Path-based Hardware Loop Prediction

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

Abstract

For microprocessors that attempt to exploit instruction level parallelism, it is necessary to have a large window of candidate instructions from which to issue from. With loop prediction, we can predict the number of times that a loop will iterate, as well as the paths that will be followed inside the loop body. With this type of prediction, several basic blocks can be prefetched and stored in a dedicated loop buffer, reducing the number of instruction cache and memory requests, while providing a large window of instructions for speculative execution.

In this paper we describe the design of a path-based loop predictor that will allow us to unroll loop iterations using a hardware mechanism. To begin to design such a mechanism, we have evaluated three different loop prediction mechanisms for accurately predicting the number of loop iterations. We also evaluate the correlation between the pattern of branch outcomes that lead up to a loop with the probability that we iterate in the loop, as well as the pattern(s) of conditional branches within loops. Finally, we propose and evaluate a design to predict the instruction stream followed during an entire loop execution. The main benefits of path-based loop predictor are: 1) the reduction in the number of instruction fetch requests, and 2) a wide window of instructions is available to issue from.

In this work we evaluate three different sets of benchmarks: mediabench, mibench and SPEC2000. Our results suggest that loop prediction can be performed in 20-25% of all loop executions, and can potentially provide us with new levels of instruction level parallelism.

keywords

Loop Prediction, Prefetch, ILP, Loop Buffer, IPC.

1 Introduction

Branch prediction is a technique used to speculate on the address of the next block of instructions to be fetched. When a branch is predicted correctly, several cycles can be saved. Sophisticated branch prediction mechanisms can be found on most microprocessors in production today.

The performance benefits obtained with prediction are due to the speculative execution of instructions on the correctly predicted path. However, predicting only a single branch limits speculation to one basic block. Some researchers have proposed to predict multiple branches per cycle [23], though the accuracy of these mechanisms falls off sharply when applied to all branches in the program.

It is well known that most programs tend to spend the majority of their execution time in a small portion of their code (commonly known at the 90/10 rule). Therefore, it is important to carefully optimize the execution of these frequently executed regions. Most of this code is located within loop constructs.

In this work, we propose and evaluate mechanisms for predicting the execution of loops based on their dynamic characteristics [4]. The proposed design extensions can be applied to a superscalar architecture or an embedded processor without modifying the underlying instruction set.

Our proposed microarchitecture features are evaluated for network, telecommunication, and general purpose applications. One goal of this study is to establish the potential impact of our ideas across different application-specific domains. The selected application possess differences in:

- the number of loops,

- the maximum loop nesting,
- varying control structures within loops.

Table 1 shows the programs selected from mediabench [15], mibench [16], and the SPECint2000 [25]benchmark suites.

Table 2 presents a number of branch characteristics for these applications, including:

- the number of loop bodies,
- average basic block size (for the entire application),
- percent of dynamic loops terminating with backward conditional branches,
- percent of dynamic loops terminating with backward unconditional branches, and
- maximum loop nest depth.

1.1 Loop Terminology

Before we begin this discussion, it is important to define some terms that will be used throughout this paper.

- *loop head* - the first instruction in the loop
- *loop tail* - the branch to a negative displacement that marks the end of the loop
- *loop body* - the instructions within the loop
- *loop iteration* - one execution of a path through the loop body, starting from the loop head and ending at the loop tail
- *loop visit* - entering a loop, executing some number of loop iterations and then exiting the loop body
- *path-to-loop* - pattern of conditional branch outcomes leading up to a loop body
- *path-in-iteration* - pattern of conditional branch outcomes in a single loop iteration
- *path-in-loop* - pattern of conditional branch outcomes in the loop body

We will utilize these terms to help explain how we can exploit the path-to-loop behavior to predict an entire loop visit execution.

From previous work [4] we observed that since loops iterate a number of times during each visit, and that during each iteration we can observe repeatable patterns, so we believe we should try predict an entire loop visit's execution using past history.

Although many loops exhibit an easily predictable behavior during execution, a significant number involve more complex patterns. The complexity of loop behavior is dependent upon the input to the program, the variables being evaluated, the number and type of control flow instructions within the loop body, and the size of the loop body [4].

When a loop is visited, different paths can be followed through the loop body. If collect the outcome of a set of n branch executions, a *path* of length n is formed. If a loop iterates, a sequence of repeated paths can be formed. Likewise, the pattern of branches leading up to a loop body (i.e., the path-to-loop) can be monitored in a similar fashion. Predicting the patterns present in the path-to-loop and the path-in-loop branches provides an opportunity for prefetching instructions from basic blocks that are linked to the branches in the predicted path. Correlating path-to-loop patterns with path-in-loop patterns can lead to entire loop prediction.

If a loop body does not contain any conditional branches, loop prediction can be trivial. If the loop body exhibits a complex behavior containing multiple branches within its loop body, accurate loop execution prediction becomes more difficult. Fortunately, conditional branch behavior tends to be correlated [19]. With this in mind, we propose to predict entire loop executions based on correlating path-to-loop branch behavior with path-in-loop behavior.

2 Related work

The identification of delays caused by the computation of branch addresses was first observed in the 1960s and formally discussed at the beginning of the 1970s in [20]. Later, at the beginning of the 1980s a study of branch strategies was presented in [24]. The first study that showed dynamic characteristics of loops was presented by Kobayashi [14]. A more recent study presented a thorough examination of the dynamic characteristics of loops [5].

Since then, branch prediction has drawn a lot of attention and several mechanisms have been proposed [2, 6, 7, 9, 11, 12, 13]. Some alternative mechanisms include the branch address cache [28], trace cache [21] and control flow prediction with a tree-like predictor [8] .

Seznec et al. have proposed predicting several branches in the same cycle to produce larger traces of instructions using multiple-block ahead prediction [23].

Table 1. Benchmarks information.

Suite	Name	Inputs	Dyn Instrs	Description
mibench	dijkstra	input.dat	254M	Dijkstra' shortest path algorithm
mibench	CRC32	large.pcm	692M	Cyclic Redundancy Check 32
mibench	patricia	large.udp	481M	A Patricia trie data structure
mibench	FFT	large.pcm	249M	Fast Fourier Transform computation
mibench	typeset	large.lout	84M	Typesetting tool with a front-end processor
mediabench	epic	test.image.pgm.E	10.8M	Experimental image data compression utility
mediabench	g721	clinton.pcm	420M	G.711, G.721 and G.723 voice compressions
mediabench	jpeg	testimg.ppm	16.5M	JPEG image compression and decompression
spec2000	gzip	smred.source	2.1B	Gzip data compression algorithm
spec2000	gcc	smred.c-iterate.i	134MB	Gcc compiler
spec2000	bzip2	lgred.source	2.1B	Bzip2 data compression algorithm
spec2000	parser	smred.in	449M	Word processor
spec2000	twolf	test	500M	Place and Route Simulator

Table 2. Branch characteristics for the benchmarks.

Benchmark	Number of Loops	Average Basic Block Size	Percent of Loops Terminating with BkwdCond	Percent of Loops Terminating with BkwdUncond	Max. NL
gzip	112	10.00	54.46	45.54	4
gcc	1219	6.35	8.12	91.88	7
bzip2	75	12.19	29.33	70.67	6
parser	306	6.06	11.11	88.89	6
twolf	241	7.93	19.50	80.50	4
CRC32	21	8.66	61.90	38.10	1
FFT	39	10.19	43.59	56.41	2
Dijkstra	39	5.37	71.79	28.21	3
epic	76	9.48	80.26	19.74	2
g721	26	8.53	84.62	15.38	2
jpeg	102	18.58	92.16	7.84	5
patricia	62	5.54	74.19	25.81	3
typeset	80	7.09	81.25	18.75	2

In multiple-block ahead, two contiguous basic blocks are predicted every clock cycle.

In our proposed mechanism we emphasize the importance of prefetching instructions contained in loops. We consider fetching a large number of basic blocks (potentially thousands).

Most well known and aggressive fetch mechanisms have one goal in common: predicting multiple branches in a single cycle. The originality in our work lines in our ability to exploit locality and correlation found in loops. Although previous mechanisms have exploited program locality, they have failed to consider how to correlate path-to-loop patterns with path-in-loop patterns.

Compiler research has also looked at exploiting data locality in loops [17, 3]. These studies have concluded that extracting data locality from loops can significantly improve performance. However, there is a large number of loops that cannot be optimized by the compiler and which should be handled in hardware. An interesting mechanism would be one that both handled instruction fetch and data fetch.

Modern DSP processors have investigated including a loop cache to implement hardware loop unrolling for well-structured loops with a maximum nesting level of four [18, 22, 26]. To manage loops, the DSP processors have incorporated special loop control instructions. The main limitation with these mechanisms is that the size of the loop buffer is many times too small. In the field of embedded computing hardware, loop caching is becoming an accepted technique [1].

In multi-threaded processors, loop locality can be exploited by executing several independent iterations of a loop on different threads [10, 27]. The techniques described by Tubella and Gonzalez for dynamic loop detection are used in our work.

The rest of this paper is organized as follows. Section 3 describes loop prediction mechanisms to accurately predict the number of loop iterations per visit. Section 4 describes the mechanism utilized to predict an entire loop execution. Section 5 presents motivating results that establish the repeatability of path-to-loop

patterns as well as path-in-loops patterns, and then shows results for entire path visit prediction. Section 6 summarizes the paper and discusses future directions for the work.

3 Predicting the number of loop iterations

To determine the predictability of iteration frequency per loop visit, we studied three prediction mechanisms:

- Last value - the last number of iterations during a loop visit is used to predict the number of iterations on the next loop visit
- Stride - the difference between the number of iterations for the last two loop visits is maintained. This number is then added to the last interation count to predict the future number of interations.
- Freq - maintains a set of frequency counters that record counts for the last n loop visits. The predicted loop iteration value is the entry with the highest frequency count.

Figure 1 shows how accurately we can predict the number of loop iterations per loop visit for the studied benchmarks. In general, the number of iterations is highly predictable using any of the 3 predictors, with Last Value slightly outperforming the other two. We were somewhat surprised that Stride did not give us an edge since we had observed strided behavior in past work [5]. A confidence mechanism could be used to improve this predictor, though Last Value seems to capture the number of iterations accurately. The results for the frequency predictor for CRC32 are due to this program not having enough branch information to adequately train the predictor.

4 Loop prediction hardware

To accurately predict the entire execution of a loop it is important to be able to accurately predict when a loop visit will occur and what path-in-loop will be followed. To capture the correlation between the path-to-loop and path behavior in the loop body, we maintain a shift register that holds the outcome of the last n branch outcomes. When the loop body is fetched, we use past path-to-loop history to predict whether the corresponding loop tail branch will be taken. The loop stack is used to track whether we are entering a loop, versus whether we are iterating in a loop body. More importantly, the loop stack captures the history for the current loop visit. When the loop iterates, the address for this branch (i.e., the loop tail) and its target (i.e., the loop head) are pushed onto a *loop stack*. A path-to-loop is detected when when we are about to fetch the loop body from memory and no information for the present loop head is found in the stack. We include the loop tail in the stack to be able update the proper loop table entry later.

A loop can be detected early in fetch. If the loop is predicted to iterate, then the loop is pushed onto the stack. Before the push is performed, the stack is interrogated for information on this loop. If information for this loop is found in the stack, then the loop is iterating; otherwise, the loop is new and should be pushed on the stack. When a loop completes all of its iterations (i.e., completes the loop visit), the loop stack communicates the loop visit information to the loop table. The information of that table is then used for predictions on future visits.

Our loop stack and loop table are both 2-level tables. The loop stack is indexed using the loop tail branch address. The configuration used in the results presented contains an 8-entry first level stack, which indexes to a second level table containing 64 entries per every first level stack entry. The stack provides 8 entries to handle loop nesting efficiently (though this number probably could be larger for some programs). The second level table contains 64 16-bit patterns. The 64 entries capture the sequence of up to 64 patterns that may occur per loop visit. A iteration field is provided that captures the number of times that any particular pattern repeats.

The loop table has a similar structure, but contains 2048 first level table entries and is indexed using an XOR of the 16-bit pattern register and the loop tail branch address. The loop table is only updated after the loop stack has recorded the history of the entire loop visit. The loop table is accessed whenever we detect we are about to visit a loop. A tag check is performed on the first level table access in the loop table to insure we have indexed into the proper entry.

On the next visit to this loop, the predicted path is compared against the actual path traversed. If we predicted correctly, an associated 2-bit counter is increment, denoting that we have confidence in this prediction. In the case where there is a mismatch, the counter is decremented, and the new path is entered in the table. If there is no room in the table for replacements, we utilize the confidence values to select an entry to replace. Figure 2 shows the organization of a loop stack and a loop table. An entire loop prediction is correct when the predicted number of iterations of

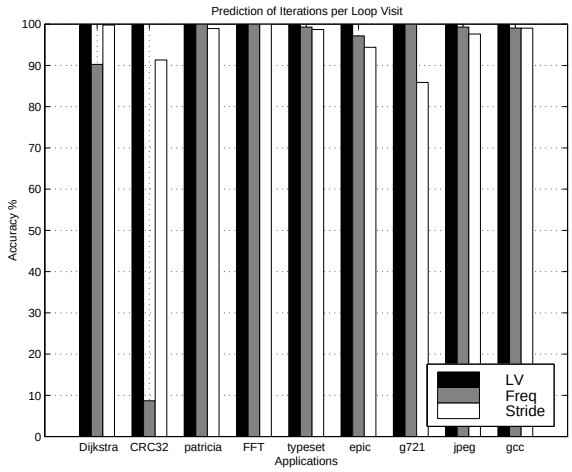

Figure 1. Prediction accuracy of the number of iterations per loop vist, comparing 3 different predictors.

the loop is correct, the predicted path-to-loop is correct and every one of the predicted path-in-iteration is correct. Next, we describe the set of experiments we perform to determine the correlation between paths-to-loop and paths-in-loops.

5 Predicting loop executions

To predict the entire execution of a loop, we need to clearly understand both path-to-loop behavior and path-in-loop behavior. A mechanism to implement whole loop visit execution must predict the number of times a loop will iterate and the path to be followed on each iteration. In this work, predicting an entire loop visit execution is based on the ideas already described in Sections 3 and 4 and on the earlier design of the *self-correlating loop predictor* proposed in [4].

5.1 Simulation environment and benchmarks

We used the ATOM tool to obtain the branch profile data presented in Tables 1 and 2. We also use the Simplescalar toolset to study our loop prediction mechanisms. We added our loop prediction table and loop stack to the Alpha EV6 version of the Simplescalar 3.0b of the toolset. All the benchmarks used here were compiled using the compiler DEC C V5.2-033 on Digital UNIX V4.0 (Rev. 564) with the compilation flags: *-non_shared -02.*

5.2 Prediction of path-to-loop and path-in-iteration

Figure 3 shows the predictability of entering a loop, given the pattern of branch outcomes (i.e., the path-to-loop) leading to the loop. The accuracy of path-to-loop prediction indicates the precision of determining in advance whether a loop will be visited. For most of the applications the prediction accuracy is 30% to 40%. The prediction of a path-to-loop is highly dependent on capturing the right length of the correlation present in the path-to-loop. A number of correlated branch prediction studies have come to the same conclusion. We look to leverage their results in future work. Figure 4 shows the accuracy for path-in-loop prediction. Here we measure the prediction accuracy of predicting all the path-in-loop patterns during a loop visit, assuming we know that we have entered the loop. So we are computing this on a complete loop visit basis. After we know that we have entered a loop, the path of branches to be traversed during each iteration is predicted. The prediction accuracy measured in this case is dependent upon the branches visited within the loop body. The high prediction accuracy obtained for CRC32 and FFT indicates that their loops did not not contain many hard to predict branches. This result can also be attributed to the fact that both applications contain a small number of loops with a maximum nesting levels of 1 and 2, respectively. Figure 5 shows the accuracy of predicting path-in-loop per loop iteration. Here, at every iteration we evaluate whether the predicted path-in-loop is correct. Again, we assume we

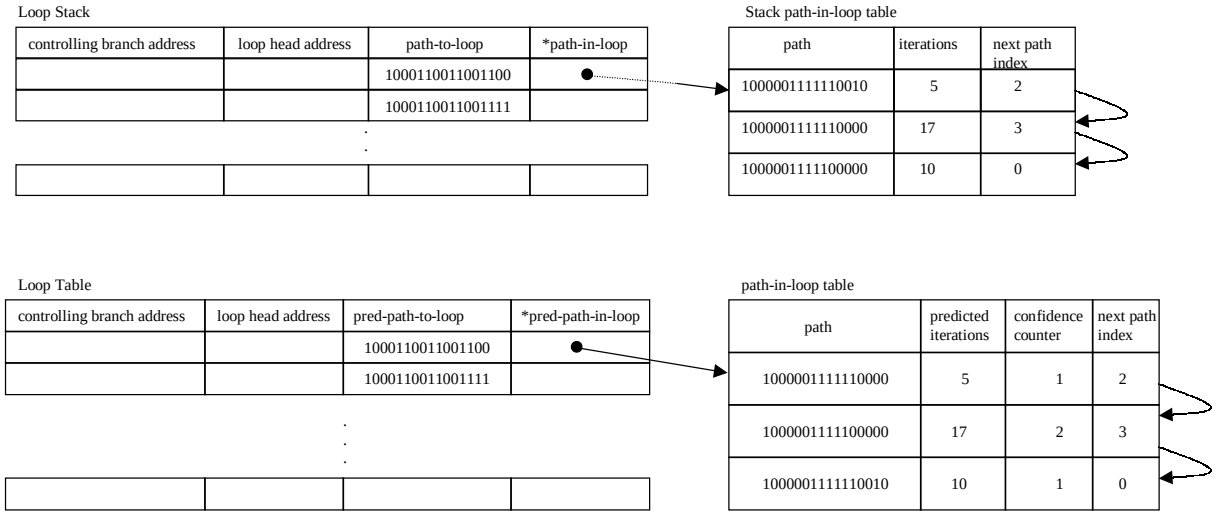

Figure 2. Loop stack and loop table organization. Both mechanisms utilize a 2-level design.

have already entered the loop and are now investigating the predictability of the sequence of the patterns (i.e., paths-in-loop) generated during each iteration. In Figure 5 we compute this on a per iteration basis, whereas in Figure 4 we computed this on a per loop visit basis (i.e., all the paths had to be correct). As expected, every bar in Figure 5 is as high as or higher than the corresponding bar in Figure 4. What is interesting is that for most of the applications, we can predict the exact pattern to be followed in the next iteration more than 50% of the time. For 4 of the applications, we could predict the path-in-iteration correctly greater than 80% of the time. So if we can improve our ability to predict when we enter the loop, we should be able to predict a large number (most) of the iterations correctly. In Figure 6 we show the prediction accuracy of predicting the number of iterations to be performed for a given path-in-iteration. This shows how accurately we can predict the number of iterations a predicted path-in-iteration will perform. To compute this, after a loop visit is complete, we compare the entries in the second level tables for the loop stack and the loop table. To produce Figure 6, we find the percent of entries that match. We increase the number of correct predictions by the number of entries that are the same for each loop visit. In all benchmarks we get over 99% correct prediction rates. This should help us considerably when performing complete loop prediction. While we expected to obtain high prediction accuracy, we did not expect to get prediction accuracy this high.

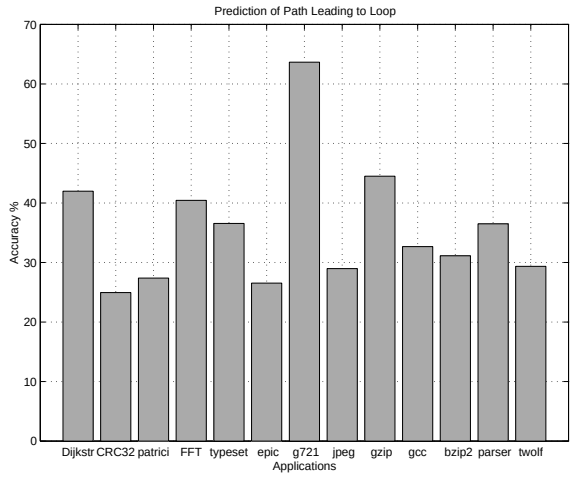

Figure 3. Prediction of entering the loop using the path-to-loop.

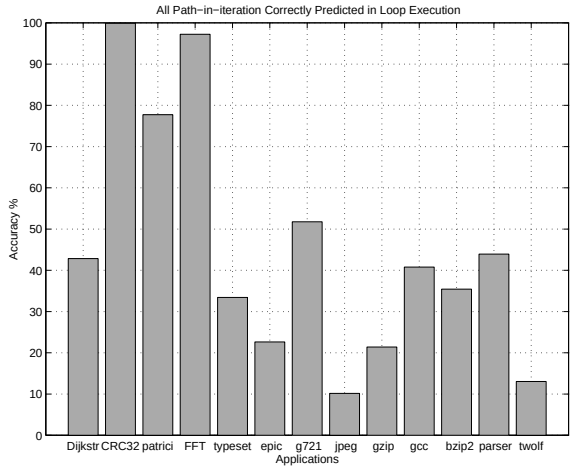

Figure 4. Prediction of path-in-iteration per loop visit.

5.3 Prediction of entire loop execution

The main goal of the previous section was to look at the individual components that will allow us to predict the execution of a complete loop visit. Next we show the results of predicting the entire execution of any loop in the program. This is, determining in advance if a loop will be entered and predicting the exact path that will be followed within the loop body during the entire vist. Equipped with this information, an aggressive prefetching mechanism can be used to fetch a large number of instructions in a single cycle. For example, assume that we use all 16 bits of path information, so then instructions from the next 16 basic blocks can be prefetched in the next cycle. After the 16 branch outcomes are committed, we can then know whether to commit the instructions from these 16 basic blocks. If we are correct a majority of the time, large performance benefits are possible.

Figure 7 shows the results of predicting an entire loop execution. We are predicting:

1. if we enter the loop,
2. the pattern of branches per each loop iteration, and
3. the number of iterations of each path-in-iteration.

Approximately 20-25% of the total loop visits can be predicted with 100% accuracy. This is encouraging

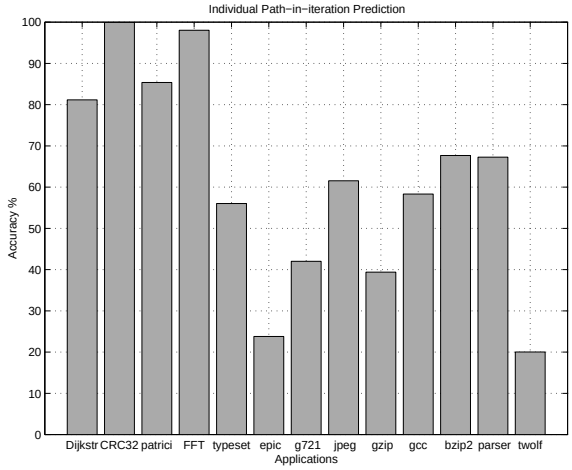

Figure 5. Prediction of path-in-iteration per loop iteration.

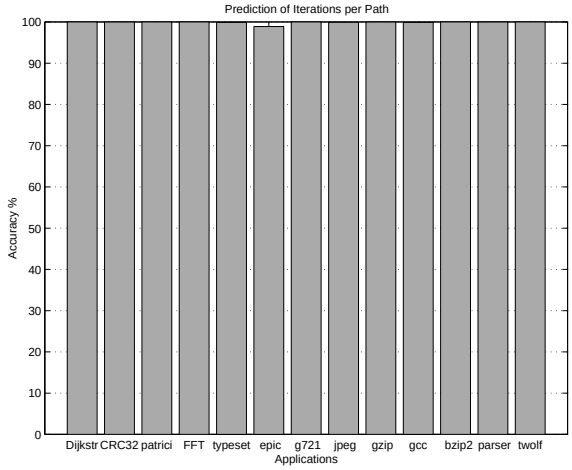

Figure 6. Prediction of number of iterations for path-in-iteration.

data, especially when we consider the results of Figure 5 which showed us that the accuracy of predicting partial loop visits can be high. Also note that we do quite well for the mibench and most of the mediabench applications, though perform much less favorably for the SPEC2000 benchmarks. This should be expected, though presents us with a challenge to address in future work.

5.4 Discussion

It was observed that applications that contain loops possessing a smaller number of control flow instructions within their bodies, smaller loop body sizes, and smaller nesting values tended to have higher accuracy for loop execution prediction. It was also observed that the number of traversed paths per loop visit varies across the benchmarks. However, for a hardware implementation, only a fixed number can be stored. In the experiments reported here, the number of recorded paths per loop visit was 64.

6 Summary

In this work we investigated our ability to unroll entire loop executions at runtime. We feel that our results are extremely promising. Our next challenge is to modify our SimpleScalar model to allow for a very large instruction window and a very wide instruction issue width, so that we can begin to exploit the instruction

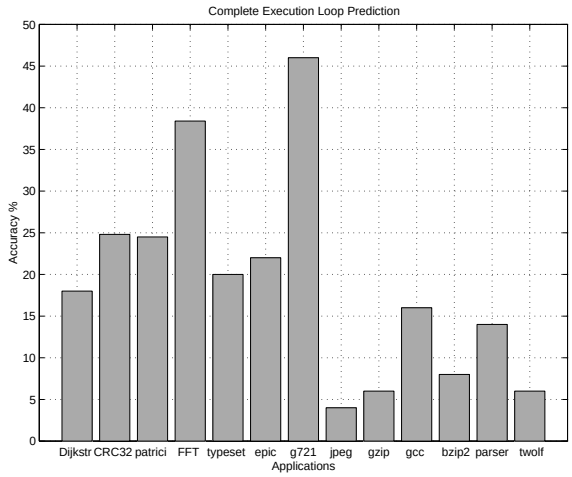

Figure 7. Loop Execution Prediction.

level parallelism that is provided by this mechanism.

While we have presented only a single design point in the overall design space, we have already explored over 256 different implementations. We plan to evaluate the chip real estate devoted to an actual implementation of our loop stack and loop table in future work.

Acknowledgements
Marcos de Alba is an associate professor of the Department of Electronic Engineering at the Universidad Autonoma Metropolitana Azcapotzalco. He has been able to pursue graduate studies thanks to the support of his advisor and Fulbright-Conacyt.

References

[1] F. Vahid. A. Gordon-Ross, S. Cotterell. Exploiting fixed programs in embedded systems: A loop cache example. In *Computer Architecture Letters*. IEEE Computer Society, January 2002.

[2] P-Y. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 274–283, Denver, CO, June 1997.

[3] 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.

[4] M. R. de Alba and D. R. Kaeli. Runtime predictability of loops. In *Fourth IEEE Workshop on Workload Characterization*, pages 91–98, Austin, TX, December 2001.

[5] M. R. de Alba, D. R. Kaeli, and E. S. Kim. Analisis dinamico de bloques iterativos. In *Tercer Congreso Internacional en Control, Instrumentacion Virtual y Sistemas Digitales.*, pages 93–106, Ciudad de Mexico, Agosto 27-31 2001.

[6] K. Driesen and U. Hlzle. Accurate indirect branch prediction. In *Proc. of the 25th International Symposium on Computer Architecture*, pages 167–178, Barcelona, Spain, July 1998.

[7] K. Driesen and U. Holzle. The cascaded predictor: Economic and adaptive branch target prediction. In *Proc. of the 31st International Symposium on Microarchitecture*, pages 249–258, Dallas, TX, November 1998.

[8] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proc. of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–263, 1995.

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

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

[11] J. T. Coffey I. K. Chen and T. Mudge. Analysis of branch prediction via data compression. In *Proc. of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, Cambridge, MA, October, 1996.

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

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

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

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *30th International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, North Carolina, 1997.

[16] J. S. Ringenberg M. R. Guthaus, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Fourth IEEE International Workshop on Workload Characterization*, pages 3–14. University of Michigan, December 2001.

[17] 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.

[18] Motorola. *SC140 DSP Core Reference Manual*, April 2001.

[19] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proc. of the fifth International Conference on Architectural Support for Programming Languages and Operating System*, volume 27-9, pages 76–84, New York, NY, 1992. ACM Press.

[20] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, 1972.

[21] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proc. of the 29th International Symposium on Microarchitecture*, pages 24–35, Paris, France, 1996.

[22] S. Sair and D. Kaeli. A study of loop unrolling for vliw-based dsp processor. In *Proc. of the Workshop on Signal Processing Systems*, pages 519–527, Cambridge, MA, October 1998.

[23] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *Proc. of the seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, Cambridge, MA, 1996.

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

[25] SPEC. Spec cpu 2000 benchmark set program. http://www.spec.org/osg/cpu2000/.

[26] Texas Instruments. *TMS320C62xx DSP Family.*

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

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