# Process Switches and Branch Prediction Accuracy

**David Chen, Bennett Lau, Jeffrey Shafer**

Department of Electrical and Computer Engineering
Rice University

**Abstract**

There were several hypotheses that motivated this research project. First, we proposed that running multiple processes pollutes the history in branch predictors due to aliasing. This pollution decreases accuracy. Second, we argued that branch prediction accuracy can be improved over existing techniques by removing this aliasing. Specifically, the history table can be partitioned to allow each process, or a subset of the most frequently executed processes, to have their own history.

This project successfully shows that aliasing in the branch predictor history does occur due to multiple processes executing, and quantifies the frequency of that problem. The accuracy of existing predictors can be improved, and adding separate history tables is an effective method of removing the aliasing problem. Specifically, a prediction improvement of 0.5 – 3% can be obtained through this technique, and this improvement is always in addition to any prediction algorithm that suffers from aliasing. The actual accuracy penalty due to aliasing is much higher than this improvement would seem to suggest. The relative infrequency of process switches, however, and the short time of approximately 2000 branches to "repair" the destructive aliasing in the branch history table after a process switch, both combine to significantly limit the potential for improvement via this technique. When the hardware overhead of multiple history tables is taken into account, this technique is not recommended for general-purpose applications

## 1. Introduction

The branch predictor accuracy is a significant component of overall performance in modern deeply pipelined superscalar machines. The penalty of a mispredicted branch can be quite high [6]. Most existing research into branch predictors, however, examines their performance in single-process programs only. As computation moves increasingly to multi-threaded applications, it is unclear if the existing research is applicable. Do multiple threads compete over the branch predictor and interfere with each other? How severe is this effect? Does a method exist to alleviate this problem?

## 2. Existing Work

Previous research into branch predictors defines some useful terminology. First, *aliasing*, as described by [9], is central to this research project. Analogous to memory caches, aliasing in branch prediction occurs when different branches are assigned to the same history counter. These instruction streams can be from the same process or different processes altogether.

Aliasing is not a permanent effect. If one branch executes for a thousand times, and then another branch aliases to its same spot in the history table and also executes a thousand times, the overall effect on prediction accuracy will be very limited, as it will only take a few cycles (depending on the depth of the history) for the new branch to "train" the predictor with the new pattern.

This aliasing can either be constructive, destructive, or neutral, depending if the new pattern alters the final prediction of branch taken or branch not taken. [9] showed that destructive aliasing is significantly more common than constructive aliasing, particularly if the instruction stream is large or the branch history table is small. However, this analysis was not performed for independent processes. Rather, it was done on different segments of the instruction stream from the same process, so it is unclear if the behavior will be similar.

Related to aliasing, [2] defined *training overhead.* This overhead refers to the fact that history counters must be "primed" to deliver the correct prediction by first observing a few repetitions of the branch. This concept is relevant because each process switch incurs a new training overhead for the branch predictor.

Branch predictor configurations for simultaneous multithreaded processors were explored in [5]. In addition to a traditional shared predictor, they examined providing separate history registers or branch history tables for each thread. They concluded that providing only a separate history register for each thread increased the prediction accuracy by eliminating harmful aliasing between execution threads, and minimized the increase in hardware requirements that completely independent predictors would require. Further, this configuration with

independent predictors increased performance even when the threads were executing the same code. Unfortunately, while the branch prediction accuracy was increased, the overall system performance was only marginally affected, because the thread-level parallelism inherent in the SMT design was effective in hiding the negative effects of branch misprediction.

[2] found that including kernel references increased the branch predictor history aliasing. This effect could alter the conclusions of past studies of prediction accuracy, such that algorithms with short branch histories would now performance better than algorithms with long histories, so that the effect of aliasing would be more rapidly overcome through training.

## 3. Multiple Process Instruction Traces

Because most existing literature on branch predictors only used single-process instruction traces, this study was conducted with multi-process instruction traces. To obtain the desired instruction traces of multi-process applications and the host operating system, the Simics simulator from Virtuatech was used. Simics is a detailed functional simulator for multiple processor architectures, including x86, PowerPC, Alpha, and Itanium. In addition to simulating the microprocessor, Simics also emulates a core subset of peripheral devices, such as the BIOS, video card, network card, and disk drive. Thus, it is possible to install and run unmodified operating systems such as Linux or Windows on Simics. Because the entire system is simulated, and thus the simulation rate can be dynamically altered, Simics enables non-intrusive profiling of the instruction stream and memory access stream without concern that the monitoring overhead will distort the data capture [3].

Simics includes a standard module that can capture an instruction stream during program execution. This stream mixes both user-level and kernel-level code, as well as multiple processes. For this project, the goal was to extend Simics to capture process IDs so we can clearly distinguish between multiple processes and study the immediate impact of a process switch on branch prediction accuracy. Because Simics functions as a hardware execution platform, it has no native understanding of a software concept such as a “process.”

A prebuilt operating system disk image from Virtutech was used. This image contained RedHat Linux 7.3 with the KDE GUI. Both the operating system and Simics were modified to obtain trace files with process IDs.

In the Linux kernel, we modified the schedule() function in the sched.c file. As a task switch occurs, a “magic function” is called into Simics which passed the new process ID as a parameter. The kernel was compiled and loaded into the simulated machine.

In the Simics source code, the trace.c file of the instruction trace module was modified. Here, an event handler was added to run whenever the magic instruction is called within the Linux kernel. When the event handler is called, it prints the new process ID to the instruction trace file. It is important to have the process ID inline in the instruction trace so that they can be analyzed at the same time by the custom branch predictor simulator.

Inside the simulated operating system, a suite of applications was installed from which memory traces were obtained. A wide selection was used because it was unclear as to which application types would produce aliasing, and whether the application type would determine whether the aliasing is constructive or destructive. The applications include the MySQL database, Apache web server, and SPEC 2000 CPU benchmark [4,1,7]. In addition, many common Linux utilities were also present in the system, such as the VI text editor and Pine email viewer.

Traces of 10 million+ instructions were taken of all applications. Traces of the web server and database server were captured during the period of time when the server was actively servicing multiple end user requests in parallel. Traces of the SPEC benchmarks were taken during the execution of the core benchmark functions, and do not include the compilation or configuration of the benchmarks that happens prior to execution. The VI text editor trace was taken when the editor was opening an existing file and searching for text, and the Pine email viewer trace was taken when the client was opening a new mailbox and displaying the first message on screen.

Each trace was parsed to determine the number of process switches that occurred in the instruction stream. The purpose of this project is to examine the effect of process switches on the branch predictor accuracy. Thus, traces with a minimal number of switches (less than 8) were rejected and not analyzed because their potential for performance improvement was minimal. Unfortunately, this rejected nearly all of the SPEC benchmarks as unsuitable for this experiment. The SPEC programs were not useful because their tight execution kernels make few, if any, blocking calls to the operation system that require immediate process switches. Two benchmarks (Gzip and Mgrid) were left in the collection of traces to evaluate their performance despite the lack of process switches, and as expected these traces turned in the worst accuracy improvement.

## 4. Branch Prediction Simulator

In order to model the behavior of common branch predictors, we have picked the best two performing predictors, namely, a global gshare predictor with a global pattern history table (Gag) and a global gshare predictor with per-process pattern history tables (a variation of Gas) suggested in [8].

Our original plan was to abstract the branch predictor module from the SimpleScalar toolset However, upon examining the SimpleScalar source code, we discovered that it was not feasible to separate the branch predictor module as there were too many dependencies across different modules, making it difficult to isolate the branch predictor from the rest of the program.

We therefore decided to write our own branch predictor algorithm using Perl. One big advantage of writing our own branch predictor was that we could customize and interface the branch predictor to our trace parser a lot easier. We also added extra features into the branch predictors. For example, our branch predictor can calculate the utilization of the pattern history table and compare the pattern history tables and more importantly, the prediction accuracy of the two different branch predictors at run-time. These additions to our branch predictor algorithm were essential for efficient data analysis on the predictors. Our branch prediction simulator is composed of two units: the trace parser and the branch prediction module.

### 4.1 Trace Parser

In order to get the traces from the user-level and system-level programs, our branch predictors were fed with a trace parser that reads instructions one by one from the trace files created by Simics. The instructions were decoded and identified as branch and non-branch instructions. The way the branch predictor decides whether a branch instruction was taken is as follows: First, the branch predictor will read the branch target address from the branch instruction. At the same time, the trace parser will also record the instruction length for the current branch instruction in order to compute the fall-through address for the current branch instruction. Second, the trace parser will read the next instruction from the trace parser. The address of the next instruction was extracted and then compared to the branch target address of the last branch instruction. If the address of the next instruction matched the target address of the last branch instruction, the branch was taken; otherwise, the branch was not taken and the address of the instruction would be exactly the fall-through address of the branch instruction.

### 4.2 Branch Prediction Module

Coupled with the trace parser is our branch prediction module that runs in back to back with the trace parser. In our experiment, we implemented two different branch predictors. The first, as shown in Figure 1, was a global branch predictor with global pattern history table. The branch history register on the left on the figure was an n-bit shift-left register used to store the direction (Taken / Not taken) of the last n branches, n being an adjustable value. The program counter contained the address of the current branch instruction. The global pattern history table contained a total of $2^n$ entries of 2-bit saturation counters used for branch prediction. The saturation counter value ranged from 0-3. The values of the 2-bit saturation counter, 0, 1, 2 and 3 represented a prediction of "strongly not taken", "weakly not taken", "weakly taken" and "strongly taken" respectively. Whenever the program encountered a branch instruction, the n-bit value in the branch history register would be XORed with the last n-bits of the program counter, which is the address of the current branch instruction. The result from the XOR operation would then be used to index into the global pattern history table. The value of the saturation counter from that specific index location was read and a prediction (strongly not taken, weakly not taken etc.) would be made based on that value. When the parser fetched the next instruction and determined whether the branch was really taken or not, the same saturation counter that made the prediction was updated with the result. If the branch was taken, the saturation counter would be incremented by 1 subject to the maximum value of 3. Otherwise, the counter would be decremented subject to the minimum value of 0. Finally, the branch history register would be shifted left in order to record the branch direction.

The second branch predictor, as shown in Figure 2, is a global branch predictor with a per-process pattern history table. The only difference from the first predictor is that instead of one global pattern history table, every process in the program has a separate set of $2^n$ entries of saturation counter values. This allows more adaptive predictions to different processes in the program and thus better prediction accuracy. After the XOR operation, the results of the XOR operation are used to index the pattern history table corresponding to the running process.

After some experiments with different configurations of the branch predictor, we decided to use a 12-bit history register and a 4096-entry ($2^{12}$) pattern history table. This configuration provided the best relative performance over a range of programs. Therefore, we built both our branch predictors with this configuration.

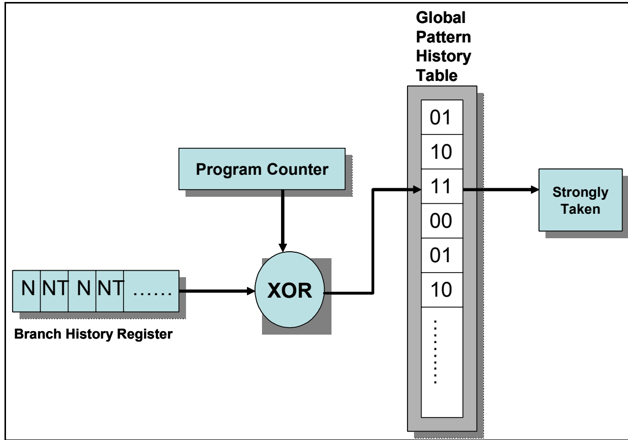


**Figure 1:** Global Branch Predictor with Global History Table

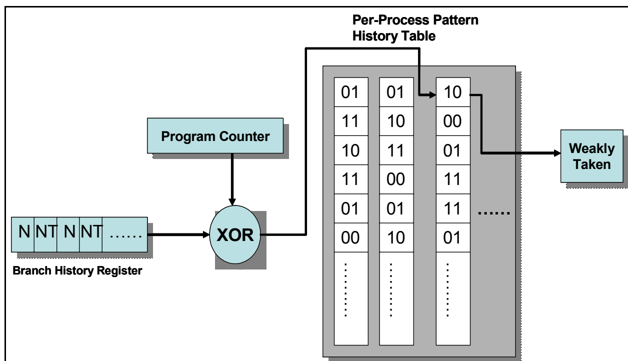


**Figure 2:** Global Branch Predictor with Per-process History Table

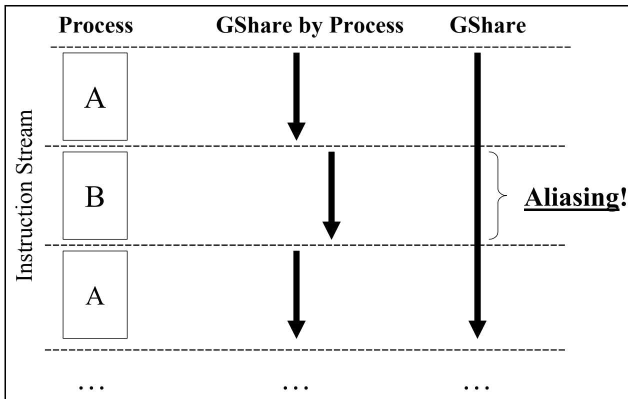


**Figure 3:** Branch Predictor Aliasing

Among the customizations and additions to our branch predictors, the functionality to compare and contrast the run-time accuracy of the branch predictors was very helpful to our analysis. The convergence graph of accuracy of the two branch predictors shed light into how the two branch predictors performed relative to each other over time.

## 5. Experimental Results

After obtaining instruction traces of multi-process systems and constructing a branch predictor simulator, a sequence of experiments was conducted. The purpose of these experiments was to determine if aliasing exists, if this aliasing degrades accuracy, to quantify precisely where this behavior occurs, and to propose solutions for eliminating aliasing.

Figure 3 illustrates how aliasing might occur between multiple processes. The first column represents the currently executing process. The second column represents a branch predictor that has a separate history table per process. Thus, when a process switch occurs, a new table is used, and when original process “A” returns to execution (perhaps several process switches later), the original history table is also reloaded. The third column represents a standard branch predictor with a single history table. Thus, when the original process “A” is replaced by process “B”, the second process naturally starts to overwrite some elements of the history table with its own history. This is called aliasing. Thus, the state of the history table when process “A” returns to execution some time later is different from the state of the history table when “A” was originally executing.

The first objective in this research project was to determine if aliasing exists in real-world programs and branch predictors. A collection of instruction traces was run through the branch predictor to compare the original GShare algorithm with the modified predictor with separate history tables by ID. A representative comparison is shown in Figure 4.

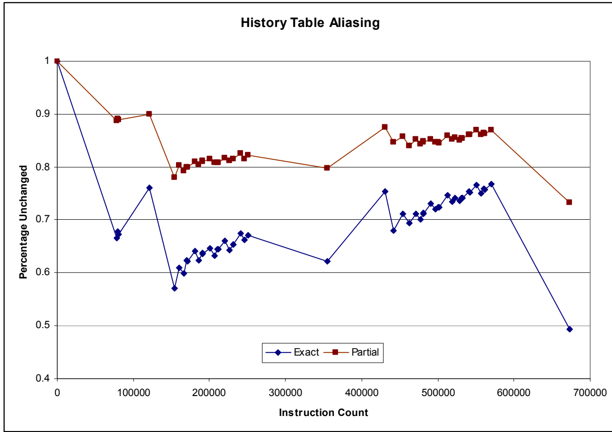


**Figure 4:** History Table Aliasing (MySQL trace)

This figure was created by comparing the ID branch predictor history file with the global branch predictor history file immediately after it enters and exits a new process. For purposes of visualization, this graph was restricted to only show when process 2736 (a MySQL server thread) enters and exits. *The two lines should be examined independently, as each line uses a different method to compare the two different predictors used in this study.* The top shows the percentage of the history table that was considered "partially" unchanged between the standard predictor and the ID-enabled predictor. A "partial" match indicates that the branch prediction result of taken or not taken was the same, but the exact value of the saturating counter in the history table was different. In contrast, the "exact" match in the bottom line indicates that the counters in the history table were exactly the same.

At instruction count zero, it is clear that both empty history tables would have the exact same contents, and thus be 100% unchanged. When the simulation starts, process 2736 has just finished and another process (or processes) is executing. When the MySQL process resumes execution at the next data point, there is a substantially lower similarity between the two branch predictors compared to when they last left the process. The only possible cause for this divergence is aliasing in the global predictor caused by other processes overwriting portions of the history table.

Note that some processes have higher aliasing rates than others. The amount of aliasing a process causes is independent of its length of execution time. If a process has a very short duration, however, the aliasing (as shown in Figure 4) will be minimal because the process executed very few instructions and branches. In this project, the emphasis is on processes that execute enough branches to produce a significant amount of aliasing.

Clearly, aliasing exists and is caused by competition for history table resources between multiple processes. It is not clear, however, if this aliasing is constructive, destructive, or neutral. To examine this issue, all the instruction traces were run through both branch predictors and compared, as shown in Table 1.

**Table 1:** Predictor Accuracy With and Without Aliasing

| **Trace Source (Primary Program)** | **GShare Predictor Accuracy (%)** | **ID Predictor Accuracy (%)** | **Accuracy Difference (%)** |
|---|---|---|---|
| Apache 1 | 88.48 | 88.68 | 0.20 |
| Apache 2 | 87.92 | 88.18 | 0.26 |
| Apache 3 | 87.79 | 88.20 | 0.41 |
| KDE 1 | 86.10 | 86.85 | 0.75 |
| KDE 2 | 88.99 | 90.48 | 1.49 |
| KDE 3 | 89.70 | 90.34 | 0.68 |
| KDE 4 | 90.73 | 91.16 | 0.43 |
| MySQL | 84.76 | 87.62 | 2.86 |
| Pine | 85.77 | 86.33 | 0.56 |
| SPEC2000 (Gzip) | 86.93 | 87.04 | 0.11 |
| SPEC2000 (Mgrid) | 86.16 | 86.31 | 0.15 |
| VI | 86.04 | 87.94 | 1.90 |
| | | **Average:** | **0.81%** |

The standard GShare predictor has aliasing because it only uses a single history table. The ID predictor uses the same algorithm, but has a separate history table per process. Thus, the only difference between the two systems is the removal of aliasing in the ID predictor, which produces an average accuracy improvement of 0.81%. Thus, because the predictor performance improved overall, the aliasing in all of the instruction traces must have been destructive on average.

Having learned that aliasing does occur and that it is destructive overall, the next question probed in this project was to determine the extent of the aliasing problem and show both where it occurs and how removing it improves predictor performance.

Figure 5 shows a sample performance comparison between the two predictor systems on the MySQL database trace. Although only one figure is included in this paper, the behaviors described are present in all of the instruction traces.

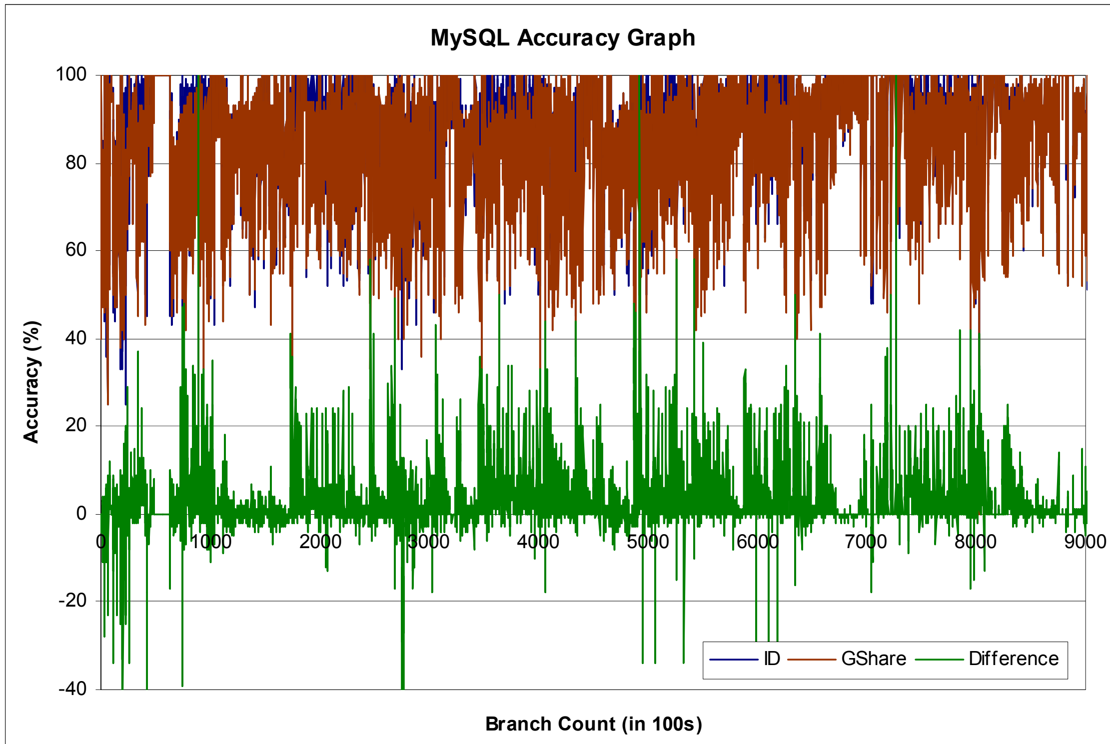


**Figure 5:** Accuracy Comparison (MySQL trace)

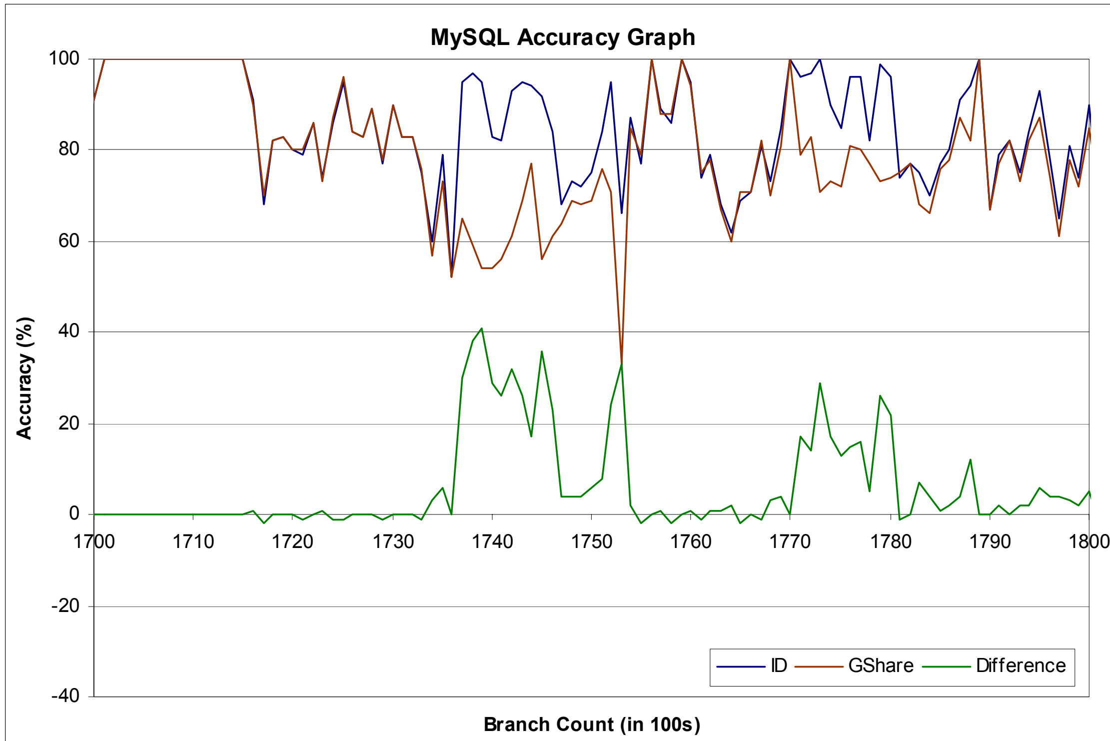


**Figure 6:** Accuracy Comparison (MySQL trace)

This graph extends for the entire duration of the instruction trace. The top two lines (mostly overlapping) centered about 80-90% represent the predictor accuracy for the standard and by-ID GShare algorithms. The bottom line centered near 0% represents the performance difference between the two predictors. Values above zero signify that the ID predictor improved accuracy, while values below zero signify that the ID predictor was less accurate than the global predictor.

It is important to note that the bottom line is more sparse than the graph above would seem to indicate, as the lines can only be plotted with a minimum thickness, and as they overlap they tend to obscure near-zero values in between. Looking at the bottom line at a higher zoom level, we notices pauses and spikes. This suggests that there are periods where both predictors behave the same, and other periods where the predictor accuracy differs significantly. Figure 6 is a close-up view of Figure 5 over a limited time frame during the instruction stream.

This figure clearly shows lengthy periods of high convergence where the difference between the two predictors is essentially zero. But, at approximately the 1735$^{th}$ (hundred) branch, the predictors wildly diverge. The ID predictor accuracy increases to near 100 percent, while the standard predictor accuracy remains roughly constant. At some later point in time, the two predictor's accuracy grows closer together and the difference again is near zero percent.

The next logic question to answer is: why does this behavior occur? The original instruction trace file was examined, and it was observed that these divergence points coincided exactly with the process switches. This is highlighted in Figure 7. Thus, the "new" process must incur a performance penalty as the history table learns its new execution pattern. After the new process executes for a period of time (i.e. the "training time" as discussed by [2]), the accuracy of the original global predictor improves to the point that it matches the by-ID predictor. The by-ID predictor, however, does not need to pay this performance penalty, as it already has a fully trained history table ready for prediction based on the last time this process executed.

The time that it takes for the accuracy of the aliased predictor to improve and match the performance of the non-aliased predictor is referred to as the *convergence time*. This concept is labeled in Figure 8.

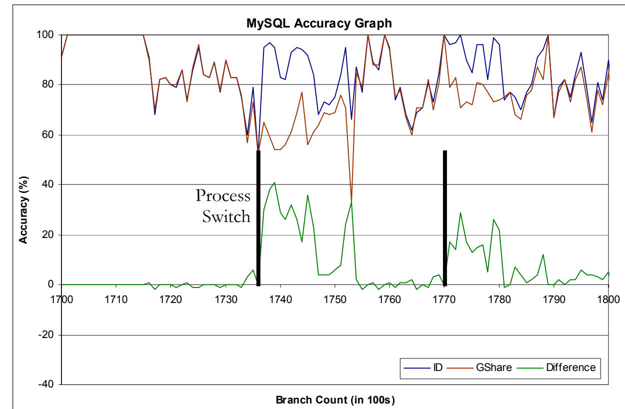


**Figure 7:** Process Switches (MySQL trace)

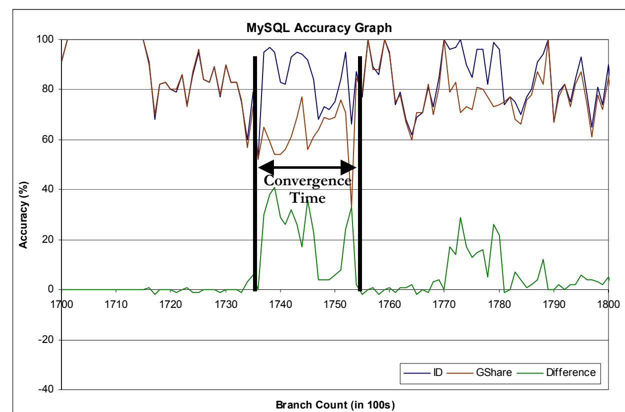


**Figure 8:** Convergence Time (MySQL trace)

The longer the convergence time is, the greater the potential for accuracy improvement is by eliminating the penalty incurred by aliasing when other processes execute. The duration of convergence is necessarily dependent on the specific processes that were executing, the time each process spent in execution, and the degree of aliasing each process produced. As an example of how the convergence varies by process, a breakdown of the KDE trace is shown in Figure 9.

The purpose of this figure is to show that, although convergence time varies, the property occurs in the general case at the beginning of all processes, and is not limited to a few rare cases in specific trace files. To obtain this graph, a special parser was written to analyze the data points shown in Figure 6, which provided the relative performance between the two competing predictors (e.g. with and without aliasing). We average the accuracy of the first 2000 branches each time the machine enters a specific process. This average was taken in increments of 100 branches so that the convergence can be observed with respect to time. This was repeated for all processes.

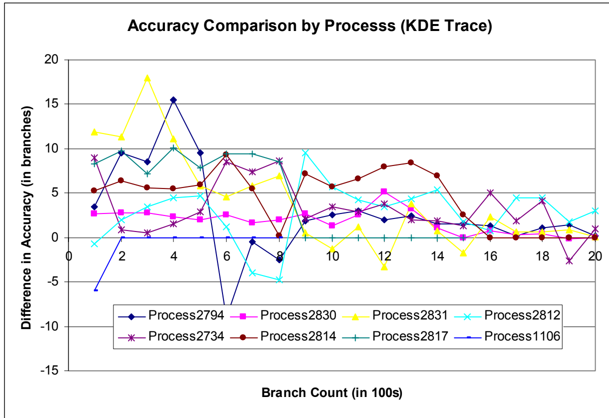


**Figure 9:** Convergence Times by Process ID (KDE Trace)

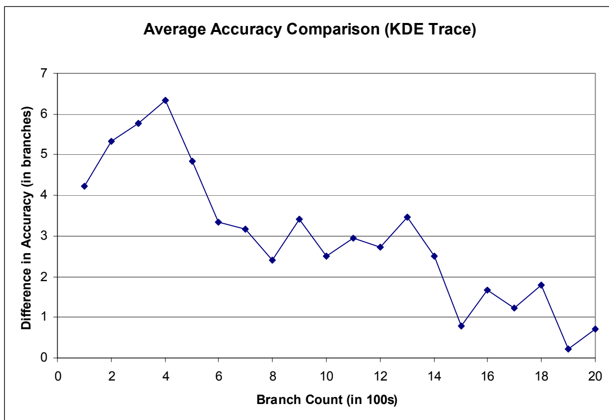


**Figure 10:** Average Convergence (KDE Trace)

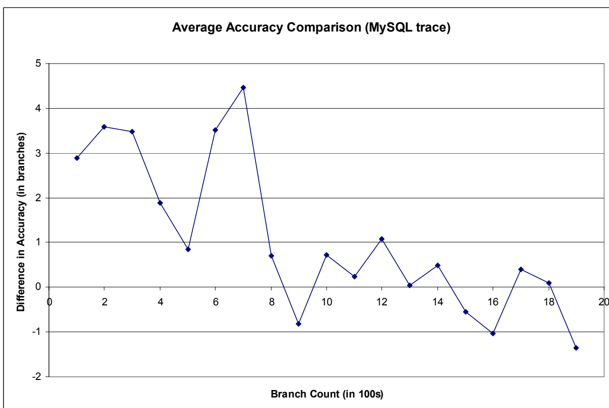


**Figure 11:** Average Convergence (MySQL trace)

Averaging the predictor performance in Figure 9 reveals a clear overall performance trend for all of the processes in that trace, as shown in Figure 10. In this figure, it is clear that all processes in the KDE trace converge to within 1% accuracy in approximately 2000 branches. Thus, aliasing occurs regularly in modern systems after every process switch. This convergence behavior held true for most instruction traces analyzed. For example, Figure 9 shows the convergence average for the MySQL trace, which indicates that it also converges to within 1% accuracy within 2000 branches, and may even achieve a reasonable level in less than 1000 branches.

A few instruction traces did not show a clear convergence pattern after averaging the performance of the individual processes. An example of this behavior is shown in Figure 12 for all of the processes and in Figure 13 for the process average.

In the by-process figure, it is clear that some processes exhibited very erratic branch predictor performance. Clearly, an irregularly performing process is difficult for any predictor algorithm to handle, and this behavior is independent of the aliasing effect that this project set out to study. Thus, simply for the purposes of illustration, the three most erratic processes from Figure 12 were removed, and a new average computed, as shown in Figure 14. Removing these from the average better illustrates the convergence behavior. It is important to note that, although the erratic processes were removed when creating this specific figure only, all process (regular or irregular) were included in Table 1 when computing the overall accuracy improvements after removing aliasing.

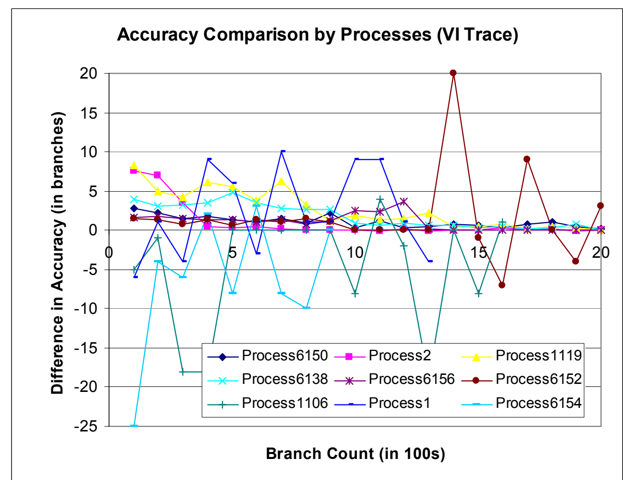


**Figure 12:** Convergence Times by Process ID (VI trace)

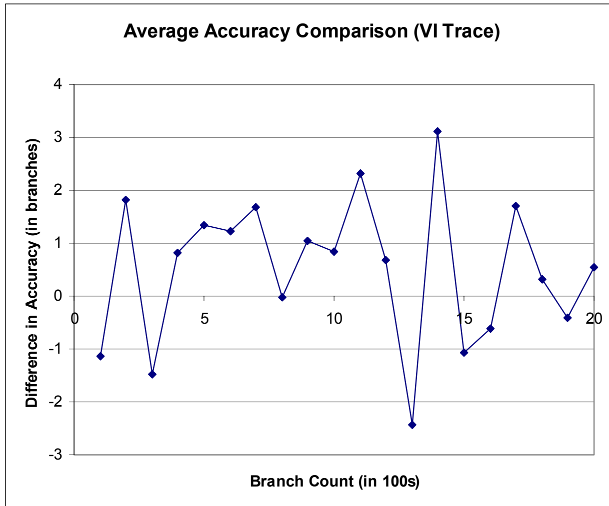


**Figure 13:** "Convergence" Average (VI trace)

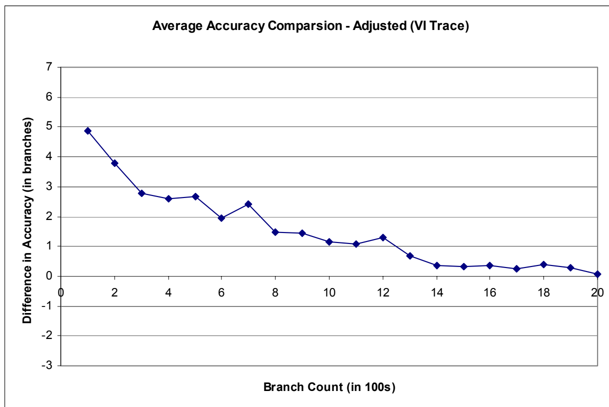


**Figure 14:** VI Trace Convergence (Adjusted Average)

## 6. Conclusions

All of the original hypotheses in this project were shown to be correct. Modern machines do frequent process switches, and traditional global branch prediction accuracy suffers after each switch. This accuracy penalty is due to aliasing in the history table, which can be eliminated by having a separate history table per process. By eliminating the aliasing penalty, the accuracy of any predictor that shares history between processes can be improved regardless of the efficiency of that algorithm.

Even with this costly solution in place, however, the overall accuracy improvement of 0.5 – 3% is quite small. The reason the improvement is so limited is not due to the actual difference in prediction accuracy, which can exhibit a large gap after a process switch. Rather, the convergence period, or the time it takes the history table to relearn the prediction patterns of the current process, is quite short at around 2000 branches. Thus, the overall accuracy improvement is highly limited by the rate of process switches in the current program set.

The results in this paper clearly indicate that, for general purpose applications, this proposed system provides both a limited benefit and comes with a high hardware cost due to the large number of parallel history tables that must be implemented and quickly accessed. Future research could identify if a separate history table is in fact needed for each and every process, or if some less-frequent or less-destructive processes could be assigned to share a history table. Although this project did use a variety of instruction traces, the data sets used were far from exhaustive. Thus, if a specific application category was found that had particularly frequent aliasing problems, this technique or a variant that only uses a limited number of history tables might be useful in improving prediction accuracy.

## 7. References


[1] Apache Software Foundation, "Apahce HTTPD Server Project", http://httpd.apache.org/

[2] Gloy, N, Young, C., Chen, B. and M. Smith, "An Analysis of Dynamic Branch Prediction Schemes on System Workloads," *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996

[3] Magnusson, P. et. Al, "Simics: A Full System Simulation Platform", *IEEE Computer*, pp. 50-58, February 2002

[4] MySQL AB, http://www.mysql.com/

[5] Ramsay, M, Feucht, C. and M. Lipasti, "Exploring Efficient SMT Branch Predictor Design", *Workshop on Complexity-Effective Design*, in conjunction with the *International Symposium on Computer Architecture*, June, 2003

[6] Sharangpani, H. and K. Arora, "Itanium Processor Microarchitecture", IEEE Micro, September/October 2000

[7] SPEC, http://www.spec.org/

[8] Yeh, T. and Y. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History", *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993

[9] Young, C., Gloy, N. and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction", *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp 276-286, June 1995