Dynamic Hammock Predication
ECE 721: Final Project Report

Sri Harsha Patnala
spatnal@ncsu.edu

Jayadeep Tunuguntla
jtunugu@ncsu.edu

1.Project Description

In this project, we explore **control dependence**, which is one of the major IPC bottlenecks in modern superscalar processors. Our idea deals with exploiting control flow instructions, specifically branches to address performance degradation issues. Despite advancements in branch prediction, some of the branches are very hard to predict, and the misprediction penalty is miserable which significantly degrades performance. Some of these branches referred to as **hammocks**, often mispredicted by the predictors, don't have many instructions in the control region (taken or not taken paths) and introduce large penalties. We target such hammock candidates and introduce **dynamic predication** without support from the ISA or the compiler. Essentially, we execute both the paths of the branch and maintain the committed state by discarding those instructions in the wrong path once the predicated branch resolves. The outcome of this project is to implement dynamic predication in the existing 721sim simulator and justify that our implementation improves performance of some targeted benchmarks. We use the 721sim as our baseline and compare with our implementation to establish this fact.

2.Simulator Accomplishments

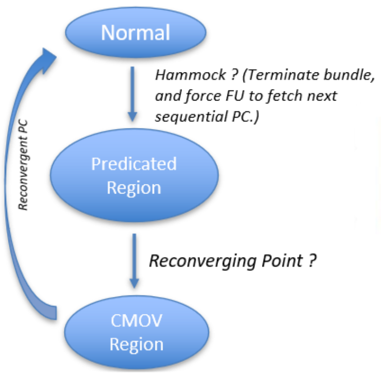
Major microarchitecture feature	**Perfect/Real Branch Prediction**		**Oracle/Speculative Disambiguation**		**Perfect/Real I$, D$**	
	if2.cc	hmmer	if2.cc	hmmer	if2.cc	hmmer
Single sided hammock	Complete	100M	Complete	100M	Complete	100M
Double sided hammock	Complete	NA	Complete	NA	Complete	NA
LSQ augmentation for then/else clause.	We tried to implement changes to LSQ to handle load/store instructions in the predicated region. However, given the time frame and complexity dealing with memory disambiguation we couldn't achieve this and still have some bugs.					
Issuing logic for CMOVs	We also tried to modify the issue logic explicitly for CMOVs, but the implementation is incomplete and has issues. This comes from the fact that we only need two sources (predicate value and corresponding source) to be ready.					

3.Microarchitecture Description

Fetch Unit

Although this isn't a microarchitectural aspect of the implementation, we perform benchmark profiling to identify hammocks. The simulator reads in a text file which entails all the necessary information of the hammock, and constructs a table, indexed by the PC of the hammock.

The fetch unit is modelled as a state machine (illustrated in Figure 1), where the state indicates region of the current fetch cycle. For each instruction in the fetch bundle, the PC is used to search into the table to check if we encounter a hammock. Fetch bundles are terminated in three scenarios, when we identify a hammock, when we see the reconverging PC (to insert CMOVs), and when all the CMOVs have been injected. The transition of state forms the basis of terminating the bundles. Also, this is where we map both the then/else/CMOV instructions to the functional simulator running ahead, which is later needed for checking at retire. Some of key challenges in fetch stage is handling the scenarios of bypassing the BTB misfetch logic (as we don't consider hammock as a branch), implicit removal of jump instructions in the then clause, and constructing the true fetch bundle based on state of fetch.


Normal
Hammock ? (Terminate bundle, and force FU to fetch next sequential PC.)
Predicated Region
Reconverging Point ?
CMOV Region
Reconvergent PC

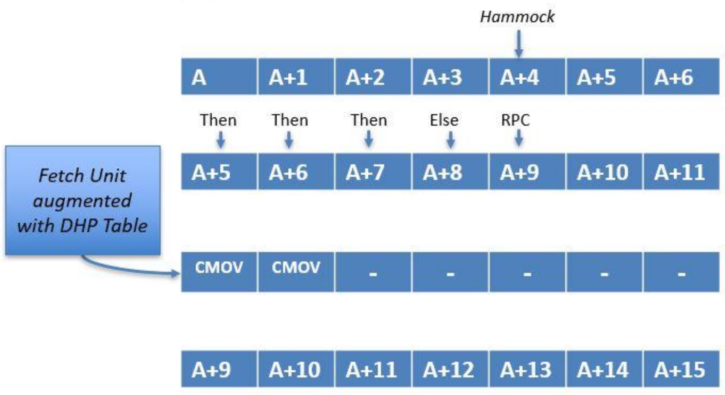


Figure1 **Figure2**

Decode

CMOVs use NOP to leverage existing opcodes in 721sim.Since we assign the instruction region type in fetch, when decoding NOPs this type is referred to assign the correct source and destination registers for CMOVs. Also, hammocks are not considered as branches, so a predicate register is assigned as a destination.

Rename and Dispatch

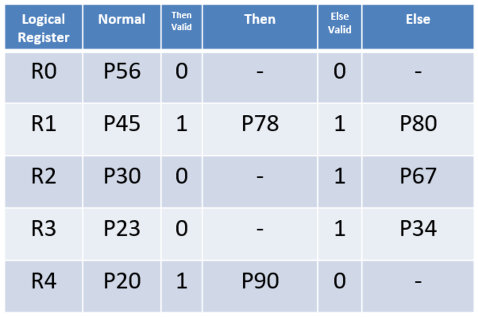
The rename map table is extended to demarcate the register mappings of the instructions before/after the hammock (fork/join context), then clause and else clause instructions. So, each register in the RMT has three versions, the fork/join version, then version and else version. To indicate if the register were ever modified by the then/else clause instructions, we assign valid bits to these mapped versions as shown in Figure 3. A special predicate register r64 also contains a mapping, which is modified whenever there is hammock. CMOVs rename the sources based on these valid bits. The active list has extra payload information, which are the instruction region type, deactivated flag, and the predication tag. Also, CMOVs are dispatched to the issue queue.

Execute and Writeback

Only two changes are needed in execute. The branch hammock has a destination register, so it needs to generate the predicate value by comparing the next PC against the computed PC, and then write this value into the PRF. For CMOV, we must choose the correct value based on predicate value and write this into the PRF. In writeback, the branch hammock broadcasts its predicated result to the active list, so that wrong path instructions are marked as deactivated. This is illustrated in Figure 4.

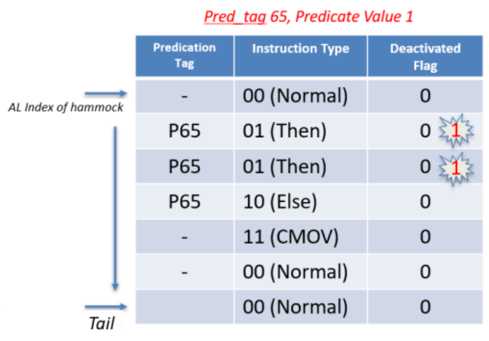
Retire

The deactivated instructions are discarded as they should not modify the committed state. However, these instructions still utilize some physical registers which are freed. CMOVs are fake retired, in a sense they maintain the committed state and free the physical registers just like any normal instruction would during retire.



Logical Register	Normal	Then Valid	Then	Else Valid	Else
R0	P56	0	-	0	-
R1	P45	1	P78	1	P80
R2	P30	0	-	1	P67
R3	P23	0	-	1	P34
R4	P20	1	P90	0	-

Figure3


Pred_tag 65, Predicate Value 1
AL Index of hammock
Tail

Predication Tag	Instruction Type	Deactivated Flag
-	00 (Normal)	0
P65	01 (Then)	0 1
P65	01 (Then)	0 1
P65	10 (Else)	0
-	11 (CMOV)	0
-	00 (Normal)	0
	00 (Normal)	0

Figure4

4.Simulator Configurations:

TABLE OF DEFAULT SUPERSCALAR PARAMETERS:

PARAMETER	VALUE
FETCH_WIDTH	8
DISPATCH_WIDTH	8
ISSUE_WIDTH	8
RETIRE_WIDTH	8
FQ SIZE	32
# BRANCH CHECKPOINTS	32
AL size	256
LQ/SQ size	32
IQ size	32

TABLE OF GRAPHS:

GRAPH NAME	MICROBENCHMARK USED	PARAMETERS VARIED
GRAPH1-IPC COMPARISION BASELINE Vs DHP	if2.cc(single sided hammock)	BRANCH PREDICTION TYPE,MEMORY DISAMBIGUITION, I,D/TRACE CACHES TYPE
GRAPH2-BRANCH MISPREDICTION RATE BASELINE Vs DHP	if2.cc(single sided hammock)	
GRAPH3-IPC COMPARISION BASELINE Vs DHP	hmmer_nph3_ref	
GRAPH4-BRANCH MISPREDICTION RATE BASELINE Vs DHP	hmmer_nph3_ref	
GRAPH5-IPC COMPARISION BASELINE Vs DHP	if2.cc(double sided hammock)	
GRAPH6-BRANCH MISPREDICTION RATE BASELINE Vs DHP	if2.cc(double sided hammock)	

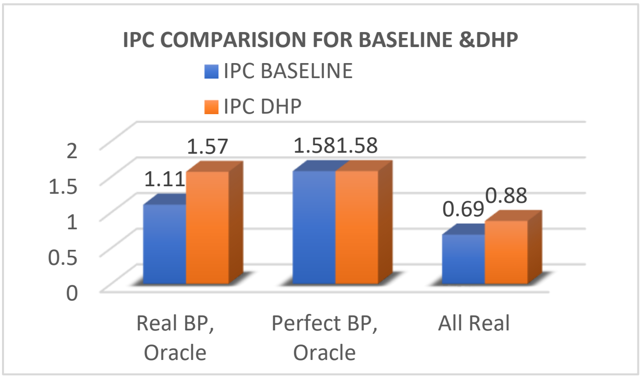
We used three different configurations in each graph.

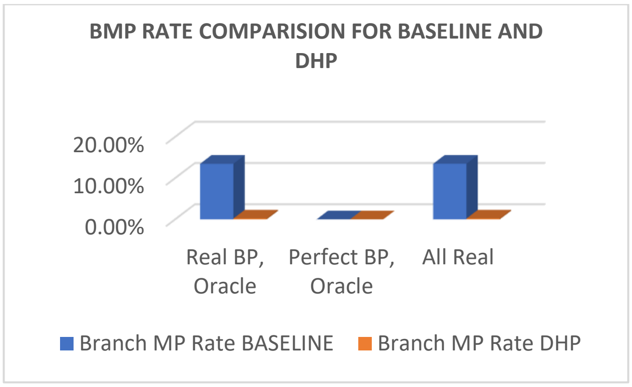
CONFIGURATION	PARAMETERS VARIED		
	BRANCH PREDICTION	MEMORY DISAMBIGUITION	I/D/T-CACHE CONFIGURATION
CONFIG1	REAL	ORACLE	PERFECT
CONFIG2	PERFECT	ORACLE	PERFECT
CONFIG3	REAL	NON-SPECULATIVE	REAL

5.Results & Analysis:

<u>5.1: PRIMARY RESULTS GRAPHS:</u>

For Single Sided Hammocks:

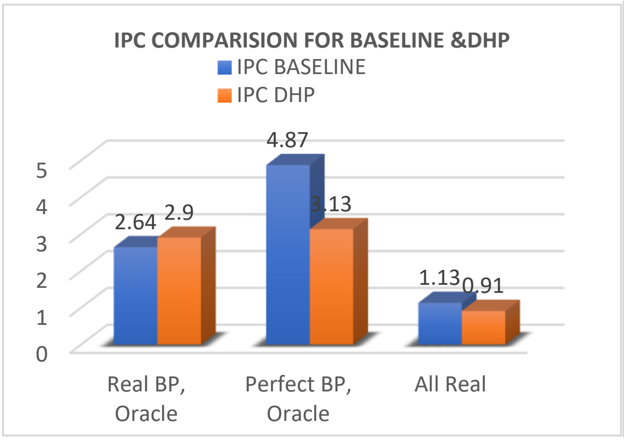


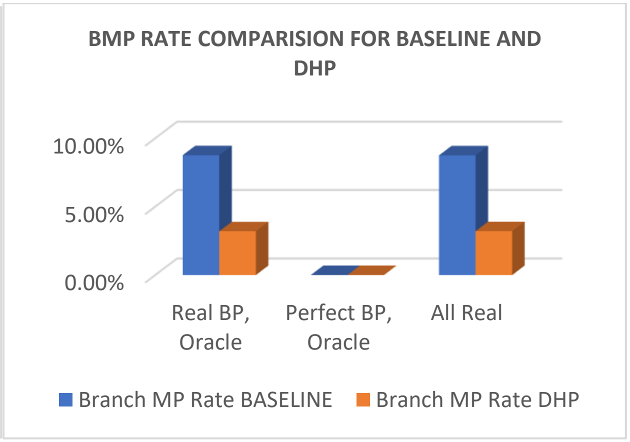


Graph 1

Graph 2

Hmmer_nph3_ref:



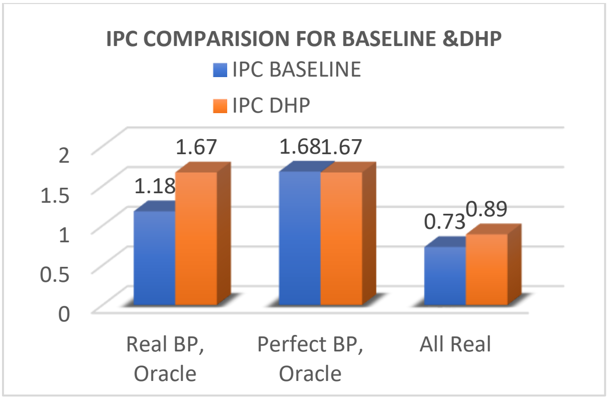


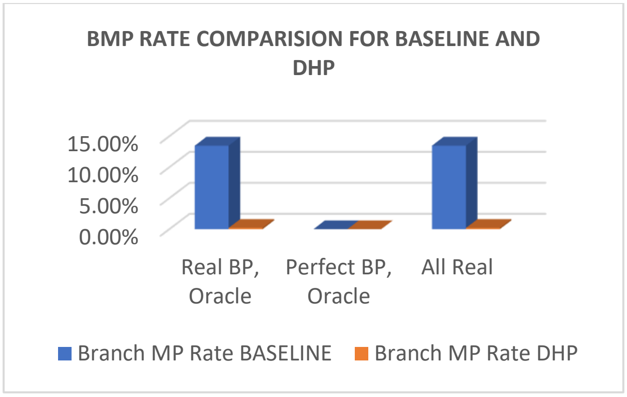
Graph 3

Graph 4

We have observed increase in IPC with DHP compared to Baseline for Real BP and Oracle Memory disambiguation configuration(Config1) in all benchmarks. For single and doubled sided hammock micro benchmarks the Real BP IPC of DHP case is almost same as Perfect BP IPC of baseline case. From this we can infer that when other branches are highly predictable and few branches are highly mis-predicting then DHP can be applied to those mis-predicting branches given their control region is small. The microbenchmarks tested above satisfies both these conditions. The branch misprediction rate has decreased when DHP is enabled in all benchmarks for all configurations which is expected.

For double sided Hammocks:





Graph 5

Graph 6

Interestingly, for graph3 there is big dip in IPC with DHP support for Config2(Perfect BP, Oracle Disambiguation, Perfect Caches),but in graph1 and graph5 they are almost same as expected. This behavior is observed because of 10% extra instructions(both CMOV and deactivated ones) executed in the pipeline stages when compared to baseline in config2. For every 10 Million instructions fetched from hmmer benchmark we are observing 1.1 Million CMOV and deactivated instructions retired at commit stage.

6.Future Work

Although the results section indicates that our implementation had performance boost in some benchmarks, we still believe it can be further improved with the following changes.

More robust fetch unit design – Current design choice of modelling a state machine in fetch forces to terminate bundles at state transitions. This causes fetch bandwidth to be the degrading factor. Maybe we can think of some alternative to address this. Or having a trace cache which could cache the entire predicated region in a single bundle may improve fetch bandwidth and thereby enhance performance.

Issuing CMOVS – CMOVs can be issued as soon as its predicate value and the corresponding source is ready. This would make the CIDD instructions after the predicated region to execute as early as possible.

Memory disambiguation – We haven’t focused on loads/stores within a predicated region, given the time we would like to explore this.

Nested Hammocks – We could investigate some techniques for implementing nested hammocks by adding some bit vectors like dependence vectors in value prediction.