DREU Research at Georgia Tech: Branch Prediction and Predication

Elba Garza

October 21, 2009: It has come to my attention that it seems my final report and presentation did not go through the first time I submitted it! Now, over a month later, I realize this and would like to note this. I'm so sorry!

Introduction

For an estimated ten weeks, I worked under the supervision of Dr. Hyesoon Kim at the computer architecture lab at the Georgia Institute of Technology.

This was my first research opportunity I had done, so there was quite a bit to learn. In every way possible, I became a graduate student. From finding an apartment to making my own food and then staying up late to finish lab reports, I did it all.

Also assigned to help me during my time of research was graduate student Minjang Kim. Sadly though, I was not able to work with him much due to his previous planned trips, including a 3-week excursion to South Korea. Due to this setback, I was not able to get much into the productive research that Minjang was heading. Instead, I learned a great amount about computer architecture.

Through DREU's questionnaires, I purposely hoped to gain more experience in the computer architecture field because I felt I had little exposure to it. Because of that though, it took longer than usual to catch up on field knowledge that was basic for everyone else.

So instead, this summer was a wonderful time of reading up on topics I'd never think of learning, much less master. Specifically, my summer entailed me learning about the usage of branch prediction and predication to prepare programs for quicker execution and smarter usage of memory and processing.

Basic Readings

Before reading up on branch prediction, Hyesoon had me refresh on basic topics such as the C++ standard template library and the x86 Intel instruction set architecture. Because Minjang's project had extensive use of containers and direct architecture manipulation, this was absolutely helpful.

Accompanying the ISA reading was an introductory lesson in compilers, especially the idea of control flow analysis.

For the future learning of branch prediction, I had to gain knowledge in the definition of what a basic block was, how they were identified, and how one identify blocks as dominators or postdominators.

From then, the readings flowed on to branch prediction, in both Gshare and bimodal form. Also, basic predication, the converting of control dependencies to data dependencies, had to be learned.

Motivation

"Dynamic predication has been proposed to reduce the branch misprediction penalty due to hard-to-predict branch instructions...

Predication eliminates branches and therefore avoids the misprediction penalty..."

- Kim, Joao, Mutlu & Patt, 2007
"Profile-assisted Compiler Support for Dynamic Predication in Diverge-Merge Processors"

Reading up on Dr. Kim's recent publications helped find the motivation for looking into both predication and branch prediction as techniques for reducing misprediction penalties.

Dynamic predication, along with basic branch prediction, can be used together to lower the possibility of executing a mispredicted branch.

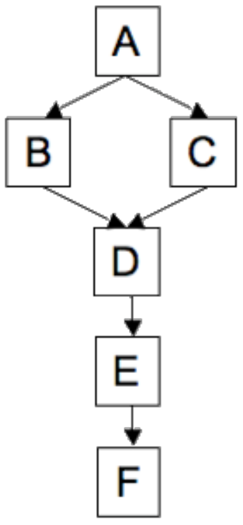
The misprediction penalty is one that we ardently try to avoid since it requires flushing of our computing pipelines and the restarting and recomputing of some of our branch instructions.

With branch prediction and predication, one can more easily find loops in programs, something that would take too long to find with other algorithms, such as tortoise-and-hare searching.

Control Flow Analysis

As stated before, my research wasn't as much hands-on as it was educational. I have very little physical results that I can show, but what I've learned is something I'll have as knowledge forever.

One of these important topics I covered was control flow analysis.



Simple hammock

Instructions can be separated into basic blocks or nodes, (A, B, C...) in a *control flow graph*. To identify basic blocks in our programs, one can follow these simple rules, as given in the "Dragon Book" by Aho & Company:

There is a directed edge from one basic block 1 to 2 another if:

1. There is a branch from the last statement of 1 to the first statement of 2, or

2. Control flow can fall through from 1 to 2 because:
   i. 2 immediately follows 1, and
   ii.1 does not end with an unconditional branch

With such rules, source code such as the code below can be separated into its respective branches:

```
begin
   prod := 0;
   i := 1;
   do begin
         prod := prod + a[i] *
b[i];
         i = i+ 1;
   end
   while i <= 20
end
```

Therefore, this source code has the following address code:

```
(1)    prod := 0
(2)    i := 1

(3)    t1 := 4 * i
(4)    t2 := a[t1]
(5)    t3 := 4 * i
(6)    t4 := b[t3]
(7)    t5 := t2 * t4
(8)    t6 := prod + t5
(9)    prod := t6
(10)    t7 := i + 1
(11)    i := t7
(12)    if i <= 20 goto (3)

(13)    …
```

Addresses (1), (3), and (13) represent the starts of new basic blocks created under the rules of the Dragon book.

With the creation of basic blocks, one can easily discern dominators within the control flow graph.

By definition,

A node, *a*, in a Control Flow Graph ***dominates*** a node, *b*, if every path from the first node to node *b* goes through *a*. It can be said that node *a* is a ***dominator*** of node *b*.

The ***dominator set*** of node *b*, **dom(***b***)**, is formed by all nodes that dominate *b*.

Also: by definition, each node dominates itself, therefore, $b \in$ **dom(***b***)**.

Definition: Let $G = (N, E, s)$ denote a flowgraph, where:

N: set of vertices
E: set of edges
s: starting node.
and let $a \in N$, $b \in N$.

1. *a* **dominates** *b*, written $a \leq b$, if every path from *s* to *b* contains *a*.

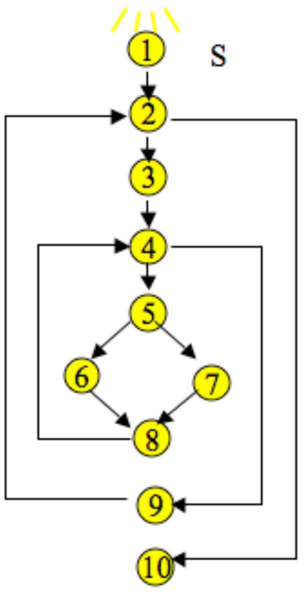
2. *a* **properly dominates** *b*, written $a < b$, if $a \leq b$ and $a \neq b$.

3. *a* **directly dominates** *b*, written $a <_d b$ if: $a < b$ and there is no $c \in N$ such that $a < c < b$.

In observing these dominators, one can do as one great lecture on this topic described:

"Imagine a source of light at the start node, and that the edges are optical fibers

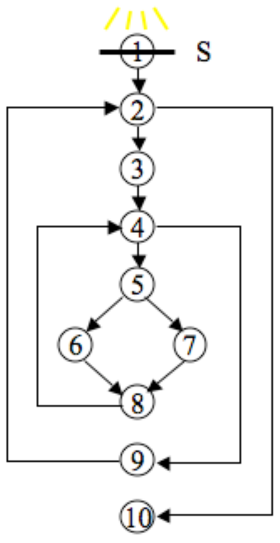
To find which nodes are dominated by a given node a, place an opaque barrier at a and observe which nodes became dark."



Then following the later instructions:

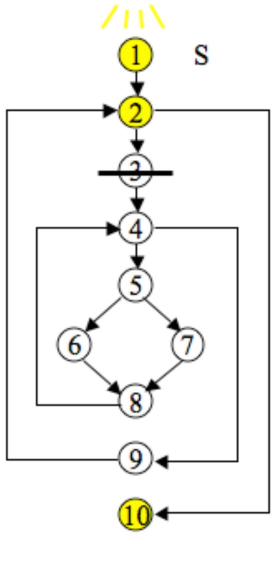
"To find which nodes are dominated by a given node a, place an opaque barrier at a and observe which nodes became dark."

For example, if we wished to see which nodes are dominated by Node 1, we place an opaque barrier there:



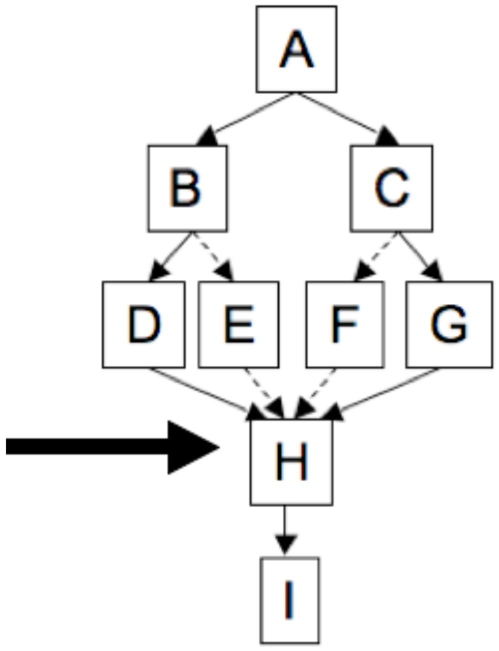
All the nodes darkened by the barrier are nodes who have Node 1 as a dominator.

Here is the technique used on Node 3:



The CFG shows that Node 3 dominates nodes 3, 4, 5, 6, 7, 8, and 9. Note though, that Node 10 is not darkened, because there is still a path present from Node 2 to Node 10 to maintain it "lit".

Using the definition of dominators, one can infer what a postdominator is.



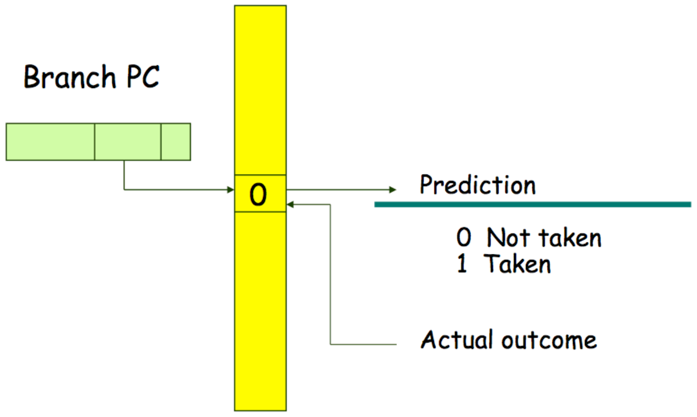
In this control flow graph, Node H can be recognized as a postdominator since rather than a path from the beginning always going through H, all paths at some point *converge* to it. In other words, all paths eventually lead to H.

Why are postdominators important? They are absolutely key because they are the foundation for which we are trying to identify loops in Minjang's project. Postdominators are noted to be present at the end of loops, and our tracer tool then analyzes and searches for postdominators are possible signals for loops.

Branch Prediction

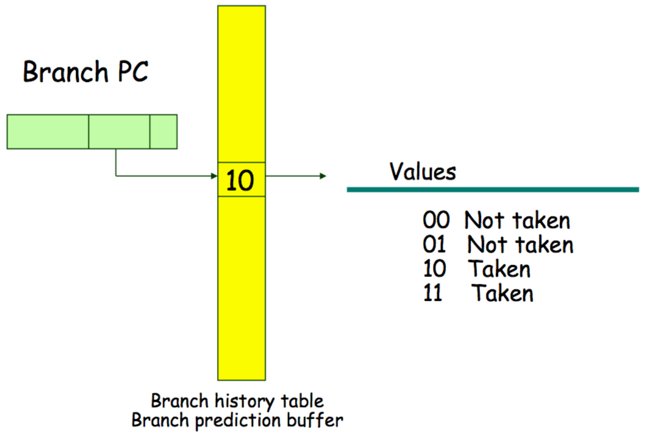
Branch prediction emerges from control flow analysis as a method to better predict the movement of a program.

In basic branch prediction, an array is used to keep record of predictions of whether a branch is taken (true) or not (false). Indexing is done through the usage of the branch PC:

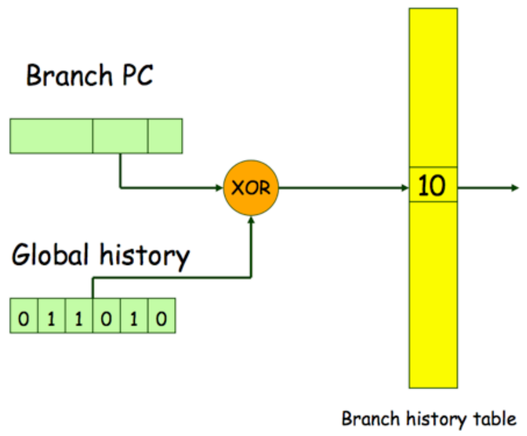


The actual result of the branch is then recorded and our next prediction changed accordingly. For example, if we guess correctly that a branch is taken (1), the prediction of it being taken remains. If it results that the branch was not taken, the future prediction is modified to be not taken (0).

Since that prediction is relatively discrete and jumps quickly from taken to not taken, a two-bit system can then be used to show strong predictions and weak ones.



With two bits, one can discern from the prediction being “strongly taken” (11) to “strongly not taken” (00). Both “weak” predictions are more susceptible to switching of prediction and reflect the fact that recent predictions have been incorrect.



Above is the model of a Gshare branch predictor, which uses previous global history and the Branch PC exclusive-OR’ed together to create the index at which our prediction is placed.

In the process of my research, I was actually able to model a simple Gshare branch predictor using Intel’s PIN tool, a dynamic instrumentation tool which allows the user to work directly with processing information.

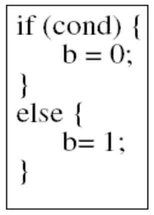
Intel’s PIN tool works by inserting code during the running time of a program to report running analysis.

Note: To see a print screen of Pin Tool, look at Slide 18 of my presentation!

Predication

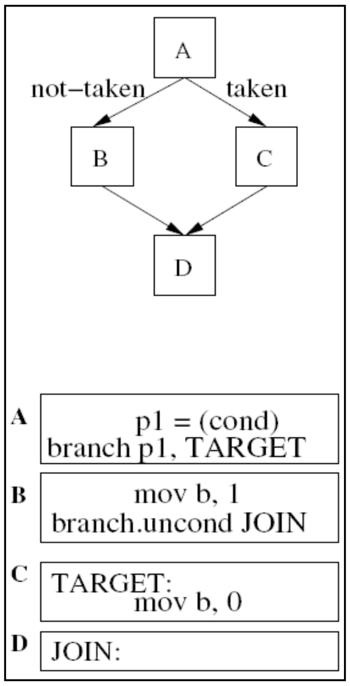
To further reduce chances of mispredicting branches during program executions, it is beneficial to employ predication.

To show how predication works, example code is given:



```
if (cond) {
      b = 0;
}
else {
      b= 1;
}
```

Code



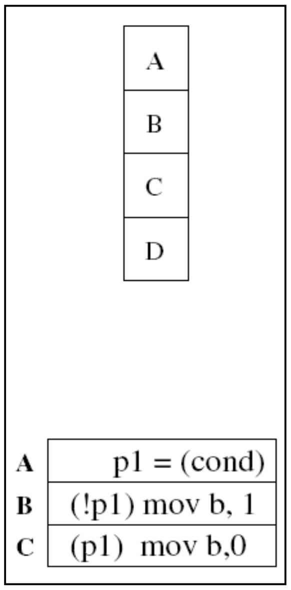
A	p1 = (cond) branch p1, TARGET
B	mov b, 1 branch.uncond JOIN
C	TARGET: mov b, 0
D	JOIN:

Branch Code

Simple code can be transformed into branch-style code as previously done. In this code, then, we have four basic blocks.

Predication removes condition dependence and causes all code to be data-dependent, making it possible for two possible outcomes to be executed simultaneously.

Note that blocks B and C represent the two possibilities that p1 is deemed to be a true and taken condition (B), or not, (C).



A	p1 = (cond)
B	(!p1) mov b, 1
C	(p1) mov b,0

The change can be summarized as follows:

"Rather than having branches (taken or not taken), the branch condition is made into a predicate, deemed either true or false... In Predication, each instruction is executed when a predicate is true. Every instruction enters the processing pipeline, but results are suppressed if the predicate is found to be false."

The benefit of predication is that two outcomes can be calculated at the same time, and depending on whether that condition is deemed to be taken or not, the proper calculation is used and the other scrapped.

Conclusion

While Minjang's project was vastly more complicated than my simple implementation of a Gshare branch prediction using PIN, I learned a vast amount of information.

There is no groundbreaking discovery that I can proudly show to DREU as a representation of my time spent in the lab, but I feel I've gained so much insight I could not possibly have collected any other way.

Next semester, I'll be taking more computer architecture classes to fulfill some technical electives for my major. I can assure you that I never would have even thought of signing up for such a class without this experience.

I thank Hyesoon Kim for her patience with me, her dedication to helping me understand, and her enthusiasm.

This summer was unforgettable in so many ways. Before this summer, I did not anticipate going to graduate school, but now, it is definitely my plan of action.

Thank you DREU!

Sources:

J.R. Allen et al., ''Conversion of Control Dependence to Data Dependence,'' Proc. 10th Ann. Symp. Principles of Programming Languages (POPL 83), ACM Press, 1983, pp. 177-189.

H. Kim et al., ''Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution,'' Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture (Micro 05), IEEE CS Press, 2005, pp. 43-54.

H. Kim et al., ''Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths,'' Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture (Micro 06), IEEE CS Press, 2006, pp. 53-64.

H. Kim et al., ''Profile-Assisted Compiler Support for Dynamic Predication in Diverge-Merge Processors,'' to be published in Proc. IEEE/ACM Int'l Symp. Code Generation and Optimization (CGO 07), IEEE CS Press, 2007.

S.A. Mahlke et al., ''Characterizing the Impact of Predicated Execution on Branch Prediction,'' Proc. 27th Ann. Int'l Symp. Microarchitecture (Micro 94), IEEE Press, 1994, pp. 217-227.