have to wait until the conditional jump instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline. The branch predictor attempts to avoid this waste of time by trying to guess whether the conditional jump is most likely to be taken or not taken. The branch that is guessed to be the most likely is then fetched and speculatively executed. If it is later detected that the guess was wrong then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay.

Branch prediction attempts to guess whether a conditional jump will be taken or not. Branch target prediction attempts to guess the target of a taken conditional or unconditional jump before it is computed by decoding and executing the instruction itself. Branch prediction and branch target prediction are often combined into the same circuitry.

Branch prediction are categories into two major types: Static and Dynamic.

Static prediction is the simplest branch prediction technique because it does not rely on information about the dynamic history of code executing. Instead it predicts the outcome of a branch based solely on the branch instruction. Static branch predictors are used in processors where the expectation is that branch behavior is highly predictable at compile time. Being able to accurately predict a branch at compile time, it is also helpful for scheduling data hazards e.g. loop unrolling. In static prediction, all decisions are made at compile time. This does not allow the prediction scheme to adapt to program behavior that changes over time. Thus affecting the accuracy and latency of the processor.

Taken and Not taken are the two most important static branch predictions.

Dynamic branch predictors on the other hand uses information about taken or not taken branches gathered at run-time to predict the outcome of a branch. Dynamic branch prediction is better than static branch prediction.

Two level, bimodal and combined are the three dynamic branch predictions that are implemented in SimpleScalar.

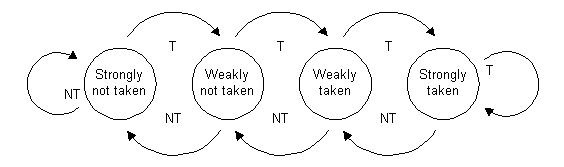
**II.DYNAMIC BRANCH PREDICTORS**

**2 bit predictors:** The predictor that fixes the two mispredictions per anomaly behaviour of the one bit predictor is called as a two bit predictor. The possible values are 00,01,10,11and the MSB bits specify what the predictions must be. The LSB/Convection bits tells how sure are we that we should be predicting what the prediction bit is saying.

00-Strong not taken 01-weak not taken

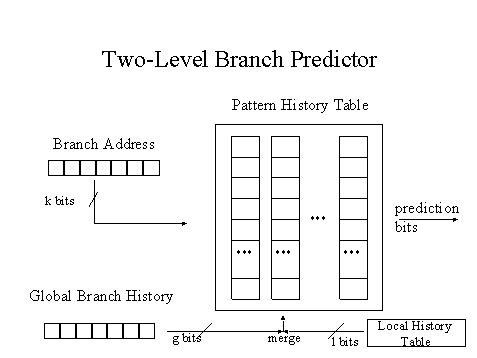
10-weak taken 11-strong taken

Only anomaly gets mis-predicted but not the dominant behaviour that follows the anomaly.



**Fig 2:** state diagram of two bit predictor.

**Correlating or 2 level predictor:** 2 bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch. Branch predictors that use the behavior of the other branches to make a prediction are called Correlating or two level predictors. Use of Global vs local behavior information. (m,n) predictor, with global info about the m branches and each of the predictors maintained as a n-bit predictor.



**Fig 3:** two level branch predictor block representation.

Suppose m=2, when we are trying to make a decision about a particular branch, we will have to look at how the earlier two branches got executed and depending on their behavior, we will decide about this branch. Looking at two branches we will again have 4 possibilities of 00,10,01,11.

So basically instead of maintaining the counter for each branch to capture a common case,

* Maintain a counter for each branch and surrounding pattern.
* If the surrounding pattern belongs to the branch being predicted, the predictor is referred as local predictor.
* If the surrounding pattern includes neighboring branches, the predictor is referred as global predictor.

**III.TOURNAMENT PREDICTOR**

**Concept:** Predict the predictor.

Hopes to select the right predictor for the right branch.

**Rule:** Use the predictor with better prediction result.

**Critical Reasons:**

* Some branch predictors perform well with one sort of programs, i.e. floating-point programs containing a lot of loops, but bad with others where different branch predictors are more successful.
* To avoid the negative effect of context switches it is also a good idea to combine a branch predictor with a large history table with another predictor with a small or no history table.

It uses 2 predictors, one based on local information and one based on global information and combine them with a selector.

It tries to implement multilevel prediction.

They are hence also called as **multi level branch predictors**

So both the predictors will be implemented and with the selection mechanism, the predictor is chosen.

Selection mechanism will vary with the requirement:

* Choose the other predictor on one mis-prediction or
* Switch the predictor only in case of two successive mis-predictions.

**Selector:** An additional 2-bit counter selector array helps to select the appropriate predictor for each branch. One bit saves the last time global correct/wrong. The other bit saves the local correct/wrong.

* **Updating the history tables:** All the branch History tables of the predictors are updated independently. (Predictor tables are updated on context switch) This is called as the default case. (optional=0). In optional case, we can skip to update the local predictor when global predictor is chosen and it provided the correct prediction.

The selector is updated accordingly on the following basis:

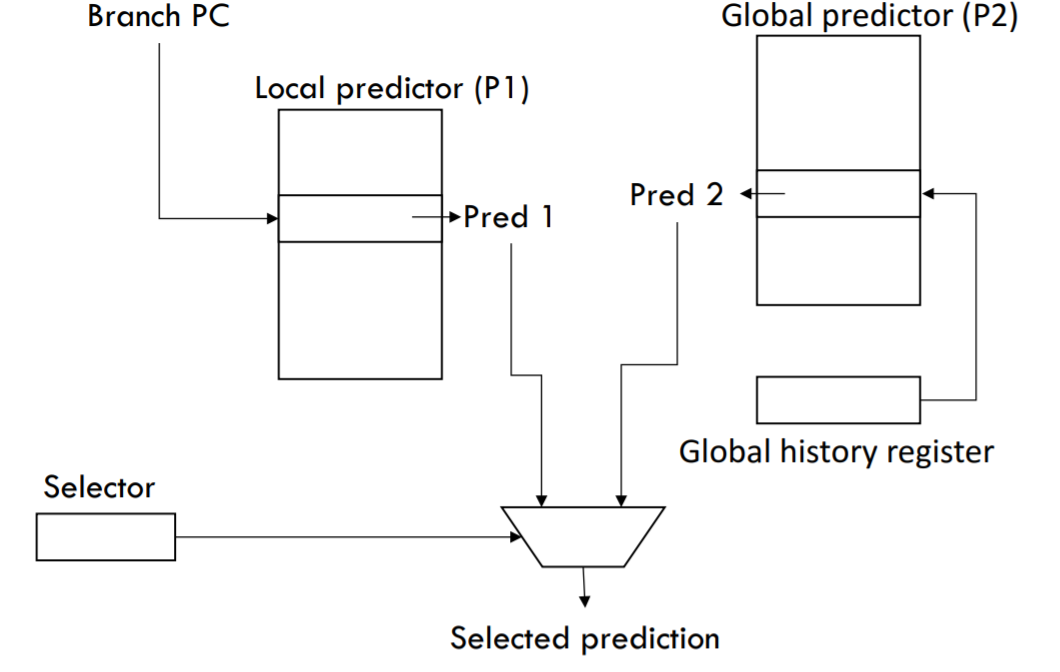


**Fig. 4:** Selector update rules.

Update the selector when two predictors disagree.

Note: In our implementation when the optional is chosen as 1, i.e. update only global history tables has some flaws as we have noticed that the miss rate was increasing at a higher rate. So we sticked to the optional value as 0 and performed the experimentation.

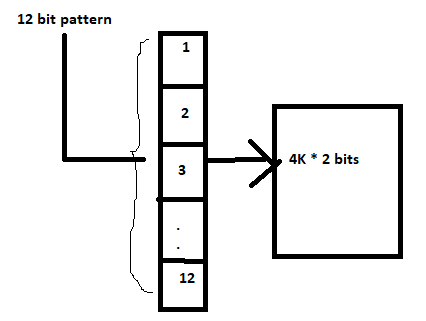
**Fig 5:** Block representation of Tournament Predictor:



The following are the implementation details of the tournament Predictor.

**Global Predictor:**12-bit pattern. It has 4k entries and is indexed by history of last 12 branches with each entry being a 2-bit entry-2bit predictors. (0-nottaken;1-taken).

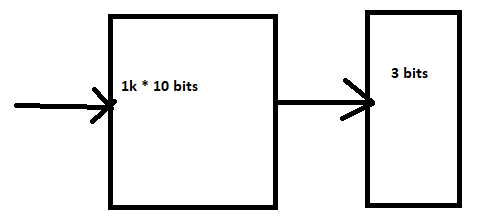
**Local Predictor:** Is a two level predictor, history table with 1k 3-bit predictor entries. (10bits)



**Fig. 6:** Global predictor block representation- size details.

a.*Top level:* a local history table consisting of 1024 10bit entries; each 10 bit entry corresponds to the most recent 10 branch outcomes for the entry. This 10 bit history allows patterns 10 branches to be discovered and predicted. i.e whether a branch was taken/not taken for 10 times in a row. History entry: 1 1 1 1 1 1 1 1 1 1 1 ->branch was taken 10 times in a row.

b. *In the next level:* A selected entry from the local history table is used to index a table of 1k entries consisting a 3bit saturating counters, which provide the local prediction. 3-bit predictor. Counter which is used to select the local / global predictor.



**Fig 7:** Block representation of the local predictor-size details.

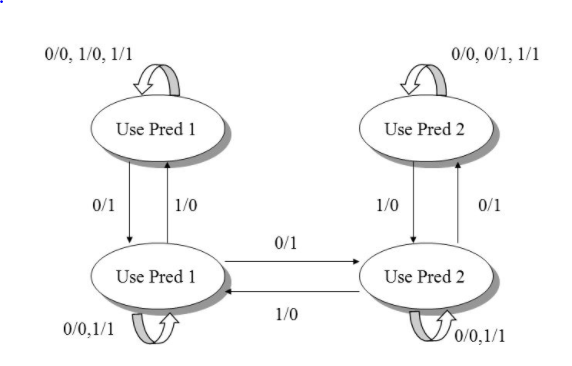
*Selector:* 4k 2bit counters from local/global predictors.Only after 2 mis predictions, the selector will change the predictor. The transition/ the selection of the predictor by the selector can be observed with a state diagram representation which is shown as in fig 8.

***IV.*IMPLEMENTATION DETAILS**

The local predictor is a two-level table which records the history of individual branches. It consists of a 1,024-entry by 10-bit branch history table. A two-level table was used as the prediction accuracy is similar to that of a larger single-level table while requiring fewer bits of storage. It has a 1,024-entry branch prediction table. Each entry is a 3-bit saturating counter. The value of the counter determines whether the current branch is taken or not taken.

The global predictor is a single-level, 4096-entry branch history table. Each entry is a 2-bit saturating counter; the value of this counter determines whether the current branch is taken or not taken.

The choice predictor records the history of the local and global predictors to determine which predictor is the best for a particular branch. It has a 4,096-entry branch history table. Each entry is a 2-bit saturating counter. The value of the counter determines if the local or global predictor is used. Increment or decrement the counter based on correct/ wrong predictions.



**Fig.8:** State diagram representation of the selector of tournament predictor.

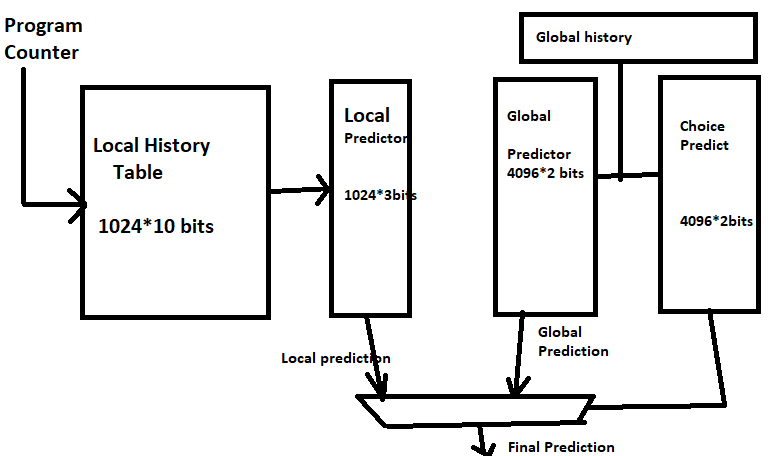
So, the design details can be outlined as follows:

*Global predictor:*12-bit global branch history register. This indexes 4k(2^12) entries of 2-bit predictors.

*Local Predictor:* Two level predictor

History table with 1k 10-bit branch history registers. Indexes into a table with 1k(2^10) 3-bit predictor entries.

*Selectors:* 4k 2-bit chunters to choose from global/local predictors.



**Fig.9:** Size representation of the predictors implemented.

For the implemenation of tournament predictor in SimpleSclar, the breapd.c, bpread.h, sim-outorder.c,sim-breapd.c files were modified.

The main changes that are to be made for functionality building and defining the prediction selector was done in bpread.c file. The global and local components along with the selector are included in this. The local component is defined so that it traverses 2level tables and uses pointers for prediction state information.

Based on the fig.8, the selector was designed to change the predictor from global to local or from local to global only after two mis predictions. This could be achieved by simple c programming. If the selector is in 00,01 states then the local predictor is chosen and if the selector is using 10,11 then global predictor is being used.

In bpread.h the sizes are defined as stated earlier. All the parameters for the tournament predictor: number of 2 bit selector entries, size of global branch history table, size of local branch history registers and size of local predictor table size. The sim-bpread.c and sim-outorder.c were modified for including the tournament predictor. The existing meta-predictor was used for making necessary modifications to meet the requirements.

**V. MODIFICATIONS FOR SIMULATION.**

**`**The following is the modification that was changed in the simulation files for the tournament predictor.

* **-bpred tournament –bpread tournament<sel\_size> <global\_regize><local\_htb\_size><local\_hrsize><optional>**

The following are the definitions of the parameters:

<sel\_size>:no of 2 bit selector entries

<gloabl\_regsize>:gives the size of global branch history register which tells the size of global predictor table size.

<local\_htb\_size> :gives the size of history table for local branch history registers.

<local\_hr\_size>:gives the size of each local branch history register and inturn the local predictor table size.

Eg : -bpred tournament -bpred:tournament 4096 12 1024 10 0

**VI. DIFFICULTIES FACED DURING IMPLEMENTATION.**

Initially we were trying to install simplescalar with our euids in the putty, but faced “make” troubles and received continuous errors that makefile was missing even though it was existing, we have tried several ways and commands but could not figure out the issue so used only one of our artemis machine. So the entire part of work was initially done on general C compilers to make sure that there were no errors during our modifications and then tried to include bits of work and check how it was functioning in one of our putty.

SimpleSacalar Hackguide was like a complete initial guide for us during the initial stages.

For implementing the table update function, as there were two cases default and optional- we could not verify the implementation of the optional case where the global history tables are updated if the predictor chose the global predictor and the prediction is correct; the local history tables update can be skipped.

**VII. RESULTS**

The following are the results that we could summarize for this predictor in caparison with the bimodal and two level predictors that are already existing in SimpleScalar.

CPI, total number of branches executed and the no. of misses was measured and tabulated.

|  |  |  |  |
| --- | --- | --- | --- |
| CPI | | | |
| **Benchmark** | **two level** | **bimodal** | **tournament** |
| anagram | 2.5095 | 2.4896 | 2.4729 |
| perl | 2.5746 | 2.5127 | 2.5275 |
| compress 95 | 1.6451 | 1.6954 | 1.6331 |

|  |  |  |  |
| --- | --- | --- | --- |
| **sim\_total\_branches** | | | |
| **Benchmark** | **two level** | **bimodal** | **tournament** |
| anagram | 1193 | 1193 | 1034 |
| perl | 6514 | 6274 | 6394 |
| compress 95 | 17390867 | 18014984 | 17270435 |

|  |  |  |  |
| --- | --- | --- | --- |
| **bpred.misses** | | | |
| **Benchmark** | **two level** | **bimodal** | **tournament** |
| **anagram** | 246 | 633 | 210 |
| **perl** | 1071 | 587 | 799 |
| **compress 95** | 1078922 | 1446253 | 846777 |

**Graphs:**

The following is the plot representing the CPI among the 3 predictors.

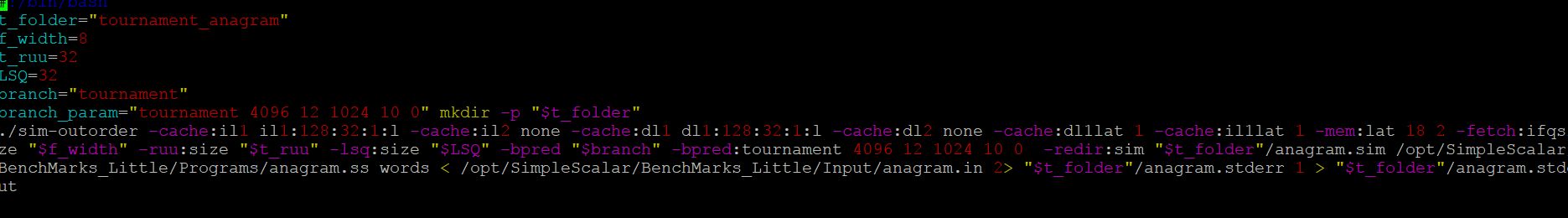
As seen in the above graph, the CPI was not any high for the tournament predictor which might be an implementation issue.

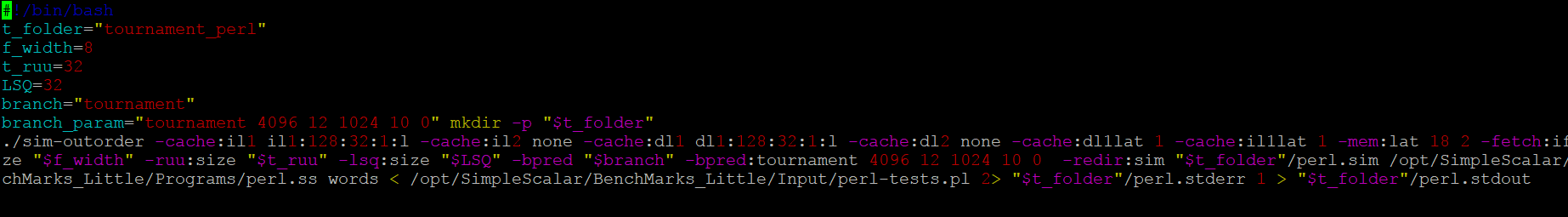
**Scripts:** The script files have been included along with the source files. The following are the sample screenshots taken while running the scripts in the three benchmarks compress95, anagram and perl.

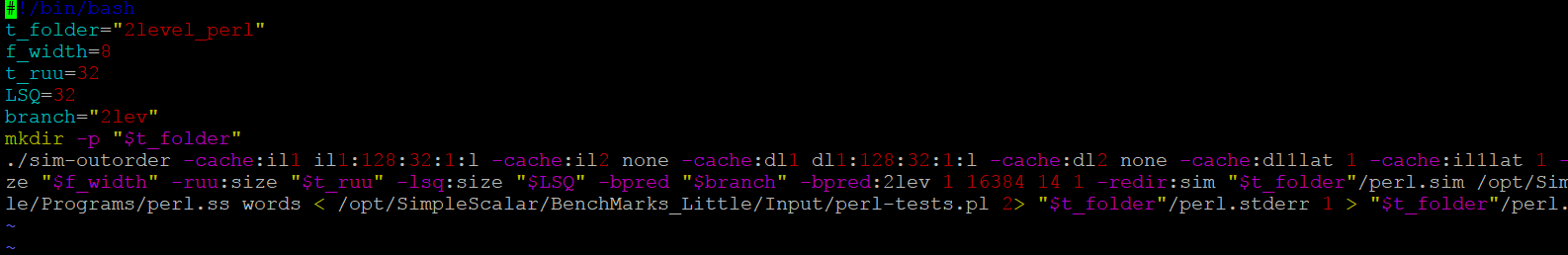
Further comparisons can be made by modifying the tournament predictor parameters like the no. of selectors.

**Advantages:** Theoretically it is said that Tournament predictor works better than the bimodal and two level predictor. This predictor is highly accurate as it uses 29k bits during prediction i.e. large number of predictin bits are used effectively.

Also, tournament predictors can achieve both the better accuracy and latency at medium sizes (8k-32kbits).







**VIII. REFERENCES**

The primary reference based on which the implementation was started was [1].

[1].two level predictor implementation.

[2]. **“**Alternative Implementations of Two-Level Adaptive Branch Prediction**”-** Tse-Yu Yeh and Yale **N.** Patt.

[3]. “Branch prediction using both global and local branch History information” – M.C Chang, Y.W. Chou.

[4]. J. E. Smith "A Study of Branch Prediction Strategies" <em>Proceedings of the 8th International Symposium on Computer Architecture</em> pp. 135-148 1981-May.

[5].  T-Y Yeh Y.N. Patt "Two-Level Adaptive Branch Prediction" <em>The 24th ACM/IEEE International Symposium and Workshop on Microarchitecture</em> pp. 51-61 1991-Nov.

[6]. Several videos on youtube and blogs also helped us to understand and gain knowledge on the implementation of the predictor.