Dynamic BTB Indexing Using Jump Offset Prediction

Vidhya C., Manimozhi I., and Jithendranath Mungara

Abstract—**Branch instructions have been the biggest contributor of reducing parallelism in pipe-lined computers. Branch prediction had improved the predictability of branch conditions. But the branch targets for indirect branches are still hard to predict as a single branch instruction can branch to different targets in case of indirect branches. Using a BTB will only improve performance of indirect branches that branch to the same target address.**

In this paper, we propose a modified indexing of BTB, that would allow, first, multiple entries for a single branch instruction to improve prediction of branches with multiple targets and second, use of the offset that the branch instruction would use as part of the index to improve the indexing of BTB. We achieve this by caching the values produced by most recently executed instructions and identifying the instruction that produced the offset value used by the branch instruction as an offset source for that branch. When the source instruction is executed the next time, the value produced is taken as the predicted offset and used along with branch instruction PC as an index into the BTB. Updating the BTB after the completion of branch instruction is also done with the same index. Using the predicted offset to augment the BTB index allows for better prediction of indirect branches.

Index Terms—**Branch prediction, BTB, dynamic indexing, indirect jump, offset prediction.**

I. Introduction

Modern pipelined processors incur pipeline penalty mainly due to control dependencies in the program flow. Current high-performance super-scalar processors use branch prediction to speculatively execute instructions beyond an unresolved branch to reduce the penalty caused by branches. Most common types of branches are conditional or unconditional and direct or indirect. It had been shown that conditional branch direction can be predicted with high accuracy using various branch prediction techniques. This makes direct branches, which have a fixed branch target, conditional or unconditional easier to predict. Indirect branches however are hard to predict as they have multiple targets corresponding to a single indirect branch instruction. Indirect jumps are being used more frequently in object oriented programming languages. With increase in programs written in such languages, the relative frequencies of indirect jumps are increasing compared to direct jumps. As indirect branches have high misprediction rate, the increase in the frequency of indirect branches, indirect branch misprediction penalty is becoming a sizeable fraction of overall branch misprediction penalty.

In this paper we introduce a technique to dynamically index BTB using the predicted value of the jump target offset. "Dynamic BTB indexing using jump offset prediction" dynamically identifies an instruction that produced the value of the offset used by the indirect jump and marks it as the offset-source instruction for that branch instruction. When that instruction produces a new value, its output is used as the predicted-offset for the jump instruction and the index into BTB is computed by hashing the PC of jump instruction and the predicted-offset.

This paper makes the following contributions:

1) Dynamic BTB indexing using jump offset prediction: a low cost, correlation based indirect jump target address prediction mechanism with high target prediction accuracy. Multiple targets of a jump instruction are stored in the BTB at indices computed by hashing the PC of the jump instruction with the output value of the offset-source instruction.
2) Target address prediction overriding using a second and more accurate target address prediction. Target address prediction overriding has not yet been proposed. Prior studies have proposed branch direction outcome overriding for conditional branches.
3) The prediction can be implemented in an out-of-order processor using existing BTB with little extra hardware support.

II. Related Work

Initial indirect jump target predictions were done using branch target buffer (BTB), which predicted the last target of the branch as the current target [1], [2]. Indirect branches often have multiple targets and thus a BTB based predictor, though simple in design, is inaccurate for predicting them.

Further work on predicting indirect branches were divided into two categories: history-based and precomputation based. A branch history based two-level predictor is the "target cache" for predicting the target address of indirect jumps. It was based on same idea as a two-level branch direction predictor [3]. A target history register records the target addresses from recently executed indirect jump instructions. To predict the target of an indirect jump instruction, the fetch address of jump instruction and the target history register is used to index in the target cache to get next target address. Upon completion of the indirect jump, the actual target address is used to update the target cache entry and the target history register.

Li *et al.* [4] proposed technique that identifies indirect

Manuscript received May 12, 2012; revised July 18, 2012.

Vidhya C is with CMR Institute of Technology, Bangalore, KA 560037, India(e-mail: mail.vidya.in@gmail.com).

Manimozhi I is with the Department of Computer Science, CMR Institute of Technology, Bangalore, KA 560037, India (e-mail: srimanisen@gmail.com).

Jithendranath Mungara is with the Department of Computer Science, CMR Institute of Technology, Bangalore, KA 560037, India (e-mail: jmungara@yahoo.com).

DOI: 10.7763/IJIET.2012.V2.186

jump instructions that are hard-to-predict and then stores their targets in the BTB using a second level index. This technique is called rehashable BTB (R-BTB) the new index is computed by hashing the target history register with the jump instruction address. Other branches including the easy-to-predict indirect branches continue to index the BTB using the first level index, the PC value. Driesen et al. [5] [6] combined multiple target predictors to build a cascaded hybrid predictor, which uses a simple predictor for easy-to-predict branches and a more complex but small predictor for only the hard-to-predict branches.

Kim *et al.* [7] proposed VPC prediction which treated an indirect branch with N different targets as N conditional direct branch instructions and used existing conditional branch predictors to predict them. To predict the target of an indirect jump instruction, the conditional branch predictor is accessed for up-to a "max-targets" times, each time for different target of the indirect branch. This iterative access stops when a specific direct-target is predicted to be taken, in which case the indirect jump is predicted to take that target address, or "max-targets" limit is reached, in which case the processor is stalled until the indirect branch is resolved. Each attempt to predict a direct-target takes one cycle, which stalls the pipeline. This causes the performance of VPC prediction to degrade significantly for programs which have indirect branches with higher number of targets.

Pre-computation based target prediction schemes try to calculate the target address ahead of the execution of branch instruction instead of trying to predict the target address. Roth et al. [8] proposed a precomputation-based prediction method specifically for virtual function calls. It captures the sequence of instructions used to generate the target addresses. Whenever the first instruction in the sequence is executed, the technique quickly executes the rest of the instructions using a separate parallel execution engine, which computes the target before the actual indirect jump instruction is fetched. A significant drawback in this technique is that it requires significant hardware for capturing the target generation instructions along with a fast execution engine to pre-compute the target ahead of time. Another drawback is that this technique is very specific to virtual call target prediction.

Recently, Farooq *et al.* [9] proposed a compiler assisted technique for indirect branch prediction called Value Based BTB Indexing (VBBI), in which the compiler identifies an instruction(which is referred to as 'hint instruction') whose output value strongly correlates with the target address taken by the jump instruction. Different hint values correspond to different target addresses of a jump instruction, and these targets are stored in the BTB at different indices computed by hashing the PC of the jump instruction with the hint value. Next time when the jump instruction is fetched, the BTB is indexed using its PC and the new hint value to get the predicted target address. VBBI was shown to produce better prediction rates than previous prediction mechanisms. However, it had the drawback that it required compiler support which causes severe overhead when it comes to adapting new technologies in computer architecture. To fully utilize the potential of the technique, all existing programs would have to be re-compiled with the modified compiler.

III. Dynamic BTB Indexing Using Jump Offset Prediction

A. Overview

"Dynamic BTB indexing using jump offset prediction" dynamically identifies an instruction that produced the value of the offset used by the indirect jump and marks it as the offset-source instruction for that branch instruction. When that instruction produces a new value, its output is used as the predicted-offset for the jump instruction and the index into BTB is computed by hashing the PC of jump instruction and the predicted-offset.

Augmenting the PC with the predicted-offset allows multiple targets to be stored in the BTB for a single branch instruction. For a jump instruction, different offset values would produce different targets and they would be stored in the BTB at the index computed by hashing the jump PC with the predicted-offset. Basis for this system comes from the idea that the jump target offsets would be computed usually just before the branch instruction is computed [7].

To identify the offset source instruction, we maintain a small map of recent values produced by instructions to their PCs called the Value-Cache. When the jump instruction executes, the actual offset value for the target address calculation is used to lookup the value cache. Value cache will provide the most recent instruction in the past that produced the offset value. We identify this instruction as the offset-source instruction for future predictions of the jump instruction. The BTB is updated by indexing using the hash of jump PC and the offset value.

When the offset-source instruction executes next time, it checks the map of source instruction pc to jump pc to identify the jump instruction for which this is the source and then updates the offset source buffer with the new value it produces. This updates the predicted-offset value the branch instruction will use to predict the next indirect jump. If the offset source instruction and the actual offset value have a good correlation, this would result in very high prediction accuracy.

B. Implementation Details

We introduce three small buffers to augment the BTB for identifying the source-offset instruction and to maintain a buffer of predicted-offset values.

1) Value cache:

A cache of most recent values produced and the PC of the latest instruction that produced the value. This cache is updated by the instructions that produce a register output (like ADD, MOV etc). Lookup on this cache is done in the write-back stage of the indirect jump instruction. The indirect jump offset value is used to look-up the recent instruction that produced the value, this instruction is used as the offset-source for that jump instruction.

2) Predicted offset buffer (POB)

This buffer is used to store the predicted offset values produced by the offset source instruction. The buffer is indexed the jump instruction PC. This buffer is updated in the write back stage when the indirect jump instruction and the offset-source instructions complete execution. The indirect

jump instruction updates the buffer with the actual offset value that was used to generate the target. When the offset-source instruction completes execution, it updates this buffer with the value it produced. This value is used as the predicted-offset for the next branch instruction target prediction. Lookup on this buffer is done by the jump instruction during the fetch stage. The jump_pc is hashed with predicted-offset value in the buffer to form the index into BTB.

3) Source jump map

This is a small map, to store the map of source instruction PCs and their corresponding jump instruction PCs. This map is indexed by the source instruction PC and updated in the write-back stage of the jump instruction. The jump instruction identifies the source instruction using the value cache and updates this map so that the source instruction would know which entry in the POB to update when it executes the next time. Look-up on this buffer is done during the fetch stage of offset-source instruction to find the jump instruction PC for which it is a offset source.

Each of the buffers are small, fast buffers and used only for indirect branch instructions in the instruction flow. Also, the buffers are accessed and modified in different stages by different group of instructions thus reducing the contention for access and improving the efficiency.

C. *Indirect Jump Instructions*

When an indirect jump instruction is executed, the following additional operations are performed. See Figure 1.

In the fetch-stage, look-up in to the predicted offset buffer is made using the branch instruction PC to find the predicted offset value. If the predicted offset obtained from the buffer is valid, it is hashed with the jump PC to generate the index into the BTB for predicting the target address of the jump instruction. When a valid predicted offset is not found, the BTB indexing defaults to just using the jump PC. The predicted offset buffer is a small very fast buffer, with just the jump PC and the offset value. Thus the look-up into the buffer in the fetch stage will be very quick, and the target prediction can be done in the fetch-stage without stalling the pipeline.

In the write back-stage, the actual offset value used by indirect jump instruction will be used to update the predicted offset buffer and to identify offset source instruction. First, the offset value is used to look-up the value cache to find the most recent instruction that produce the offset value. This instruction is the offset source and the entry is created in source jump map. Second, the offset value is also updated in predicted offset buffer to be used as the default prediction of offset, the next time the indirect jump instruction will be executed.

Updating the offset-source instruction is not done if the prediction from existing source instruction resulted in a correct target address prediction. This is done so that the correct offset-source instruction is not modified by other instructions that produce the same value as the offset value during some branch execution.

D. *Offset Source Instructions*

When an offset-source instruction is executed, the following additional operations are performed. See Figure 2.

In the fetch-stage, look-up in to the source jump map buffer is made using the source instruction PC to check if the instruction is actually a source instruction for some indirect jump instruction. This information along with the jump PC is captured in the pipeline buffers and passed on to the write back stage. Keeping the information about the source instructions in the source-jump map indexed by the source instruction PC enables a fast lookup to check for a source instruction when the instruction is fetched.

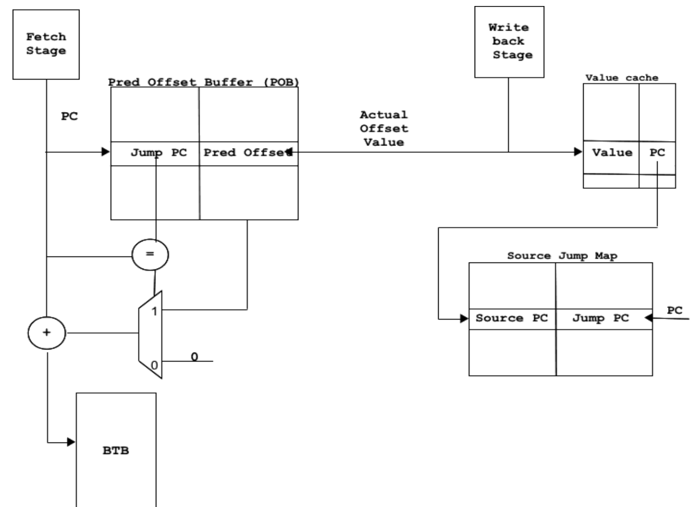

Fig. 1. Execution of indirect jump instruction

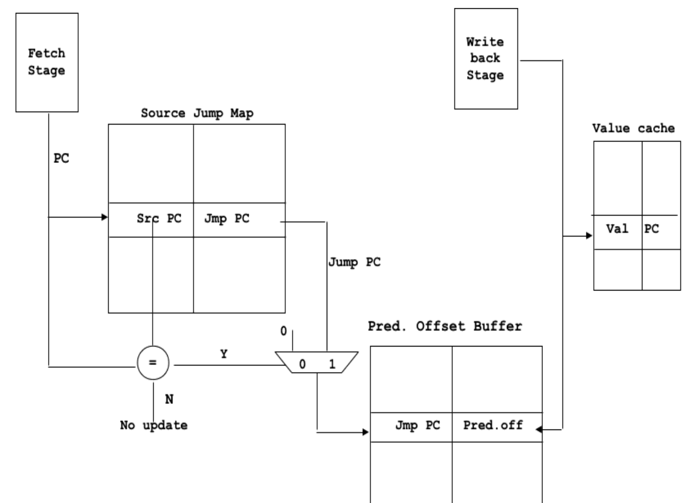

Fig. 2. Execution of offset-source and value producing instructions

In the write back-stage, the jump PC is used to index the predicted offset buffer and the predicted offset value is updated with the value produced by offset-source instruction. This offset value will be used by the subsequent indirect jump instruction for predicting the target address. There is a potential contention scenario here, when more than one jump instruction might end-up associating with the same instruction as its offset source. However, this is a very unlikely scenario as this could happen only when indirect jump instructions based on same condition occur very close to each other.

E. Instructions Producing Register-Output

Other instructions that produce a register output value are identified using the op-code and they update the value-cache using the value produced and their PC upon completion in write back-stage. Only instructions that produce a register output are considered because the offset value in an indirect jump instruction is a register. Filtering based on op-code to identify such instructions is done so that the load on the value-cache buffer will be as low as possible.

IV. Evaluation Methodology

Simple-scalar processor simulator toolset which features a wide issue super-scalar processor is used as the simulation environment. The processor on which the system is built is a out-of-order issue, pipelined super-scalar processor. We plan to extend the simple-scalar simulator to implement and analyse our branch prediction technique. The performance analysis of the predictor and comparison with other prediction schemes will be done using the Spec2000 CPU benchmark suites.

V. Conclusion

This paper proposed and evaluated the Dynamic BTB indexing using jump offset prediction for predicting target addresses of indirect jumps. The key idea of the new scheme is to store multiple targets of an indirect jump in the BTB at different indices computed by hashing the jump PC with the output value of a instruction whose output strongly correlates with the offset used for computing the target address taken by the indirect jump instruction. As such, the new indexing scheme enables the use of existing BTB structure to predict the targets of an indirect jump without requiring an extra structure specialized for storing multiple indirect jump targets.

References

[1] P. Chang, E. Hao, and Y. N. Patt, “Target prediction for indirect jumps,” *ISCA-24*, pp. 274–283, 1997.

[2] J. Lee and A. Smith, “Branch prediction strategies and branch target buffer design. Computer,” vol. 17, no.1, pp. 6–22, Jan. 1984.

[3] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive training branch prediction,” MICRO-24, pp. 51–61, 1991.

[4] T. Li, R. Bhargava, and L. K. John, *Rehashable BTB: An adaptive branch target buffer to improve the target predictability of java code*, 2002.

[5] K. Driesen and U. H’olzle, *Multi-stage cascaded prediction*, Euro-Par, 1999.

[6] K. Driesen and U. H¨olzle, “The cascaded predictor: economical and adaptive branch target prediction,” *MICRO-31*, pp. 249–258, 1998.

[7] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, “VPC prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization,” pp. 424–435, 2007.

[8] A. Roth, A. Moshovos, and G. S. Sohi, “Improving virtual function call target prediction via dependence-based pre-computation,” *ICS-13*, pp. 356–364, 1999.

[9] M. Farooq, L. Chen, and L. K. John, “Value based BTB indexing for indirect jump prediction,” *IEEE 16th International Symposium on High performance Computer Architecture(HPCA)*, 2010.

Vidhya C. is a post-graduate student at CMR Institute of Technology under Vishveshwarya Technology University, Bangalore. She is born and raised in Tamilnadu. Vidhya received her B.E. degree in Computer Science and Engineering from Anna University, Tamilnadu in 2009. She worked as a Lecturer in the department of computer science at Vivekananda Institute of Technology, Tamilnadu between 2009-2010.