Loop Branch Analysis of Championship Branch Prediction Traces

Ali Yasser Ismail

August 11, 2014

1 Motivation

Every year the JILP Workshop on Computer Architecture holds a Branch Prediction Championship where contestants develop branch predictors and attempt to achieve the best accuracy. The branch predictors run on a simulator that iterates through 40 traces/programs to test their accuracy. The mean accuracy of all 40 traces is calculated to determine the overall performance of the branch predictors [1].

In order to perform well in this competition, it is advantageous to identify the branches, determine their behavior, and design a predictor around this behavior to increase performance accuracy. According to Sherwood et al. [2], programs with deeply nested loops can create a significant amount of mispredictions. Many accurate branch predictors utilize global and local histories of previous branch activity to make accurate predictions for future activity. Issues arise because global and local histories do not contain enough resources to capture the entire pattern of deeply nested loops. Sherwood et al. gives a general example of when this occurs: if the loop termination is of the pattern $((1)^{N}0)^{m}$, where 1 represents taken and 0 not taken, then this pattern cannot be captured if N is larger than the local history register. This pattern also cannot be captured by a global history G^1 that is smaller than N because it will not have the visibility to see the entire loop[2].

In order to attain this visibility, a profiler was designed to track the different types of branches, their histories, and targets. With this information, the branches were categorized to point a predictor designer in a more refined direction. For example, rather than focus on all conditional branches that contain dead or glacial executions, a designer can focus more on the conditional branches with irregular behavior. This will bring the designers efforts to more complex problems and set aside the easier tasks of predicting branches that always go taken or have a very predictable behavior.

Likewise, the profiler provides global information on branch behavior. The traces may not make all conditional calls consecutively. There may be other branch activity that surrounds conditional branching. For example, there may be a direct call branch to a function, a few conditional loops, and then a return branch that returns to the main program. By studying the global behavior, the predictor designer will gain insight on the main program and the function calls.

All in all, branch analysis give a predictor designer excellent visibility on the competition traces, which will allow them to focus their efforts in design and give them ideas on how to

develop a predictor. The profiler provides visibility that can reduce unwanted phenomena such as the compounding effect of mispredictions caused by deeply nested loops. It also provides insight on the branches execution activity and behavior. This paper will describe the functionality of the profiler and use the SHORT-INT-1.cbp4.gz trace from the CBP competition to serve as an example of what insight it is capable of providing. Section 2 describes related work that helped give ideas in creating a profiler, section 3 describes the objective of this work, section 4 describes the methodology to developing the profiler, section 5 describes future work, and section 6 conclusion.

2 Related Work

Sherwood et al [2] studied the dramatic effects of deeply nested loops with high percentage execution. As discussed earlier, deeply nested loops can cause a large amount of mispredicts. To eliminate these mispredicts, they developed a Loop Termination Buffer (LTB), hardware capable of detecting the termination of loops branches. By detecting potential terminating loops, a predictor can use this information to help it make decisions. By analyzing the loop trip count (number of times in a row the loop is taken) and loop iteration count (the number of times the branch has been since it was last not taken), the LTB can accurately identify deeply nested loops.

Similarly, Patent US590573[3] describes a counter based branch prediction system. Yet again, counters are used to track the branch behavior at a per-branch basis. Tracking the run-time results of programs and their branching behavior allows a predictor to make a more informed decision. This prediction system, like the TLB, tries to identify loops and make decisions based on their loop counts.

3 Objective

The objective of this project was to develop a profiler that provides useful information on the traces provided by the Championship Branch Prediction (CBP) simulator. Branch types that include, conditional, unconditional, return, and direct call, are profiled based on branch percentage, execution percentage, and behavior. Branch behavior includes dead, glacial, loop, and irregular. Profiling behavior at the per-branch level for these traces gave fine grained insight on the functionality of each trace and which branches to focus on.

4 Methodology

4.1 Profiler Foundation

The CBP profiler utilizes a map of vectors data structure to profile activity on a per-branch basis. The map and vector data structures are provided by the C++ standard template library. The program counter (PC) serves as the key into the map data structure and the vector of that map key contains a series of counters. The counters are used to track the

behavior, execution activity, etc. The rest of this section will describe in detail the tracked information.

4.2 Branch Make-Up

The first information provided by the profiler is the branch make-up of a trace. It lists the number of branches and execution information of each of the following branch types:

- Conditional Branches
- Unconditional Branches
- Direct Call Branches
- Return Branches

The profiler uses a map of vectors, as described in section 4.1, to capture the PC of a branch, record the OPTYPE(branch type), execution count, and branch history. Figure 1 provides the partial output of the profiler that shows the PC branches, the number of their executions, their branch type, an branch history. For example, the branch with the PC 4212027 was executed 753 times, is of branch type 6 (Conditional Branch), and went taken 2 times, then went not taken, then went taken 2 times, and etc.

```
4211819 : 251 3 251
4211832 : 251 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4211843 : 251 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4211872 : 251 3 251
4211883 : 251 6 251
4211958 : 251 3 251
4211970 : 251 4 251
4211980 : 251 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4211987 : 251 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4212027 : 753 6 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
4212031 : 251 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 1: Raw stats (PC, execution count, branch history) of branches for the SHORT-INT-1.cbp4.gz trace.

Figure 2 shows a second output file of the profiler that indicates the % of branches of each branch type and % of total execution for each branch type. 83 % of the execution is made up of conditional branches for the example trace. Based on this information, a designer could make the conclusion that most of their focus should be on conditional branches, since they make up most of the execution. Also, it is interesting to note that this trace only contains 730 branches that are executed a large number of times (4989008 executions). Since there are a little amount of branches, there is not a lot for the designer to focus on.

Figure 2: Profiler generated branch stats for the SHORT-INT-1.cbp4.gz trace.

4.3 Branch Behavior

The current objective for the profiler is to identify the following branch behaviors:

- Dead Branches
- Glacial Branches
- Loop Branches
- Irregular Branches

By identifying these behaviors, the focus on 730 branches can be reduced. The dead, glacial, and loop behaviors are very predictable, thus, allowing the designer to design around their patterns. Also, by studying the behavior of the irregular branches, even greater accuracy can be achieved. Figure 3 contains the general branch break down for all the behaviors. Listed are the basic metrics for each behavior. They include:

- # of branches of that behavior type
- % of branches of that behavior type
- % of execution made up of that behavior type

4.3.1 Dead Branches

The first behavior the profiler identifies is dead branch behavior. Dead branches are consecutive all takens or consecutive all not takens. This behavior is very predictable and easily handled by most current branch predictors. The profiler analyzes the branch history shown

in Figure 1 to identify any dead branches. When they are identified, they are stored in their own map of vectors and the rest of the branches are stored in a new map of vectors to be analyzed later. Figure 3 contains useful metrics on dead branches. For the SHORT-INT-1.cbp4.gz trace, dead branches make up 592 branches of the 730 branches. This is roughly 81.1% of the branches. In addition, dead branches make up 41.1% of the total execution of this trace. All in all, this means the predictor designer can push the focus towards the remaining 138 branches, which make up the remaining 58.9% of the execution. Again, these branches are dead and their pattern is easy to detect. They comprise a large percentage of the workload, thus, the designer should push the focus on the remaining execution.

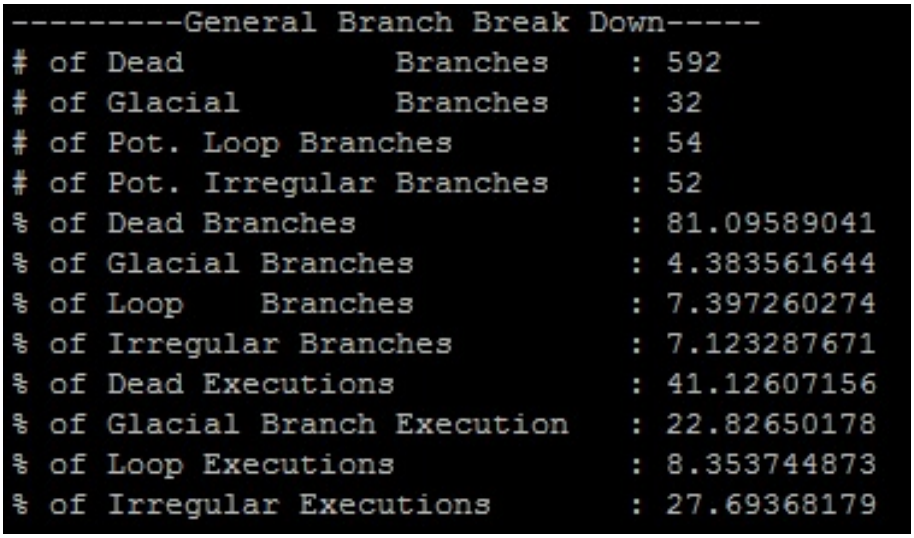

Figure 3: Output of "9_branch_stats.txt"

Similarly, the profiler puts extra focus on the dead branch identification. It lists the types of branches that are dead branches, which is shown in Figure 4. Compare the totals in Figures 2 and 4, and notice how all unconditional, direct call, and return branches are all taken dead branches. Also, all not taken dead branches are conditional branches. This suggests that the remaining 138 branches to focus on are all conditional branches.

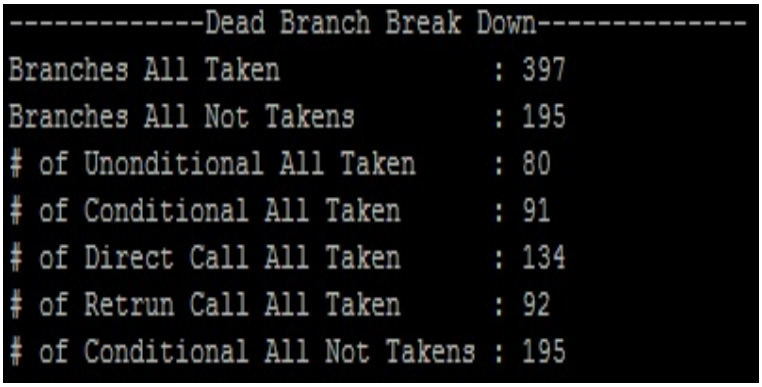

Figure 4: Output of "9_branch_stats.txt"

4.3.2 Glacial Branches

The next behavior the profiler identifies is glacial branch behavior. Glacial branches have a high percentage of not takens and a very low percentage of not takens. The profiler is set to identify branches with 85% execution of not takens. This behavior, like dead branches, is very predictable. When most of the execution of a branch is not taken, then the rest of the execution can become negligible.

Figure 3 shows the analysis on glacial branches done by the profiler. There are 32 glacial branches that make up 22.8% of the execution. This narrows down even further the focus of the predictor of the designer. There is now only 106 branches to focus on, that make up only 36.1% of the original total execution.

4.3.3 Looping Branches

The next behavior the profiler identifies is potential loop branch behavior. There is a good reason to include the word "potential" when listing the loop branches. The profiler currently contains a basic algorithm to identify the most obvious looping branches or branches that may potentially contain a pattern. Take a look at the branch history of PC 4212052 in Figure 5. Remember that the first value is the execution total (1506) and the second value is the branch type (6). The rest of the values are its branch history. This branch is a conditional loop that is taken 5 times, then not taken. It is a single loop and its behavior is obvious and identified by the profiler. Now take a look at the PC 4216183. There is a sequence of 1 0 0 0 0 0 1 0 0 0, where 1 means taken and 0 means not taken, for the first 10 executions. The next sequence is exactly same, however, the sequence after that is 1 0 0 0 0 0 1 0 0 1. There is a slight difference at the end of this sequence. At this point, the pattern is broken and this pattern seems to be irregular, however, take a look at further execution of this branch in Figure 6. A new pattern has emerged (1 0 0 0 0 0 1 0 0) and it is only slightly different than the starting pattern. It seems that some of the branches that initially irregular are not so irregular as shown.

Figure 5: Loop branches identified by the profiler

Figure 6: Further execution history of the branch with PC 4216183.

By identifying these patterns, nested loops become clearer and noticeable. A branch predictor can use the information of the profiler to identify these looping branches and make more accurate predictions. In addition, unlike the loop identifiers in section 2, more complex patterned loops are identified, which would result in more accurate branch prediction. Figure 3 shows that there are a total of 54 potential looping branches that comprise of 8.4 % of the total execution. Currently, 92.8% of the branches, which comprise of 72.3 % of the total execution, contain predictable behavior. The rest of the execution, 27.7 %, is deemed irregular.

4.3.4 Irregular Branches

The final behavior the profiler identifies is the irregular branch behavior. These branches do not contain any noticeable behavior and can be very difficult for a designer to design around. Figure 3 shows that there are 52 branches with this behavior and they comprise of 27.7% of the total execution. The profiler lists these branches to give a designer the ability to eye ball the activity and develop ideas to design around the irregularity. Figure 7 shows some of the branches with irregular behavior. Some of these branches are very small, such as the last 9 branches listed in Figure 7. They make up of a very small percentage of the execution and could likely be ignored. Some of the branches seem irregular, but contain a slight pattern. For example, take a look at branch 4284387. The execution has a slight pattern in that the branch loops an n amount of times until it exits. Also, most of the time n is 162. This branch also contains many executions and designing around this behavior would not be difficult.

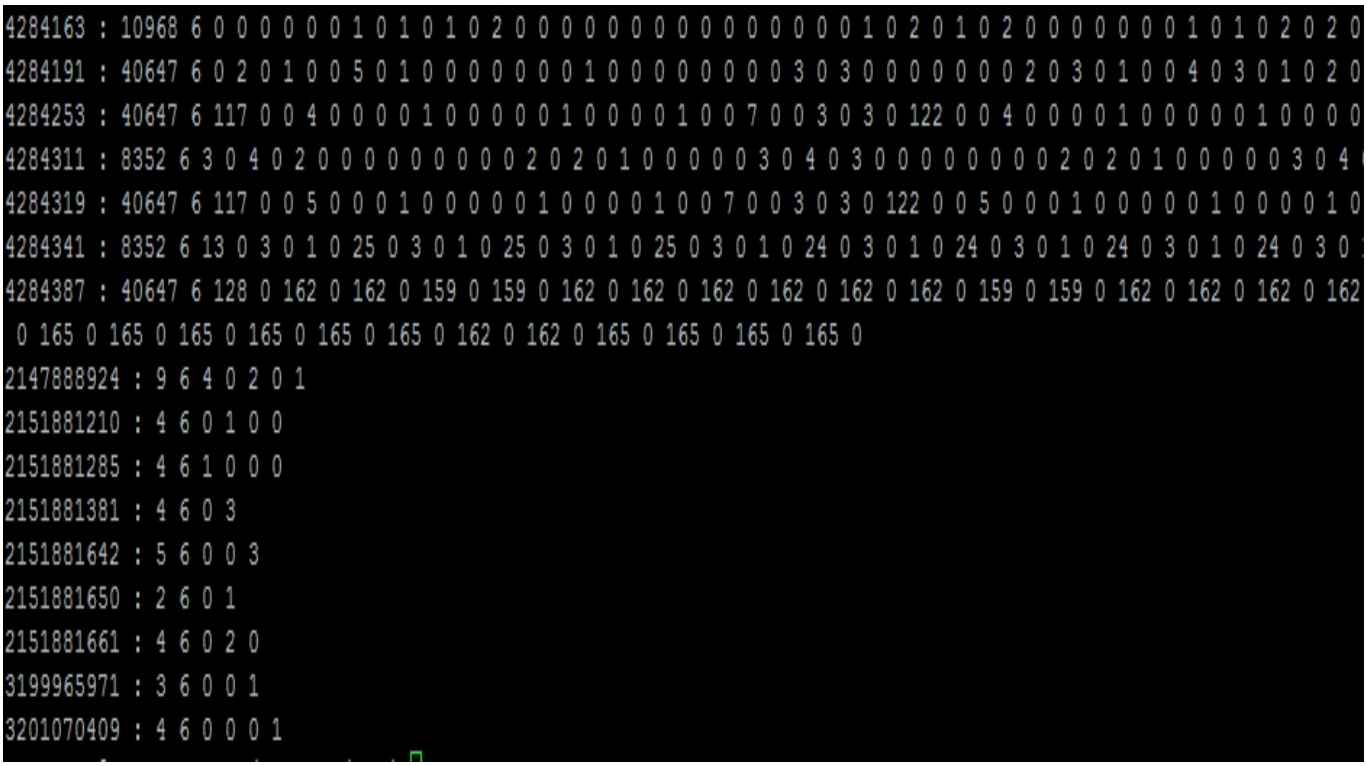

Figure 7: Branches with irregular behavior.

4.4 Global Branch History

One of the goals of this project was to identify elements of the global behavior of the branches. More specifically, detecting when the trace makes a function call, what occurs in this function call, and when the function returns. In order to achieve this functionality, there needs to be a way to detect function calls and when the main program returns from these function calls.

Currently the profiler analyzes the trace as it runs and identifies any direct call branches. When a direct call has been identified, this means a potential function call has been made and the profiler will begin to record activity. The profiler keeps a count of direct and return calls. Once the number of return calls matches the number of direct calls, the profiler will quit recording activity. It will start recording again once it identifies a direct call.

It is was unknown if the return calls directly correspond to direct calls. To investigate this, the profiler was designed to print out the identity of the branch calls recorded between a direct call and its alleged termination. In addition, once a return call is identified, its target branch is recorded instead of its identity. Figure 8 shows two histories of potential function calls. The first value 3, for each branch, indicates the first direct call. The 6's indicate conditional branch activity after the first direct call and the large values 4212558 and 4213303 indicates return calls. As mentioned earlier, these large values are the PC of the target branches for the return calls. Notice that the PC of the direct call plus 5(PC of direct call + 5) is the PC of the target branch for every return call. This is a very good sign that there is a correlation between direct calls and return calls. In addition, the profiler prints out the counts of direct calls and return calls for these direct call histories. Figure 9 shows that 8 out of 9 of these call histories have matching direct to return call counts. The values on the left correspond to return calls and the values on the right correspond to direct calls. This is another good sign that there is a correlation between direct and return calls, however, notice that the first branch in Figure 9 does not have matching counts.

```
4212553 : 3 6 6 6 6 6 6 6 6 6 6 6 4212558 3 6 6 6 6 6 6 6 6 6 6 4212558 3 6 6 6 6 6 6 6 6 6 6 4212558
4213298 : 3 6 6 6 6 6 6 6 6 6 6 6 4213303 3 6 6 6 6 6 6 6 6 6 6 4213303 3 6 6 6 6 6 6 6 6 6 6 4213303
```

Figure 8: Partial histories of two direct calls.

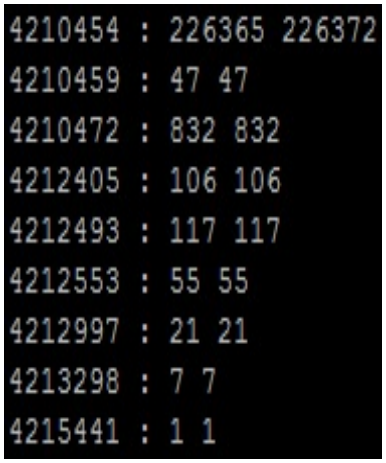

Figure 9: Direct and return call counts for all the potential function call histories.

Likewise, take a look at Figure 10. This is the partial history of the direct call that does not have matching counts. The PC 4210454 refers to PC of the first direct call. The PC 4211819 refers to the PC of a second consecutive direct call. The PC 4211824 refer to the target branch of a return call and is expected to correlate to the second direct call. It does (PC of direct call + 5), thus, there still is correlation where the counts do not match.

Figure 10: Partial history of the first direct call branch.

4.5 Other Miscellaneous Profiler Information

The profiler was designed to provide a few other miscellaneous information that could prove to be handy for some of the traces. This information includes:

- Top 10 most executed branches
- Branches with matching execution patterns
- Complete target branch history

This information may not identify interesting information about a trace, but it is available. If most of the execution activity is invested in a few branches, then the top 10 branch information will identify this. If there are many branches, identified by the matching execution information, with the same execution patterns, then a branch predictor could treat these branches as one large branch. The complete target branch history is the least interesting, at least for the example trace. It did, however, identify that the unconditional and conditional branches never changed their target branch. This may be different for other traces, but it does show the simplicity of the example trace.

5 Future Work

First and foremost, the profiler needs to be built up even further. Work needs to be done to determine the best threshold for glacial branch identification. A more complex pattern detection algorithm needs to be implemented to detect internel patterns in branches that, at a first glance, seem like irregular branches. Also, the direct call history discussed in section 4.4 must be further investigated to determine the exact correlation between direct and return calls.

Once the profiler is completed, it should be used to profile all 40 traces. Once all 40 traces are profiled, a branch predictor could be designed to cater to the 40 traces. If the profiler is developed well enough, a very accurate branch predictor would be the result. Similarly, new ideas may emerge while profiling the competition traces. New functionality will be built into the profiler and reveal even more information on the traces.

6 Conclusion

All in all, the profiler has excellent functionality to provide information on the CBP competition traces. It provides the branch make-up of a trace, branch behavior, global branch history, and other possibly useful miscellaneous stat information. The branch make-up describes the type of branches that make-up the trace. The branch behavior profiling describes how much of the execution and how many branches are dead, glacial, looping, and/or irregular. Also, the global branch history showed a correlation between direct and return calls. Finally, the miscellaneous profiler describes the top 10 branches, branches with matching execution patterns, and a complete target branch history for every branch. The profiler was used to profile the SHORT-INT-1.cbp4.gz, to show the type of insight that can be gained by using the profiler.

References

[1] (2014) Championship branch prediction. [Online]. Available: http://www.jilp.org/cbp2014

[2] T. Sherwood and B. Calderr, "Loop termination prediction," in *3rd International Symposium on High Performance Computing(ISHPC2K)*, October 2000.

[3] G. Sheaffer, "In a pipelined computer microprocessor," Jun. 1 1999, uS Patent 5,909,573. [Online]. Available: http://www.google.com/patents/US5909573