In Proceedings of the 3rd International Symposium on High Performance Computing (ISHPC2K), October 2000, (c) Springer-Verlag.

Loop Termination Prediction

Timothy Sherwood   Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,calder}@cs.ucsd.edu

Abstract

Deeply pipelined high performance processors require highly accurate branch prediction to drive their instruction fetch. However there remains a class of events which are not easily predictable by standard two level predictors. One such event is loop termination. In deeply nested loops, loop terminations can account for a significant amount of the mispredictions. We propose two techniques for dealing with loop terminations. A simple hardware extension to existing prediction architectures called *Loop Termination Prediction* is presented, which captures the long regular repeating patterns of loops. In addition, a software technique called *Branch Splitting* is examined, which breaks loops with iteration counts above the detection of current predictors into smaller loops that may be effectively captured. Our results show that for many programs adding a small loop termination buffer can reduce the missprediction rate by up to a difference of 2%.

1 Introduction

Branch prediction is the architectural feature which allows the front-end of the processor to continue fetching instructions in the presence of control flow changes. Branch prediction predicts the directions of branches during fetch, so the fetch engine knows which cache block to fetch from in the next cycle. When the wrong direction is predicted, the whole pipeline following the branch must be flushed, significantly impacting performance.

Accurate branch prediction uses the past history of branches to predict the future behavior of a branch. Early branch prediction architectures used an N-bit saturating counter to predict the direction of each branch [11]. Using only an N-bit counter accurately predicts branches which are biased in either a taken or not-taken direction, but looses prediction accuracy if the branch exhibits a more complex pattern. To capture these patterns local and global branch history prediction were proposed [16, 14].

Local branch history can be used to accurately predict a branch by storing the last L directions for a given branch, and using this to index into a 2nd level Pattern History Table (PHT) [16, 14]. The PHT is a table of N-bit saturating counters used to predict the direction of the branch. Local branch history increases prediction accuracy by capturing arbitrary patterns that are repeated within the last L times the branch was executed.

Global branch history uses correlation information between branches to accurately predict a branch's outcome. A history of the last G executed branch directions are kept track of in a global history register.

The global history register is then used to index into a pattern history table of 2-bit counters to predict the branch direction.

It is hard for local and global histories to capture loop terminations of the pattern $((1)^N 0)^m$, where 1 represents a taken branch and a 0 represents a fall-through branch. These patterns cannot be captured by a local history predictor if N is larger than L (the local history size). Loop termination can only be captured using global history if N is smaller than G[1] or there is a unique branching sequence right before the loop termination. For the programs examined in this study, 43% of the executed branches are loop branches, and 7% of the executed loop branches are mispredicted on average.

In this paper we propose to use a Loop Termination Buffer (LTB) to predict branch patterns of type $((1)^N 0)^m$. The LTB keeps track of branches with this behavior, and predicts when the pattern changes (terminates). This allows us to achieve up to 100% prediction accuracy for loop branches after a short warm up period.

In addition, we examine the potential of using a software approach called Branch Splitting to correctly predict loop branches. Local branch history can only accurately predict loop terminations for branches that execute less than L times, where L is the number of bits used for the local history. For loops that have an iteration count larger than L, the loop guarding branch can be split into two or more branches all which have an interaction count less than L, as long as the product of new iteration counts equals the old iteration count. The new branches' patterns will then fit into local history and will have all of their backwards and termination behavior accurately predicted.

The rest of the paper is organized as follows. Section 2 describes Loop Termination Prediction and our Loop Termination Buffer implementation. Section 3 describes Branch Splitting. Simulation methodology is described in section 4. Section 5 evaluates the performance of Loop Termination Prediction and Branch Splitting. Section 6 describes prior research for loop termination prediction. Finally, we summarize our contributions in section 7.

2 Loop Termination Prediction

Traditionally loops are thought of as the steady state operation of execution, and at first glance loop termination seems to account for only a small portion of the total amount of branches seen. However, this is an important part of a branch prediction architecture, because a regular loop has the miss rate which is the inverse of it's iteration count. Since the branch prediction accuracy of most processors is already in the high nineties, loop branch mispredictions can account for a large fraction of the remaining branch mispredictions.

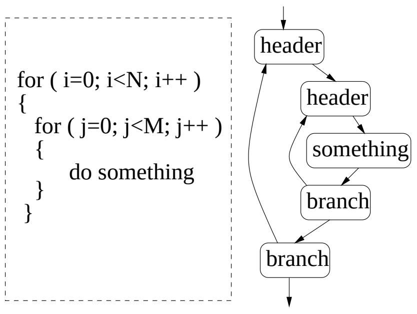
In this section, we propose using a very simple architecture extension which can predict how many times a loop branch will iterate to provide loop termination prediction.

2.1 Predicting Loop Termination

To allow instruction fetch to continue without stalling, each cycle a traditional branch prediction architecture predicts (1) if the current instruction fetch contains a branch, (2) the direction of the branch, and (3) provides the branch target address to be used in the I-cache fetch in the next cycle. This prediction is typically performed given only the current instruction cache fetch address each cycle.

The goal of our research is to predict when a loop branch terminates its looping behavior. Therefore, in addition to the above prediction information, during branch prediction we must (4) determine that the branch is a loop branch, and (5) predict if it has terminated or not. In order to provide this prediction information, the branch has to be labeled as a loop branch in the branch prediction architecture and the loop's trip count must be predicted.

[1] Assuming that there are not any branches internal to the loop.



```
for ( i=0; i<N; i++ )
{
  for ( j=0; j<M; j++ )
  {
      do something
  }
}
```

Figure 1: Doubly nested for loop

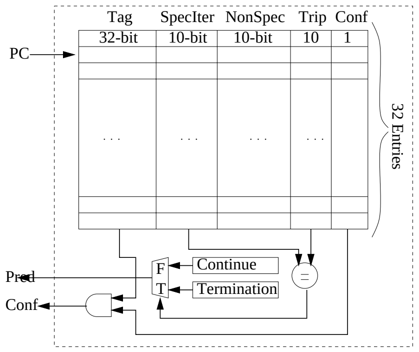


Figure 2: Loop Termination Buffer

2.1.1 Identifying Loop Branches.

Loop branches can be identified in hardware by either having special loop branch instructions (effectively having the compiler mark a branch as loop branch), or identifying the loop branches by looking at the sign bit of their displacement.

For this study, we targeted loops with the conditional branch at the bottom of the loop as shown in the doubly nested loop code example in figure 1. The number of loop branches generated this way depends upon the compiler being used. The Compaq C and FORTRAN Alpha compilers (with -O4 optimization) compiles most `for`, `while`, and `do` loops with the conditional branch check at the bottom of the loop with a negative displacement.

For our programs, loop branches usually have a negative displacement, and we dynamically predict that a conditional branch is a loop branch if it has a negative displacement. While this is not a perfect definition of a loop conditional branch, we used it to dynamically classify in hardware which branches are loop branches, so that we may know which branches should attempt to use loop termination prediction.

2.1.2 Predicting Loop Trip Count and Termination.

In order to correctly predict loop termination the branch predictor needs to (1) predict the loop trip count for the branch, and (2) keep track of the current loop iteration for the loop branch.

The *loop trip count* is the number of times in a row the loop branch is taken. The trip count can be a constant determined at compile-time for some loops, constant for a given run of an application but only determined at run-time for that run, or dynamically changing during a program's execution. Predicting the loop trip count can catch all three of these types of loop branches.

We use a *loop iteration* counter to record the current iteration count – the number of times the branch has been taken since it was last not-taken. The iteration counter is used to (1) predict when the loop has terminated, and (2) to initialize the loop trip count.

The loop termination prediction information described above could be stored directly into a Branch Target Buffer used for predicting target addresses during branch prediction [9, 15]. Instead of doing this, in our study we examine having a small associative buffer on-the-side to accurately predict the loop termination for loop branches.

2.2 Loop Termination Buffer

The *Loop Termination Buffer* (LTB) is a small hardware structure capable of detecting the impending termination of loop-branches. The prediction mechanism takes advantage of the fact that many loops have trip counts that do not change often over the course of execution. Take for example the doubly nested loop in figure 1. If we assume a traditional predictor which has been properly warmed up, the inner loop will cause N branch misses, and the outer loop will cause one, for a total of N+1 misses. The inner branch has a highly regular pattern of being taken N-1 times and then falling through the Nth time. This last time, the loop's termination, is an event which we wish to correctly predict.

As can be seen in figure 2, the LTB has five fields: a tag field to store the PC of a branch; a *speculative* and *non-speculative* iteration count to store the number of times the branch has been taken in a row; the loop trip count field to track the number of consecutive times the loop-branch was taken before the last not-taken; and a confidence bit indicating that the same loop trip count has been seen at least twice in a row.

2.2.1 During Branch Prediction.

To access the LTB, the fetch PC, which is used to perform the normal branch prediction, is also used to index in parallel into the LTB. If there is a tag match, the speculative iteration counter is checked against the trip count. If they are equal, the branch is a candidate for predicting termination. The confidence bit is then tested. If it is set, the loop branch is predicted to be not-taken (exiting the loop). If instead the speculative iteration counter and the trip count are not equal, then the speculative iteration count is speculatively incremented by one. We count from 0 up with the iteration counter, instead of down, so that we have the iteration count available in the counter for initialization of the trip count.

2.2.2 When Resolving a Branch.

A branch is considered for insertion into the LTB when it completes and its branch direction is resolved. When a branch resolves and is found to not be in the LTB, it is inserted into the LTB if determined to be a loop-branch and if it is mispredicted by the default predictor. In our study, we assume backwards conditional branches are loop-branches. When inserted into the LTB, all of the LTB entries counters are initialized to zero. Since an entry is inserted into the LTB when a loop-branch is found to be mispredicted, there are no outstanding branches and this allows the speculative and non-speculative iteration counters to start out synchronized.

During resolution, a taken branch found in the LTB has its non-speculative iteration counter incremented by one.

A not-taken loop-branch found in the LTB during branch resolution updates its trip count and confidence bit. If the non-speculative iteration counter is equal to the trip count stored in the LTB, then the confidence bit is set, otherwise it is cleared. The non-speculative iteration count is then incremented by one and is copied to the trip count. The speculative iteration count is then set to the current speculative iteration count minus the non-speculative iteration count. The speculative iteration count is reset to this value because the same loop-branch may have already been fetched again before the not-taken branch resolves. Finally, the non-speculative iteration counter is reset to zero.

One reason for having two iteration counters, as shown in figure 2, is to recover the iteration counters during a branch misprediction. When a branch misprediction occurs, all of the non-speculative iteration counters copy their values into the speculative iteration counters. Therefore, this synchronizes the speculative and non-speculative counters. This approach is similar to prior architectures proposed to recover branch prediction state during a misprediction [10].

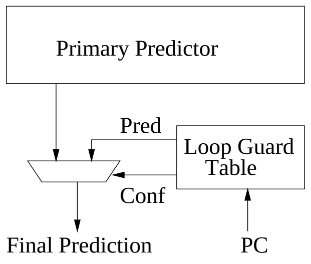
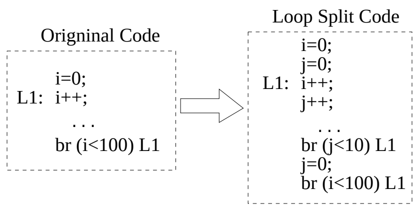


Figure 3: Using the Loop Termination Buffer for Branch Prediction. The primary predictor is used except in the case where the loop termination predictor predicts loop exit and is highly confident.



```
    i=0;
L1: i++;
     . . .
    br (i<100) L1
```

```
    i=0;
    j=0;
L1: i++;
    j++;
     . . .
    br (j<10) L1
    j=0;
    br (i<100) L1
```

Figure 4: Example of Branch Splitting. The branch in the original code will cause a branch misprediction 1 out 100 times the branch is executed. The transformed code can fit in the local history and will correctly predict both loop branches.

2.3 Loop Termination Predictor

Applying the loop termination buffer to branch prediction is a fairly straight forward process. When a loop branch is predicted to terminate, the branch is predicted as not-taken. Otherwise the branch is predicted taken. Although a more integrated approach of loop termination prediction into existing branch predictor architectures is possible (e.g., putting the counters into the branch target buffer), we examined using a separate buffer to concentrate on the effect of loop termination prediction.

Looking to figure 3 we can see how the branch predictors are combined at a high level. The final predictor generates a prediction for every branch, however the primary predictor will be overridden if the loop termination buffer generates a confident prediction. Recall from above that a confident prediction is generated if the branch PC is found in the LTB and it has seen the same trip count twice in a row.

3 Branch Splitting

A local branch history can only accurately predict loop terminations for branches that execute less than L times. For example, let us assume that the local history size is 12. If the predictor is predicting a loop branch whose iteration count is 100 (i.e. the pattern $(1^{99}0)^M$) as in figure 4, then the local history will not be able to predict the loop exit.

Because the local branch history can only accurately predict loop terminations for branches that execute less than L times, where L is the number of bits used for the local history, breaking the loop into multiple smaller loops will allow local history to correctly predict them. For loops that have an iteration count larger than L, the loop guarding branch can be split into two or more branches all which have an interaction count less than L, as long as the product of new iteration counts equals the old iteration count. The new branches' patterns will then fit into local history and will have all of their backwards and termination behavior accurately predicted.

Continuing with example above, let us split the branch into two branches, B1 with iteration count of 10, and B2 with iteration count of 10. This will create a doubly nested loop, whose loop body will be executed the same number of times as the original loop. Both the branch patterns will now be $(1^{9}0)^M$ which can be easily captured in the local history. This is very much akin to the technique of loop tiling, except here the resource we are tiling for is the branch predictors local history.

To apply branch splitting, we need accurate knowledge of the loop bounds, in order to create new bounds with little or no cleanup code. If cleanup code is generated, then this may more than offset the branch mispredictions saved by the main loop. In addition, when applying branch splitting one needs to be aware of the increased pressure splitting the loop branch will create on the branch predictor, since it will increase the number of entries used.

4 Methodology

To perform our evaluation we gathered results for 13 of the SPEC95 INT and FP benchmarks. The benchmarks were compiled on a Alpha AXP-21164 processor under OSF/1 V4.0 using the DEC C and FORTRAN compilers. All benchmarks were compiled at full optimization (`-O4 -ifo`). Since we are using full optimization, some loops in the programs were unrolled and software pipelined.

For the results in this paper we used ATOM [12] to instrument the programs and gather simulation results. The ATOM instrumentation tool has an interface that allows the elements of the program executable, such as instructions, basic blocks, and procedures, to be queried and manipulated. In particular, ATOM allows an "instrumentation" program to navigate through the basic blocks of a program executable, and collect information about registers used, opcodes, branch conditions, and perform control-flow and data-flow analysis. Programs were executed for a total of five billion instructions or program termination.

5 Results

To evaluate the benefit of loop termination prediction we examine the branch prediction performance of two predictors based on McFarling's meta predictor [8].

McFarling's meta predictor has a *meta* chooser table of 2-bit counters to choose between *bimodal* and *gshare* branch prediction. We use a bimodal table of 2-bit counters indexed by the PC to produce the bimodal prediction. In addition, we use a global history register XORed with the branch PC as an index into a table of 2-bit counters to provide the gshare prediction. The meta table is also indexed with the PC, and the 2-bit counter keeps track of which predictor (the bimodal predictor or gshare predictor) is more often correctly predicting the branch, and then the corresponding prediction is used.

The predictor with local history, called here the local/global chooser or LGC, is very similar to the McFarling predictor with the exception that the bimodal predictor is replaced with two tables for local history prediction. A per branch history is tracked in a first level *local* history table, which is then used to index into a second table of 2-bit counters. This extra level of indirection allows local branch history patterns to be captured. The major difference between the LGC and the branch predictor found on the Alpha 21264 [7] is that LGC uses the PC to index into the chooser, as opposed to using the global history.

5.1 Loop Characterization

We begin our analysis with a characterization of the loop behavior in the programs we examined. Figure 5 shows the percent of conditional branches executed which we classified as loop branches. These are executed branches having a negative sign displacement as described in section 2.1.1. The graph also shows the fraction of these executed branches that were found in our loop termination buffer (Hardware Loop Detection) during execution. Then layered on top of that is the percent of executed branches that had branch splitting applied to them. For `swim`, almost 100% of its executed branches were loop branches. These were all found in the LTB, and had branch splitting applied to them. For `applu`, 63% of its executed branches were classified as loop branches, 34% of the executed branches were found in the LTB, and 9% of the executed branches had the branch splitting optimization applied to them.

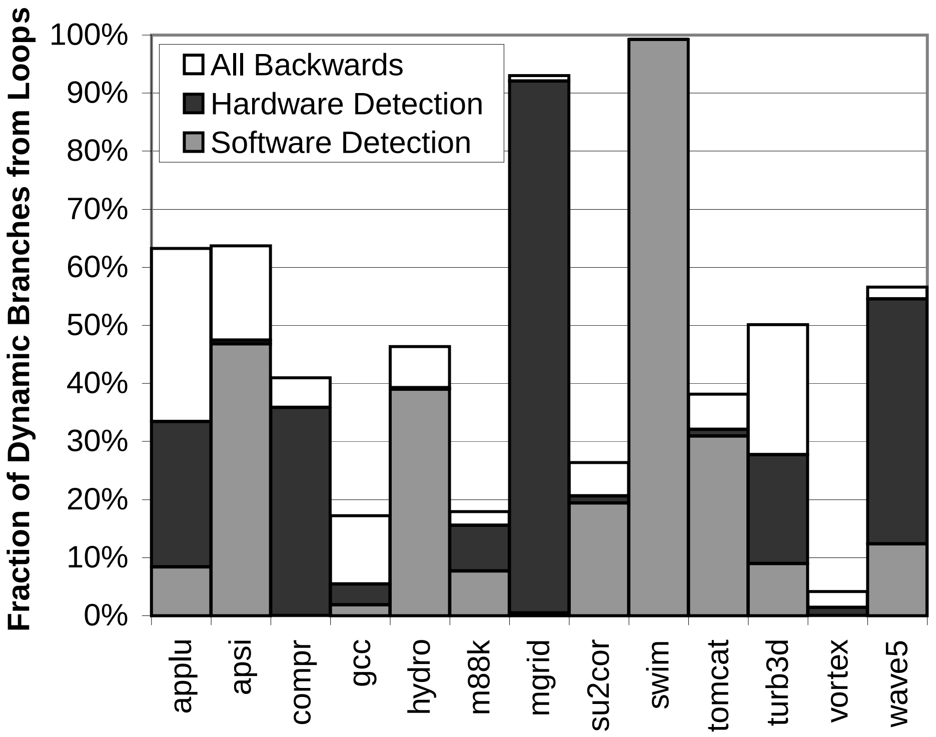


Figure 5: Percentage of conditional branches executed which are looping branches, as detected by being backwards, or as detected by hardware or software

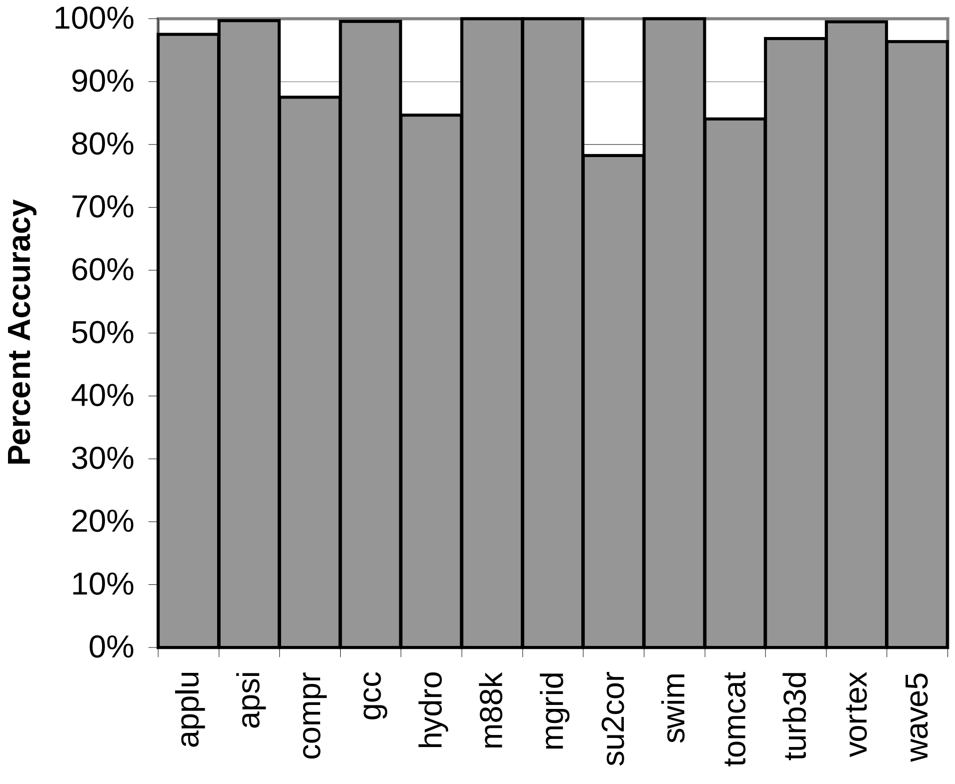


Figure 6: Confident Prediction Accuracy. The prediction accuracy of the LTB when it returns it’s prediction as confident.

application	0-9	10-19	20-39	40-69	70-99	100-199	200-399	400-999	1000+
applu	70.15%	0.00%	29.85%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
apsi	16.31%	0.00%	7.93%	59.20%	0.00%	11.88%	4.66%	0.01%	0.00%
compress	99.97%	0.00%	0.03%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
gcc	32.39%	0.00%	23.47%	20.49%	1.97%	5.67%	5.29%	10.73%	0.01%
hydro2d	2.01%	0.00%	0.03%	0.01%	43.94%	0.63%	9.72%	43.66%	0.00%
m88ksim	18.80%	0.00%	1.14%	0.03%	0.01%	0.08%	79.92%	0.00%	0.00%
mgrid	2.17%	0.00%	11.57%	86.25%	0.00%	0.00%	0.00%	0.00%	0.00%
su2cor	22.84%	0.00%	0.53%	1.07%	1.24%	69.54%	0.00%	4.77%	0.00%
swim	0.00%	0.00%	0.00%	0.00%	0.00%	0.20%	0.00%	99.80%	0.00%
tomcatv	4.03%	0.00%	0.00%	0.00%	0.00%	45.94%	0.00%	50.03%	0.00%
turb3d	60.60%	0.00%	36.89%	2.50%	0.00%	0.00%	0.00%	0.00%	0.00%
vortex	82.61%	0.00%	1.97%	3.69%	1.64%	7.98%	1.24%	0.87%	0.00%
wave5	3.19%	0.00%	0.00%	2.63%	0.00%	32.96%	6.56%	54.66%	0.00%

Table 1: Percent of executed loop branches that had an average trip count shown in each column header. For example, the results for `apsi` show that loop branches, which had a loop count between 40 and 69 iterations, accounted for 69% of the executed loop branches in the program.

We further examine these loop branches by breaking them down in terms of their iteration count. Table 1 shows the breakdown of loop branches in terms of the iteration count represented by the range in the column headers. These numbers are calculated by taking the average trip count for each static branch, multiplying this by the number of times the branch was executed, and then dividing this by the total number of loop branches executed. For example, the results for `mgrid` show that 88% of the loop branches executed were executed in a loop which went from 0 to a loop trip count somewhere between 40 to 69.

In looking at the results and source code for `compress`, we saw that 97% of the loop branches that were executed were in a loop which iterated between 0 and 7. This is the reason for the dominating short trip counts in `compress`. The Compaq compiler did not unroll this loop, perhaps due to a rarely taken `break` statement inside the loop. For this program, local (per-branch) history will correctly predict the loop branches, because the trip count is less than the local history size (see figure 7). But for many of the other programs, local branch history can not correctly capture all of the loop termination. The local history length is not sufficient to capture a trip count in the range of 40 to 69 iterations as in `mgrid`.

5.2 Branch Splitting

To examine the potential of Branch Splitting as presented in section 3, we profile each branch over the execution of the application. We track the number of mispredictions from each branch, along with information on the number of iterations between loop exists, and the regularity of the pattern found. A backwards branch is said to be regular if the branch direction pattern is $((1)^N 0)^m$ over the *entire* execution of the program, so the iteration count was always $N + 1$. If the branch is regular and the cycle count is larger than the local history size, then we applied branch splitting to the loop branch. The light gray part of the bar in figure 5 shows the percent of executed branches splitting was applied to.

Table 2 shows the potential reduction in branch mispredictions from applying branch splitting. The first two columns are the before and after overall branch misprediction rates, and the third column is the difference between them. The next column shows the total number of static branches in the executable. The last column is the percent increase in static branches when branch splitting was used. While the number of dynamic branches should not change significantly, any extra static branches can lead to more collisions in the table. Some applications such as `apsi` do quite well, achieving an absolute decrease in misprediction

app	miss rate	after splits	difference	static branches	% branch inc
applu	2.79%	2.52%	0.27%	1153	4.08%
apsi	2.64%	1.33%	1.31%	1642	4.45%
compress	10.96%	10.96%	0.00%	182	1.65%
gcc	9.07%	9.03%	0.04%	15952	0.42%
hydro2d	0.33%	0.09%	0.24%	1667	7.26%
m88ksim	4.67%	4.19%	0.48%	1017	0.10%
mgrid	1.64%	1.61%	0.03%	1172	0.34%
su2cor	4.15%	3.83%	0.32%	1717	3.38%
swim	0.20%	0.00%	0.20%	983	3.26%
tomcatv	0.99%	0.63%	0.37%	891	3.03%
turb3d	2.98%	2.65%	0.32%	1274	1.18%
vortex	1.69%	1.69%	0.00%	6259	0.18%
wave5	0.74%	0.30%	0.44%	1794	2.84%

Table 2: Effects of Branch Splitting. The first two columns are the before and after overall branch misprediction rates, and the third is the difference between them. Static branches is the total number of branches in the executable. The last column is the percent increase in static branches when branch splitting is used.
rate of 1.3%.

5.3 Loop Termination Prediction

In this section, we examine the reduction in branch misprediction rate from adding our Loop Termination Buffer to the Meta predictor, represented by `Meta` and `Meta+LTP`, and to the LGC predictor, represented by `LGC` and `LGC+LTP`. For the results gathered, each branch prediction table (Meta, Local, Bimodal) has 32K entries per table. For the loop termination predictor, we simulate adding a small 32 entry fully associative LTB with random replacement to the base predictor. The size of this predictor (for both the tag and data) is less than 256 bytes.

The confidence prediction accuracy of LTP is shown in figure 6, and is independent of the base predictor used. This is the percent of loop branches that were predicted to be accurate by the confidence counter in the LTB. Recall from section 2 that the confident bit is set when the the branch PC is found in the table and it has seen the same iteration count two times in a row. The reason that the prediction accuracy is not near 100% for all applications is that some loops have break statements, and/or they change the number of times they iterate during the execution of the program.

We next examine the effects of LTP on the prediction accuracy of loop branches and all executed branches. Figure 7 shows the branch misprediction rates for only the loop branches, and figure 8 shows the misprediction rate in terms of all executed branches. Results are shown using 32K entry tables for the base Meta and LGC predictor, and when adding our LTB to the predictor. `Compress`, as discussed above, has a small inner loop that contains a break statement and is not unrolled and pipelined by the Compaq C complier. This loop escapes the Meta prediction but is fully captured in the LTB. It is also captured by the LGC predictor in its local history. The results in figure 8 show that we reduce the overall miss rate for `mgrid` from 1.7% down to 0.1% using loop termination prediction. Figure 9 shows the code example of one of the routines in `mgrid` which accounted for 22% of the branch mispredictions. Almost all of these were eliminated using loop termination prediction.

Table 3 shows the average number of instructions between mispredicted branches with and without loop termination prediction for each of the 3 simulated systems. The results show a large increase in additional ILP exposed by eliminating the mispredictions due to loop termination branches.

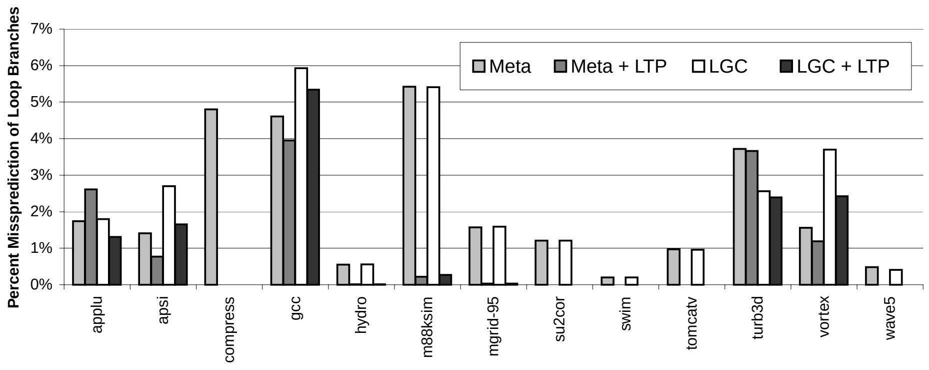


Figure 7: Loop Misprediction Rate. The prediction accuracy of McFarling's Meta predictor and a local/global chooser with 32k entry tables, with and without loop termination prediction for only loop branches.

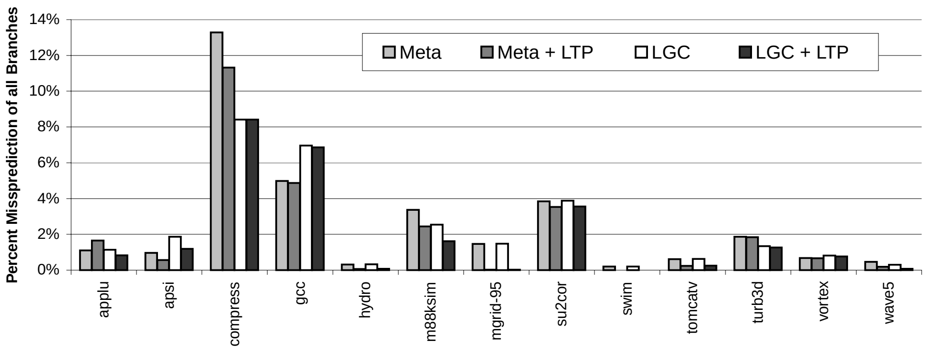


Figure 8: Overall Misprediction Rate. The prediction accuracy of Meta and LGC with 32k entry tables, with and without loop termination prediction for all branches.

```
      SUBROUTINE RESID(U,V,R,N,A)
      INTEGER N
      REAL*8 U(N,N,N),V(N,N,N),R(N,N,N),A(0:3)
      INTEGER I3, I2, I1
      DO 600 I3=2,N-1
      DO 600 I2=2,N-1
      DO 600 I1=2,N-1
 600  R(I1,I2,I3)=V(I1,I2,I3)
     >      -A(0)*( U(I1,  I2,  I3  ) )
     >      ... additional matrix computations left out of example
      CALL COMM3(R,N)
      RETURN
      END
```

Figure 9: Looping code example from `mgrid`, which accounts for 22% of the mispredicted branches.

name	32k Meta	32k Meta+LTP	32k LGC	32k LGC+LTP
applu	1969	1310	1898	2617
apsi	3394	5863	1755	2745
compress	54	63	85	85
gcc	147	151	105	107
hydro2d	5206	26277	5071	23871
m88ksim	311	430	414	650
mgrid	4370	197808	4320	215815
su2cor	333	363	331	360
swim	25002	2721829	24994	2635741
tomcatv	3731	9356	3691	9111
turb3d	1508	1532	2102	2244
vortex	1562	1599	1306	1397
wave5	6844	16752	10421	45375

Table 3: Average number of instructions between a mispredicted branch for each of the predictors and table sizes simulated.

5.4 Compiler Interaction

Our loop termination prediction architecture is a highly accurate structure that rarely makes incorrect predictions. However, one fairly common loop type will cause problems. Loops with `break` or `continue` instructions. For example, a loop with a `break` statement in it will have the same trip count several times in a row, and then terminate abruptly. While these types of loops are still predicted with higher accuracy than in a traditional predictor, they prevent perfect prediction, even with confidence. Table 2 shows the results of eliminating mispredictions for only loop branches (using branch splitting) for only branches that always have the same iteration count. Figure 8 shows that we achieve a much higher reduction in misprediction rate using hardware loop termination prediction. This is because the LTB is accurately predicting loops whose iteration count change over the lifetime of the program.

A common compiler optimization to improve memory performance it to tile loops. This optimization creates smaller inner loops to traverse over the data, keeping as much data in the cache creating reuse and reducing cache misses. This creates much more loop termination behavior, which could be correctly predicted by our loop termination buffer. In addition, our branch splitting results indicate that it may be worthwhile to also take into consideration the size of the branch history registers for branch performance, when tiling a loop for memory performance. For the results gathered in this paper, the compiler did not perform any tiling.

6 Related Work

Branch prediction research has concentrated on reducing aliasing into branch prediction tables, and to increase accuracy by using chooser/meta predictors which allow branches to be predicted correctly by both local or global history, based upon confidence information [1].

6.1 Other Branch Predictors for Predicting Loop Termination

Recently, researchers have examined using prior committed values [4] or value prediction in combination with traditional branch prediction to improve branch prediction accuracy [2, 3]. These predictors can accurately predict loop terminations for long loop histories, since they predict the branches based on past or predicted values. They are more general and can potentially capture more data correlation for branches besides loop termination, although at the cost of adding large buffers to store values or value differences. Our loop termination buffer approach accurately predicts loops even with just the addition of a small 32 entry prediction buffer (less than 256 bytes in size). In addition, they are fundamentally different when predicting loop terminations from LTB. While the value-based approaches predict the input operands (or their difference), the LTB predicts loop trip counts and keeps track of the current speculative loop iteration of the branch. The benefit our LTB approach is its simplicity and low cost, and it can easily be added to an existing predictor with very little overhead or increase in cycle time.

6.2 Predicting Loop Iterations for Speculative Thread Generation

Knowing the speculative loop iteration is an important area of research for speculative threaded execution. Tubella and Gonzalez [13] examined adding a *Loop Execution Table* (LET) and *Loop Iteration Table* (LIT) to a speculative threaded processor, to provide prediction information for loop iterations. The LET is used to predict the number of iterations for each loop to guide the generation of speculative threads. Whereas the LIT is used to keep track of information related to the live-in registers and memory locations from the last loop iteration.

Our loop termination buffer is very similar to the function of the loop execution table. Since their goal was to guide speculative thread generation, they did not examine using the LET for branch prediction. Our LTB has a confidence predictor, which is not in the LET, to determine when to use the loop termination prediction or the original branch predictor. In addition, the LTB keeps track of a speculative and non-speculative loop count, which is used to restore the loop count in the LTB in case of a misprediction.

6.3 Loop Counting Branch Instructions

Some architectures have special instructions for loop branches. The IBM Power PC [5] has a loop-based branch instruction that branches based on the value of a Count Register. The Intel IA-64 [6] has an identical branch called the Counted Loop Branch. For both of these architectures, when a loop-branch instruction is executed, if the count register is non-zero, then the register is decremented and the branch is taken. Otherwise, the branch is not-taken (the loop is terminated).

The PowerPC and IA-64 architectures currently do not use this loop-based branch instruction for LTP. In order to use it for loop termination prediction, an architecture would need to speculatively keep track of the loop count value. Each time the loop-branch is predicted, the speculative loop count would then be decremented. When the speculative count value reaches zero, the branch would be predicted as terminated. In actuality, a stack of speculative count values would needed, so that the count register could be saved between procedure calls. This is if the loop-branch instruction can be saved in order to allow different nested procedures to use the loop-branch register.

A stack-based speculative loop count predictor would be very similar to the Loop Termination Buffer we propose. Even so, the Loop Termination Buffer would be preferred, since it can capture more types of branches than those that are implemented using the loop-based branch instruction, and we can achieve a high degree of loop branch coverage with a LTB of size less than 256 bytes.

7 Summary

Loop terminations can constitute a large percentage of the mispredictions generated by current branch predictors. In this paper we proposed two schemes for dealing with this problem, the Loop Termination Buffer (LTB) and Branch Splitting.

The LTB is a hardware mechanism that detects and predicts branch patterns of the form $((1)^N 0)^m$. The LTB tracks branches with this behavior and then informs the predictor when it has found such a pattern. The addition of the LTB allows regular loop guarding branches to be predicted with 100% accuracy. However, we also find a significant benefit from the LTB for branches that have a loop trip count that changes over the lifetime of the program. The LTB reduced the loop mispredictions from 5.4% down to 0.2% for `m88ksim`, and aided the overall prediction accuracy by a significant amount for most programs.

In addition, we examined the potential of using a software approach called Branch Splitting to correctly predict loop branches. For loops that have iteration counts larger than may be captured by local history, the loop guarding branch may be split into two or more branches all of which have an interaction count that would be captured by local history. This approach resulted in only small reductions in misprediction rates for some programs, since we only applied it to loops that had the same iteration count for the execution of the whole program. For `apsi`, branch splitting decreased the misprediction rate from 2.6% down to 1.3%.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF CAREER grant number CCR-9733278, by DARPA/ITO under contract number DABT63-98-C-0045, and a grant from Compaq Computer Corporation.

References

[1] A. Eden and T. Mudge. The YAGS branch prediction scheme. In *31st International Symposium on Microarchitecture*, pages 69–77, December 1998.

[2] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *31st International Symposium on Microarchitecture*, December 1998.

[3] Gonzalez and Gonzalez. Control-flow speculation thorough value prediction for superscalar processors. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[4] T. Heil, Z. Smith, and J.E. Smith. Improving branch predictors by correlating on data values. In *32nd International Symposium on Microarchitecture*, November 1999.

[5] IBM. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, 1994.

[6] Intel. *IA-64 Application Developer's Architecture Guide*. Intel Corporation, Order Number 245188-001, 1999.

[7] R.E. Kessler, E.J. McLellan, and D.A. Webb. The alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, October 1998.

[8] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[9] C.H. Perleberg and A.J. Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, 1993.

[10] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 259–271, December 1998.

[11] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–148. ACM, 1981.

[12] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[13] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *4th International Symposium on High Performance Computer Architecture*, February 1998.

[14] T. Yeh. Two-level adpative branch prediction and instruction fetch mechanisms for high performance superscalar processors. Ph.D. Dissertation, University of Michigan, 1993.

[15] T. Yeh and Y. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 129–139, December 1992.

[16] T.-Y. Yeh and Y. Patt. Two-level adaptive branch prediction. In *18th Annual International Symposium on Computer Architecture*, pages 51–61, May 1991.