Runtime Predictability of Loops

Marcos R. de Alba
David R. Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, USA
mdealba,kaeli@ece.neu.edu

Abstract

To obtain the benefits of aggressive, wide-issue, architectures, a large window of valid instructions must be available. While researchers have been successful in obtaining high accuracies with a range of dynamic branch predictors, there still remains the need for more aggressive instruction delivery.

Loop bodies possess a large amount of spatial and temporal locality. A large percentage of a program's entire execution can be attributed to code found in loop bodies. If we retain this code in a buffer or the cache, we do not have to refetch this code on subsequent loop iterations. Loops tend to iterate multiple times before exiting, thus providing us with the opportunity to speculatively issue multiple iterations.

While some loops can be unrolled by a compiler, many contain conditional branches. The number of times a loop iterates may be dependent on a program variable. These issues can hinder our ability to speculatively issue multiple iterations of a loop. If we are able to profile loops during runtime, we can use this information to more accurately issue speculative paths through loop bodies.

In this paper we present a characterization of loop execution across the SPECint2000 benchmark suite. We intend for this study to serve as a guide in the selection of design parameters of a loop path predictor. We characterize the patterns exhibited during multiple visits to a loop body. We present the design of a table that records path-based loop execution history and allows us to predict multiple loop iterations dynamically.

1 Introduction

Branch prediction has been studied extensively over the past 20 years [2, 4, 8, 11, 14, 18]. The goal of a dynamic branch predictor is to predict the address of the next instruction to be fetched. When the prediction is correct, the target instruction stream can be fetched prior to the resolution of the branch; when the prediction is wrong, a penalty will be imposed to squash the speculatively fetched/executed instructions and to compute/fetch the correct instruction address. Fortunately, present conditional branch predictors produce accurate predictions [18].

We are presently designing a very aggressive, high-IPC, machine architecture [17]. We have found that the success of our design will rest on our ability to provide a wide window of valid instructions to issue. While branch predictors can greatly aid instruction delivery, there still remains the need to increase the window of available instructions.

Programmers frequently use loop constructs to implement iterative algorithms. The instruction level parallelism (ILP) contained in loops represents a large percentage of the total ILP in the program. For this reason it is important to consider how best to explore this ILP and provide a large window of instructions to the processor.

To exploit the ILP present in a loop, we need to effectively *unroll* the loop at runtime, providing multiple iterations of the loop to the instruction window. Loop unrolling has been well studied [1, 9, 12]; compiler writers have clearly demonstrated the value of this technique [7, 10]. While compilers can unroll simple loop constructs, they are not able to unroll a loop that contains data-dependent indices, control-dependent loop bodies or irregular nesting.

In this paper we characterize the dynamic behavior of loops. We attempt to capture important characteristics that can be used to predict the paths executed through a loop body upon each visit to a loop. We are particularly interested in the probability that a loop will iterate, and the particular path executed during a given iteration. This information will help to predict how to speculatively issue multiple loop iterations. This can also be used to reduce the burden of refetching the loop. Our ultimate goal is to improve instruction delivery beyond aggressive branch pre-

diction so that wide-issue architectures do not stall, waiting for instructions.

This paper is structured as follows. In Section 2 we review related work on loop prediction and loop termination prediction. In Section 3 we develop a set of terminology for describing loop characteristics. In Section 4 we discuss the methodology used to capture and evaluate loops, and Section 5 presents results. Section 6 provides a discussion of some of the design implications as a result of our study, and Section 7 summarizes the paper.

2 Previous work

Sherwood and Calder [13] propose a technique to enhance the prediction accuracy of branches associated with loops. *Loop termination* attempts to detect branch instructions that are associated with loops. When the inner loop in a pair of nested loops terminates, a wrong branch prediction is encountered. Since this termination may occur many times (it is inside an outer loop), a number of mispredictions can be encountered. Two-bit predictors will not capture the termination since they can only capture a small number of iterations (dependent upon the length of the pattern history register). Sherwood and Calder describe a solution called *branch splitting*, which splits loops with a large number of iterations into two or more loops containing a smaller number of iterations.

Kobayashi [6] describes a technique to detect loops at runtime. More recently, Tubella, Gonzalez and Marcuello [3, 16] study loop detection in the context of multithreaded processors. In [16], they propose a method to dynamically detect the presence of loops. In [3], they propose to execute different iterations of a loop in different threads to provide improved fetch bandwidth.

The goal of this work is to evaluate whether it is possible to predict the paths that will be followed over multiple iterations of a loop. Profiling is performed at runtime. Besides computing statistical information on many loop features, we also study the correlation between subsequent visits to the same loop. We begin by attempting to build a taxonomy for loop characteristics, then present a series of results for the SPECint2000 benchmark suite.

3 Loop Characteristics

Loops occur in program flow whenever control resumes at a negative displacement. Programmers utilize loops to carry out iterative algorithms or repetitive actions, looping multiple times to carry out the desired operation. The collective set of instructions contained inside the loop is called the *loop body*. The first instruction in the loop body is called the *loop head*.

Well-structured loops iterate for a constant number of times and always execute the same loop body (i.e., the loop body is absent of any non-deterministic control flow). *Ill-structured loops* contain conditional control flow inside their loop bodies. These loops are difficult to unroll at compile time, so we will use profiling to establish the predictability of these loops.

Variable-dependent loops iterate a variable number of times, determined by a run-time dependent variable. These loops are also difficult to unroll at compile time, so we will again use profiles to speculate on the number of iterations on subsequent loop visits.

Visiting a loop is defined as entering a loop body from outside of the loop body. A loop *iteration* is defined as traversing a path through the loop body one time. A *loop path* defines the sequence of instructions executed during a single iteration of a loop. Typically a loop will *iterate* multiple times during a single loop visit. A *loop pattern* defines the sequence of paths taken during a single visit to the loop, capturing the sequence of paths taken through the loop body on each iteration.

Now that we have defined these terms, we will begin to use them to discuss the loop characteristics present in a set of programs. These characteristics include the number of iterations per loop visit, as well as the path traversed (measured in the number of instructions executed) per iteration. We also study the predictability of the number of iterations and the predictability of the number of instructions executed per iteration.

This information will allow us to predict how future loop visits and iterations will behave. The behavior of loops is somewhat diverse; some loops exhibit a strictly static behavior across multiple iterations or multiple visits, while others behave radically different (making them more difficult to predict). We have also identified correlation between different visits to a loop, as well as the effects of loop nesting on predictability.

4 Methodology

In this work we study the looping characteristics present in the SPECint2000 benchmarks. While other programs may contain a larger number of well-structured loops, our goal is to understand loop behavior present in general purpose programs. This class of the programs tends to present more ill-structured and variable-dependent loops.

We leverage the method proposed by Tubella and Gonzalez [16] to detect loops at run time. Once a loop is detected, we can then record a significant amount of information at run time.

We compile and run our programs on the Compaq Alpha 21264 processor. We use the native DEC C V5.9-008 compiler with the `-non_shared -O2` options. The `-O2`

switch enables loop unrolling, inlining and code replication. These options play a critical role in determining the predictability of loops. Loop unrolling will unroll loops that are deterministic in nature, leaving the ill-structured and variable-dependent loops to be predicted by a loop predictor.

To capture loop characteristics, the ATOM instrumentation tool [15] is used. We capture statistics, avoiding program initialization (the first 100M-250M instructions), and capture a sample of 500M-1B instructions, depending on the benchmark being studied.

5 Results

Table 1 shows the benchmarks and input files used, the instructions executed and the number of unique dynamic (i.e., executed) loops analyzed for each benchmark program.

Table 1. Benchmarks from SPECint2000 selected, input datasets used, number of instructions analyzed and number of unique loops captured.

Benchmark	Input	Instrs	Loops
300.twolf	ref	1000M	130
go	5stone21.in	500M	498
176.gcc	166.i	500M	1192
164.gzip	input.random	500M	23
256.bzip2	input.random	500M	31
197.parser	ref.in	500M	302

Figure 1 shows the distribution of control-flow related instructions. The graph shows the contribution of each type of branch/jump/call/return instruction. Instructions types profiled are as follows:

- Forward conditional branches, $fwdcnd$
- Backward conditional branches, $bwdcnd$
- Forward unconditional branches, $fwdunc$
- Backward unconditional branches, $bwdunc$
- Forward jumps, fwd
- Jumps to subroutine, $jsr/call$
- Branches to subroutine, $bsr/call$
- Returns from subroutine, ret

The instructions of particular interest in our study are backward conditional and backward unconditional

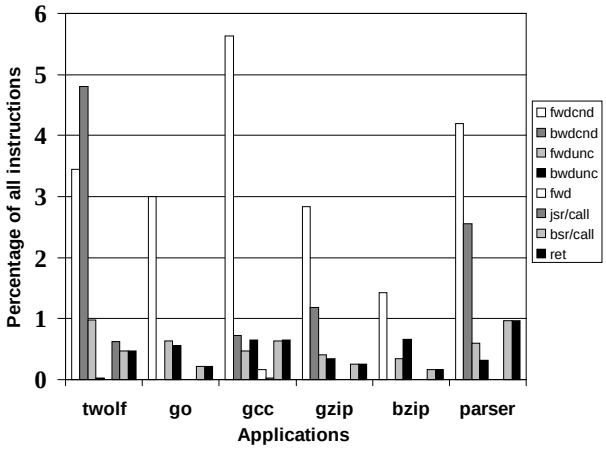

Figure 1. Mix of control flow instructions.

branches. These branches mark the end of loop bodies. The frequency of these branches in relation to other instructions in a program gives us a good sense of the loop density in each benchmark.

In most cases, programs containing loops that iterate a large number of times, also contain fewer loops. A good example of this case is in *gcc*, where the number of dynamic loops is large (as we can see in Table 1), but each loop executes a much smaller number of times (as we can see in Table 2). The opposite case can be observed for *gzip*.

The history we record about each loop includes the number of loop visits, the different patterns present in the loop body, and the number of iterations executed. The majority of the loops found in our set of benchmarks follow a small number of repeating patterns, and these patterns are constructed from subpatterns. This is the premise for us to consider aggressive path prediction utilizing hardware resources.

Figure 2 shows the ratio of conditional to unconditional branches that control the studied loops. We can see that the mix of instructions varies across the different benchmarks. *Go* and *bzip2* mainly consist of backward unconditional branches, versus *twolf* and *gzip* which contain a larger proportion of backward conditional branches.

Figure 3 shows the distribution of loop visit frequencies. As stated before, the benchmarks that contain a smaller number of loops also contain loops that are visited a large number of times (e.g., *twolf*, *gzip* and *bzip* contain 130, 23, 31 static loops, respectively, and produce the highest numbers of loop visits).

Table 2 shows the average and weighted mean of the

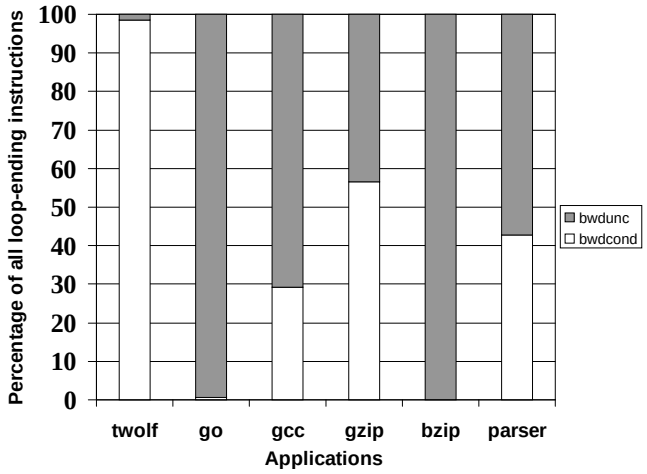

Figure 2. Instruction mix of conditional and unconditional instructions that terminate loops.

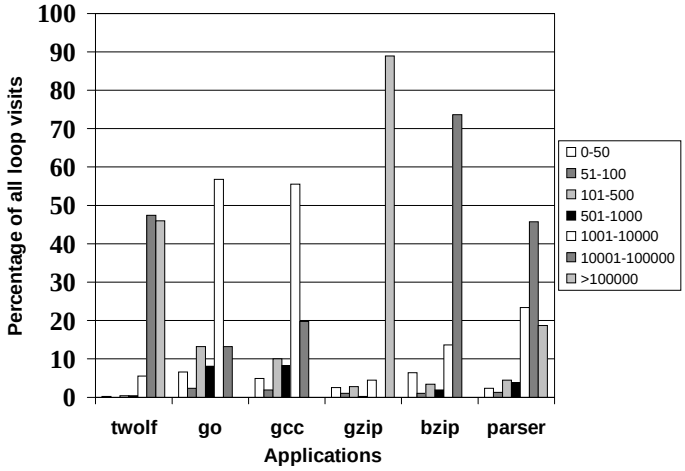

Figure 3. Distribution of loop visit frequencies.

number of iterations per loop iteration for the set of benchmarks. The weight used in column 3 is the frequency that each loop is iterated. We can also notice the trend that more frequently iterated loops contain a small number of instructions.

Table 2. Average number of iterations per loop visit and weighted average (weighted by the number of iterations).

Benchmark	Avg. # of its per loop visit	Weighted Avg.
twolf	790.61	6.05
go	14.61	3.45
gcc	22.58	3.43
gzip	5816.76	8.90
bzip2	73846.64	5.76
parser	477.13	4.83

Figure 4 shows a breakdown of the number of loop iterations on a per loop visit basis. We can see that for *go*, *gcc* and *gzip*, more than 90% of the loops iterate between 0 and 50 times. For *twolf* and *parser*, close to 60% of their loops perform between 0 and 50 iterations. This will make the design of a hardware-based loop prediction mechanism more challenging since temporal locality will be high, but not as high as we would like. *Bzip* is a special case, since more than 80% of its loops iterate more than 10000 times.

Figure 5 shows the distribution of the number of instructions executed per iteration. One thing to note here is the difference between *gzip* and the rest of the applications. *gzip* contains a few short, hot paths. For 4 out of 6 applications, larger loops (greater than 1000 instructions per iteration) dominate the statistics. While this may seem surprising, remember that the compiler is performing unrolling and inlining at compile time. This will substantially increase the length of a loop body, and will also eliminate many of the *easy to predict* loops.

Table 3 shows the prediction rate for the number of instructions executed on average per iteration. Notice that the prediction rate is at or below 50% in some benchmarks (i.e., *go* and *gcc*). To generate these results we measured only the last value seen (similar to a single bit of history used in some early dynamic branch predictors) [5]. This suggests that we might need to utilize a more sophisticated mechanism, so that the accuracy can be improved. The main purpose of this measurement was to show that even a simple loop predictor can predict the number of iterations most of the time.

Table 4 shows the average probability that a loop will iterate. Our prediction is only based on the last behavior of the loop. Although we might expect the accuracy to be above 95% for most benchmarks, the results obtained show that many loops in SPECint2000 are hard to predict accu-

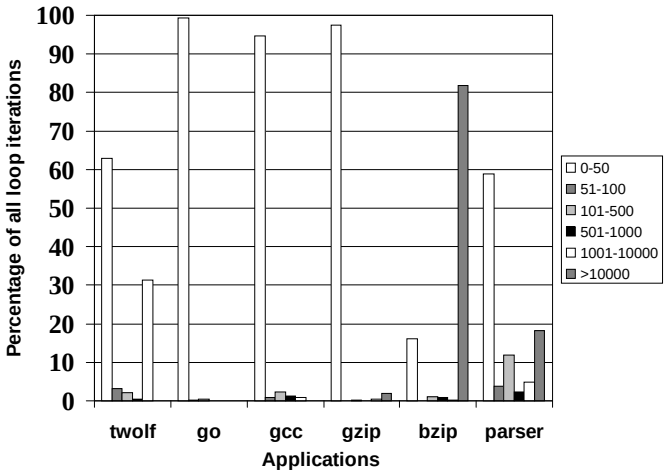

Figure 4. Loop iteration distribution. Each bar indicates a range in the number of loop iterations.

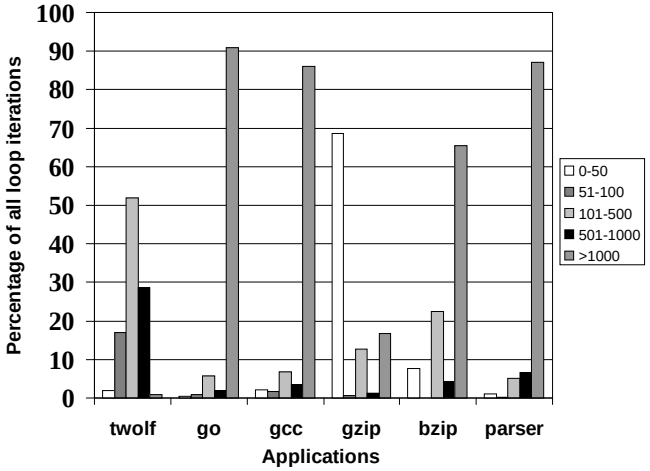

Figure 5. Distribution of the number of instructions per iteration. Each bar indicates a range in the number of instructions executed per iteration.

Table 3. Average prediction rate that the number of instructions per iteration will repeat (last iteration-value predictor).

Benchmark	Prediction rate
twolf	0.70
go	0.37
gcc	0.50
gzip	0.65
bzip2	0.62
parser	0.52

Table 4. Average prediction rate that a loop iterates again.

Benchmark	Prediction rate
twolf	0.82
go	0.70
gcc	0.68
gzip	0.87
bzip2	0.81
parser	0.74

rately. The main reason for this is that some nested loops are hard to predict. The prediction accuracy can be improved using a more elaborate methodology, as was suggested in [13]. We also are investigating more elaborate predictors for nested loop branches.

Table 5. Maximum, average and, weighted mean nesting depths of loops.

Benchmark	MaxNL	AvgNL	WeightNL
twolf	2	0.51	1.06
go	10	3.06	2.54
gcc	7	1.92	1.61
gzip	2	0.86	1.04
bzip2	5	1.25	1.51
parser	6	1.61	1.21

Table 5 presents loop nesting depth information for our set of benchmarks. While obtaining this data, we observed that loops with higher nesting depths (e.g., 5-6), exhibit a more predictable behavior than those with smaller loop depth values. In addition, it was observed that it is easiest to predict the innermost loops (these loops typically have the highest frequency of execution in the loop nest, and thus, have a highly predictable nature). Table 6 presents predictability versus loop nest depth for *gcc*. As we can see, loops at a nesting depth of 6 are highly predictable.

To improve upon the predictability of the number of it-

Table 6. Predictability of the number of iterations versus loop nest depth.

Nesting Depth	Prediction rate for the # of its
1	40.52
2	62.73
3	71.67
4	77.68
5	90.28
6	98.96

erations and the predictability of the number of instructions per iteration, we attempt to identify patterns exhibited across multiple iterations of a single loop visit. For all visits of a single loop, there exist a sequence of one or more patterns, and within each pattern there exist one or more *(instruction executed per iteration, number of iterations)* pairs. The number of instructions executed will be replaced in our loop predictor hardware with a shift register that records branch outcomes contained in this loop iteration. To determine the predictability of a loop, we record the frequency of each pattern and the frequency of each pair within the pattern. We then use the frequency of the entire loop visit pattern to predict the sequence of patterns to be exhibited upon future loop visits. The loop visit history is updated upon loop exit. We use the most frequently exhibited pattern to predict future behavior. We call this technique *self-correlating loop visit prediction*.

One loop characteristic which we exploit in our strategy is that the most frequently executed loops are also the most predictable loops. A loop is less predictable if it contains a large number of patterns or contains a large number of pairs. Similarly, a loop is more predictable if it contains a small number of patterns, with a small number of pairs within each pattern, and these pairs vary largely in frequency (i.e., a few loop patterns dominate).

Figure 6 shows the distribution of iteration predictability. The predictability of the number of iterations for each iteration is based on using a complete loop history, as described above. These results differ from the predictability values in Tables 3 and 4, since now we utilize a loop pattern captured on a per loop visit to predict individual loop iteration characteristics. We report on predictability on an individual iteration basis. We divided the predictability distribution into 10 ranges, from 0 to 100%, in increments of 10. The bar for *twolf* shows that, when using self-correlating loop visit prediction, close to 63% of the loop iterations are predicted correctly 90-100% of the time. Nearly 13% exhibit a predictable pattern 80-89% of the time. While it was clear from Tables 3 and 4 that there exists predictability in a single loop, we can now see that by capturing correlation

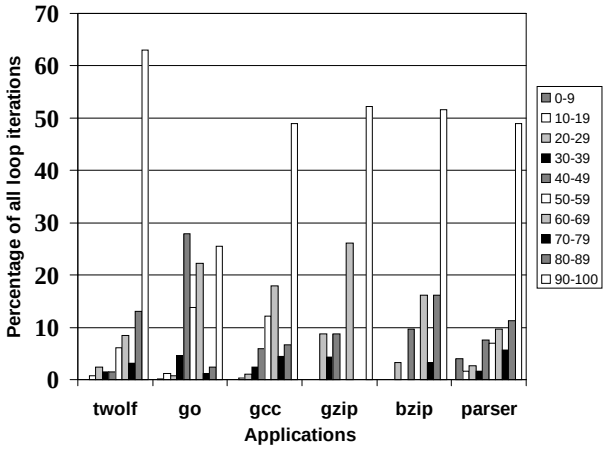

Figure 6. Distribution of iteration predictability. Predictability is computed on a per iteration basis.

patterns for a single loop, we can increase the predictability of the entire loop.

Figure 7 shows the predictability of the number of instructions contained in each loop iteration, again using our self-correlating loop visit predictor. Again, we can see that *twolf* exhibits high predictability. Results are strikingly similar to Figure 7, which demonstrates that we can accurately predict both the number of iterations and the paths executed.

6 Discussion

The main goal of this study was to produce statistical results of dynamic loop characteristics and use them to propose reasonable design parameters for a hardware-based loop predictor. We have purposely not constrained the number of patterns or loops that we can record information for, just so we have an indication of the amount of predictability present in the loop patterns present in these applications.

The objective of dynamically unrolling more loops is to maximize the throughput of the fetch unit and to increase the utilization of the available processing units. In our current high-ILP design [17], instruction fetch bandwidth is presently a limiting factor. A loop path predictor provides the capability to speculatively execute several iterations of a loop.

In this study we observed that in 5 of the 6 benchmarks, more than 50% of the loop iterations are easy to predict (having a predictability rate above 90%) and that for all of

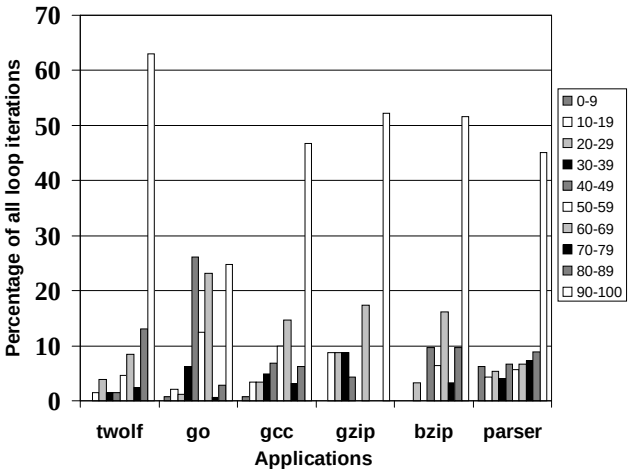

Figure 7. Distribution of instructions per iteration predictability.

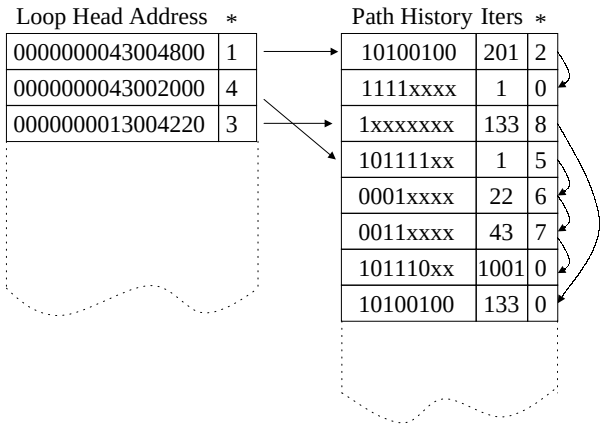

Figure 8. Hardware design of a self-correlating loop predictor.

the benchmarks more than 50% of the loop iterations have predictability rates of more than 80%. Our results also suggest that we can predict the number of iterations and the associated path for more than half of the loops visited. In addition, the study also showed that many loops contain loop bodies that exhibit more complicated iteration patterns. We are proposing that we can detect these patterns at run time using pattern-based correlation.

6.1 Design parameters of loop predictor

From the results obtained in this study it is observed that the vast majority of loops execute more than 50 instructions, and in many cases, more than 1000 instructions. Constructing dedicated loop buffers of this size is possible, but may consume a considerable amount of real estate. We instead suggest augmenting either the instruction fetch and/or the instruction window logic to retain these instructions, recognizing that their execution is currently within an iterative structure, and that there is a high probability that we will reissue these same instructions shortly.

Figure 8 shows the preliminary design of a dynamic self-correlating loop predictor. The mechanism contains a two-level table design. The first level allows us to index into the table using the loop head address. When an instruction fetch is issued for a block containing a loop head address, we will then use the loop predictor mechanism for predicting branches.

The second level of the table contains a sequence of entries, each storing the unique patterns exhibited by the loop. From our characterization of loops, we have found that the most frequently executed loops contain only a few patterns, and that these patterns tend to repeat a number of times. Repetitions of a single pattern of a loop iteration are captured in the `Iters` entry in the table. We can chain together entries in the second level table using indices. This effectively captures state transitions during a single loop visit. Entries are updated on each execution of this loop. Once we have recorded a complete loop visit history, we can use it upon subsequent loop visits. In future work we plan to provide multiple first-level table entries for a single loop head address, using the pattern of branch outcomes exhibited prior to entering the loop to select a particular first-level table entry.

7 Conclusions

The locality present in loops makes them attractive to propose more aggressive strategies to improve instruction delivery. While compilers have recognized the benefits provided by unrolling loops, many loops can not be unrolled at compile time due to their structure. In this paper we have investigated the runtime predictability of these ill-behaved and variable-dependent loops. We have looked at both using a last value approach, as well as a self-correlating loop visit predictor. Predicting entire loop visits has the poten-

tial to expose substantial amounts of ILP. We have also included the design of a mechanism called a self-correlating loop predictor, that will allow us to accurately detect loop patterns.

Future studies on the predictability of loops should not only focus on the predictability of loop branches, but also produce the right path through the loop body and the total number of iterations for the entire loop visit. The combination of branch prediction, compiler-based loop unrolling and procedure inlining, coupled with hardware-based loop prediction and loop unrolling promises to improve instruction delivery to wide-issue microprocessors.

Acknowledgments

Marcos de Alba is a Professor of the Electronic Engineering Department at the *Universidad Autonoma Metropolitana* and is a Fulbright grantee for Doctorate studies at Northeastern University. David Kaeli is supported by a grant from the *Ministry of Education, Culture and Sports* of Spain and by a grant from the National Science Foundation.

References

[1] J. Davidson and S. Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 125–132, New York, NY, 1995. ACM Press.

[2] N. Gloy, C. Young, J. B. Chen, and M. D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proc. 23rd Annual Intl. Symp. on Computer Architecture*, pages 12–21, 1996.

[3] A. Gonzalez and P. Marcuello. Dependence speculative multithreaded architecture. Technical report, Universitat Polytecnica de Catalunya, 1998.

[4] D. Kaeli and P. Emma. Improving the accuracy of history-based branch prediction. IEEE Transactions on Computers, 46(4):469–472, April 1997.

[5] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, volume 19-3, pages 34–42, New York, NY, 1991. ACM Press.

[6] M. Kobayashi. Dynamic characteristics of loops. *IEEE Transactions on Computers*, 33(2):125–132, 1984.

[7] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23, pages 318–328, 1988.

[8] J. Lee and A. Smith. Branch prediction strategies and branch target buffer design. IEEE Computer 17(1), January 1984.

[9] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. Hwu, P. Chang, and T. Kiyohara. Compiler code transformations for superscalar-based high-performance systems. In *Proceedings Supercomputing '92*, pages 808–817, Minn., MN, 1992. IEEE.

[10] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions in Programming Languages and Systems*, 18(4):424–453, 1996.

[11] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 76–84, New York, NY, 1992. ACM Press.

[12] S. Sair and D. Kaeli. A study of loop unrolling for vliw-based dsp processor. Proceedings of the Workshop on Signal Processing Systems, October 1998.

[13] T. Sherwood and B. Calder. Loop termination prediction. In *Proceedings of the 3rd International Symposim on High Performance Computing*. Springer-Verlag, October 2000.

[14] J. E. Smith. A study of branch prediction strategies. In *Proceedings of 8th Symposium in Computer Architecture*, pages 135–148, Minneapolis, MN, 1981.

[15] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. DEC Western Research Laboratory Technical Report, 1994. Technical report.

[16] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, Las Vegas, NV, January 1998.

[17] A. Uht, D. Morano, A. Khalafi, M. de Alba, T. Wenisch, M. Ashouei, and D. Kaeli. Ipc in the 10's via resource flow computing with levo. Technical Report TR No. 092001-001, University of Rhode Island, 2001.

[18] T. Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *The 20th Annual International Symposium on Computer Architecture*, pages 257–266, Goteborg, Sweden, 1993.