Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters

Michal Revucky
michal.revucky@aon.at

January 6, 2007

Abstract

The paper [EG03b] deals with two methods, which are used to reduce mispredictions of indirect branches in virtual machine interpreters. Interpreters designed for efficiency can spent a lot of their execution time recovering from misprediction of indirect branches. Branch target buffers is a common method of reducing mispredictions, however the accuracy of this method in existing interpreters is only 2% - 50%. The proposed methods of improving the branch prediction accuracy of BTBs for interpreters are: replicating virtual machine (VM) instructions and the use of superinstructions[1]. The use of these techniques results in speedups by a factor of 3.17 over efficient threaded code interpreters and speedups by a factor of 1.3 over techniques relying on superinstructions only.

1 Introduction

Interpreters are popular language implementation technique, when choosing this approach there are certain advantages like *Ease of implementation*, *Portability* and *Compilation Speed* on the other hand the *Execution Speed* of a program suffers by a factor of ten slowdown for general-purpose programs over native code produced by an optimizing compiler [HATW99]. The Reason for this slowdown is that modern interpreters execute a huge number of indirect branches (up to 13% of the executed instructions). A misprediction of an indirect branch on modern architectures is expensive (it costs about 10 cycles on a Pentium III and up to 20 cycles on a Pentium IV). This fact leads

[1]combing a sequence of instructions into a single instruction

to the problem that even an efficient interpreter can spend more than the half of its execution time in recovering from branch mispredictions [EG01].

The best indirect branch predictor in today's processors is the branch target buffer (BTB). BTBs mispredict 50%-63% of the executed branches on threaded code interpreters and 81%-98% in switch based interpreters.

The rest of this paper will deal with:

- Interpreters and their utilization of BTBs.
- Improving the branch prediction accuracy with the new method of replicating VM instructions.
- Evaluation of this new method as well of the existing super instruction method.

2 Background

2.1 Efficient Interpreters

If we want to implement an efficient interpreter it should be fast interpreting a program, performing large number of simple operations rather than spend most of its time in native code library. On programs implementing a large number of simple operations, interpreters are slowest compared to native code compilers. Interpreters are divided into a front-end and a back-end. The reason for this is to avoid parsing the source over and over. The front-end (compiler) therefore produces VM code which has a flat layout and may be executed efficiently by the back-end.

The execution of one VM instruction consists of accessing its arguments performing the function of the instruction and dispatching (fetching, decoding and starting) the next VM instruction. Dispatch is common to each VM instruction and inefficient dispatch leads to inefficient interpreters. A dispatch of one VM instruction in efficient interpreters can be implemented by 3 native instructions (including the indirect branch), this leads to a high occurrence of indirect branches in the VM instruction mix.

There are two popular VM instruction dispatch methods:

Switch dispatch it is implemented by using a huge *switch* statement, each case for each instruction implemented by the virtual machine. Although this method is not very efficient [EG01] it must be used when building a VM according to ANSI C.

Threaded code represents a VM instruction as address of the routine that implements the instruction [Be73]. The code for dispatching the next VM instruction consists of fetching the VM instruction, jumping to the fetched address and incrementing the instruction pointer. This technique may be implemented with GNU C using the labels-as-values extension.

2.2 BTBs and Interpreters

Over the time CPU pipelines grew in order to support faster clock rates and out-of-order superscalar execution. Straight-line code are executed very fast on such CPUs, but their weaknesses are the branches, since they are resolved late in the pipeline (stage n), it would take n cycles until a branch is a the same stage in the pipeline if there was no branch, that's why they affect the start of the pipeline.

The occurrence of this problem is reduced by the usage of branch prediction. A widely spread branch predictor in modern CPUs is the branch target buffer (BTB). An idealised BTB contains one entry for each branch and predicts that the branch jumps to the same target as the last time the branch was taken (see Fig. 1).

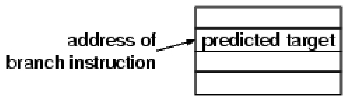
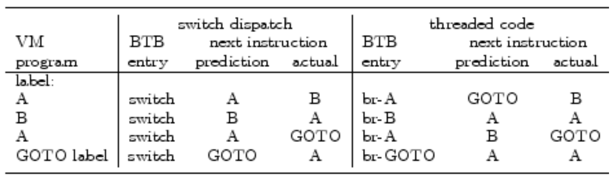


Figure 1: Branch Target Buffer

As research has shown BTBs mispredict 81%-98% of indirect branches in switch-dispatch interpreters and 57%-63% of indirect branches in threaded code interpreters. The reason for this difference is that in threaded code each VM instruction has its own dispatch sequence. On the other hand the switch dispatch has a single dispatch code and breaks are compiled into unconditional branches to this common dispatch code.

When you consider the code fragment in Fig. 2. Imagine that that the loop was executed at least once. With the switch dispatch there is only one BTB entry involved. When jumping to native code of VM instruction A, the BTB entry is updated to point to that native code routine. When the next VM instruction is dispatched the BTB predicts A, which in our case is wrong and the VM instruction B would be correct. In this example the BTB always predicts wrong.



	switch dispatch			threaded code		
VM	BTB	next instruction		BTB	next instruction	
program	entry	prediction	actual	entry	prediction	actual
label:						
A	switch	A	B	br-A	GOTO	B
B	switch	B	A	br-B	A	A
A	switch	A	GOTO	br-A	B	GOTO
GOTO label	switch	GOTO	A	br-GOTO	A	A

Figure 2: BTB Prediction on a small VM program

On the other hand for threaded code, each VM instruction has its own indirect branch as well as its own BTB entry. Consider the code for threaded code in Fig. 2, for example when VM instruction B is dispatched for the first time the BTB br_B entry is updated to A and the branch prediction will be correct this also applies to GOTO. Since the VM instruction A appears twice in this code fragment its branch prediction using BTB will never be correct, because br_A will alternatingly be updated to B and GOTO, but never correctly.

The rest of the paper will deal with interpreters using separate dispatch branches.

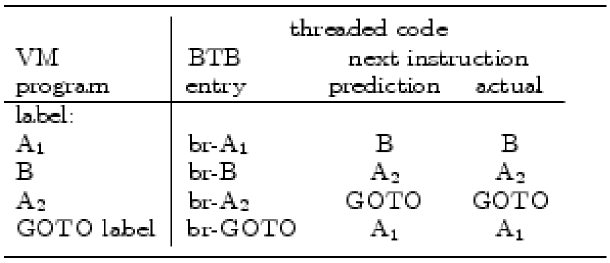
3 Improving Prediction Accuracy

Mispredictions are common when a VM instruction appears in the working set of an interpreted program more than once. If a VM instruction appears only once, the BTB will predict its dispatch correctly. Two methods were developed or reused in order to reach this aim.

3.1 Replicating VM Instructions

While replicating VM instructions, it is avoided to have several VM instructions spread across the working set of an interpreted program. With replication of VM instructions there are copies for a single instruction. The aim is that a replica appears only once, then its branch will be predicted correctly, since each instruction will have its own BTB entry.

In Figure 3 you may see how this works. The VM instruction A has now two replicas A_1 and A_2. Both of these copies have its own dispatch branch and its own entry in the BTB. In our Example A_1 is always followed by B and A_2 is always followed by $GOTO$ and the dispatch branches for A_1 and A_2 will be always predicted correctly, after the interpreter executed the loop for the first time.

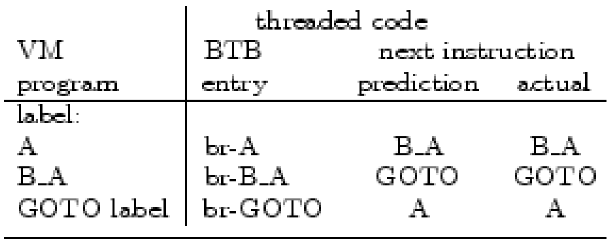


VM program	threaded code		
	BTB entry	next instruction prediction	next instruction actual
label:			
A_1	br-A_1	B	B
B	br-B	A_2	A_2
A_2	br-A_2	GOTO	GOTO
GOTO label	br-GOTO	A_1	A_1

Figure 3: Replicating VM Instructions

3.2 Superinstructions

When using superinstructions several VM instructions are combinded into a single superinstruction. This approach was use to reduce the size of VM code. It has also positive effects on branch prediction accuracy. When you consider Figure 4 it gets quite clear why this is the case. The sequence B and A is combined into a single superinstruction B_A. And now there are only unique instructions in this code fragment, every dispatch of each instruction has its own BTB entry and therefore after the first iteration of the loop the targets will be predicted correctly.



VM program	threaded code		
	BTB entry	next instruction prediction	next instruction actual
label:			
A	br-A	B_A	B_A
B_A	br-B_A	GOTO	GOTO
GOTO label	br-GOTO	A	A

Figure 4: BTB Accuracy and Superinstructions

4 Implementation

There are two ways of implementing these two methods, the *static* and the *dynamic* approach. With the *static* approach the interpreter writer provides replicas and/or superinstructions at interpreter build-time. This is usually done by supplying C code to the interpreter. With the *dynamic* approach the interpreter front-end generates VM code and produces replicas and/or superinstructions.

5 Speedups

The new techniques provide nice speedups over existing techniques provided by efficient interpreters. A speedup by a factor of 2.08 is reached with *static both* (a combination of replication and super instructions, implemented with the static approach). With *static super* (superinstructions only) a factor of 3.17 is reached.

6 Conclusion

VM code contains many indirect branches. Since common branch predictors do not always predict an indirect branch correctly, software techniques have to be implemented in order to exhaust a branch predictor entirely. This summary presented such two techniques. Both techniques brought a remarkable speedup of the executed VM code.

References

[EG03b] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In SIGPLAN '03 Conference on Programming Language Design and Implementation, 2003.

[HATW99] J. Hoogerbrugge, L. Augusteijn, J. Trum and R. van de Wiel. A code compression system based on pipelined interpreters. *Software-Practice and Experience*, 29(11):1005-1023, Sept. 1999

[EG01] M. A. Ertl and D. Gregg. The behaviour of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001*, pages 403-412. Springer LNCS 2150, 2001.

[Be73] J. R. Bell. Threaded code. *Commun. ACM,* 16(6):370-372, 1973.