Long-Latency Branches: How Much Do They Matter?

Abhas Kumar, Nisheet Jain, Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur 208016
INDIA
Email: abhask, nisheet, mainakc@cse.iitk.ac.in

***Abstract*— Dynamic branch prediction plays a key role in delivering high performance in the modern microprocessors. The cycles between the prediction of a branch and its execution constitute the branch misprediction penalty because a misprediction can be detected only after the branch executes. Branch misprediction penalty depends not only on the depth of the pipeline, but also on the availability of branch operands. Fetched branches belonging to the dependence chains of loads that miss in the L1 data cache exhibit very high misprediction penalty due to the delay in the execution resulting from unavailability of operands. We call these the *long-latency branches*. It has been speculated that predicting such branches accurately or identifying such mispredicted branches before they execute would be beneficial. In this paper, we show that in a traditional pipeline the frequency of mispredicted long-latency branches is extremely small. Therefore, predicting all these branches correctly does not offer any performance improvement. Architectures that allow checkpoint-assisted speculative load retirement fetch a large number of branches belonging to the dependence chains of the speculatively retired loads. Accurate prediction of these branches is extremely important for staying on the correct path. We show that even if *all* the branches belonging to the dependence chains of the loads that miss in the L1 data cache are predicted correctly, only four applications out of twelve control speculation-sensitive applications selected from the SPECInt2000 and BioBench suites exhibit visible performance improvement. This is an upper bound on the achievable performance improvement in these architectures. This article concludes that it may not be worth designing specialized hardware to improve the prediction accuracy of the long-latency branches.**

I. Introduction

Dynamic control speculation, or more traditionally known as branch prediction, is one of the most important factors controlling the end-performance delivered by a microprocessor. Since instructions execute late in the pipeline, it is necessary for the fetcher to speculate about the path that would be taken by a branch instruction immediately after it is fetched. Of course, a prediction can be made at any point between the time a branch is fetched and the time it executes. However, an early correct prediction avoids excessive number of bubbles in the pipeline. The number of cycles between the time a prediction is made and the time the branch executes is known as the branch misprediction penalty because, if the branch is mispredicted, the instructions fetched into the pipeline during this time are on the wrong path and must be drained out of the pipe without contributing anything to the progress of the execution. For example, in a microprocessor, if the branch misprediction penalty is n cycles and the fetch width is k, the number of useless instructions fetched because of a branch misprediction is nk. As expected, for deep pipelines, such as the one in the Intel Pentium 4 [6], where n is large, branch prediction accuracy becomes extremely important. The situation worsens if a mispredicted branch cannot execute as early as possible due to unavailability of operands. The misprediction penalty reaches a maximum when the branch belongs to the dependence chain of a load instruction that misses in the L2 cache. It is most likely that by the time the branch executes and the misprediction is detected, the active list (or the re-order buffer) would be full with wrong path instructions. It has been speculated that these branches could hurt performance [10], [20], [21] and therefore, special predictors should be designed to improve the prediction accuracy of such branches.

In this paper we explore the performance impact of branches that not only belong to the dependence chain of a load missing in the L2 cache, but also belong to the dependence chain of a load missing in the L1 data cache. Of course, the loads missing in the L2 cache are only a subset of the loads that miss in the L1 data cache (we consider architectures with inclusion in the cache hierarchy). We call the branches belonging to the dependence chains of the loads missing in the L1 data cache the long-latency branches. We find that such branches have near-zero impact on performance in traditional pipelines with state-of-the-art branch predictors because only few of them are mispredicted. Also, due to a finite size of the active list not too many branches belonging to the dependence chain of a missing load can be fetched in the pipe while the load is outstanding. For architectures, such as Runahead execution [15] and CLEAR [12], which allow checkpoint-assisted early load retirement, many more branches belonging to the dependence chain can be brought into the pipeline. We estimate an upper bound on performance improvement that can be achieved if *all* the branches belonging to the dependence chain of every load taking a L1 data cache miss could be predicted accurately. Our simulation results show that only one application out of twelve control speculation-sensitive applications selected from the SPECInt2000 and BioBench suites shows a 13% improvement in performance. Three other applications achieve an improvement in the range 5.6% to 7.1%. The remaining eight applications are largely unaffected.

In the next section we briefly discuss some relevant work. In Section II we detail our simulation environment. Section III discusses the impact of long-latency branches on traditional pipelines while Section IV presents an overview of the Runahead and the CLEAR architectures and establishes,

through simulations, an upper bound on the achievable performance improvement in these architectures, if all long-latency branches are predicted correctly. In Section V we conclude.

A. Related Work

In this paper we explore the impact of branches belonging to the dependence chains of loads missing in the L1 data cache. If the values of these loads could be predicted, the dependent branches could be executed speculatively based on the predicted value. Some work has been done in exploiting general operand value prediction to improve control speculation [2], [5], [7]. Selective reversal of branch predictions by discovering correlations with predicted operand values is explored in [2]. Early computation of branches through operand value prediction is presented in [5]. The authors also explore a hybrid scheme by combining a conventional branch predictor (gshare) and a branch execution unit taking inputs from an operand value predictor. The branch difference predictor introduced in [7], correlates branch outcomes with the history of differences between the branch operand values. A rare event predictor remembers the abnormally behaving history patterns in a cache-like structure so that interference in the main predictor can be reduced. In contrast to these studies, in this paper we explore the performance impact of the long-latency branches.

Wrong path events such as null pointer dereferences, unaligned accesses, access permission violations, TLB misses, illegal opcodes, division by zero can be used as a hint of branch misprediction and an early reversal of prediction can be initiated [3]. For long-latency branches, such a scheme can be extremely useful in reducing the high misprediction penalty, provided some exceptional event does happen on the wrong path. Our simulation results show that mispredicted long-latency branches are extremely small in number.

II. Simulation Environment

This section presents the details of the simulator and the applications we have used to evaluate the impact of long-latency branches. Table I presents the parameters of the MIPS ISA-based simulated architecture. We simulate an aggressively clocked deep pipeline so that we can gauge the maximum benefit that the processor might enjoy if the long-latency branches could be correctly predicted. The fetcher accesses the BTB for every instruction in the fetch stage and proceeds with the BTB outcome, which only tells the fetcher about the fate of a branch the last time around. A BTB miss forces the fetcher to continue sequentially (i.e. the BTB outcome in such cases is the fall-through PC). After an instruction is decoded, the conditional branch predictor is looked up, if the instruction is a branch. The predicted outcome overrides the BTB outcome, if two predictions do not agree. In this case, all the instructions fetched from the BTB target are marked squashed and the fetch PC is steered according to the conditional branch prediction. Our simulated pipeline assumes the existence of an adder in the decode stage to compute the branch target from the instruction's PC-relative offset. This is used in conjunction with the prediction to steer the fetch PC. We always update the global history register speculatively as in the Alpha 21264 [11] and recover it from branch stack checkpoint in case of a branch misprediction. A branch misprediction is detected after the branch executes which is at least 31 cycles from the time it is fetched. The register map is recovered from the branch stack immediately and fetching begins from the correct path in the next cycle. We allow at most 48 outstanding non-committed branches at any point in time (this is one-fourth of the maximum number of in-flight instructions).

TABLE I
Simulated architecture

Attribute	Value
Frequency	4 GHz
Pipe stages	38
Front-end/Commit width	4/8
BTB	256 sets, 4-way
Branch predictor	several (see Table II)
RAS	32 entries
Branch misprediction penalty	31 cycles (minimum)
Active list	192 entries
Branch stack	48 entries
Integer/FP register	224/224
Integer/FP queue	48/48 entries
Unified load/store queue	64 entries
ALU	8 (two for addr. calc.)
Integer mult./div. latency	6/35 cycles
FPU	3
FP multiplication latency	2 cycles
FP division latency	12 (SP)/19 (DP) cycles
ITLB, DTLB	128/fully assoc./LRU
Page size	4 KB
L1 instruction cache	32 KB/64B/4-way/LRU
L1 data cache	32 KB/32B/4-way/LRU
Unified L2 cache	2 MB/128B/8-way/LRU
MSHR	16+1 for retiring stores
Store buffer	32
L1 cache hit	6 cycles
L2 cache hit	25 cycles (round trip)
Memory system	
Memory controller frequency	2 GHz
System bus width	64 bits
System bus frequency	2 GHz
SDRAM access time	80 ns (row buffer miss) 40 ns (row buffer hit)
Number of banks	16
SDRAM bandwidth	6.4 GB/s

The memory system is assumed to have an on-die integrated memory controller clocked at half the speed of the main core. The 16-way banked SDRAM module is clocked at 800 MHz, and can transfer 64 bits to the memory controller on both the clock edges (DDR), thereby achieving an aggregate bandwidth of 6.4 GB/s. A 32-bit physical address is divided into the following parts and decoded accordingly by the memory controller. The least significant 3 bits are offset into an 8-byte column, the next 12 bits are used as the column number, the next 4 bits are the bank address for the simulated 16 banks, the next 13 bits are the row address. The mapping of address bits to row, column, and bank is similar to the one suggested in [16]. To avoid bank conflicts between a miss request and a

writeback originating from the eviction due to the same miss, the bank number is calculated by XORing the bits [18:15] with bits [21:18] of the address, the latter being the least significant four bits of the L2 tag [22].

The simulated branch predictors are detailed in Table II. We present them for completeness, but we use only the 2Bc-gskew predictor for the major portion of this study (see next section for details). We do not consider over-riding branch predictors, such as the prophet-critic predictor [4], in this study. For this study we fixed the branch predictor storage budget to 16 KB. Accordingly, we scale up the original Alpha 21264 Tournament predictor [11]. While computing the configuration of the path-perceptron predictor, we had to take into account the non-negligible checkpoint storage it requires to record all the in-flight partial sums. We found that this takes up about 2.5 KB of storage for the configuration shown in Table II. Note that the 2Bc-gskew predictor is optimized with different history lengths for different components as suggested in [17]. We find that this optimization has a significant impact on performance.

TABLE II
SIMULATED BRANCH PREDICTORS

Predictor	Confi guration
Gshare [13]	16-bit global history, 2-bit saturating counters
Tournament [11]	4096-entry SAg, 12-bit local history, 3-bit saturating counters; 14-bit global history, 2-bit saturating counters; 16384-entry chooser, 2-bit saturating counters
Perceptron [9]	46-bit global history (excluding bias bit), 8-bit 2's complement weights, 348 perceptrons
Path-perceptron [8]	33-bit global history (excluding bias bit), 8-bit 2's complement weights, 397 perceptrons
2Bc-gskew [14], [17], [18]	16384-entry BIM, G0, G1, META, 2-bit each, 6-bit global history for BIM, 24-bit global history for G0, 48-bit global history for G1, 6-bit global history for META

In this study we use eleven applications from the SPEC2000 integer suite, eight from the SPEC2000 floating-point suite, and three from the BioBench suite [1]. Our simulator currently cannot handle `252.eon`, `177.mesa`, `178.galgel`, `187.facerec`, `189.lucas`, `191.fma3d`, and `200.sixtrack` from the SPEC2000 suite. The selected SPEC2000 applications are simulated with the `ref` input sets. We have selected `002.tiger`, `003.clustalw`, and `protdist` of `006.phylip` from the BioBench suite. These applications are run with inputs `sitchensis.fa`, `tufa420.seq`, and `tufa420.phy`, respectively. All the applications are run for one billion representative instructions as determined by SimPoint [19], when fed with the basic block vectors for intervals of length one billion instructions.

III. CHARACTERIZING THE LONG-LATENCY BRANCHES

In this section we present the results of our simulations showing the characteristics of the long-latency branches. First, we pick one of the five branch predictors discussed in Table II. Next, we prune the set of the applications based on their sensitivity to control prediction. Figure 1 presents the misprediction rates of the predictors for all the twenty three applications. For each application we show the misprediction rates for gshare, Tournament, perceptron, path-perceptron, and 2Bc-gskew predictors, from left to right in that order. The integer applications (CINT) are grouped towards the left, the floating-point applications (CFP) appear in the middle, and the BioBench applications are towards the right of the bar chart. For each of the CINT, CFP, and BioBench groups, we also show the average misprediction rates of the predictors. For CINT, 2Bc-gskew is the clear winner achieving an average misprediction rate of slightly over 4%. Also, for individual integer applications, 2Bc-gskew secures the first place in all but `gzip`, where gshare gets a slight edge over 2Bc-gskew. For CFP, perceptron and path-perceptron are tied at the first place while for BioBench, gshare and 2Bc-gskew are tied at the first place. Since the floating-point applications are not much sensitive to control speculation, we select the 2Bc-gskew predictor for further study.

The next step is to select the interesting applications and focus on them. Table III shows the IPC of all the twenty three applications with the 2Bc-gskew predictor as well as a perfect predictor. The left half of the table shows the performance of the integer applications while the right half shows that of the floating-point and the BioBench applications. As expected, the floating-point applications are least sensitive to control speculation and a 2Bc-gskew predictor achieves IPC equivalent to the perfect predictor. Only `wupwise`, `applu`, and `ammp` show small gap in performance between the two predictors. Also, `phylip` from BioBench falls in the same category. Among the integer applications only `mcf` does not show any performance improvement with the perfect predictor. Based on these results, we pick the remaining ten integer applications, and `tiger` and `clustalw` from BioBench for further inspection.

Figure 2 presents the distribution of the selected twelve applications based on how sensitive they are to control speculation and how much opportunity they offer in terms of correctly predicting the long-latency branches. On the horizontal axis we plot the IPC difference between a perfect predictor and the 2Bc-gskew predictor. On the vertical axis we plot the percentage of mispredicted branches that can be categorized as long-latency with the 2Bc-gskew predictor. A branch has a long latency if it belongs to the dependence chain of a load that misses in the L1 data cache. Note that the branches that enter the pipe after the data cache refill completes cannot be categorized as long-latency, even though they belong to the dependence chain of the load. Therefore, it is clear that this percentage has a close relationship with the amount of resources that a microprocessor has. More resources allow more instructions to be fetched while the load miss is outstanding. For our simulated pipeline (which we believe is quite aggressive in terms of resources, given the current commercial standards), we find that none of the twelve applications offer much opportunity in terms of optimizing the long-latency branches. The applications that are placed

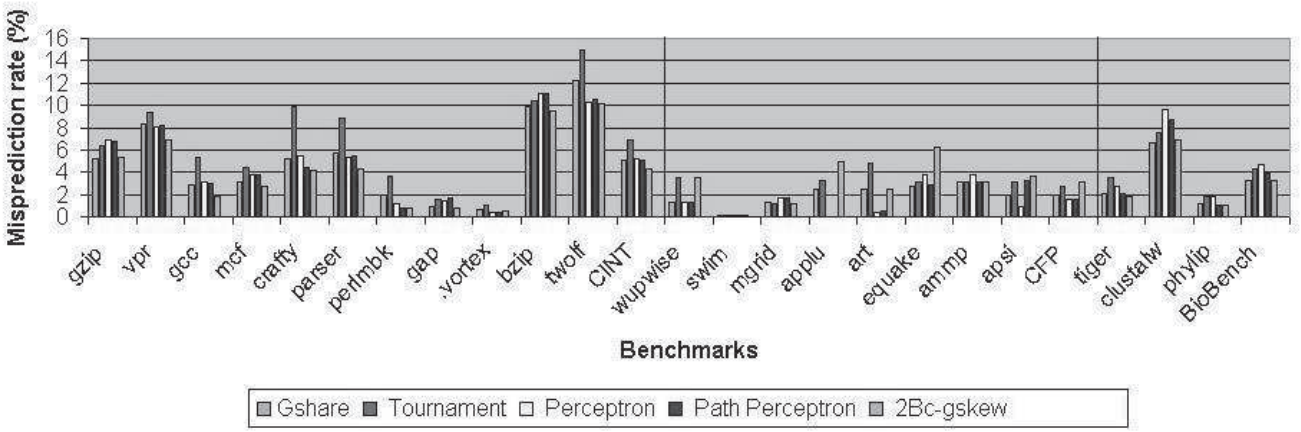

Fig. 1. Misprediction rates for various branch predictors in a 38-stage pipeline

TABLE III

IPC COMPARISON BETWEEN 2BC-GSKEW AND PERFECT PREDICTORS

CINT App.	2Bc-gskew IPC	Perfect IPC	CFP and Bio App.	2Bc-gskew IPC	Perfect IPC
164.gzip	0.46	0.70	168.wupwise	0.72	0.74
175.vpr	0.44	0.60	171.swim	0.18	0.18
176.gcc	0.49	0.70	172.mgrid	0.59	0.59
181.mcf	0.03	0.03	173.applu	0.47	0.48
186.crafty	0.63	0.99	179.art	0.12	0.12
197.parser	0.47	0.64	183.equake	0.21	0.21
253.perlbmk	0.39	1.12	188.ammp	0.42	0.44
254.gap	0.36	0.57	301.apsi	0.42	0.42
255.vortex	0.96	1.17	002.tiger	0.45	0.50
256.bzip	0.36	0.57	003.clustalw	0.65	1.10
300.twolf	0.28	0.40	006.phylip	0.83	0.84

TABLE IV

IPC WITH PERFECT PREDICTION OF THE LONG-LATENCY BRANCHES

Application	2Bc-gskew IPC	PerfectLLB IPC	Application	2Bc-gskew IPC	PerfectLLB IPC
002.tiger	0.45	0.44	197.parser	0.47	0.47
003.clustalw	0.65	0.65	253.perlbmk	0.39	0.39
164.gzip	0.46	0.46	254.gap	0.36	0.36
175.vpr	0.44	0.44	255.vortex	0.96	0.93
176.gcc	0.49	0.49	256.bzip	0.36	0.36
186.crafty	0.63	0.57	300.twolf	0.28	0.28

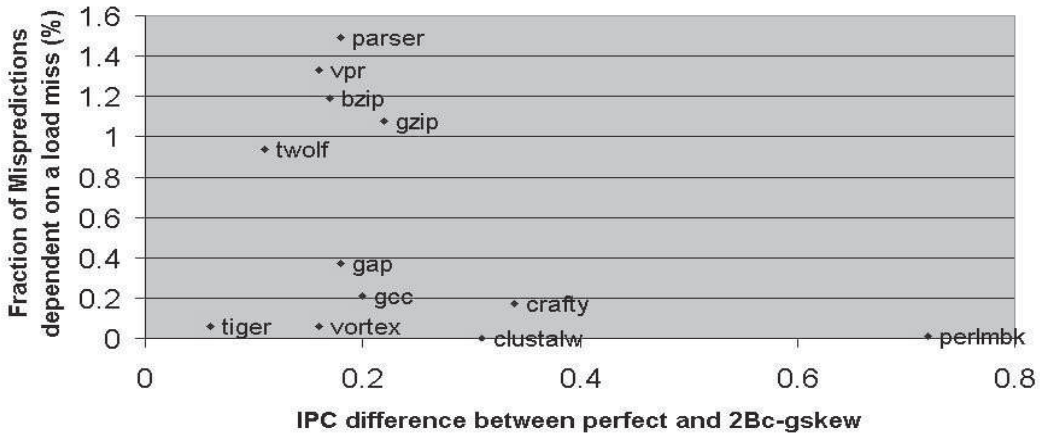

Fig. 2. Distribution of applications based on sensitivity to long-latency branches in a 38-stage pipeline

towards the upper right corner of the plot are the most promising ones because they not only have enough room for improvement in terms of control speculation, but also have a sizable percentage of mispredicted long-latency branches. However, in this case we find that `perlbmk` has the maximum IPC gap between the 2Bc-gskew and the perfect predictors, but has near-zero mispredicted long-latency branches. On the other hand, `parser` has the maximum percentage of mispredicted long-latency branches, but this is only about 1.6%. Therefore, we do not expect much performance gain even if we predicted all the long-latency branches correctly for these applications. These results are presented in Table IV. For brevity, the base IPC with the 2Bc-gskew predictor is also included along with the perfect long-latency branch (PerfectLLB) IPC.

Table IV confirms our hypothesis that long-latency branches are not critical for performance. Predicting all these branches correctly does not improve performance at all. What is surprising is that for three applications (`tiger`, `crafty`, `vortex`), the performance actually degrades after predicting all the long-latency branches correctly. Since this effect is most prominent in `crafty`, we explored it a little further. We found that the L1 instruction cache miss rate increased from 1.63% to 2.78% after correctly predicting all the long-latency branches in `crafty`. Further, 60% of these extra dynamic instructions that were missing in the cache belonged to the wrong path of some mispredicted long-latency branch in the base 2Bc-gskew predictor. However, these instructions on the alternate path were used after a while. After we eliminate all the mispredictions of the long-latency branches, these instructions are accessed less frequently and the LRU replacement policy of the instruction cache evicts these cache blocks before they are accessed on the correct path. We also pinpointed the code section where this phenomenon takes place most frequently (in the chain of `if-elseif-else` within the first `while` loop of the `Swap` function) and found that the branch behavior in that region is extremely data-dependent and erratic. In summary, the wrong path of the most frequently mispredicted long-latency branch is traversed more frequently than the correct path in `crafty`.

IV. Estimating the Impact on Runahead and Clear

In the last section we pointed out that the opportunity offered by an application in terms of correcting long-latency branches depends on how many such branches can be fetched while a missing load is outstanding. Due to the finite size of the active list in traditional architectures this number is quite limited, as shown in our results. The Runahead execution [15] and Checkpointed Load with EArly Retirement (CLEAR) [12] allow a long-latency load to be retired speculatively so that new instructions can be fetched into the pipe and the active list can continue to move smoothly. In the Runahead execution, the destination register of the load is marked "invalid" and this state propagates along the dependence chain. Any instruction receiving an "invalid" operand does not execute and proceeds to retirement immediately. However, none of these instructions are allowed to modify the architectural state. When the load refill completes, the processor flushes the pipe, rolls back to a register checkpoint taken at the time of speculatively retiring the load, and starts fetching from the load. Effectively, Runahead execution allows warming up of the pipeline structures such as the caches and the predictors while the missing load is outstanding.

CLEAR augments load-value prediction with early retirement. When a long-latency load comes to the head of the active list, a value predictor is consulted and the load is retired with the predicted value written to the destination register. Thus, with correct value prediction, CLEAR can offer significantly higher performance than Runahead execution.

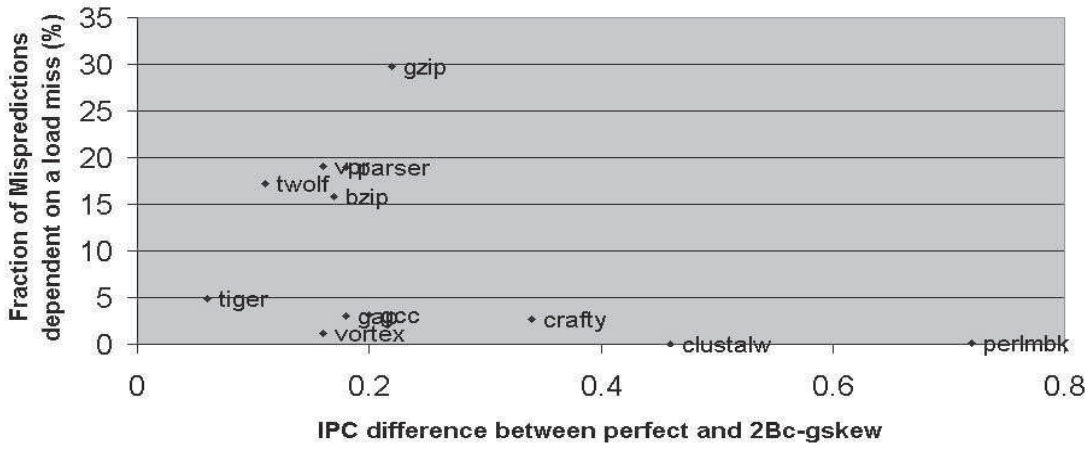

Fig. 3. Distribution of applications based on sensitivity to long-latency branches in a 38-stage pipeline with unpruned dependence tree

Still on mispredictions, it has to roll back to a checkpoint taken at the time the load was retired speculatively. However, even in that case CLEAR enjoys the prefetching effect of Runahead.

In both Runahead and CLEAR, staying on the correct path while a missing load is outstanding is extremely important. Otherwise, execution along the wrong path for such a long time would pollute the caches and the predictors. In Runahead execution this is a bigger problem because the branches dependent on long-latency loads cannot even execute and will have to rely on predictions only. The first such mispredicted branch is called the divergence point. In CLEAR a wrongly predicted value can still continue the execution along the correct path depending on the branch condition. For example, a `bgez` branch will be executed correctly for any non-negative predicted value. Nonetheless, correctly predicting all these branches dependent on the long-latency loads would be beneficial.

We gauge the maximum possible gain that can be achieved in these architectures by correctly predicting *all* the branches belonging to the dependence tree of a load that takes an L1 data cache miss. Figure 3 shows the distribution of the applications in a similar fashion as Figure 2, but the vertical axis now plots the percentage of the mispredicted branches that are on the dependence chain of loads missing in the L1 data cache. Therefore, a mispredicted branch, that comes into the pipeline even after the load has refilled, is considered for inclusion in this statistic, if it depends on the load. Thus, we consider the unpruned dependence tree with a load missing in the L1 data cache at the root to generate this plot. All mispredicted branches belonging to such a tree are considered in this plot. The simulations are carried out on the traditional pipeline without Runahead or CLEAR support, but we maintain the dependence tree of each load missing in the L1 data cache to achieve similar effects.

In Figure 3 we notice that the percentages on the y-axis are much higher now, as expected. However, applications with high IPC gap between the 2Bc-gskew and the perfect predictor still fail to offer much opportunity. From the plot we observe that `gzip`, `vpr`, `parser`, `twolf`, and `bzip` are the most promising ones. Table V presents the IPC after correctly predicting all the branches dependent on loads missing in the L1 data cache. It also shows the baseline IPC with the 2Bc-gskew predictor. Interestingly, `gzip` shows a significant improvement in IPC (13%) as predicted by Figure 3, while `vpr`, `bzip`, and `twolf` show some visible improvement in IPC (5.6% to 7.1%). Only slight performance improvement is achieved by `gap` and `vortex`. In summary, under the ideal conditions where *all* the branches, belonging to the dependence chain of a load that misses in the L1 data cache, can be predicted correctly, four out of twelve applications show more than 5% IPC improvement. However, it is not clear how many of these branches would really come under the L2 cache miss shadow in the Runahead or CLEAR architecture. Once the refill completes, the subsequent branches in the dependence chain will get the operands quickly and can verify the prediction, as usual. The improvement projected in Table V is, indeed, an upper bound on the achievable performance gain because of two reasons. First, we consider the branches in the dependence chain of loads missing not only in the L2 cache, but also in the L1 data cache. Second, the dependent branches that enter the pipeline even after the load refills are also considered. However, it is interesting to note that as the memory latency increases relative to the processor speed, we would approach the IPC improvement depicted in Table V.

V. Conclusions

In this paper we have explored the performance potential of correctly predicting the long-latency branches i.e. the branches

TABLE V

IPC WITH PERFECT PREDICTION OF THE LONG-LATENCY BRANCHES IN UNPRUNED DEPENDENCE TREE

Application	2Bc-gskew IPC	PerfectLLB IPC	Application	2Bc-gskew IPC	PerfectLLB IPC
`002.tiger`	0.45	0.45	`197.parser`	0.47	0.47
`003.clustalw`	0.65	0.65	`253.perlbmk`	0.39	0.39
`164.gzip`	0.46	0.52	`254.gap`	0.36	0.37
`175.vpr`	0.44	0.47	`255.vortex`	0.96	0.97
`176.gcc`	0.49	0.49	`256.bzip`	0.36	0.38
`186.crafty`	0.63	0.63	`300.twolf`	0.28	0.30

that belong to the dependence chains of loads missing in the L1 data cache. We find that for traditional pipelines these branches are not critical for performance. Due to the limited size of the active list, not too many mispredicted branches depending on the missed loads actually come into the pipe. For architectures where the size of the active list is virtually extended by checkpoint-assisted early load retirement, we find that only one application (`gzip`) enjoys 13% performance improvement in the ideal situation where *all* the branches belonging to the dependence chains of missing loads are correctly predicted. Three other applications (`vpr`, `bzip`, and `twolf`) achieve up to 7.1% performance improvement.

We conclude that although a large performance gap remains between the state-of-the-art branch predictors and a perfect predictor, the branches belonging to the dependence chains of the long-latency loads should not be the target for improving control speculation. Even though these branches, if mispredicted, suffer from a large misprediction penalty, they are often predicted correctly.

REFERENCES

[1] K. Albayraktaroglu et al. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 2–9, March 2005.

[2] J. L. Aragón et al. Selective Branch Prediction Reversal by Correlating with Data Values and Control Flow. In *Proceedings of the 19th International Conference on Computer Design*, pages 228–233, September 2001.

[3] D. N. Armstrong et al. Wrong Path Events: Exploiting Unusual and Illegal Program Behavior for Early Misprediction Detection and Recovery. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 119–128, December 2004.

[4] A. Falcón et al. Prophet/Critic Hybrid Branch Prediction. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 250–261, June 2004.

[5] J. González and A. González. Control-fbw Speculation through Value Prediction for Superscalar Processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 57–65, October 1999.

[6] G. Hinton et al. The Microarchitecture of the Pentium 4 Processor. In *Intel Technology Journal*, Q1, 2001.

[7] T. H. Heil, Z. Smith, and J. E. Smith. Improving Branch Predictors by Correlating on Data Values. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 28–37, November 1999.

[8] D. A. Jiménez. Fast Path-based Neural Branch Prediction. In *36th Annual International Symposium on Microarchitecture*, pages 243–252, December 2003.

[9] D. A. Jiménez and C. Lin. Neural Methods for Dynamic Branch Prediction. In *ACM Transactions on Computer Systems*, **20**(4): 369–397, November 2002.

[10] T. Karkhanis and J. E. Smith. A Day in the Life of a Data Cache Miss. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues*, May 2002.

[11] R. E. Kessler. The Alpha 21264 Microprocessor. In *IEEE Micro*, **19**(2):24–36, March 1999.

[12] N. Kirman et al. Checkpointed Early Load Retirement. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 2–13, February 2005.

[13] S. McFarling. Combining Branch Predictors. *Technical Note*, DEC Western Research Laboratory, June 1993.

[14] P. Michaud, A. Seznec, and R. Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 292–303, June 1997.

[15] O. Mutlu et al. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 129–140, February 2003.

[16] S. Rixner. Memory Controller Optimizations for Web Servers. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 355–366, December 2004.

[17] A. Seznec. An Optimized 2Bc-gskew Branch Predictor. *Technical Report*, September 2003.

[18] A. Seznec et al. Design Trade-offs for the Alpha EV8 Conditional Branch Predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 295–306, May 2002.

[19] T. Sherwood et al. Automatically Characterizing Large-scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[20] J. E. Smith. Is There Anything Left to Learn about High Performance Processors? *Keynote Talk in 17th Annual ACM International Conference on Supercomputing*, June 2003.

[21] L. Spracklen and S. G. Abraham. Chip Multithreading: Opportunities and Challenges. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 248–252, February 2005.

[22] Z. Zhang, Z. Zhu, and X. Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *Proceedings of the 33rd International Symposium on Microarchitecture*,pages 32–41, December 2000.