ABSTRACT

GANDHI, JAYNEEL. FabFetch: A Synthesizable RTL Model of a Pipelined Instruction Fetch Unit for Superscalar Processors. (Under the direction of Eric Rotenberg).

High-performance superscalar processors examine a large pool of speculative instructions, called the dynamic instruction window, to exploit instruction-level parallelism (ILP). Scaling performance requires a larger and more accurate dynamic instruction window, which, in turn, requires a more accurate branch predictor. Achieving higher accuracy typically requires increasing the size of the branch predictor. Unfortunately, this may cause an increase in the processor's cycle time. A fast cycle time can be preserved by pipelining the branch prediction logic. This is not straightforward, however. The branch predictor uses the program counter (PC) of the current fetch block (among other context) to produce information that is needed to predict the PC of the next fetch block. This means that the inputs to the branch prediction logic depend on its outputs. This dependency loop within the branch prediction logic renders naive pipelining ineffective. Accordingly, if we want to use a larger branch predictor without increasing cycle time, then we need to apply sophisticated approaches to effectively pipeline the next PC loop.

Several approaches to effectively pipeline the branch prediction logic have been proposed in the last decade and they have been shown to have close to the same accuracy as their unpipelined counterparts and sustain a bandwidth of one prediction per cycle. Previous evaluations were based on cycle-level simulators implemented in high-level languages. However, pipelining the branch prediction logic is complex, and

the full extent of this complexity can only be revealed by implementing it at the level of hardware design.

Along these lines, in this thesis, I design, verify and evaluate synthesizable register-transfer-level (RTL) models of pipelined branch predictors to confirm their effectiveness in hardware, in terms of cycle time, accuracy and bandwidth. The motivation for RTL modeling is three-fold:

1. *Cycle time evaluation*: RTL models account for all control and datapath logic, unforeseen logic complexity that comes with pipelining the branch prediction logic, and imbalances that may arise in the pipelining of this logic. All of these factors are essential for determining the cycle time reduction of pipelining the branch prediction logic. This is not possible with abstract simulators.
2. *Validation of accuracy and bandwidth*: Due to the level of detail of RTL modeling, it is the most convincing way to confirm the accuracy and bandwidth claims of proposed techniques for pipelining the branch prediction logic.
3. *Generating synthesizable RTL designs of arbitrary fetch units for core customization*: The RTL models developed in this thesis are the basis for FabFetch, a novel tool within the FabScalar toolset for automatically generating synthesizable RTL designs of arbitrary fetch units, that differ in their fetch width, pipeline depth and sizes of structures. The overall FabScalar toolset enables tailoring major dimensions of a superscalar processor to workloads. The FabFetch tool developed in this thesis specifically enables tailoring the major dimensions of the fetch stage. The fetch stage is critical with respect to core

customization because the branch predictor's accuracy has a profound impact on exposing ILP and thus indirectly shapes the dimensions of other pipeline stages. In particular, having pipelined variants of the branch prediction logic opens up the design space to larger BTBs and BPs, which, in turn, expands the range of superscalar processor designs that are worth considering.

FabFetch: A Synthesizable RTL Model of a Pipelined Instruction Fetch Unit for Superscalar Processors

by
Jayneel Gandhi

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2010

APPROVED BY:

_____	_____
Eric Rotenberg Committee Chair	Gregory Byrd
_____	
Rhett Davis	

DEDICATION

To my parents, relatives and friends…

BIOGRAPHY

Jayneel Gandhi was born on April 21, 1987 in Mumbai, India. He received his bachelor's degree in Information and Communication Technology (ICT) from Dhirubhai Ambani Institute of Information and Communication Technology (DA-IICT), Gandhinagar, India in May 2008. During his undergrad, he worked as intern in ST Microelectronics Pvt. Ltd., Greater Noida, India in their Frontend Technology and Manufacturing (FTM) Department.

Currently, Jayneel Gandhi is a Master's Student in Computer Engineering at North Carolina State Univeristy (NCSU) under the guidance of Dr. Eric Rotenberg. He is affiliated with Center for Efficient, Scalable and Reliable computing (CESR) at NCSU. He is interested in the area of Computer Architecture and VLSI Design.

He will be continuing his graduate studies at University of Wisconsin at Madison in Electrical and Computer Engineering Department from fall '10.

ACKNOWLEDGMENTS

I would like to thank my parents for encouraging me throughout my life to have a good education. I would like to thank my undergraduate advisor, Dr. Chetan D. Parikh who instilled the value of understanding the problem and then attacking it with different viewpoints, which helped me in my research.

It was my privilege to work with Dr. Eric Rotenberg, my advisor at NC State University. He not only helped me gain knowledge during my master's but also gave my degree a direction. It was really enthralling experience working with him over the last two years.

I would like to thank my colleagues at NCSU and CESR for giving inputs during my research and thesis: Niket K. Choudhary, Tanmay A. Shah, Hiran Mayukh, Sandeep Navada, Hashem Hashemi, Muawya Al-Otoom, Saunder Rajan, Abhishek Dhanotia, Shivam Priyadarshini. I would like to thanks my roommates and friends for their support during my stay at Raleigh: Anirudh Acharya, Kinjal Bhavsar, Vasant Agrawal, Mitesh Ghelani, Varun T.K., Akshay Pavagada, Hiren Patel, Ajay Mulchandani and Vicky Deliwala.

This thesis was supported in part by NSF grant No. CCF-0811707, Intel and IBM. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Superscalar processors are being used in systems that vary from large high-performance servers to small low-power hand-held devices. Superscalar processors exploit instruction-level parallelism (ILP) to provide higher performance. A superscalar processor has multiple function units that can execute multiple instructions in parallel in a cycle. The processor must find enough independent instructions each cycle to fully utilize the parallel function units. The processor maintains a large pool of instructions, called the dynamic instruction window, which it examines to identify ready-to-execute instructions. Because of data dependencies among instructions, a large dynamic instruction window is typically needed, on the order of 100s of instructions, to identify enough independent instructions that can execute in parallel.

Branch instructions change the control-flow of the program and become a barrier in creating a large dynamic instruction window. The target address of a branch instruction is not known until it executes. A naive approach is to wait until the branch instruction executes, before fetching the next instruction. Unfortunately, branch instructions are so frequent in a program that this naive approach prevents forming a large dynamic instruction window. If instruction fetching stalls until a branch instruction executes, then the dynamic instruction window cannot grow beyond 5 to 8

instructions. Table 1-1 shows that the average number of instructions between dynamic branches (dynamic basic block length) is only 5.78.

Table 1- 1: Average dynamic basic block lengths for the 100 million Simpoints [1] of the SPEC 2000 integer benchmarks.

Benchmark	Avg. dynamic basic block length
bzip	5.87
crafty	6.93
gap	6.32
gcc	5.77
gzip	5.54
mcf	4.03
perl	5.00
parser	5.00
twolf	5.72
vortex	6.00
vpr	7.45
Average	5.78

Therefore, to maintain a large dynamic instruction window of 100s of instructions, the processor must accurately predict the target addresses of branch instructions. Branch prediction logic is typically made up of three major components:

1. Branch target buffer (BTB): The BTB identifies branches among the instructions currently being fetched, called the *fetch group*, and provides the taken targets of these branch instructions.
2. Conditional branch predictor (BP): The BP predicts the direction (taken/not-taken) of conditional branches within the fetch group.

3. Next program counter (PC) logic: This logic uses all of the information from the BTB and BP, to predict the next program counter (PC). The next PC is the starting address of the fetch block to be fetched in the next cycle.

A key point is that the BTB and BP use the current PC (among other context) as an index into their respective tables, to produce information that is needed to predict the next PC. Thus, there is a critical dependency loop present in the branch prediction logic. Figure 1-1 shows a high-level view of the logic involved in predicting the next PC every cycle. In particular, Figure 1-1 highlights the next PC dependency loop. We will call the branch prediction logic, along with the logic for fetching instructions from the instruction cache (I-cache) (not shown in Figure 1-1), as Fetch Stage 1. Fetch Stage 2 (not shown in Figure 1-1) has the logic for extracting the fetch group from the instructions supplied by the I-cache as well as the logic for recovering from a BTB miss.

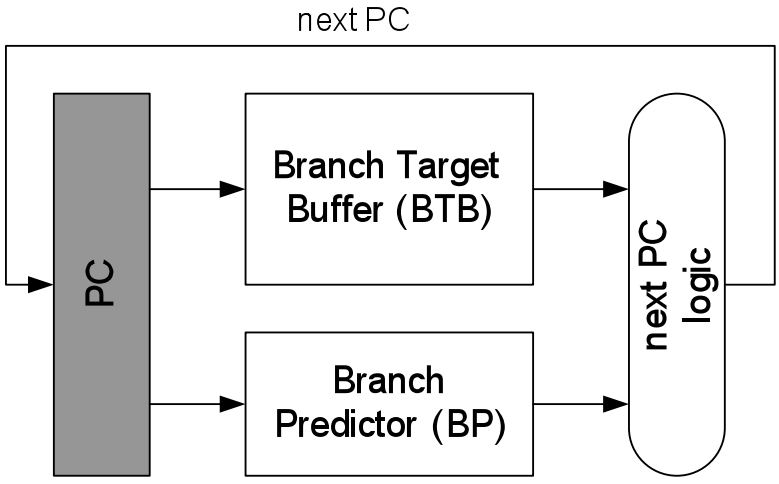

Figure 1- 1: High-level view of branch prediction logic in a superscalar processor.

Scaling performance requires a larger and more accurate dynamic instruction window, which, in turn, requires more accurate branch prediction logic. To enable this, both the BTB and BP need to be larger. Unfortunately, this increases the total propagation delay through the Fetch Stage 1, which may cause an increase in the processor's cycle time (lower clock frequency).

Accordingly, if we want to use larger branch prediction tables (BTB and BP) without increasing cycle time, then we need to somehow pipeline the branch prediction logic. Let us look at the cycle-level timing diagram of a conventional 1-cycle branch predictor (Figure 1-2) to see the intricacies in pipelining the branch predictor.

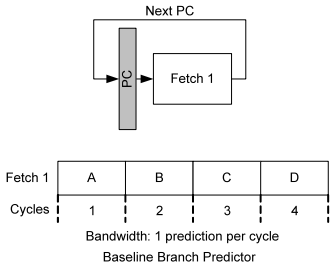

Figure 1- 2: Timing diagram for a conventional 1-cycle branch predictor.

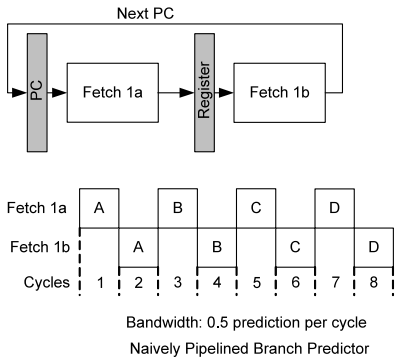

Figure 1- 3: Timing diagram for a naively pipelined 2-cycle branch predictor.

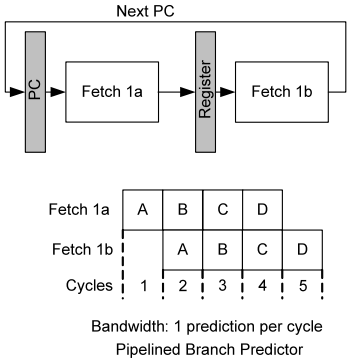

Figure 1- 4: Timing diagram for a pipelined 2-cycle branch predictor.

We can see in Figure 1-2, that the next PC loop causes a dependency between the current PC and the next PC. For example, the PC of fetch block A is used to predict the PC of fetch block B, the PC of fetch block B is used to predict the PC of fetch block C, and so on. If we naively pipeline the logic in Fetch Stage 1 into two sub-stages, as

shown in Figure 1-3, no benefit would be derived: although cycle time is ideally halved, we now need to wait 2 cycles until the prediction initiated by fetch block A produces the PC of fetch block B, and only then can fetch block B initiate the next prediction. That is, naive pipelining doubled frequency but this effect is neutralized by a reduced bandwidth of 0.5 predictions per cycle. We need to apply clever approaches to effectively pipeline the next PC loop, so that we maintain a bandwidth of 1 prediction per cycle as in the unpipelined case (without losing prediction accuracy). The implication is that we need to be able to start B before A finishes, as shown in Figure 1-4. This can be achieved if A is used to predict *C*, instead of B (B is used to predict D instead of C, and so on). This is called Block Ahead Prediction [2].

The pipelining of the next PC logic can be split into two separate parts: pipelining the BTB and pipelining the BP. Seznec et al. proposed Block Ahead Prediction [2] to effectively pipeline the BTB and BP for 2 cycles. Subsequently, Jiménez focused on pipelining just the BP and considered arbitrary pipeline depths for it. A particular implementation, called gshare.fast [3], is a pipelined version of gshare [4] with a limited number of PC bits in the index. This encouraged Seznec et al. to propose Effective Ahead Pipelining [5], which pipelined the BTB and BP separately in Block Ahead Prediction, yielding arbitrary depths for BP coupled with a two-cycle pipelined BTB.

The pipelined branch predictors cited above were shown to have close to the same accuracy as their unpipelined versions and sustain a bandwidth of one prediction

per cycle. Previous evaluations were based on cycle-level simulators implemented in high-level languages, however. Pipelining the branch prediction logic is complex, and the full extent of this complexity can only be revealed by implementing it at the level of hardware design.

Along these lines, in this thesis, I design, verify and evaluate synthesizable register-transfer-level (RTL) models of pipelined branch predictors to confirm their effectiveness in hardware, in terms of cycle time, accuracy and bandwidth. My fetch unit designs are integrated into the RTL model of an overall superscalar processor derived from the FabScalar toolset [6][7]. The motivation for RTL modeling is three-fold:

1. *Cycle time evaluation*: RTL models account for all control and datapath logic, unforeseen logic complexity that comes with pipelining the branch prediction logic, and imbalances that may arise in the pipelining of this logic. All of these factors are essential for determining the cycle time reduction of pipelining the branch prediction logic. This is not possible with abstract simulators.
2. *Validation of accuracy and bandwidth*: Due to the level of detail of RTL modeling, it is the most convincing way to confirm the accuracy and bandwidth claims of proposed techniques for pipelining the branch prediction logic.
3. *Generating synthesizable RTL designs of arbitrary fetch units for core customization*: The RTL models developed in this thesis are the basis for FabFetch, a novel tool within the FabScalar toolset [6][7] for automatically

generating synthesizable RTL designs of arbitrary fetch units, that differ in their fetch width, pipeline depth and sizes of structures (BTB, BP, and I-cache). The overall FabScalar toolset enables tailoring major dimensions of a superscalar processor (superscalar width, pipeline depth, etc.) to workloads. The FabFetch tool developed in this thesis specifically enables tailoring the major dimensions of the fetch stage. The fetch stage is critical with respect to core customization because the branch predictor's accuracy has a profound impact on exposing instruction-level parallelism and thus indirectly shapes the dimensions of other pipeline stages. In particular, having pipelined variants of the branch prediction logic opens up the design space to larger BTBs and BPs, which, in turn, expands the range of superscalar processor designs that are worth considering.

1.1 Contributions

In this thesis, I have designed, verified and evaluated RTL models of two different fetch unit designs.

The first fetch unit design implements the original form of Block Ahead Prediction [2]. In the original form of Block Ahead Prediction, the BTB and BP are pipelined together, as shown in Figure 1-5. The pipeline depth of the branch prediction logic is two cycles.

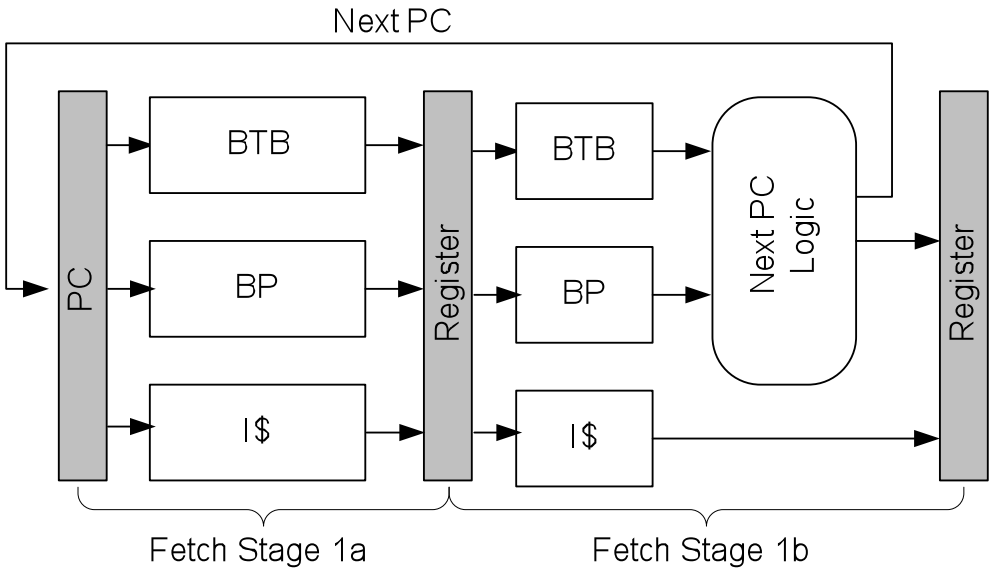

Figure 1- 5: Original form of block ahead prediction.

It is impractical to pipeline the BTB deeper than two cycles because ahead-prediction of branch targets becomes unwieldy [2][5]. The two-cycle limitation is not so much of an issue from the standpoint of the BTB because performance is not overly sensitive to the size of the BTB, unless it is grossly undersized. At most, the BTB only needs to be as large as the number of static branches in the program. Moreover, even if the BTB is mildly undersized, the penalty of a BTB miss is typically much lower than the penalty of a branch misprediction.

Unfortunately, the two-cycle limit of the first fetch unit design is a problem from the standpoint of the BP. The taken/not-taken accuracy of a contemporary BP, that exploits global branch history, is very sensitive to the size of the BP. Scaling the processor's performance is heavily dependent on the ability to scale the BP. Fortunately, the BP is more scalably pipelinable than the BTB, *i.e.*, it is relatively easier

to pipeline the BP arbitrarily deep. But first, the BP pipeline must be separated from the BTB and I-cache.

The second fetch unit design separates the BP pipeline from the BTB and I-cache, as shown in Figure 1-6. This enables the BP to scale arbitrarily large while maintaining a fast cycle time. Seznec et al. described the basic approach to separately pipelining the BP and BTB [5]. Our specific design is distinguished in that our BP is *decoupled* from instruction fetching. It is decoupled because the BP is indexed only when the current fetch block has a conditional branch in it and it is indexed by the conditional branch's PC rather than the fetch block's PC. The decoupling is facilitated by a queue between the BP and the Fetch 1 stage.

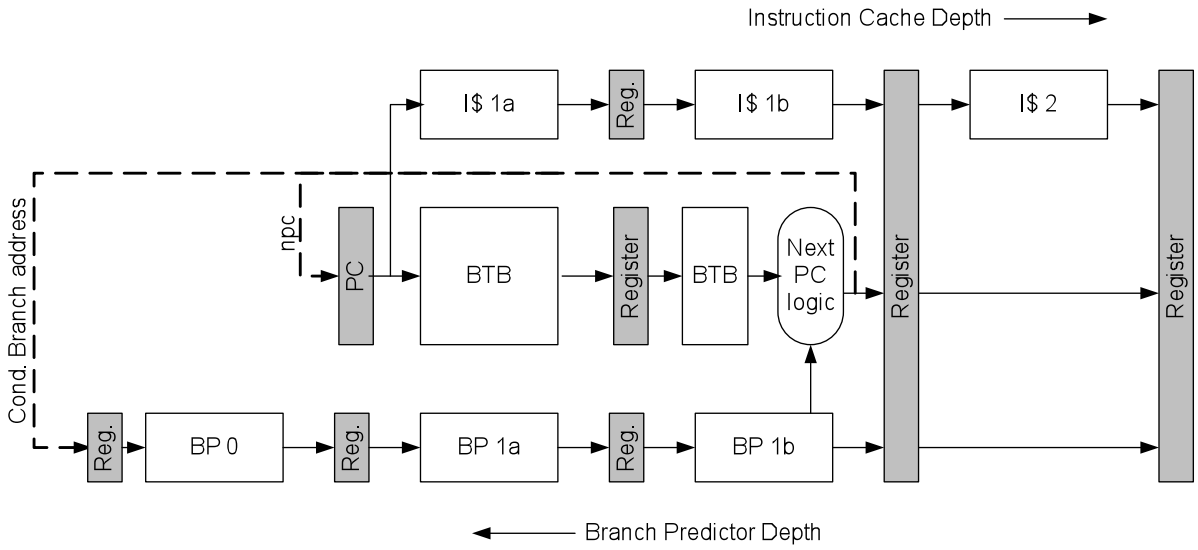

Figure 1- 6: Model of decoupled effective ahead pipelining (DEAP).

For both fetch unit designs, I did the design in two phases. In the first phase, I familiarized myself with the overall approach by implementing it in a C++ cycle-level simulator. In addition to gaining familiarity with the approach, positive results from the

first phase - a negligible drop in accuracy and a bandwidth of one prediction per cycle - motivated moving to the second phase. In the second phase, I implemented the detailed synthesizable RTL model using the Verilog hardware description language. All RTL models have been successfully integrated with FabScalar-generated cores to facilitate whole-processor simulation of standard benchmarks. Thus, the RTL models of the two fetch units have been verified in the context of whole pipelines.

The contributions of this thesis are as follows:

1. Design, verification and evaluation of a synthesizable RTL model of a 2-cycle pipelined fetch unit employing the original form of Block Ahead Prediction [2], in which the BTB and BP are pipelined together.
2. Design, verification and evaluation of a synthesizable RTL model of a fetch unit employing Decoupled Effective Ahead Pipelining (DEAP), in which there is a separate N-cycle BP pipeline. A novel aspect is the decoupling of the N-cycle BP pipeline from instruction fetching, as explained above. The N-cycle BP pipeline could conceivably be combined with either a 1-cycle or 2-cycle BTB pipeline. The current design only implements the former and the latter is left for future work.
3. FabFetch, a novel toolset within the FabScalar framework [6][7]. With FabFetch, we can automatically generate synthesizable RTL designs of arbitrary fetch units, which differ in their fetch width, pipeline depth and sizes of structures. FabFetch adds multiple dimensions to the fetch stage in the

FabScalar framework. The FabFetch-generated designs have been verified by simulating them with FabScalar-generated cores for 100 million instruction Simpoints [1] of 6 SPEC 2000 integer benchmarks.

For the 2-cycle pipelined fetch unit employing Block Ahead Prediction [2], a 40% reduction in cycle time was observed for all fetch widths with a meager 1.6% drop in instructions per cycles (IPC) with respect to the baseline 1-cycle predictor. In the fetch unit featuring DEAP, accuracy decreases by less than 0.6% and IPC decreases by only 3.3% due to pipelining the BP for 4 cycles. DEAP can be used to pipeline the BP for many cycles without much loss in IPC and accuracy. Moreover, due to pipelining, we get a many-fold increase in frequency. Both of the above implementations are able to sustain a bandwidth of 1 prediction per cycle.

1.2 Outline

I will first go through the methodology adopted to implement and study the pipelining techniques, in Chapter 2. It will be followed by a description of the baseline fetch unit design, in Chapter 3. This is the design against which we will be comparing our results. We will see how the I-cache can be pipelined and its performance impact, in Chapter 4. I will present the design of the Block Ahead Predictor along with its performance results in Chapter 5. I will describe the design of decoupled effective

ahead pipelining followed by its performance results in Chapter 6. I will finally summarize and look at future work in Chapter 7.

CHAPTER 2

Methodology

I have used the FabScalar framework [6][7], which provides synthesizable RTL implementations of complete superscalar cores following a canonical structure. The canonical structure of FabScalar is given in Figure 2-1. In addition to RTL models of superscalar cores, the FabScalar framework comes with a configurable C++ cycle-level simulator.

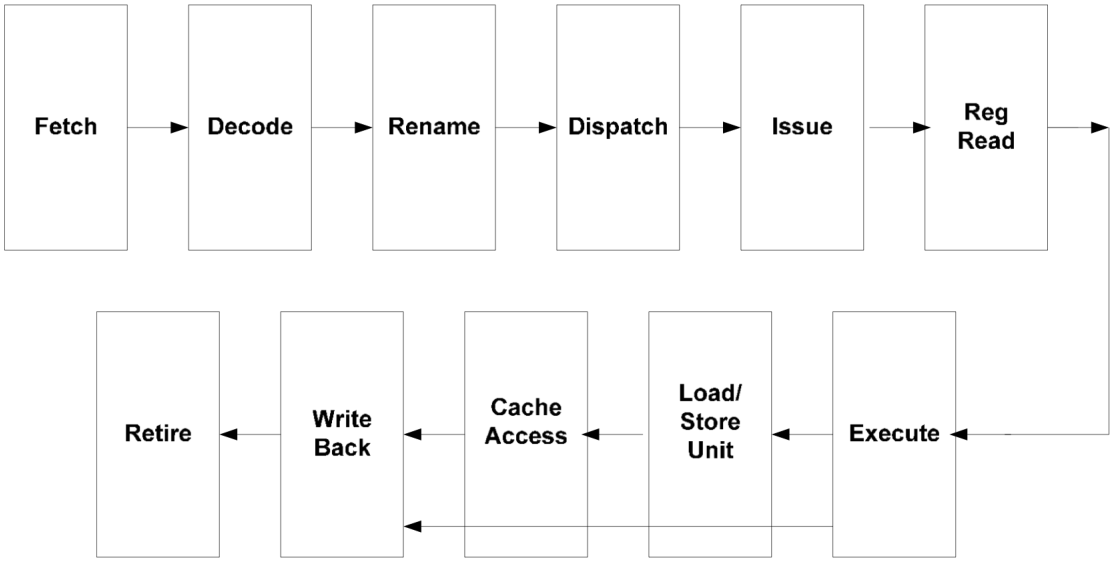

Figure 2- 1: Canonical pipeline stages of a superscalar processor [6].

I designed various pipelined versions of the fetch stage to support large branch predictors with a fast cycle time. I first implemented various pipelined fetch units in the C++ simulator to understand the logic involved and also quickly evaluate their performance, as a precursor to designing the corresponding RTL models. After studying

the impact of pipelining the branch prediction logic in the C++ simulator, the synthesizable RTL model was implemented using the Verilog hardware description language. The synthesizable RTL model was then translated into scripts for automatically generating synthesizable RTL models of the fetch stage with different pipeline depths, fetch widths, and structure sizes. These scripts constitute the FabFetch tool. FabFetch was integrated with other FabScalar scripts for generating whole cores for simulation and verification.

2.1 Functional Verification and Synthesis

I used the EDA tools shown in Table 2-1 for functional verification and synthesis.

Table 2-1: EDA environment for ASIC flow.

Phase	EDA tool used
functional verification	Cadence NC-Verilog, vers. 06.20-s006
synthesis and timing	Synopsys Design Compiler, vers. X-2005.09-SP3

For functional verification, I used the C++/Verilog co-simulation environment that came with the FabScalar toolset. The co-simulation environment is a functional simulator (does not model the processor) written in C++ that runs concurrently with the Verilog simulator of the superscalar processor. The two are independent and the functional simulator assists in checking and debugging the Verilog simulator by comparing the results of instructions as they retire from the processor. Verilog simulation results are presented for the 100 million instruction SimPoints [1] of six SPEC2000 integer benchmarks. The other six integer benchmarks have some floating-

point instructions in their SimPoints and FabScalar-generated cores do not have floating-point support as of now.

We used the FreePDK 45nm technology and standard cell library [8] for synthesis and timing. Custom-designed RAMs are an essential part of the fetch stage: they make up the BTB and BP tables. For the purposes of functional verification, RAMs are implemented as behavioral modules in the Verilog model. For synthesis and timing, however, the behavioral modules are replaced by LEF hard macros to properly model their access times. I used the FabMem tool [9] of the FabScalar toolset to obtain access times for RAM structures.

2.2 Variants of Pipelined Fetch Units

FabFetch produces synthesizable RTL models of different fetch units. Each fetch unit has different degrees of pipelining key structures in the Fetch-1 stage. Table 2-2 shows (1) three fetch unit designs, (2) which of the three designs are supported by the C++ cycle-level simulator and the Verilog simulator, and (3) the depth of pipelining the BTB, BP, and I-cache in each design.

Table 2- 2: Fetch unit designs generated by FabFetch.

Fetch Unit Design	Simulators		Depth			Fetch Width
	C++	Verilog	BTB	BP	I-cache	
Pipelined I-cache	✓		1	1	N	1 to 8
Block Ahead Predictor	✓	✓	2	2	2	1 to 8
Decoupled Effective Ahead Pipelining (DEAP)	✓	✓	1	N	1	1 to 8

2.3 Branch Predictors and Performance Metrics

To evaluate the performance of pipelining the branch prediction logic, I will use a large BP having 2^{16} entries. Since each entry is a 2-bit counter, the BP has a cost of 16KB. I will use two different branch prediction algorithms: bimodal and gshare [4].

Pipelining the large BP may degrade its accuracy. Nonetheless, in general, its accuracy is still significantly higher than the accuracy of a smaller 1-cycle BP. I will measure the reduction in accuracy due to pipelining the BP.

The decrease in accuracy due to pipelining the BP causes a reduction in instructions-per-cycle (IPC). But, the loss in accuracy is compensated by a large boost in frequency that in turn causes improvement in overall performance of the processor. Therefore, to measure the overall performance improvement I will use two major metrics:

1. Instructions-per-cycle (IPC)
2. Cycle time

CHAPTER 3

Baseline Fetch Unit

The baseline fetch unit consists of two stages: Fetch Stage 1 and Fetch Stage 2. Each of them is 1-cycle deep. Figure 3-1 shows the model of the baseline fetch unit.

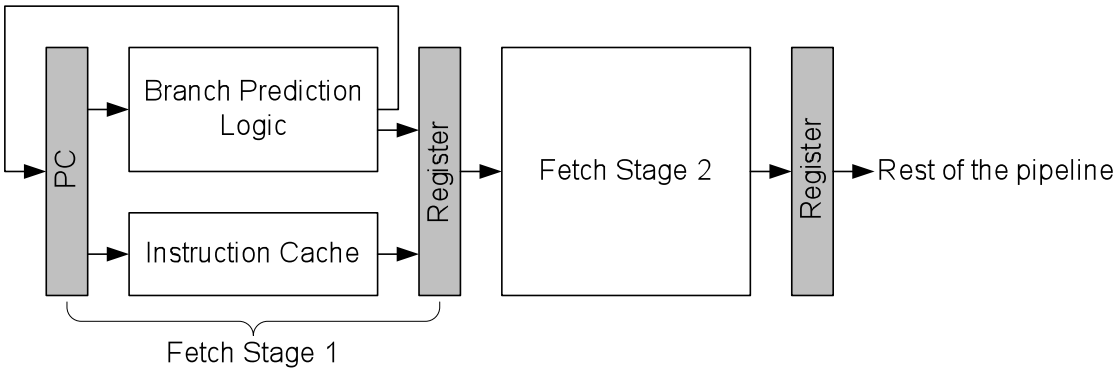

Figure 3- 1: Baseline instruction fetch unit.

Fetch Stage 1 consists of the branch prediction logic, along with the logic for fetching instructions from the instruction cache (I-cache). The I-cache and branch prediction logic in Fetch Stage 1 are shown separately since they will be pipelined separately in later chapters. Fetch Stage 2 has the logic for extracting the fetch block from the instructions supplied by the I-cache as well as the logic for recovering from a BTB miss.

3.1 Design of Fetch Stage 1

The basic function of the fetch unit is to produce an address every cycle to fetch instructions from the instruction cache (I-cache) and to fetch instructions from the I-

cache to feed them to the rest of the pipeline. Figure 3-2 shows the design that produces the next PC in Fetch Stage 1.

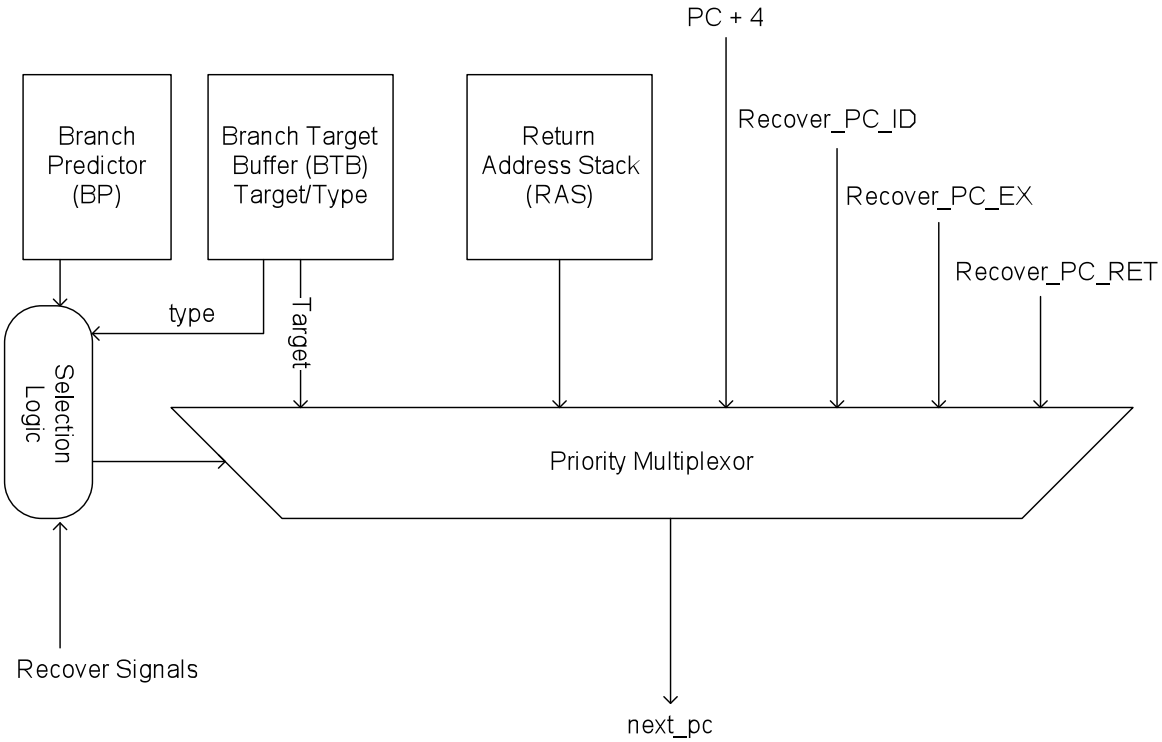

Figure 3- 2: Next PC logic in the baseline Fetch Stage 1.

The I-cache is accessed in parallel with the next PC being generated. The problem is that we need to know which instructions being fetched from the I-cache are branches, if any, and what are their target addresses, to produce the PC of the next fetch block in the next cycle, so as not to delay fetching the next fetch block. The BTB is the table that stores this information. The BTB stores the branch PC with its taken target and its type (jump, call, return, or conditional branch). If a PC hits in the BTB, it will prove that the corresponding instruction is a branch and will provide its taken target and its type. This information is used to select the next PC. For a 4-wide fetch

unit, we need to check for BTB hits for 4 consecutive PCs. The banked BTB design shown in Figure 3-3 facilitates the multiple PC checks in the BTB with just 1 read port in each of 4 banks.

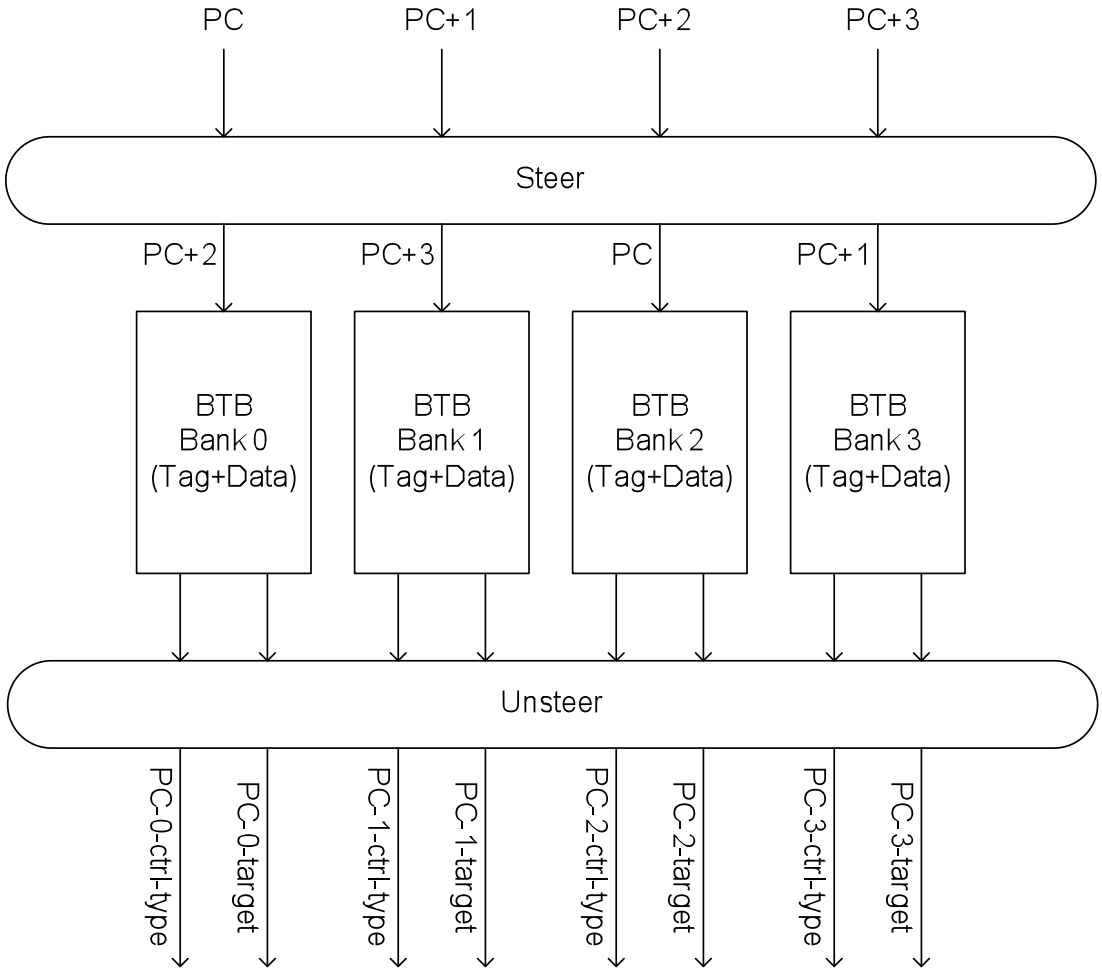

Figure 3- 3: Baseline BTB design for a 4-wide fetch processor.

The BTB gives the taken targets for calls, jumps and conditional branches. For deciding whether conditional branches are taken or not taken, the BP gives a prediction for each conditional branch among the fetched instructions. For the baseline design, we use a table of two-bit Smith Counters indexed by the PC or the so-

called bimodal predictor [10]. We need to access multiple counters in order to get predictions for all the instructions being fetched in the current cycle. If the BTB hits for a particular instruction and it signals that the instruction is a conditional branch, then the respective branch prediction is selected and is used to determine the next PC. The design shown in Figure 3-4 is used to produce 4 predictions per cycle for a 4-wide fetch unit.

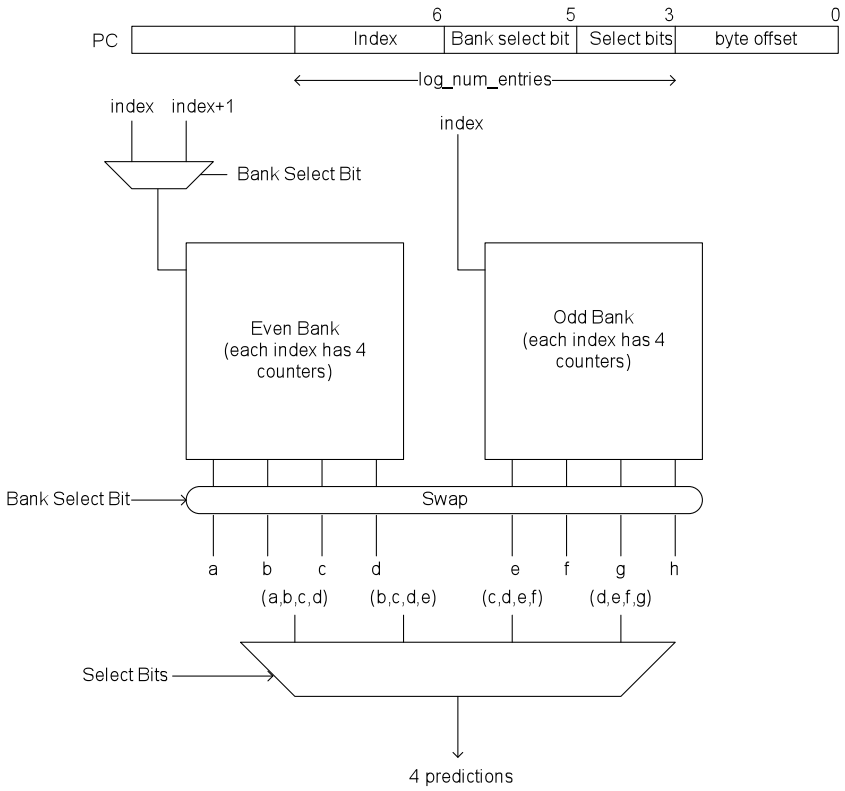

Figure 3- 4: Baseline BP design for a 4-wide fetch processor.

The target addresses for calls and jumps are given by the BTB directly. The target addresses for conditional branches can be selected (BTB target or fall-through) as per the prediction given by the predictor employed. Return targets are a little difficult

to predict with a BTB. A function can be called from many call sites and the branch prediction logic has to produce the return site corresponding to the call. Calls and returns follow a simple last-in-first-out (LIFO) behavior. Thus, we employ a small structure called the Return Address Stack (RAS) [11] to predict the targets of returns. Calls pushes the address of the return site onto the stack and returns pop them out. Thus, when the BTB signals that the instruction being fetched is a return, it takes the target provided by the RAS. Figure 3-5 shows the priority logic in detail that is shown in Figure 3-2.

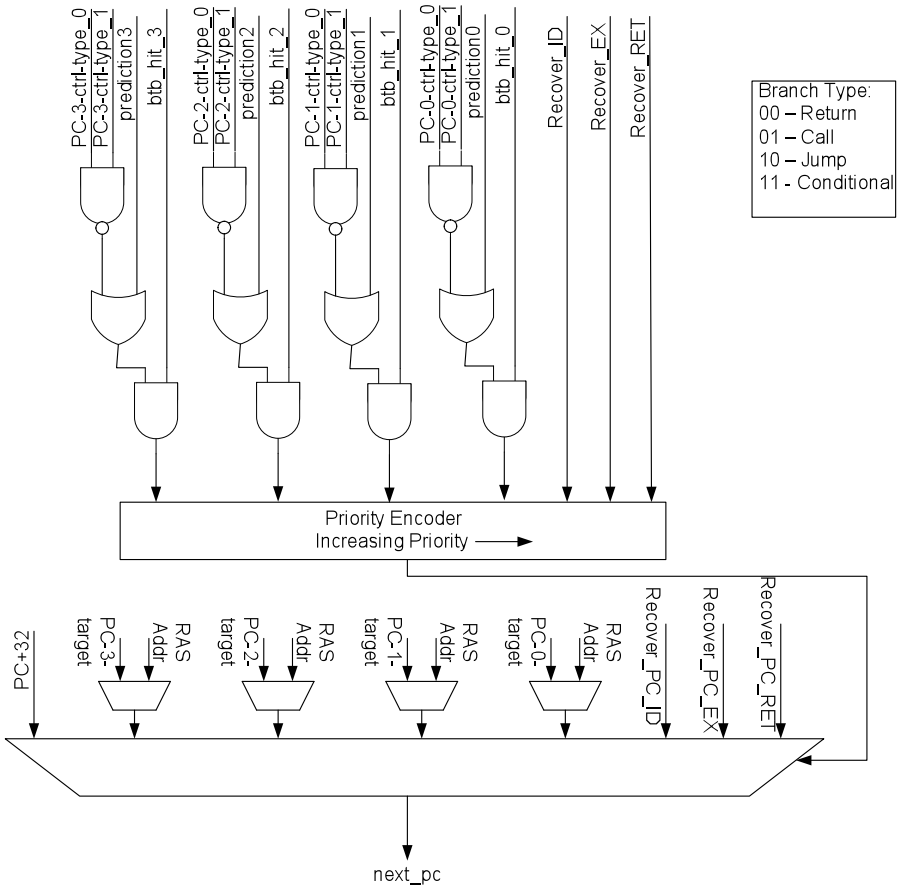

Figure 3- 5: Priority logic for generating the next PC.

The I-cache is designed to fetch N sequential instructions per cycle where N is the fetch width of the processor (N=4 in the baseline fetch unit). To achieve this for any fetch block PC (even PCs not aligned on cache line boundaries), the I-cache has odd and even banks that each supply N instructions, which is conceptually similar to the BP organization. Figure 3-6 show the I-cache datapath for a 4-wide instruction fetch unit and its distribution between the Fetch-1 and Fetch-2 stages. One full cycle is given for the I-cache access and the swap of the cache lines from the two banks in the correct order. The alignment of instructions is done later as a part of Fetch-2.

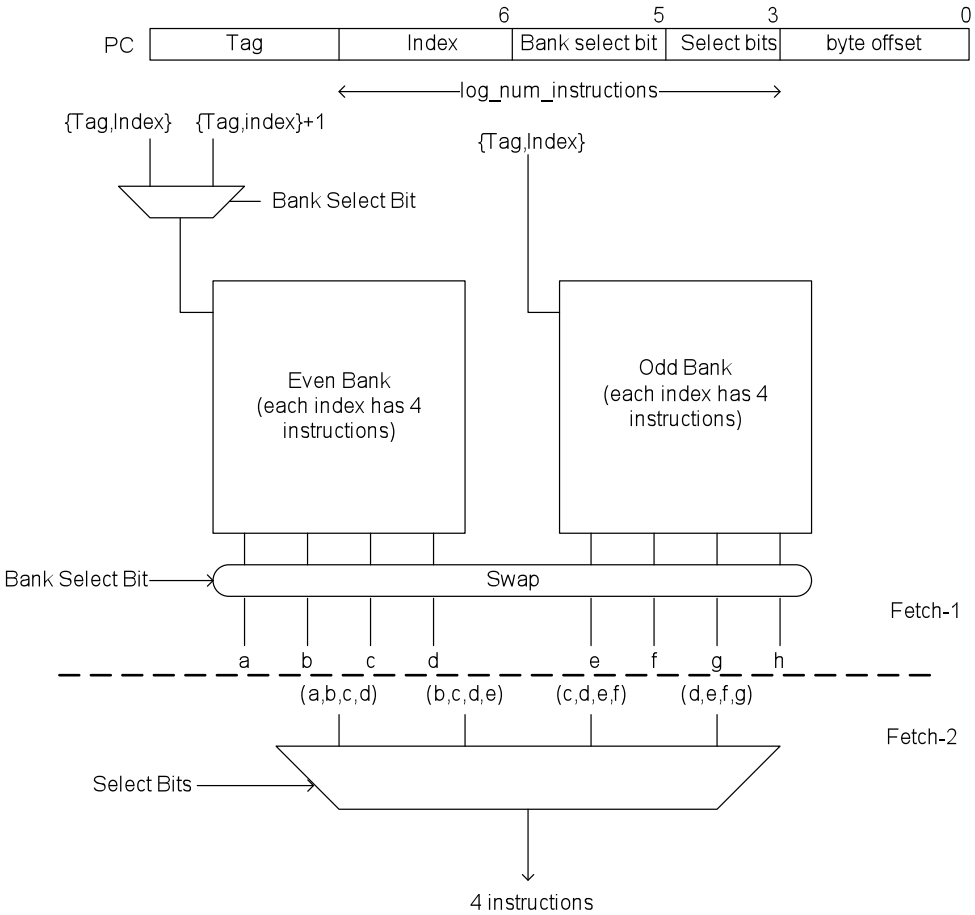

Figure 3- 6: Design of instruction cache for a 4-wide fetch processor.

3.2 Design of Fetch Stage 2

Fetch Stage 2 has the logic for extracting the fetch block from the instructions supplied by the I-cache as well as the logic for recovering from a BTB miss. The instructions that are fetched and aligned in this cycle are pre-decoded. Pre-decode is used to detect branches and to calculate their target addresses. In case the BTB missed on the first predicted-taken branch, if any, among the fetched instructions, then a recovery signal is sent to Fetch Stage 1 to restart from the missed branch's taken-target. Fetch Stage 2 also maintains a FIFO Branch Queue (CTIQ), which has all the in-flight branches in the processor for (1) updating the BP non-speculatively in program order and (2) rolling back BP state such as the global history register (GHR) if used by the branch prediction algorithm. It also maintains a checkpointed copy of the Return Address Stack (RAS). It is required to correct the RAS when there is a BTB miss on a call or return. It sends the required instruction packet (i.e., the fetch block) down the pipeline.

3.3 Performance of Baseline Fetch Unit

We will quantify the performance of the baseline 4-wide fetch unit in a baseline 4-issue superscalar processor as proposed in [6]. Four measurements are collected in the Verilog simulator, and shown in Table 3-1: performance in instructions per cycle (IPC), misprediction rate of conditional branches, misprediction rate of all other branches (jumps, calls, and returns), and overall branch misprediction rate.

Table 3- 1: Performance of the baseline 4-wide fetch unit in a baseline 4-issue superscalar processor.

Benchmark	IPC	Cond. Branch Misprediction Rate	Misprediction Rate of all other branch types	Overall Branch Misprediction Rate
bzip	0.76	10.01	26.71	10.99
gap	0.73	9.47	65.23	16.40
gzip	0.68	10.87	52.88	14.10
mcf	0.74	12.42	43.99	12.52
parser	0.76	10.83	26.04	12.21
vortex	0.76	0.52	79.74	8.80

We can see that the bimodal predictor generally reaches a 90% accuracy for conditional branches. We can also see that the rest of the branches are more difficult to predict, like returns via the RAS and indirect jumps and calls via the BTB; although they are less frequent than conditional branches, these are the next bottleneck in restricting the accuracy of the speculative window.

CHAPTER 4

Pipelining the Instruction Cache

The instruction fetch unit is like the heart of the processor that is pumping instructions into it. The I-cache is the component that provides a continuous supply of instructions to the instruction fetch unit to keep the processor alive. Instruction cache misses are like heart attacks for the processor. The front-end has to stall and the processor is left with the remaining in-flight instructions to be executed until the miss is resolved. To minimize the effect of I-cache misses on processor performance, the I-cache needs to be large enough so that it can hold the footprint of the application code that the processor is running. With aggressively pipelined processors, it is possible that a sufficiently-large I-cache cannot be accessed in 1 cycle. It needs to be pipelined into several stages. The idea is to pipeline the RAM of the cache. But, the logic of Fetch-1 cannot be changed or pipelined, since we need to get a next PC every cycle. So, we pipeline the I-cache as shown in Figure 4-1.

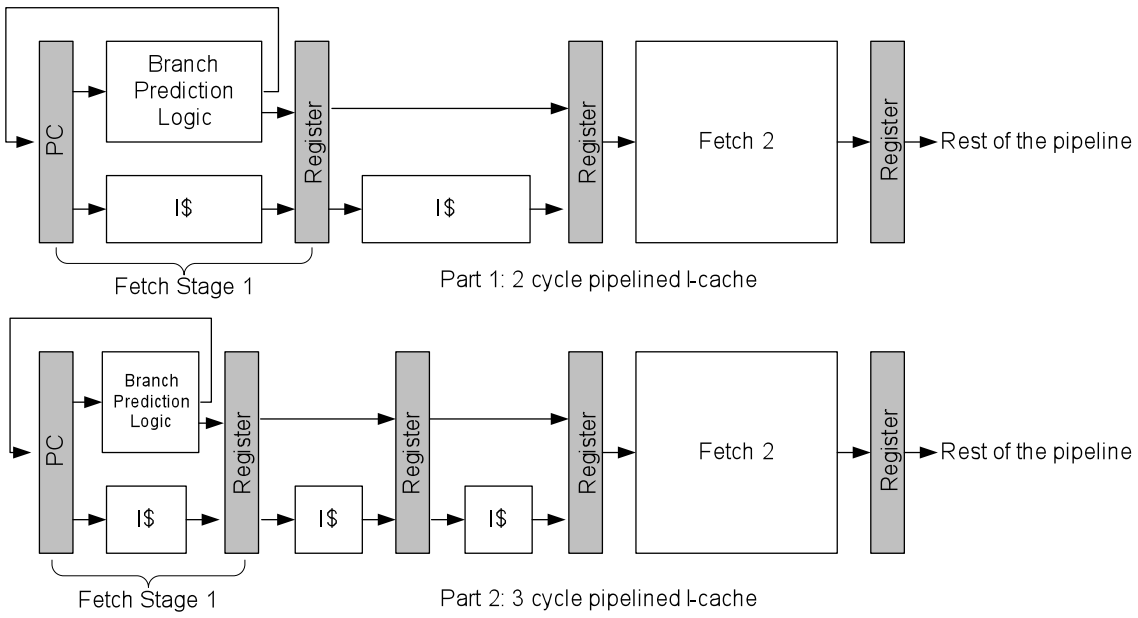

Figure 4- 1: Model of instruction fetch unit with pipelined instruction cache.

4.1 Performance Impact of Pipelined Instruction Cache

Pipelining the I-cache adds cycles between Fetch Stage 1 and Fetch Stage 2, as shown in Figure 4-1. Fetch Stage 2, which detects BTB misses, is delayed until we get the instructions from the I-cache. Thus, the loop that corrects BTB misses, from Fetch Stage 2 to Fetch Stage 1, increases from 1 cycle to X cycles where X is the pipeline depth of the I-cache. In other words, the BTB miss penalty (in cycles) is increased. The impact of increasing the BTB miss penalty is most noticeable in applications suffering frequent BTB misses.

The major impact is seen on the misprediction loop. Every branch instruction will take more cycles to reach the execute stage and resolve its target address. But if

our prediction is most of the time accurate, then the impact of pipelining the I-cache can be hidden.

To study the impact of lengthening the BTB miss and branch misprediction loops, let us take a 4-issue superscalar processor with ideal caches (always cache hit) and change the pipeline depth of the I-cache. The following experiments were performed with the C++ cycle-level simulator. The BTB has 512 entries and is 4-way set-associative. The branch predictor is 2^{16}-entry gshare. The results are shown in Table 4-1 and Figure 4-2.

Table 4- 1: IPCs with different pipeline depths of the I-cache (512-entry BTB).

Benchmark	**I-cache Depth**					
	1	**2**	**3**	**4**	**5**	**6**
bzip	1.15	1.12	1.08	1.06	1.04	1.01
crafty	0.95	0.9	0.86	0.81	0.77	0.72
gap	1.12	1.07	1.01	0.96	0.91	0.87
gcc	1.12	1.04	0.96	0.88	0.82	0.76
gzip	1.02	0.99	0.96	0.89	0.86	0.83
mcf	1.44	1.43	1.41	1.39	1.36	1.32
parser	1.12	1.09	1.01	0.98	0.94	0.91
perl	0.98	0.93	0.87	0.82	0.77	0.72
twolf	0.96	0.92	0.89	0.85	0.82	0.78
vortex	1.34	1.29	1.21	1.11	0.89	0.82
vpr	1.27	1.24	1.21	1.18	1.16	1.14

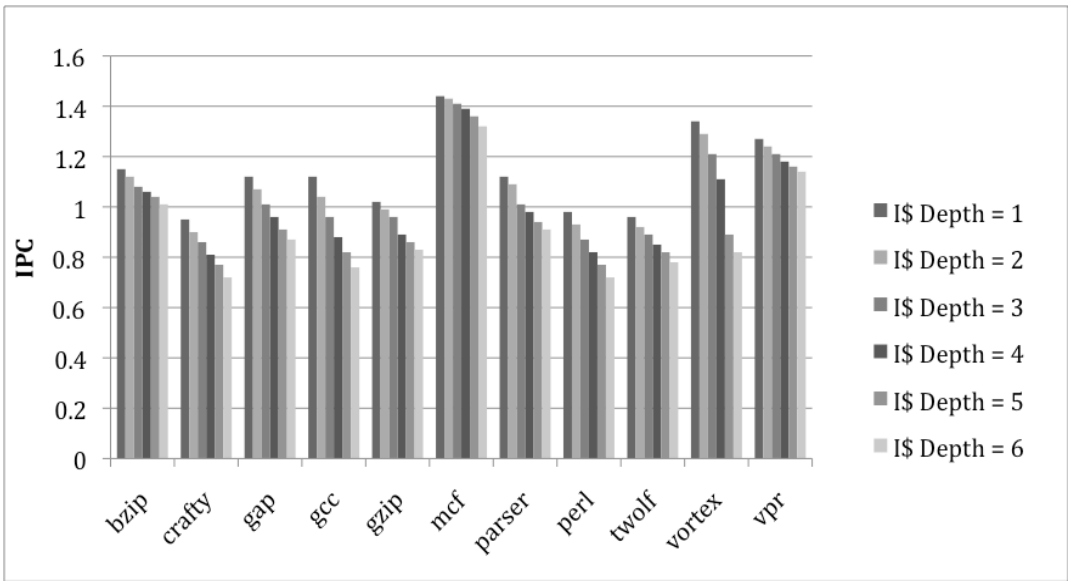

Figure 4- 2: IPCs with different pipeline depths of the I-cache (512-entry BTB).

We can see from the experiment that if we have ideal caches, then the effect of pipelining the I-cache has a big impact on the performance on the processor. We need to quantify whether this is due to increasing the BTB miss loop or the branch misprediction loop. For that, we will consider a 32K-entry 4-way set-associative BTB, larger than in the previous experiment, and again vary the I-cache depth.

Table 4- 2: IPCs with different pipeline depths of the I-cache (32K-entry BTB).

Benchmark	I-cache Depth					
	1	2	3	4	5	6
bzip	1.15	1.12	1.08	1.06	1.04	1.01
crafty	0.96	0.93	0.9	0.88	0.85	0.83
gap	1.13	1.09	1.05	1.00	0.96	0.92
gcc	1.16	1.12	1.09	1.04	1.00	0.96
gzip	1.03	1.00	0.97	0.93	0.87	0.85
mcf	1.44	1.43	1.41	1.39	1.36	1.32
parser	1.12	1.09	1.01	0.98	0.95	0.92
perl	1.00	0.96	0.93	0.89	0.86	0.81
twolf	0.96	0.93	0.9	0.88	0.85	0.82
vortex	1.36	1.3	1.24	1.14	0.93	0.85
vpr	1.27	1.25	1.22	1.19	1.17	1.14

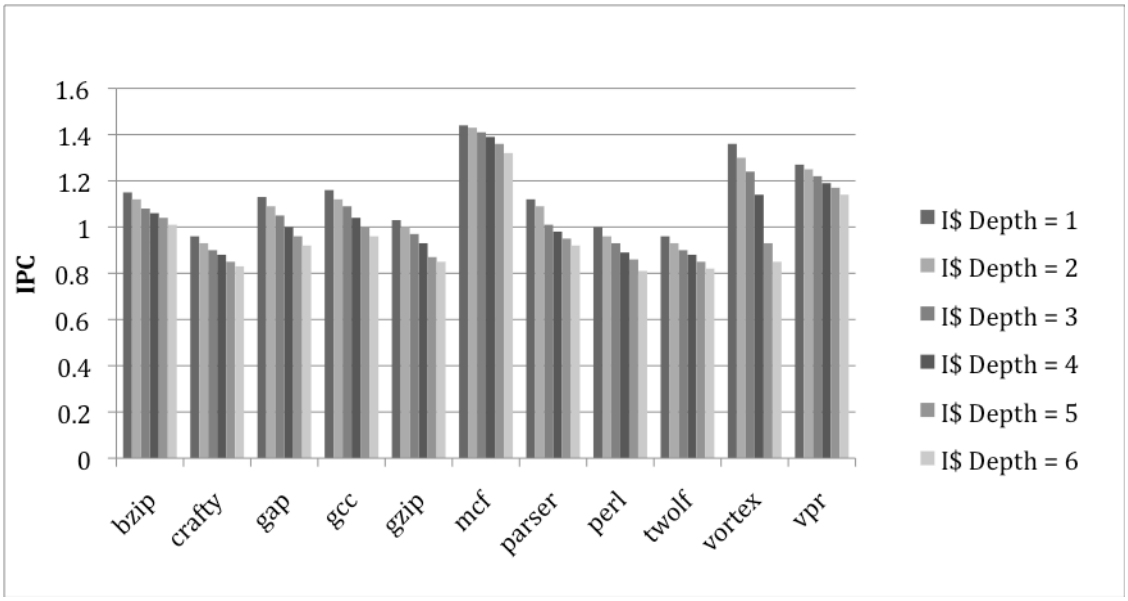

Figure 4- 3: IPCs with different pipeline depths of the I-cache (32K-entry BTB).

Since we still observe significant IPC degradation even with a much larger BTB, we infer that for all the benchmarks the major drop in performance is due to the increase in the misprediction loop.

Table 4- 3: BTB hit rate for a 512-entry 4-way set-associative BTB.

Benchmark	BTB Hit Rate
bzip	99.97
crafty	80.87
gap	90.97
gcc	74.89
gzip	99.99
mcf	100.00
parser	99.87
perl	86.93
twolf	93.30
vortex	96.63
vpr	100.00

To confirm this, let us see the BTB hit rates of all the benchmarks with a 512-entry 4-way set-associative BTB, shown in Table 4-3. Consider benchmarks with a high BTB hit rate. For these benchmarks, there is little impact due to increasing the BTB miss loop as evident from no difference in performance in Table 4-1 and Table 4-2. But in benchmarks like gcc, which has a high BTB miss rate, the impact of extending the BTB miss loop is higher. Summing up, if the BTB is sufficiently large, the impact on performance due to a pipelined I-cache will mostly be driven by the depth of misprediction loop.

Ultimately, since the I-cache is the heart of the processor that keeps on pumping instructions, we need to make it large enough to hold the program footprint but not so large that its benefit is overtaken by the disadvantages of pipelining it to preserve the processor's cycle time.

CHAPTER 5

Block Ahead Pipelining

In the baseline fetch unit, the branch prediction logic produces the next PC in 1 cycle. There is a need to pipeline the branch prediction logic to accommodate a large BTB and BP. The model of Fetch Stage 1 for block ahead prediction is shown in Figure 5-1. The idea and implementation of block ahead prediction was introduced in [2].

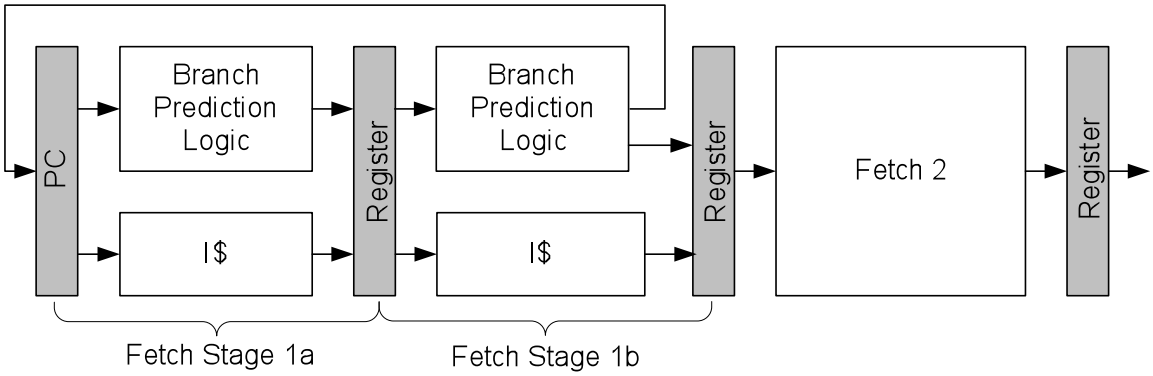

Figure 5- 1: Model of instruction fetch unit for block ahead prediction.

We can see from the model that if we pipeline in this format, then we need to use an older PC to predict the next block of instructions. The difference between a conventional fetch unit and block-ahead fetch unit is shown with the help of fetch blocks in Figure 5-2. To explain the blocks being fetched every cycle shown in Figure 5-2, we adopt the same naming convention used in [2]. This convention will be used throughout this chapter:

1. *Z names the fetch block.*

2. *Zi is the PC of the 1st instruction in the fetch block Z.*

3. *Zz is the PC of the last instruction in the fetch block Z.*

4. *z is the size of the fetch block Z.*

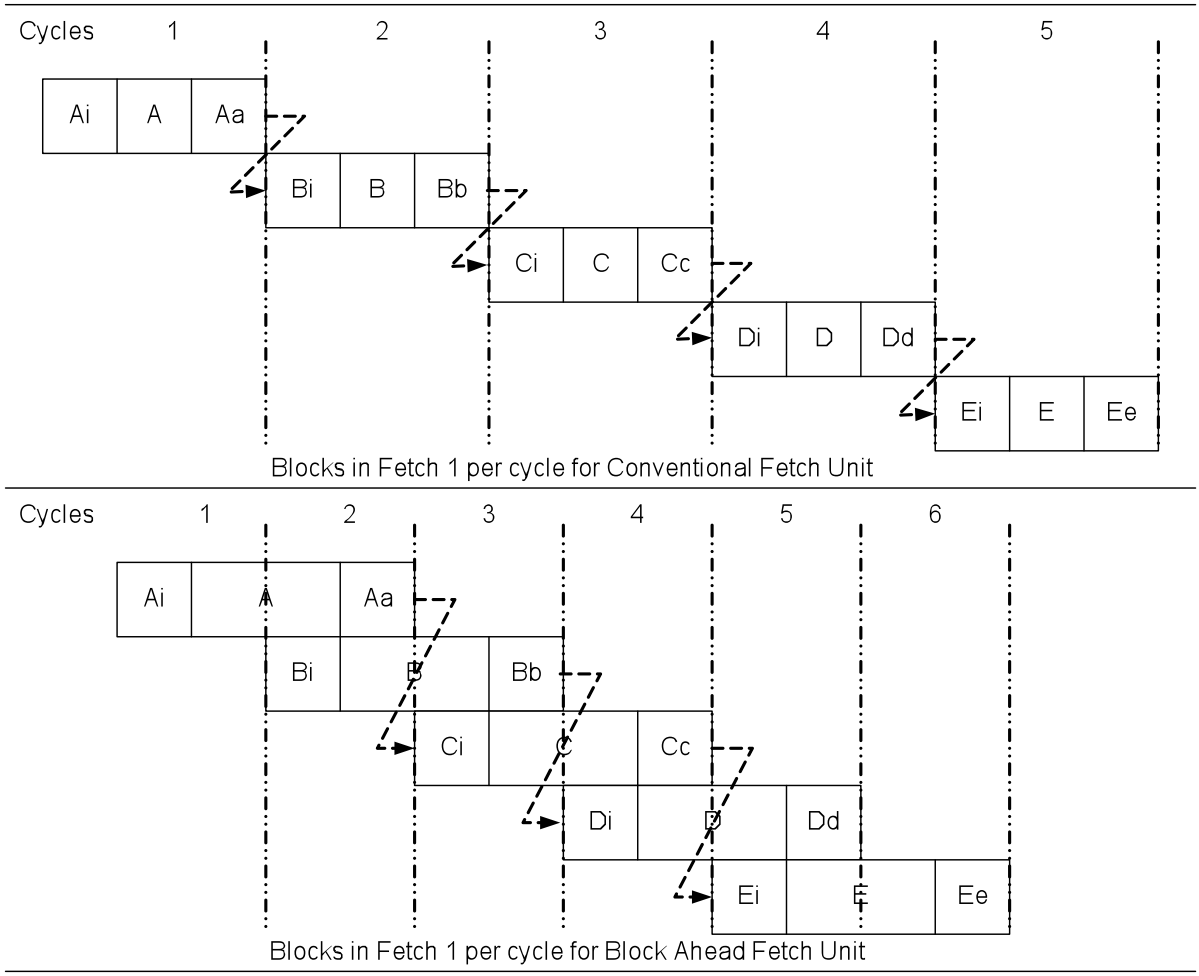

Figure 5- 2: Blocks being fetched every cycle for conventional vs. block-ahead fetch unit.

Let us see how a conventional BTB works. Let us take the block A from the above figure. In case Aa is a control instruction, its BTB hit would provide:

(Aa ==> Target address of Aa, Type of control instruction for Aa)

Similarly, to get the prediction for the control instruction Aa for the conventional case, we will use the index of Aa and get the counter from the predictor table (assuming bimodal predictor). This information is used by the Fetch Stage 1 priority logic to produce the address Bi, the start PC of the next fetch block B.

Such an organization does not work for the block-ahead fetch unit. If we look at Figure 5-2, the BTB access and the predictor access for Aa does not finish in time to predict Bi, the start PC of the next fetch block B. It can only be used to predict the start PC of B's successor, Ci (start PC of fetch block C). This is the link shown in Figure 5-2 (Aa predicting Ci). We need to design the BTB and the branch predictor to help in producing the program counter Ci using Aa in two cycles. Throughout the next section we will overhaul the design of the baseline fetch unit and all the structures associated with it to support 2-cycle block-ahead prediction.

5.1 Design of Block-Ahead Fetch Unit

In this section, I will discuss each and every structure and design them in detail, so that we can in the end build the whole system to support block ahead prediction with these components.

5.1.1 Design of Branch Target Buffer

To make the BTB work as we discussed in the section above, we need to see how we can change the current BTB design to make it possible. Let us take a control-flow graph of a program with conditional branches ending the fetch blocks.

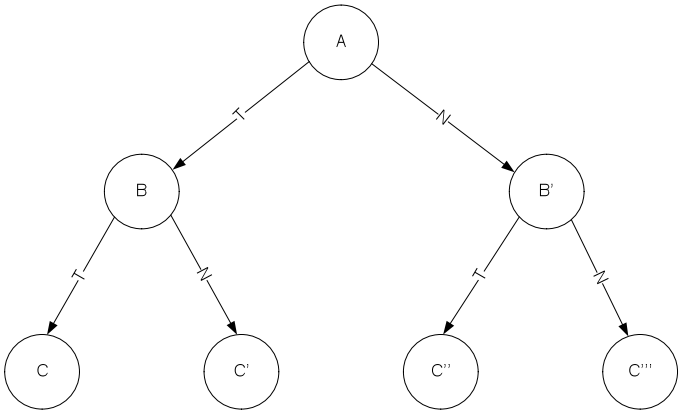

Figure 5- 3: A control-flow graph of a program.

In the conventional BTB case, there will be 2 BTB entries for the targets of B and B′. These entries are:

1. Bb ➔ Ci, conditional branch (type)
2. B′b′ ➔ C′′i, conditional branch (type)

We need to predict the start addresses of all 4 possibilities from Aa as per the direction taken by Aa and prediction of Bb or B′b′. The 4 possibilities are Ci, C′i, C′′i and C′′′i.

If we are able to change the Tag of the conventional BTB to detect the cases, then we will be able to produce all the 4 possibilities. To represent Bb, we can move it back to Bi, since we allow only 1 branch per fetch block. Aa can determine Bi and B′i, if we also give what direction it will take: T and N, respectively. There will be 2 entries in our new BTB:

1. {Aa,T} ➔ Ci, b, conditional branch
2. {Aa,N} ➔ C′′i, b′, conditional branch

Here b and b′ are the fetch block lengths of B and B′. Let us see why we need the fetch block lengths. Suppose there is fetch block X preceding fetch block A, and it did hit in the BTB. The BTB hit of fetch block X would provide *a* (the size of fetch block A). When we start to access the BTB and the BP, we start with Ai and not Aa, so we access a fetch-width number of entries in both the structures and post-select when fetch block X sends the information about its length *a*. Figure 5-4 shows how the forwarding of the fetch block size works.

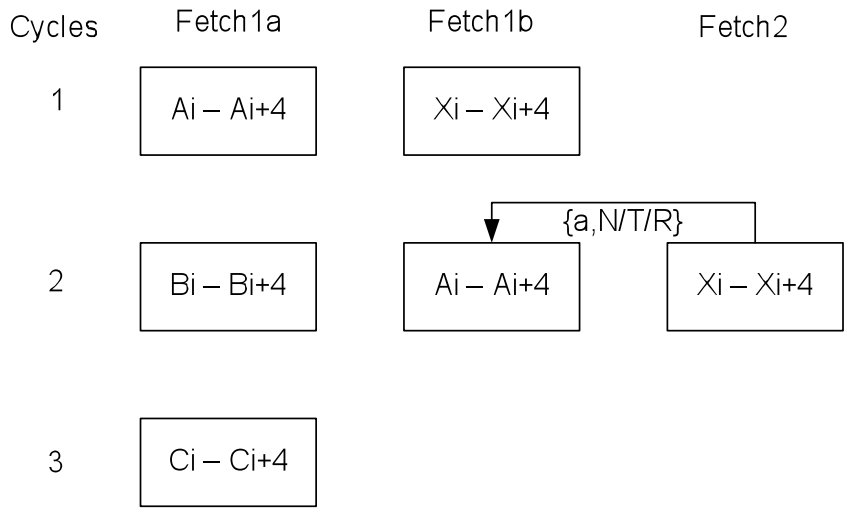

Figure 5- 4: Fetch block length being forwarded for post-selection in Fetch Stage 1.

Thus, our BTB tag is composed of the last PC of the previous fetch block and the transition taken by the last PC of the previous fetch block. (Going back to our example, fetch block A is the previous fetch block with respect to both fetch blocks B and B′, hence, the BTB tags for B and B′ are {Aa,T} and {Aa,N}, respectively.) There are three types of transitions possible: taken (T), not taken (N) and return (R). The use of return (R) will be explained in the next Subsection 5.1.2. The information stored in the

BTB for the current fetch block will be the taken-target address of its branch, the type of its branch and the position of its branch in the fetch block (i.e., fetch block size). This is the same information in a conventional BTB entry except for the fetch block size. The number of bits added to an entry increases by just two bits for a 4-wide fetch unit since we need two bits to indicate the position of the branch in the fetch block. Also, two bits are added per entry in the tag to indicate the transition type of the previous fetch block's branch.

Tag + Index	**Information**
Aa,T	Ci,b,conditional

Figure 5-5 shows the critical-path BTB read that is pipelined into two cycles. We will see how the priority multiplexor changes later (Section 5.1.6).

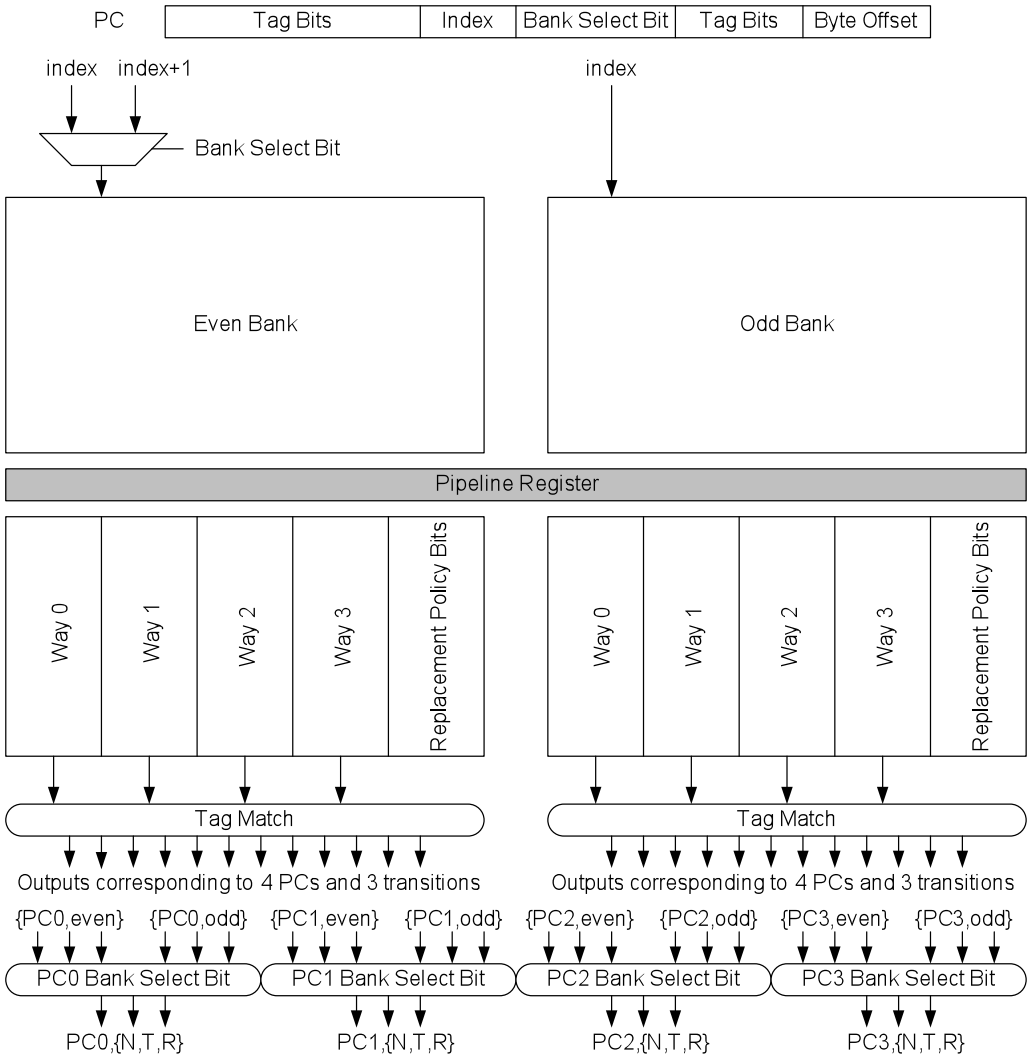

Figure 5- 5: BTB read for a 2-cycle 4-wide fetch unit with block ahead prediction.

5.1.2 Design of Return Address Stack

Let us see the case of a conventional RAS and then derive what cases would be needed with a block ahead scheme.

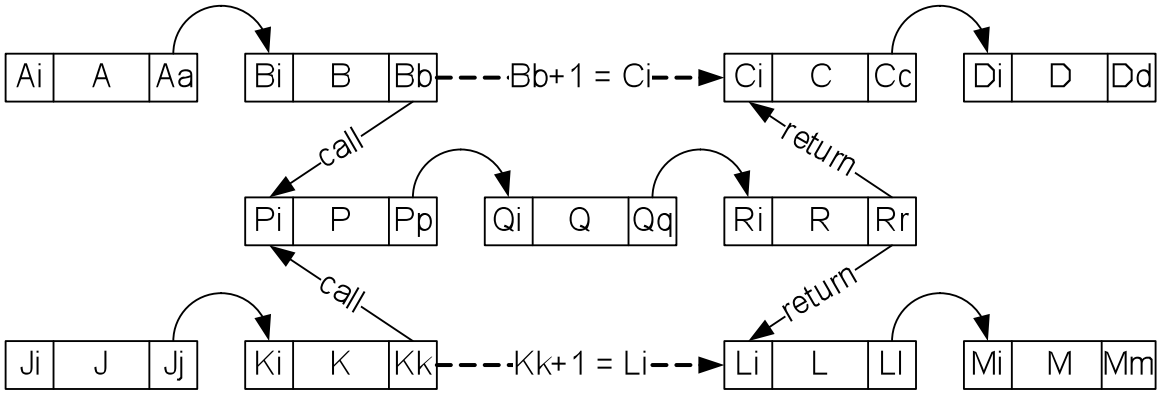

Figure 5- 6: Program flow with two call-sites (Bb and Kk) to a function starting with Pi.

Let us take the above figure as reference and see how the RAS works. There are two call-sites in the figure to the function starting with Pi. There are two BTB entries corresponding to Bb and Kk, which indicate that the control instructions are of the call type and provide their targets (Pi). Thus, the RAS will push Bb+1 (Ci) or Kk+1 (Li) according to call-site. When the function ends, the BTB hit provides that the control instruction (Rr) is of the return type, but the target for a return instruction is not fixed. The return target depends on the address from where the call was made. Thus, we pop an address from the RAS, which will give us the return target for the return. The RAS is a LIFO structure since nested calls and returns follow that format.

Let us enumerate how the 2-cycle block ahead prediction needs to predict the addresses.

1. Aa/Jj ➔ Pi
2. Bb/Kk ➔ Qi
3. Qq ➔ Ci/Li

4. Rr ➔ Di/Mi

Ci and Li are the addresses that are predicted by the RAS. These are the return addresses for the two call sites Bb and Kk, respectively. We need a mechanism through which we can predict Di and Mi from Rr. These are also dependent on the call sites Bb and Kk, respectively. Thus, we need two stacks to handle both the cases. The second stack is for predicting Di and Mi from the return instruction Rr. The second stack is called the Secondary Address Stack (SAS) [2]. The calls and returns are handled in 2-cycle block ahead prediction as shown in Figure 5-7.

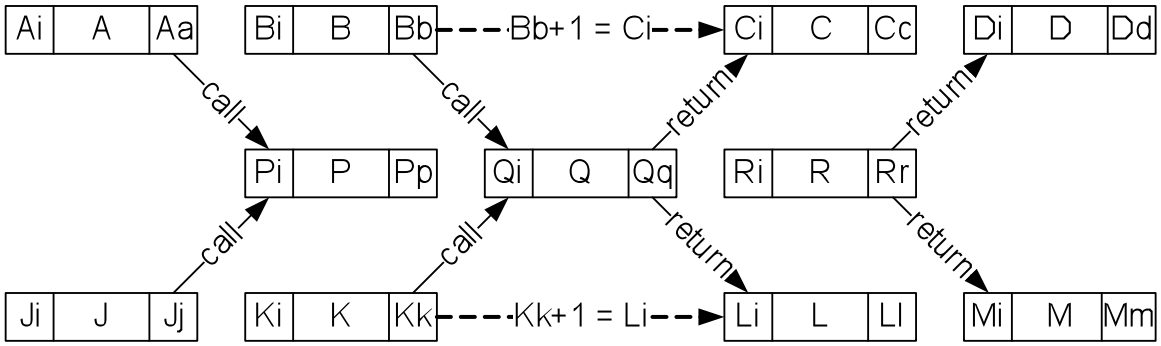

Figure 5- 7: 2-cycle block ahead prediction servicing calls and returns from two different call sites.

Let us see what each BTB hit would provide and actions associated with each BTB hit.

1. {Aa,transition} ➔ {Pi,call,b}
2. {Jj,transition} ➔ {Pi,call,k}
3. {Bb,T} ➔ {Qi,type of Pp,p}
4. {Kk,T} ➔ {Qi,type of Pp,p}
5. {Bb,R} ➔ {Di,type of Cc,c}

6. {Kk,R} ➔ {Mi,type of Ll,l}
7. {Pp,transition} ➔ {Ri,type of Qq,q}
8. {Qq,transition} ➔ {garbage,return,r}

Whenever Aa or Jj has a BTB hit, they see that there is a call in the next fetch block. They push onto the RAS: Bb+1 (Ci) or Kk+1 (Li), respectively. This is done in the next cycle, during Fetch Stage 2, when we know the size of the next fetch block (See Figure 5-4), b or k, depending on from where the call is being made.

Bb and Kk create two BTB entries each. The first entry is for the usual T transition which produces the start PC of the second fetch block in the called function, Qi (items 3 and 4 above). Interestingly, Bb and Kk can be viewed as each having another block ahead target corresponding to the second fetch block after the call-site continuation: Di and Mi, respectively. The second BTB entry is for representing this additional block ahead target and is distinguished with the return type (R) (items 5 and 6 above). If Bb or Kk hits for the return type (R), we will push the whole BTB entry onto the SAS. This defers the use of this BTB entry until the function returns to the call site.

Let us now see the return case. A BTB hit on {Qq,N/T} will say that the next fetch block ends with a return. Thus, we will pop from the RAS and get Ci/Li that we had pushed earlier. In the next cycle, we will receive the size of fetch block R (r) as well as the type of Rr which is a return. Thus, we will pop the BTB entry that was pushed onto the SAS previously and associate it with Rr. The BTB entry popped from the SAS will provide us with the target address Di/Mi.

5.1.3 Handling Misses from the Branch Target Buffer

Fetch Stage 2, otherwise known as the pre-decode stage, is used to detect BTB misses. This is the stage where we receive the instructions that were fetched from the I-cache and align and mask them to extract the fetch block. We then pre-decode the instructions to get information about branch instructions. This information is used to detect BTB misses. In the conventional case, we use the PC of the branch instruction to access the BTB and send the BTB hit information from Fetch Stage 1 to Fetch Stage 2 with the instruction packet. In Fetch Stage 2, we detect whether or not a valid instruction detected as a branch instruction by the pre-decode logic had a BTB hit in the previous cycle. If it was a BTB miss and the branch is predicted to be taken, we send a recovery signal and restart fetch from the predicted path.

In the case of 2-cycle block ahead prediction, the PC of the branch instruction and the PC used to access the BTB and give information on hit or miss of the branch instruction are separated in time. By the time we pre-decode the branch instruction, the PC used to access the BTB would have moved to the decode stage or farther ahead in the pipeline (in case of bubbles introduced due to recovery). Thus, we need to store the information of the preceding fetch block to service the BTB miss.

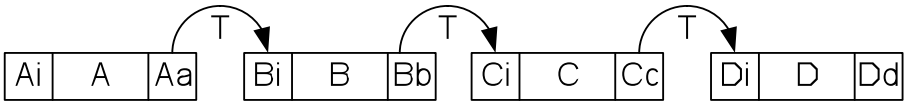

Figure 5- 8: Partial control-flow graph of a program.

Let us suppose from the above figure, {Aa,T} missed for branch instruction Bb in cycle t. Thus, the fetch unit predicts Bi+4 instead of Ci, because the BTB says there are no branches in fetch block B due to a miss. At this time (cycle = t), fetch block B is currently in Fetch Stage 1a. To know about the miss of {Aa,T} for branch Bb, fetch block B needs to be in pre-decode. Fetch block B would be in the pre-decode stage in cycle t+2. By this time, fetch block A would have moved ahead in the pipeline; it contains Aa which had missed in the BTB. Thus, we need to keep track of the fetch block ahead of fetch block B to even note a miss on Bb. The information should be able to provide the tag {Aa,T} as well as hit/miss for Bb. Thus, we store this information in Fetch Stage 3, which is actually in parallel with the Decode Stage (or the rest of the pipeline). Figure 5-9 shows how Fetch Stage 3 fits in the pipeline.

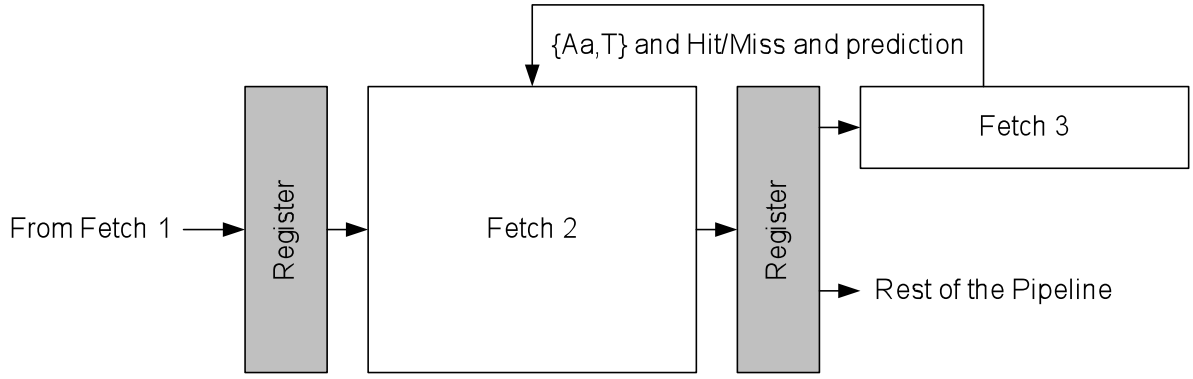

Figure 5- 9: Information passed from the fetch block preceding the branch instruction. (The preceding fetch block is in Fetch 3 and the branch instruction is in Fetch 2.)

The next step after the miss is detected is to restart fetching from the correct path. Let us revisit the previous example of {Aa,T} missing for branch instruction Bb.

The pre-decode stage (Fetch 2) will provide us with the taken target of the branch instruction. We need to select either the taken or not-taken target as per the prediction given for Bb. The prediction is among the information being sent from Fetch 3 (shown in Figure 5-9) since Aa previously accessed the BP to obtain the prediction for Bb. In the example, the taken target is selected for Bb (Ci), the recovery signal is asserted to load it into the program counter, and Fetch 1 is flushed.

Thus, in cycle t+2 we detect the BTB miss and we restart with Ci in cycle t+3. But there are no instructions valid in Fetch Stage 1b to produce a PC to follow Cc. This is due to the fact that fetch block B has now moved ahead in the pipeline and is no longer in fetch stage 1b, when fetch block C is in fetch 1a. There are now bubbles introduced between fetch block B and C due to the flush on the BTB miss of fetch block A (for Bb). To handle this case, we also need to predict the target of fetch block C in the pre-decode stage following the target of Bb. The information of the BTB hit on fetch block B is sent down the pipeline from Fetch Stage 1b to Fetch Stage 2. This facilitates the determination of Di by using the hit on {Bb,T} and the prediction obtained by Bb for branch instruction Cc. When implemented in synthesizable RTL, I observed that this logic creates the critical path for the pre-decode stage (Fetch Stage 2).

The last part of servicing a BTB miss is to recover the RAS in the conventional case and the RAS and SAS in the block-ahead case. In both cases, we keep

checkpointed copies of the stacks. These are delayed versions of the stacks and golden with respect to Fetch Stage 2.

When there is a BTB miss in the conventional fetch unit, we perform the push and pop if necessary on the checkpointed RAS (if the BTB miss was on a call or return, respectively) and then copy the checkpointed RAS to the RAS. This is needed since the RAS may be corrupted by returns and calls on the wrong path until the BTB miss is serviced (see Figure 5-10).

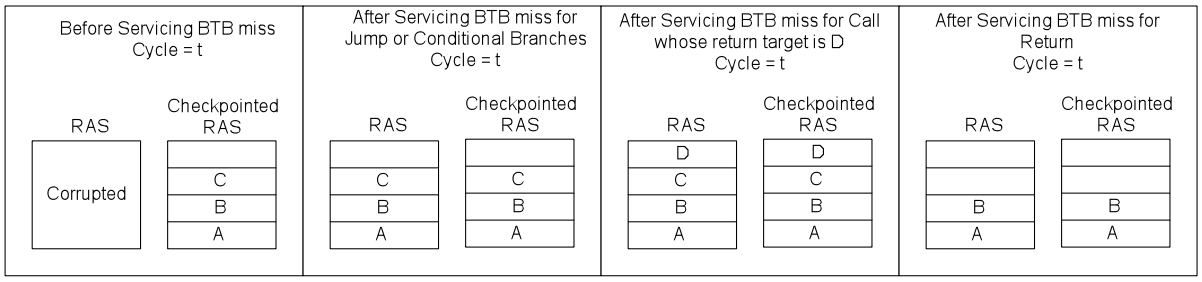

Figure 5- 10: One cycle RAS recovery model for baseline instruction fetch unit.

In the block-ahead fetch unit, both the RAS and SAS are vulnerable to corruption. We need to keep checkpointed copies of both stacks, which is golden at Fetch Stage 2. In restoring both the stacks, we need to perform pushes and pops to the checkpointed stacks if necessary according to the BTB miss and then copy them to the working stacks. But since the working stacks are in Fetch Stage 1b, we do not need the stacks in the next cycle. So, we can pipeline the recovery. We can let the checkpointed stacks get updated with the miss and then flash copy the stacks in the next cycle. It would be ready for use before any valid instruction reaches Fetch Stage 1b (see Figure 5-11).

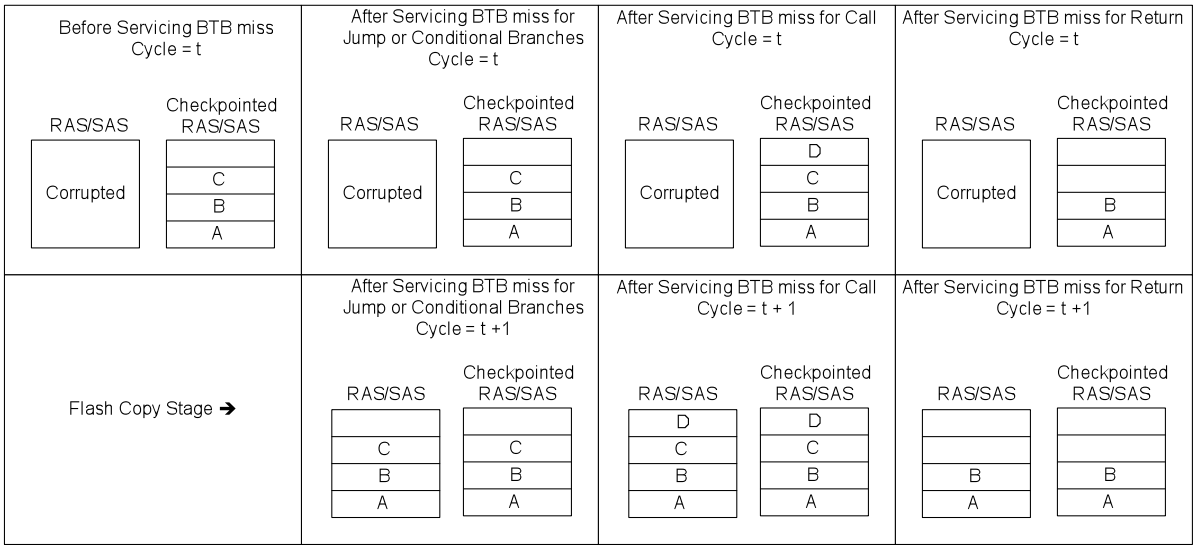

Figure 5- 11: Two cycle RAS/SAS recovery model for instruction fetch unit implementing block ahead prediction.

5.1.4 Handling Branch Mispredictions

The instruction fetch unit is used to speculate beyond various types of branch instructions. The speculation gets resolved when the branch instruction is executed. There are cases where our predictions are wrong. Thus, we need a process for flushing the pipeline and restarting from the correct path. As we saw in Chapter 1, this process is costly in terms of performance.

In the conventional fetch unit, we receive the correct target of the mispredicted branch along with the respective recovery signal. These are used to restart fetch along the correct path. In the block-ahead fetch unit, we also receive the correct target along with the recovery signal. Let us suppose, we signal the branch misprediction for flush and recovery in cycle t and the correct target is used to start Fetch Stage 1a in cycle t+1. We do not have any valid instructions in Fetch Stage 1b to predict the next PC,

thus we can predict only the fall-through address. But we know that often, this default next PC is not correct. Let us take an example and see how we can fix it.

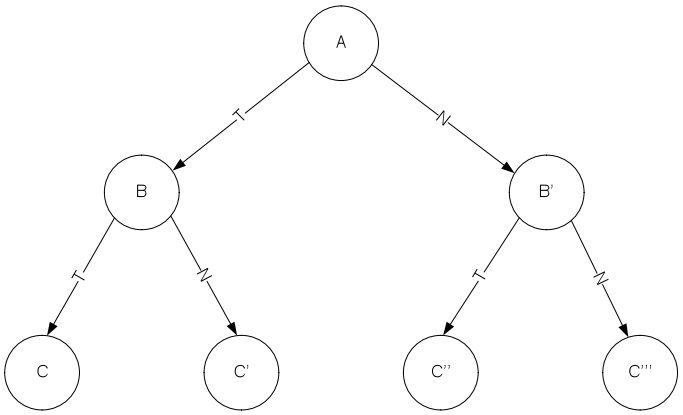

Figure 5- 12: Control-flow graph of a program.

Let us assume that the branch instruction Aa is predicted to be not-taken by its preceding fetch block. So the program goes on the wrong path to B'i. When Aa executes, it sees that it has gone down the wrong path to B'i (Aa + 1). So, it sends the correct target (Bi) with required recovery signals in cycle t, to restart fetching. Fetch Stage 1a starts fetching fetch block B in cycle t+1. Since Aa was originally predicted to be not-taken, we had obtained the BTB entry {Aa,N} corresponding with the prediction for B'b' (assuming taken for the example, produced by Aa) to predict C''i. Along with this calculation, we could have also calculated the target with a hypothetical hit on the BTB entry {Aa,T} with the same prediction for Bb (since the prediction for Bb and B'b' is made by Aa) to predict Ci. If we calculate the alternate target (Ci) in parallel with predicted target (C''i), we can start to fetch from Ci in cycle t+2 after the misprediction is detected.

To facilitate the above recovery process, information about the alternate BTB entry, {Aa,T} in our ongoing example, must be saved when the prediction is made and restored when the misprediction is detected. This information includes whether or not the alternate BTB entry is even available (it may not be) and, if so, its contents. The output register of Fetch Stage 3 is saved in the CTIQ for every branch: this state has all the information needed for recovery (see Figure 5-9).

5.1.5 Optimization for a Specific Case

In the block-ahead fetch unit, we cannot predict more than 1 branch per cycle. In general, this means we need to discard all instructions after the first branch, among the instructions fetched in a cycle. There is a special case that can be optimized, however. The case is where there is only one branch among the fetched instructions and it is predicted not-taken. In this special case, we can keep all the instructions and save refetching a few instructions. This optimization is proposed in [2]. Keeping 1-bit in the BTB with the entry for the branch can convey this case (no other branches after the branch on its not-taken path). The bit indicates whether or not the branch is the last among the fetched instructions. In case the branch is predicted not-taken, the fetch block is as wide as the fetch width and the next PC is calculated accordingly: PC + fetch width.

5.1.6 Putting it all together

Throughout the whole section we saw how all the structures work and how each and every recovery scenario works in the block-ahead fetch unit. Let us combine everything and see how the whole system works in Figure 5-13. The next PC logic with the priority logic is shown in Figure 5-14.

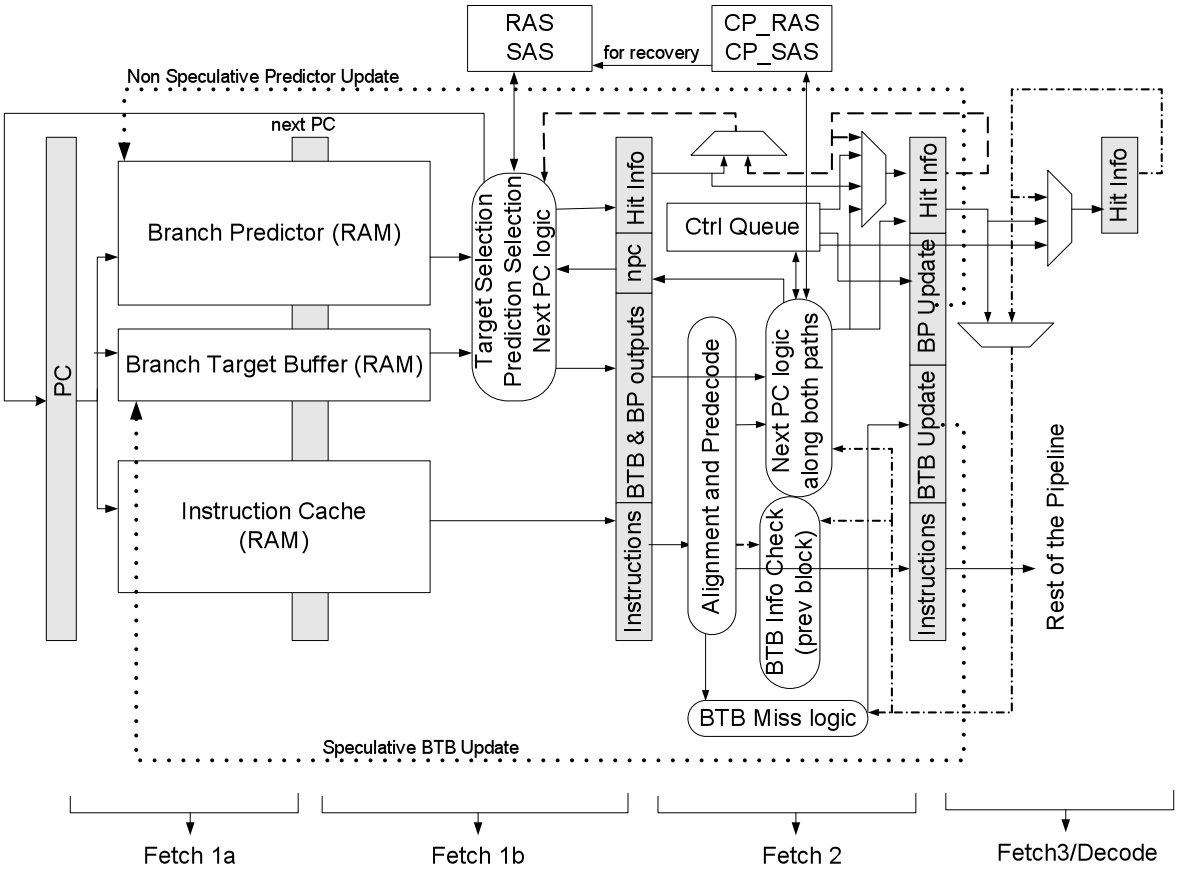

Figure 5- 13: Instruction fetch unit implementing block ahead prediction.

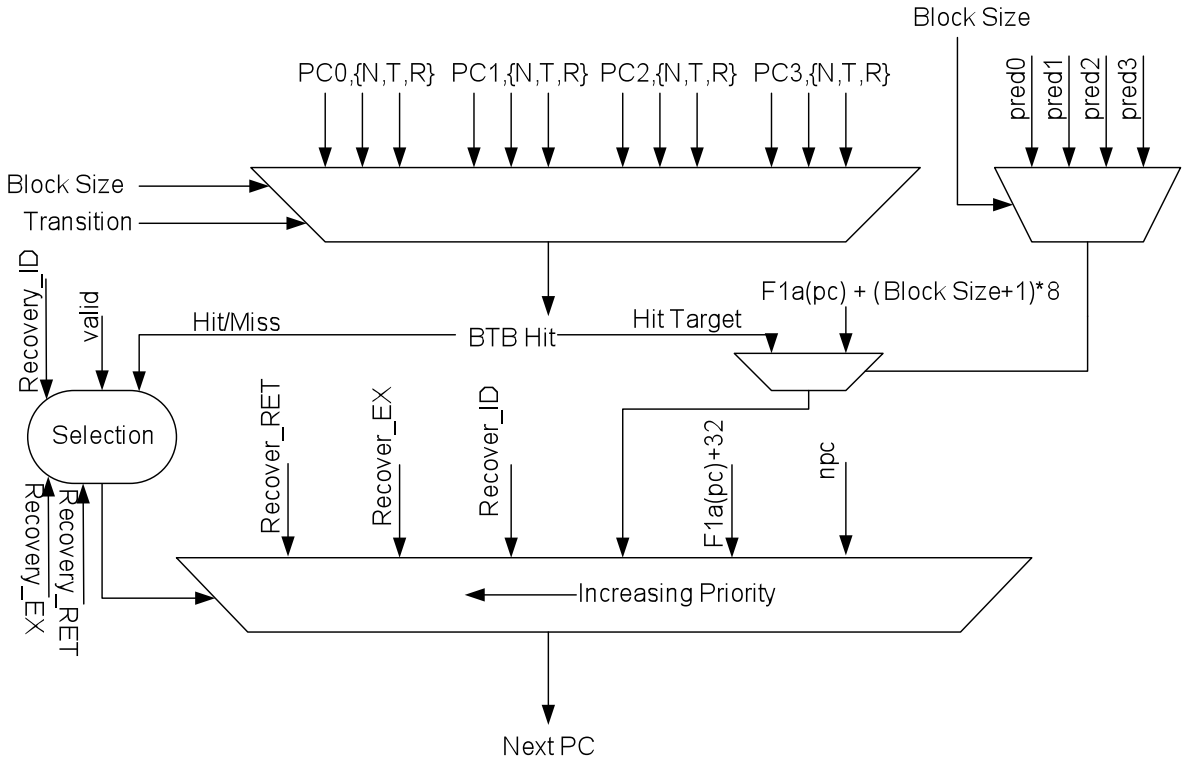

Figure 5- 14: Next PC logic for instruction fetch unit implementing block ahead prediction.

5.2 Performance of Block Ahead Prediction in C++ Simulator

We will initially evaluate block ahead prediction using the C++ cycle-level simulator that was developed to understand the working of the scheme. Accuracy of the pipelined BP is analyzed in Section 5.2.1. Hit ratio of the pipelined BTB is analyzed in Section 5.2.2. Finally, overall performance in terms of IPC is analyzed in Section 5.2.3.

5.2.1 Impact on Conditional Branch Predictor Accuracy

Table 5-1 and Figure 5-15 show the misprediction rates of the 1-cycle baseline fetch unit and the 2-cycle block-ahead fetch unit. The BP is a 16KB bimodal predictor. The BTB has 1024 entries and is 4-way set-associative.

Table 5- 1: Misprediction rates of the baseline and block-ahead fetch units. (16KB bimodal BP, 1024-entry 4-way set-associative BTB)

Bimodal Predictor		
Benchmark	**Baseline**	**Block Ahead**
bzip	9.99	9.92
crafty	11.42	10.58
gap	9.45	9.58
gcc	9.02	9.83
gzip	10.81	11.72
mcf	12.48	7.08
parser	10.82	10.46
perl	6.35	6.43
twolf	17.41	19.26
vortex	1.52	2.42
vpr	12.96	12.46
Average	**10.20**	**9.98**

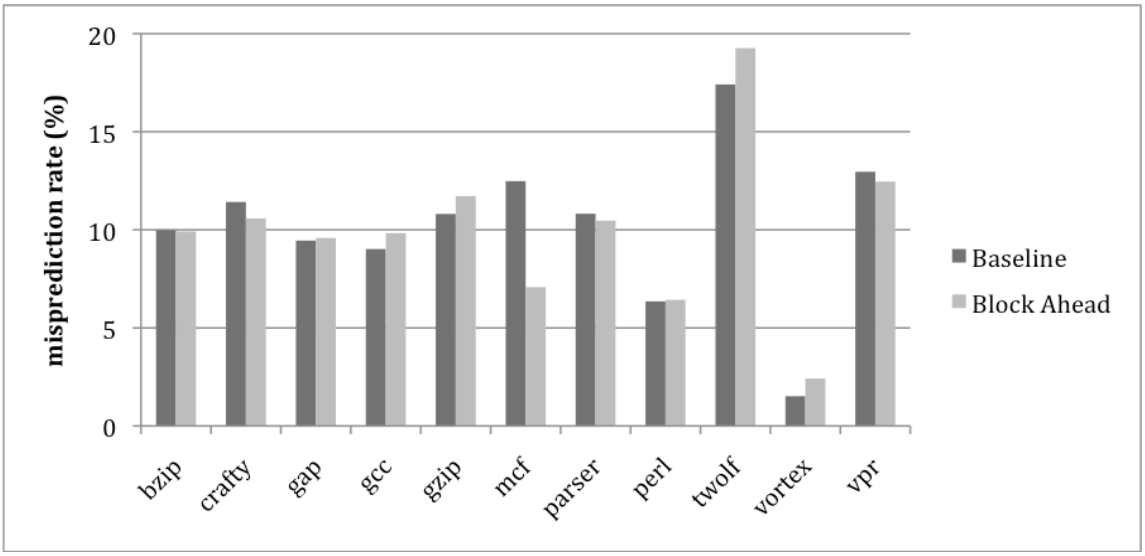

Figure 5- 15: Misprediction rates of the baseline and block-ahead fetch units. (16KB bimodal BP, 1024-entry 4-way set-associative BTB)

There are two effects that come into play for the bimodal BP since the branch is predicted by the preceding fetch block's last PC.

1. Same counter is used to predict two different branches. A conditional branch Aa is used to predict conditional branches on both the paths in fetch blocks B (Bb) and B′ (B′b′). This phenomenon decreases the accuracy of the predictor.
2. A branch can be reached from multiple fetch blocks. So the prediction of the branch is done by multiple different indices/counters. This leads to an increase in the accuracy of the predictor, due to a little bit of implicit path and/or global history.

We see the tradeoff between these two phenomena in the results of Table 5-1: some benchmarks have lower misprediction rates with the block-ahead fetch unit and others have higher misprediction rates, compared to the baseline fetch unit. On average, applying block ahead prediction to the bimodal BP reduces its misprediction rate from 10.2% to 9.98%. So, on average, the second phenomenon outweighs the first.

The two phenomena are muted by a gshare BP. Table 5-2 and Figure 5-16 show misprediction rates of the baseline and block-ahead fetch units with a 16KB gshare BP and 1024-entry 4-way set-associative BTB.

The global history register used for making predictions could not be designed to have the latest bit of history even with post-selection. So, I had to omit the most recent

branch's prediction from the global history register. This leads to higher misprediction rates for the gshare BP in the block-ahead fetch unit compared to the baseline fetch unit, for all but two of the benchmarks. Nonetheless, the accuracy of the pipelined gshare BP is much better than that of the pipelined bimodal BP.

Table 5- 2: Misprediction rates of the baseline and block-ahead fetch units. (16KB gshare BP, 1024-entry 4-way set-associative BTB)

GShare Predictor		
Benchmark	**Baseline**	**Block Ahead**
bzip	10.34	9.89
crafty	5.32	7.54
gap	2.28	3.12
gcc	4.24	5.15
gzip	8.68	9.54
mcf	2.43	3.42
parser	5.61	6.26
perl	2.08	2.41
twolf	11.02	12.48
vortex	0.32	1.13
vpr	10.19	10.09
Average	**5.68**	**6.46**

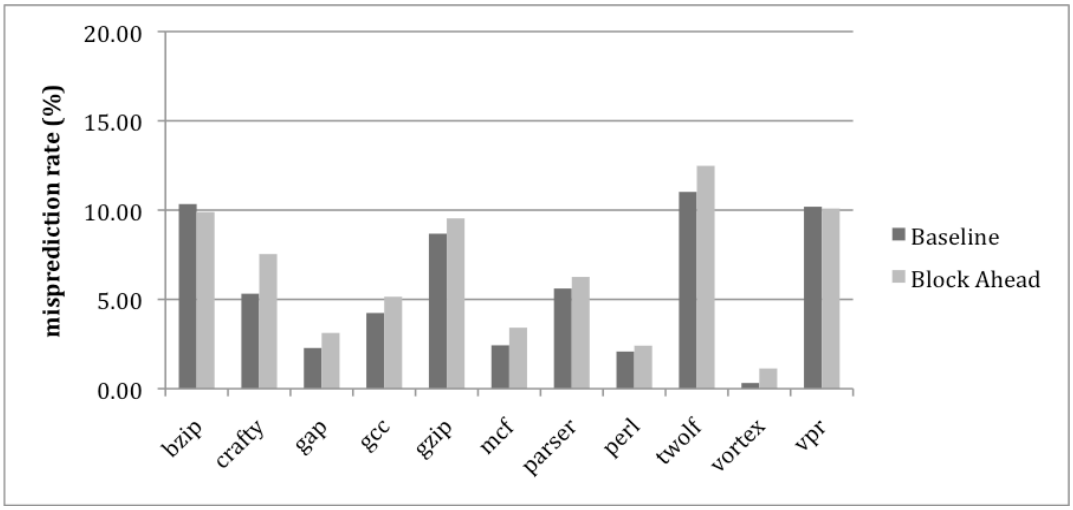

Figure 5- 16: Misprediction rates of the baseline and block-ahead fetch units. (16KB gshare BP, 1024-entry 4-way set-associative BTB)

5.2.2 Impact on Performance of the Branch Target Buffer

Table 5-3 and Figure 5-17 show the BTB hit ratios of the baseline and block-ahead fetch units. These are retirement hit ratios, i.e., only branches that are ultimately retired go into the hit ratio measurement. The results are for a 16KB gshare BP and 1024-entry 4-way set-associatuve BTB.

There are two impacts that can be seen in the results.

1. Since we do speculative updates of the BTB in the block-ahead fetch unit, we will be using more BTB entries than the baseline fetch unit which does non-speculative updates. This leads to a lower BTB hit ratio, as measured for retired branches.
2. Multiple fetch blocks can reach same branch. Thus, a single BTB entry in the baseline fetch unit grows to multiple BTB entries in the block-ahead fetch unit. This also leads to a lower hit ratio.

Table 5- 3: BTB hit ratios of the baseline and block-ahead fetch units. (16KB gshare BP, 1024-entry 4-way set-associative BTB)

Benchmarks	**Baseline**	**Block Ahead**
bzip	1.00	0.98
crafty	0.94	0.83
gap	0.98	0.88
gcc	0.88	0.80
gzip	1.00	0.97
mcf	1.00	0.99
parser	1.00	0.95
perl	0.94	0.85
twolf	0.99	0.97
vortex	0.99	0.96
vpr	1.00	0.96
Average	**0.97**	**0.92**

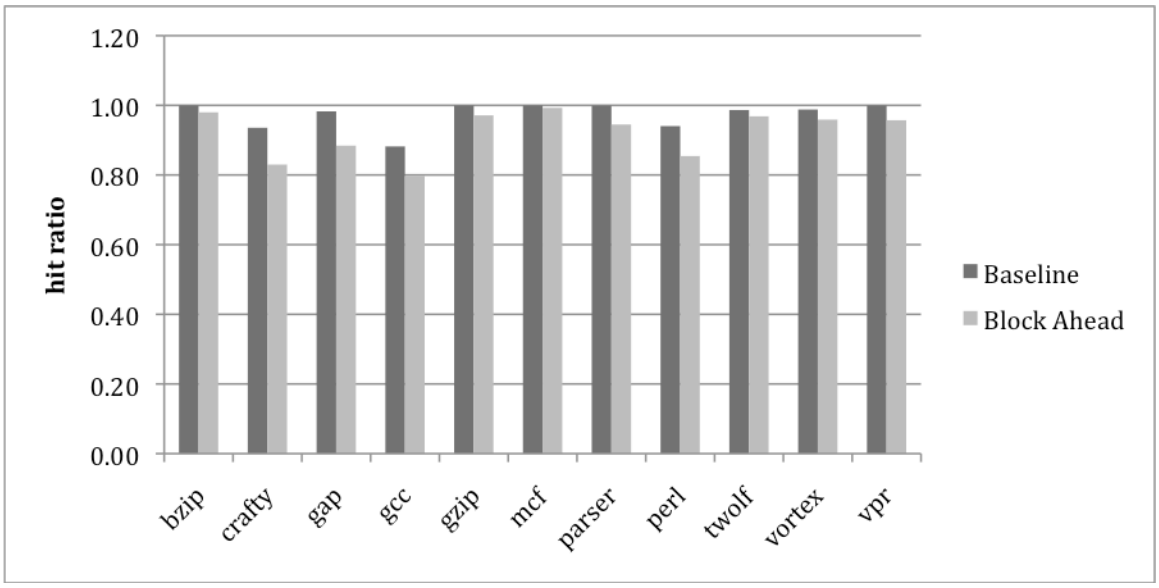

Figure 5- 17: BTB hit ratios of the baseline and block-ahead fetch units. (16KB gshare BP, 1024-entry 4-way set-associative BTB)

5.2.3 Impact on IPC

In this section, we compare the performance, in IPC, of the processor with the baseline fetch unit versus the block-ahead fetch unit. We can see that the IPC varies as the accuracy of the BP varies. (See Table 5-1 and Table 5-2 for accuracies of the branch predictors.) The block-ahead bimodal performs quite close to the baseline bimodal, and the block-ahead gshare performs considerably better. Moreover, the block-ahead variants are pipelined for higher frequency, an aspect considered in the next section where we evaluate the synthesizable RTL model of the block-ahead fetch unit.

Table 5- 4: IPCs of the baseline and block-ahead fetch units.

Predictors	**Bimodal**		**Gshare**
Benchmarks	**Baseline**	**Block Ahead**	**Block Ahead**
bzip	1.03	0.99	0.99
crafty	0.95	0.95	1.01
gap	0.88	0.84	0.95
gcc	0.88	0.85	0.94
gzip	0.95	0.91	0.99
mcf	0.75	0.87	1.02
parser	0.86	0.86	0.96
perl	0.81	0.76	0.82
twolf	0.84	0.79	0.91
vortex	1.16	1.17	1.23
vpr	0.95	0.95	1.04
Average	**0.91**	**0.90**	**0.99**

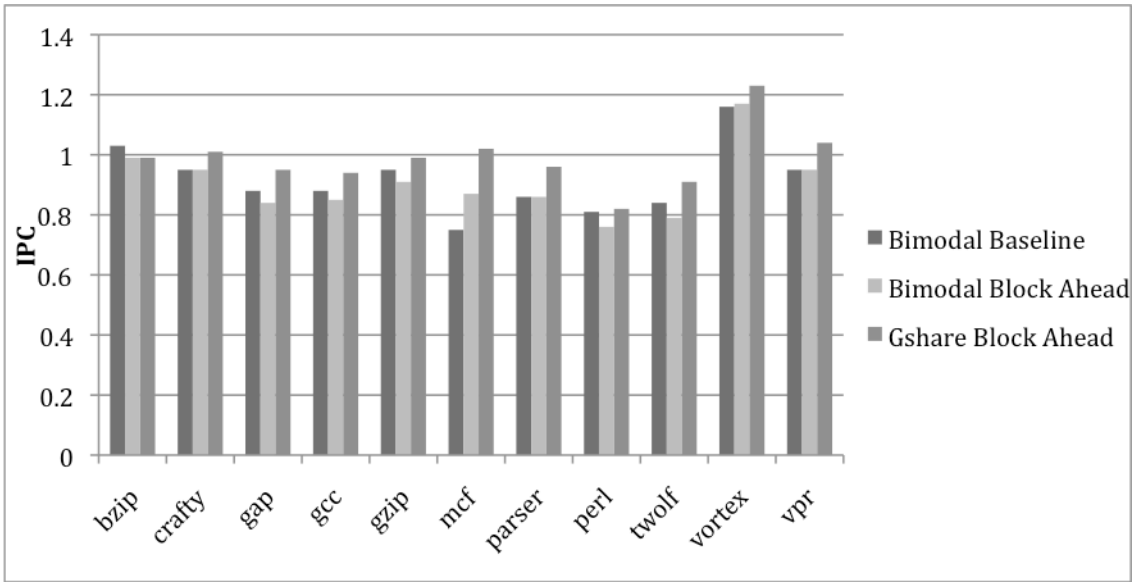

Figure 5- 18: IPCs of the baseline and block-ahead fetch units.

5.3 Performance of RTL Implementation of Block-Ahead Fetch Unit

To see the advantage of pipelining the branch prediction logic, consider a target frequency of 1GHz. As per the synthesized fetch unit, we are able to fit a 256-entry 4-

way set-associative BTB into the cycle time, which generally performs poorly in terms of hit ratio. But for the block-ahead fetch unit we have two cycles to access for the BTB and perform the next PC calculation. Thus, we can fit a 1024-entry 4-way set-associative BTB in the block-ahead fetch unit. The same argument goes for the BP as well.

Alternatively, if we keep the sizes of the BTB and BP fixed, then with pipelining the branch prediction logic we can see an increase in the frequency of the Fetch Stage 1. If other pipeline stages are also aggressively pipelined, this would increase the frequency of the processor as a whole. Figure 5-19 shows the improvement in Fetch Stage 1 delay with block ahead prediction, for different fetch widths of the processor.

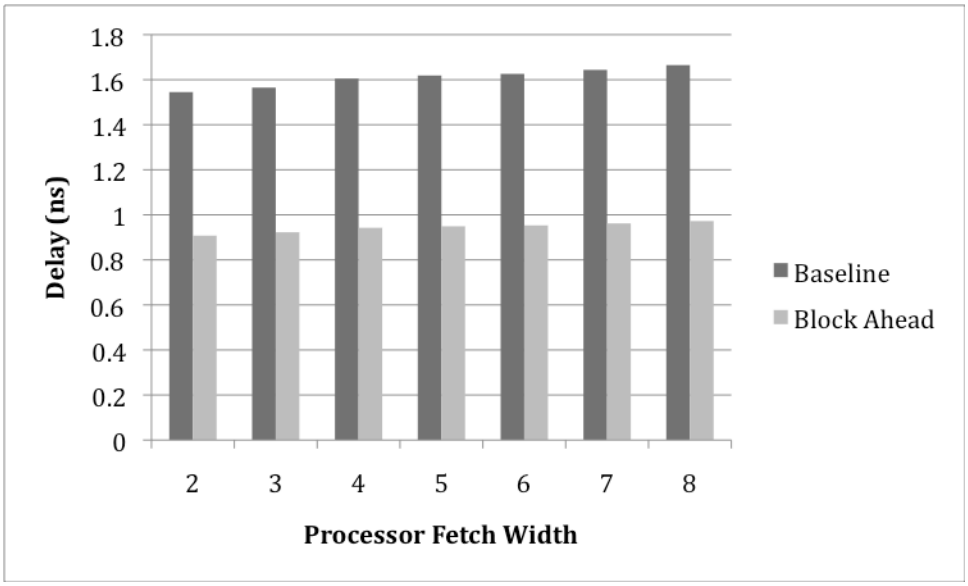

Figure 5- 19: Fetch Stage 1 delays for the baseline and block-ahead fetch units.

Table 5-5 shows IPCs and misprediction rates from Verilog simulations of a 4-way superscalar processor with the block-ahead fetch unit. The results are for a 16KB bimodal BP and a 1024-entry 4-way set-associative BTB.

Table 5- 5: .Results from Verilog simulations of a 4-way superscalar processor with the block-ahead fetch unit. (16KB bimodal BP, 1024-entry 4-way set-associative BTB)

Benchmark	IPC	Cond. Branch Misprediction Rate	Misprediction Rate of Other Branch Types	Total Branch Misprediction Rate
bzip	0.78	10.08	45.65	12.18
gap	0.70	10.32	72.07	17.99
gzip	0.62	11.80	77.56	16.86
mcf	0.83	6.87	94.84	7.15
parser	0.73	10.23	51.30	13.97
vortex	0.70	1.43	86.91	10.37

Figures 5-20 and 5-21 compare the IPCs and accuracies, respectively, of the baseline and block-ahead fetch units. Again, these results are from Verilog simulations.

The positive result is that the IPCs of the two fetch units are quite close, with block-ahead performing a bit better or a bit worse on different benchmarks. This result confirms the effectiveness of this technique for pipelining the branch prediction logic. Coupled with the higher frequency enabled by pipelining, the performance improvement of the block-ahead fetch unit is substantial.

Comparing Figure 5-15 (bimodal misprediction rate in C++ simulator) and Figure 5-21 (bimodal misprediction rate in Verilog simulator), it is apparent that the RTL and C++ models are in very close agreement, both in terms of trends across the benchmarks and absolute misprediction rates.

We can see from Table 5-5 that the accuracy of other types of branches is quite low. This is due to the fact that we chose to flush RAS and SAS on a misprediction. This leads to more return mispredicts (loss of information in RAS) and BTB misses (loss of information in SAS).

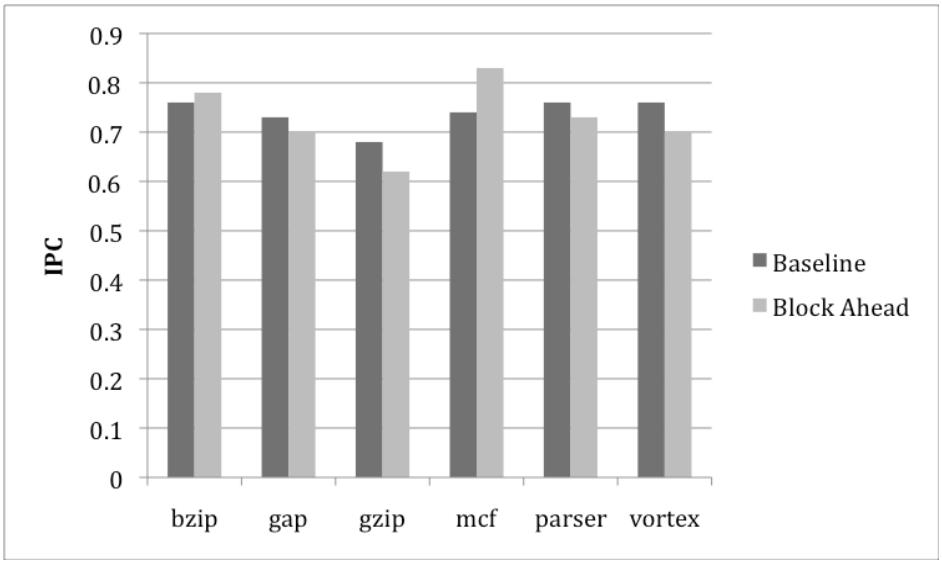

Figure 5- 20: IPCs of the baseline and block-ahead fetch units. (16KB bimodal BP, 1024-entry 4-way set-associative BTB)

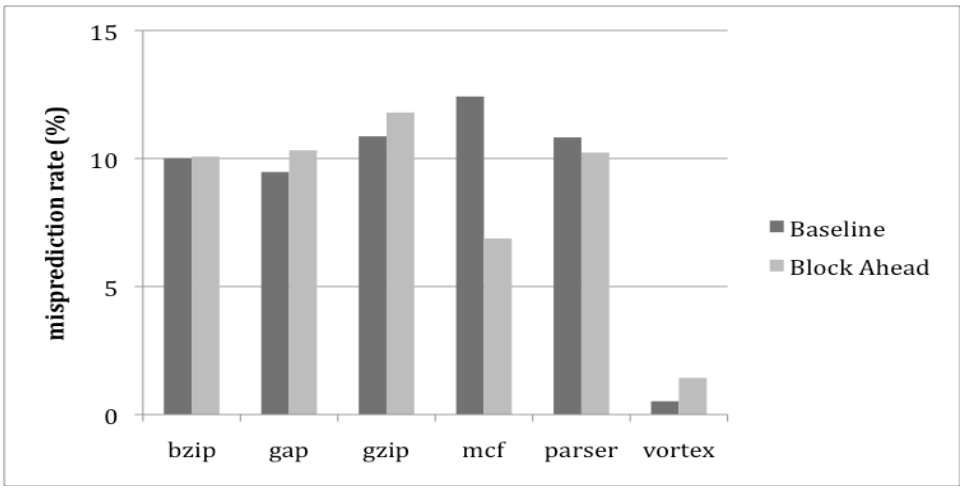

Figure 5- 21: Misprediction rates of the baseline and block-ahead fetch units. (16KB bimodal BP, 1024-entry 4-way set-associative BTB)

CHAPTER 6

Decoupled Effective Ahead Pipelining

Pipelining the BTB is not very scalable: it is complicated and expensive to pipeline the overall branch prediction logic, including the BTB, more than two cycles [2]. At the same time, in Chapter 4, we saw that the BTB is less critical to performance than the BP for a number of reasons. First, a BTB miss does not affect performance as much as a branch misprediction, due to the fact that the BTB miss can be detected and corrected quite early in the pipeline for most branches (except for indirect branches).Second, the BTB size is related to the static program size and so it does not need to be especially large. The opposite is true for the BP: the misprediction penalty is debilitating to performance (mispredictions are resolved in the execute or retire stage, depending on the aggressiveness of the recovery model) and accuracy improves substantially with ever larger BPs (if using global branch history, for example).

Therefore, we need to decouple the branch predictor pipeline from the instruction fetch pipeline so that we can pipeline the BP as deep as necessary. Parikh et. al. [12] concluded that it is better to put more power into the branch predictor to produce more accurate predictions, since doing so reduces execution time and the overall energy-delay product. To improve accuracy, large and complex branch predictors have been proposed, such as L-TAGE [13]. But these branch predictors do

not fit in 1 cycle in a deeply pipelined processor. So, we need techniques to effectively pipeline them, to achieve a fast cycle time, 1 prediction per cycle, and high accuracy.

6.1 Related Work

The basic idea to pipeline the branch predictor, as described by Seznec et al. in their work on block ahead prediction [2] and effective ahead pipelining [5], is to predict a branch using the last PC of a prior fetch block. The idea is shown in Figure 6-1.

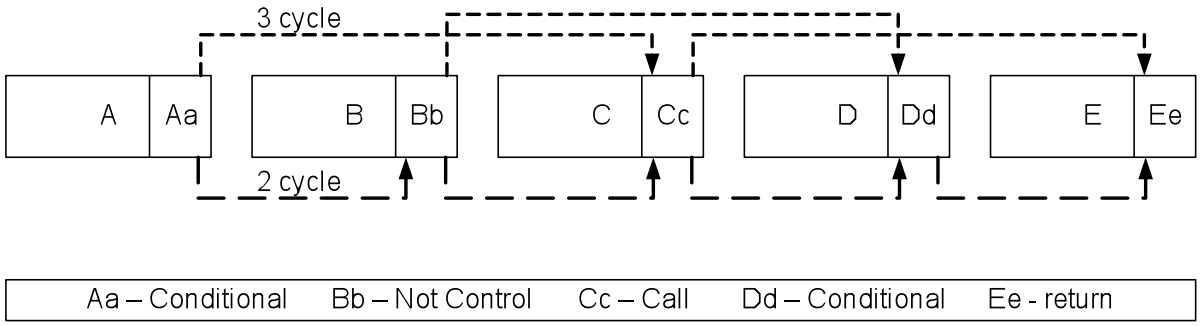

Figure 6- 1: Working of block ahead prediction.

Block ahead prediction, as originally proposed, is based on fetch block boundaries to predict branches. If the predictor index includes global branch history, we need to play tricks to keep the global branch history bits at a fixed distance. In particular, the history is polluted with bits from every fetch block end. This results in a fixed distance with a little lower accuracy.

When a block ahead predictor uses global branch history, one or more of the most recent history bits are not yet available at the time the predictor is accessed, due to prior pipelined accesses that have not finished yet. This is dealt with by using an

abbreviated history to read out multiple candidate predictions and then post-selecting one of them when the most recent global branch history bits become available at the end of the predictor pipeline.

For a gshare predictor that is pipelined into two cycles, and for a fetch width of 4 instructions, we will need to read eight counters from the pattern history table. Figure 6-2 shows which bits are missing that we will get by the end of the predictor access for post-selection. As the predictor is pipelined deeper, the number of missing history bits increases. As the fetch width is increased, the number of missing PC bits increases. These two portions of missing information can be overlapped by storing the history bits in the opposite order than proposed in [4].

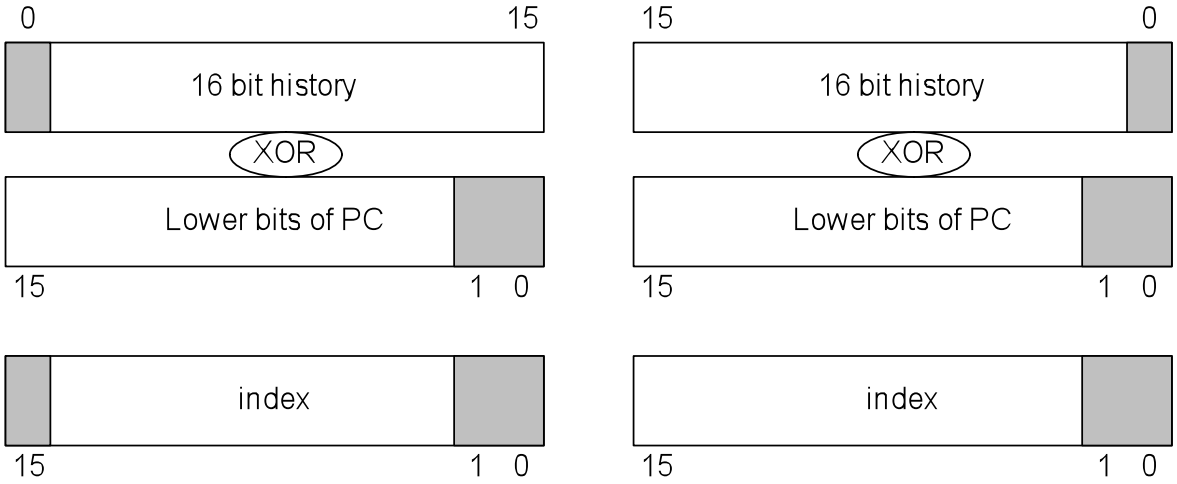

Figure 6- 2: Index calculation for block ahead predictor.

Efficient misprediction recovery requires checkpointing some of the candidate predictions that were initially read out. This is required since we do not want to re-access the predictor to read counters that we have read already. Also, we do not want

to wait until the branch predictor pipeline refills. If we re-access the predictor, we will increase the misprediction loop by {depth of branch predictor – 1} cycles. An increase in the misprediction loop, as seen earlier, hampers performance. The number of predictions to be checkpointed only depends on the number of missing history bits, i.e., {the depth of the predictor pipeline – 1}.

The second technique proposed by Daniel A. Jiménez [3] leads to better accuracies than block ahead predictor since it uses the current branch PC instead of some distant PC to predict the branch. The idea is to index the predictor table with old history only to get candidate entries. From those candidate entries, according to the BTB hit, we select the correct prediction. Figure 6-3 shows the concept used by [3].

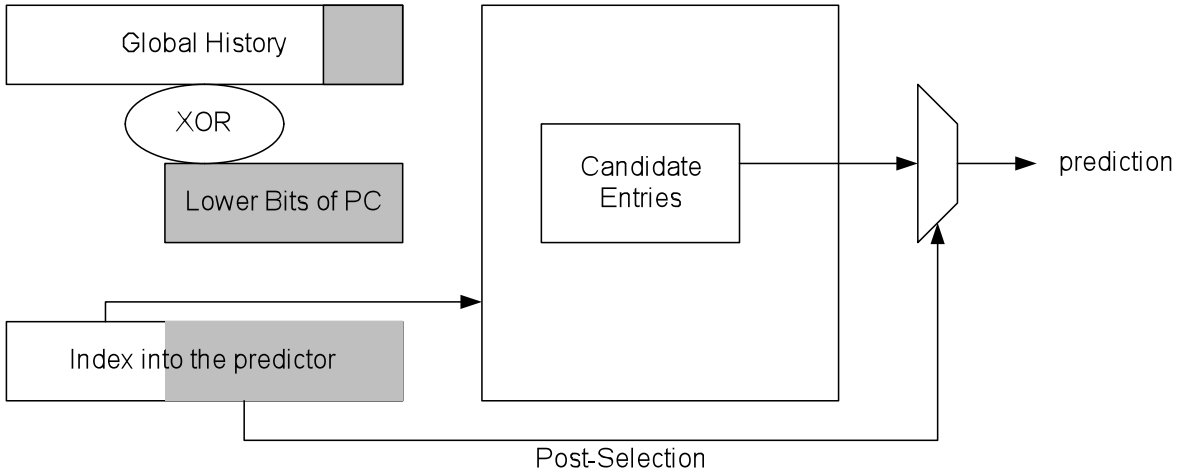

Figure 6- 3: Concept behind gshare.fast proposed by D. Jiménez.

The number of PC bits used and the depth of the branch predictor decide the candidate entries to be read from the predictor. In [3], he used 9 bits of the current branch PC and a depth of 4, which lead to reading out 512 candidate entries (2^9) from the predictor. We need to checkpoint all the entries to recover from a misprediction in

1 cycle. This is required since the branch PC can change as well as history bits. The checkpoint of 512 entries as well as selection from 512 entries requires a large checkpoint and fast muxing to do it in 1 cycle.

I propose a design that uses both ideas. I call the design *Decoupled Effective Ahead Pipelining (DEAP)*. I will explain the design implemented in FabScalar [6][7] in the next section.

6.2 Design of the Decoupled Effective Ahead Pipeline

To decouple the predictor from the rest of the fetch unit (i.e., decouple it from Fetch Stage 1), we need to find a fixed point in the instruction stream to initiate the read into the predictor and get the prediction in time for Fetch Stage 1 to consume the prediction. We know that we need to predict only conditional branches. So we can use the addresses of the conditional branches only to initiate reads. But since we need predictions in time for the conditional branches, we need to introduce distance between which address is used to predict which conditional branch. The concept is explained in Figure 6-4.

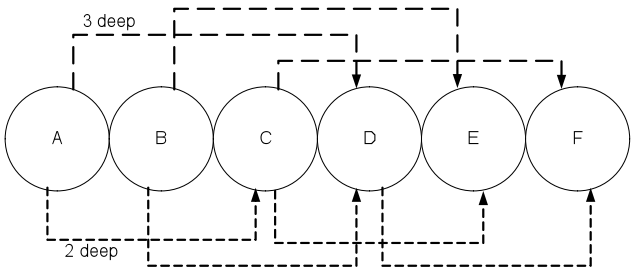

Figure 6- 4: Basic idea behind decoupled effective ahead pipelining.

In Figure 6-4, A, B, C, etc. are addresses of conditional branches in the instruction stream. The predictor's depth determines the distance between the conditional branch used to access the predictor and the conditional branch being predicted. For example, for a 2-deep predictor, B is used to predict D, whereas, for a 3-deep predictor, A is used to predict D.

6.2.1 Limit Study

To gauge the accuracy of this approach, I implemented the DEAP technique in a functional simulator with just the branch predictor modeled. I refer to this as a "limit study" since it abstracts away implementation details yet still faithfully models the DEAP indexing strategy. The predictor evaluated is a 2^{16}-entry gshare predictor. The limit study enables us to quickly measure the extent to which accuracy degrades with the DEAP indexing strategy.

Table 6-1 and Figure 6-5 show the results of the limit study. We can see that the misprediction rate of the large predictor increases by a small amount as it is pipelined deeper, which will have a mild impact on IPC.

Table 6- 1: Misprediction rates of a 2^{16}-entry gshare predictor for different depths of the predictor, in the limit study.

Benchmarks	Baseline	1 deep	2 deep	3 deep	4 deep
bzip	10.34	10.30	10.46	10.44	10.48
crafty	5.32	5.38	5.48	5.71	5.67
gap	2.28	2.70	2.92	2.94	2.93
gcc	4.24	4.47	4.72	4.81	4.84
gzip	8.68	8.71	8.70	8.70	8.71
mcf	2.43	2.43	2.42	2.43	2.42
parser	5.61	5.68	5.73	5.74	5.78
perl	2.08	2.14	2.34	2.47	2.66
twolf	11.02	11.00	11.15	11.37	11.32
vortex	0.32	0.46	0.84	0.88	1.53
vpr	10.19	10.18	10.24	10.20	10.24
Average	**5.68**	**5.77**	**5.91**	**5.97**	**6.05**

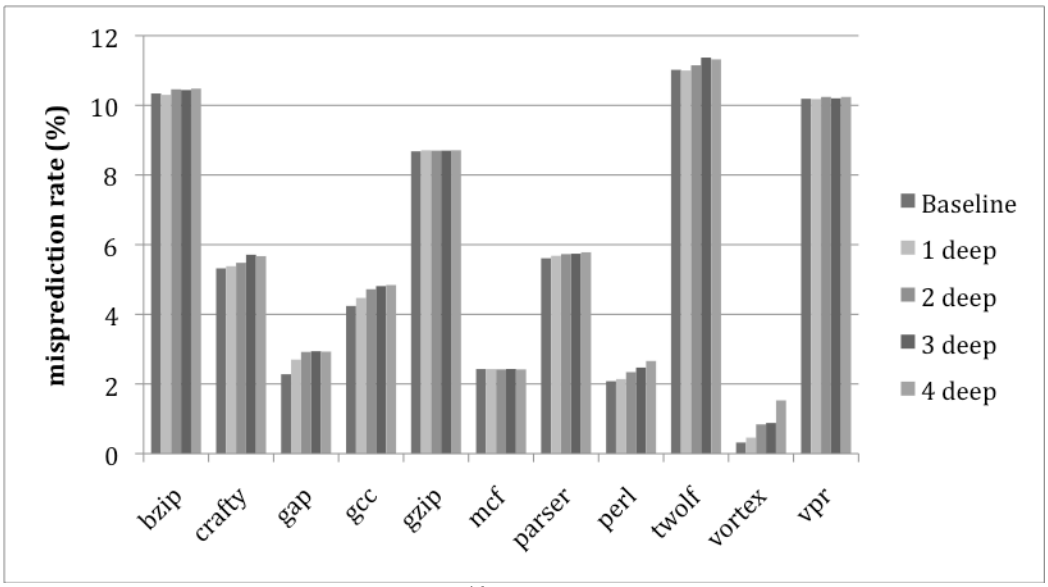

Figure 6- 5: Misprediction rates of a 2^{16}-entry gshare predictor for different depths of the predictor, in the limit study.

6.2.2 Pipeline Structure

Figure 6-6 shows a high-level view of 2-deep and 3-deep BP pipelines that are decoupled from Fetch Stage 1. BP pipeline stages are annotated as follows. The final

stage of the BP pipeline is called BP1 because it corresponds with Fetch Stage 1. Preceding stages are numbered BP0, BP-1, etc., the farther back we go.

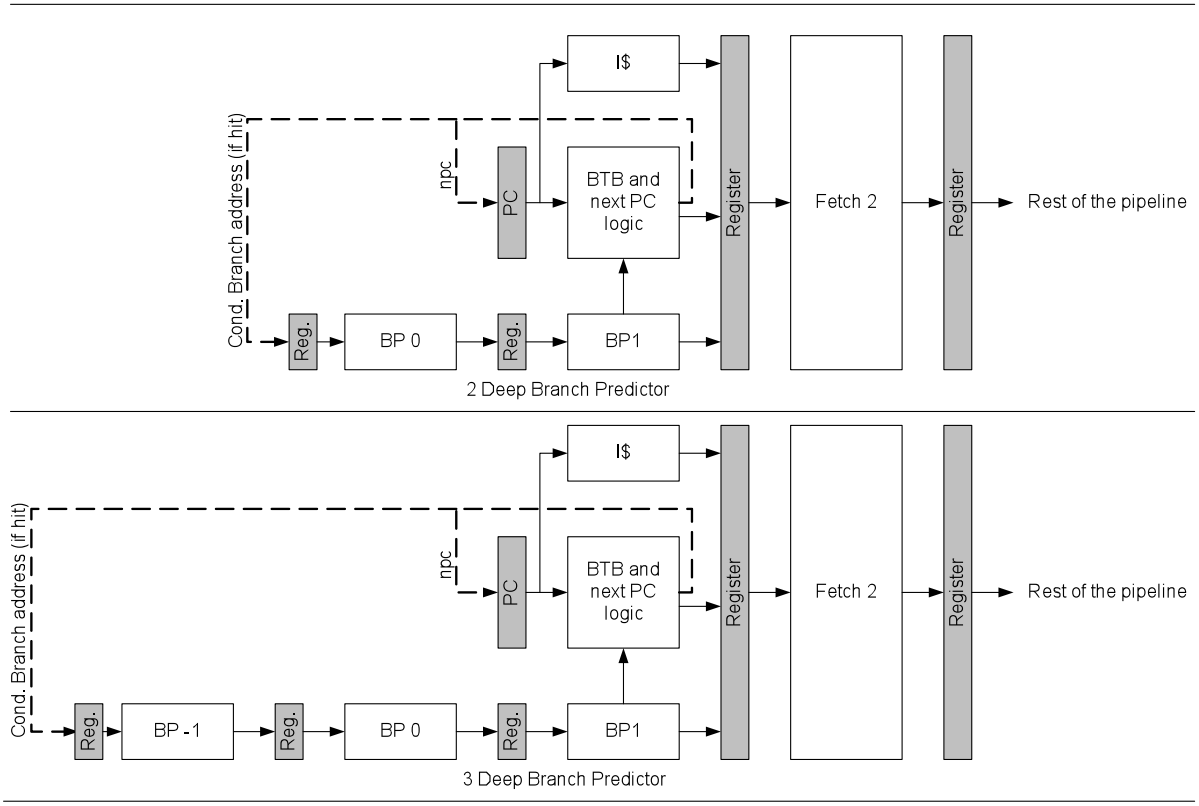

Figure 6- 6: Model of the fetch unit with DEAP.

6.2.3 Decoupling the Branch Predictor with a Queue

We need a queue to decouple the branch predictor from the rest of the pipeline. This queue will act as a buffer in case the branch predictor runs ahead of the rest of the pipeline.

Let us see how this queue will work in tandem with the rest of the pipeline by using an example. To simplify describing the example, we assume global history is not

used and that the BTB always hits. We will introduce global history and BTB misses later in the section.

The example instruction stream is shown in Figure 6-7. In Figure 6-7, fetch blocks are numbered (1 through 12) and conditional branches are indicated with capital letters (A through H). Note that some fetch blocks end in conditional branches and others do not.

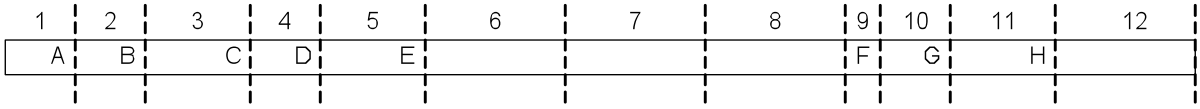

Figure 6- 7: Instruction stream with fetch blocks and conditional branches marked.

For the example, let us assume a 2-deep branch predictor. When the processor starts, the predictions for branches A and B cannot be determined since there are no blocks previously so we will use the decoupling queue to hold arbitrary initial predictions for these 2 branches (my design initializes the queue with taken predictions). The initial state of the BP pipeline is shown in Figure 6-8 as “initial”.

Figures 6-8 through 6-11 show the evolution of the example cycle by cycle. Three structures are shown in each cycle: the 2-deep BP pipeline (BP0,BP1), the Fetch Stage 1 (F1), and the decoupling queue between them. (The initial head and tail pointers of the queue are arbitrary and as it happens they show the wrap-around case of the circular buffer.)

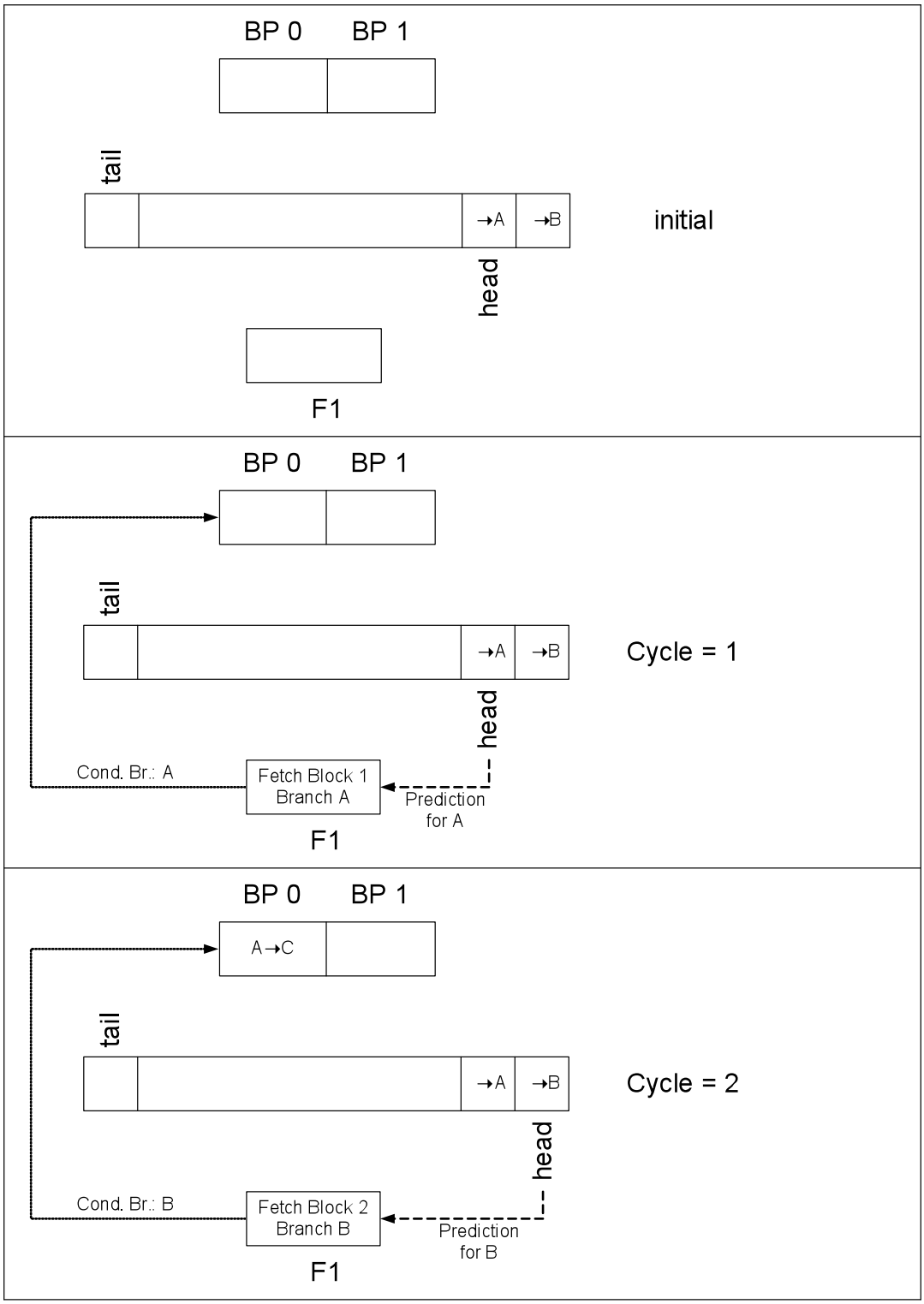

Figure 6- 8: Cycle by cycle change in state of the pipeline for decoupled effective ahead pipeline (Part 1).

Cycle 1: F1 fetches block 1 from the I-cache. F1 will have a BTB hit on conditional branch A (see Figure 6-7). F1 will read a prediction for branch A from the queue that we initialized to start the pipeline. It will produce the next PC to fetch from block 2. Since we have read from the queue, we will also increment the head of the queue. There was a hit for conditional branch A in F1. So we will trigger a read from the branch predictor in the next cycle with address A. See the state of the pipeline during cycle 1 in Figure 6-8.

Cycle 2: F1 fetches block 2. It will have a BTB hit on branch B (see Figure 6-7). We will use the prediction from the queue and increment the head. F1 will produce the next PC to fetch from block 3. We will also trigger a read from the branch predictor in the next cycle with B as the address. See the state of the pipeline during cycle 2 in Figure 6-8.

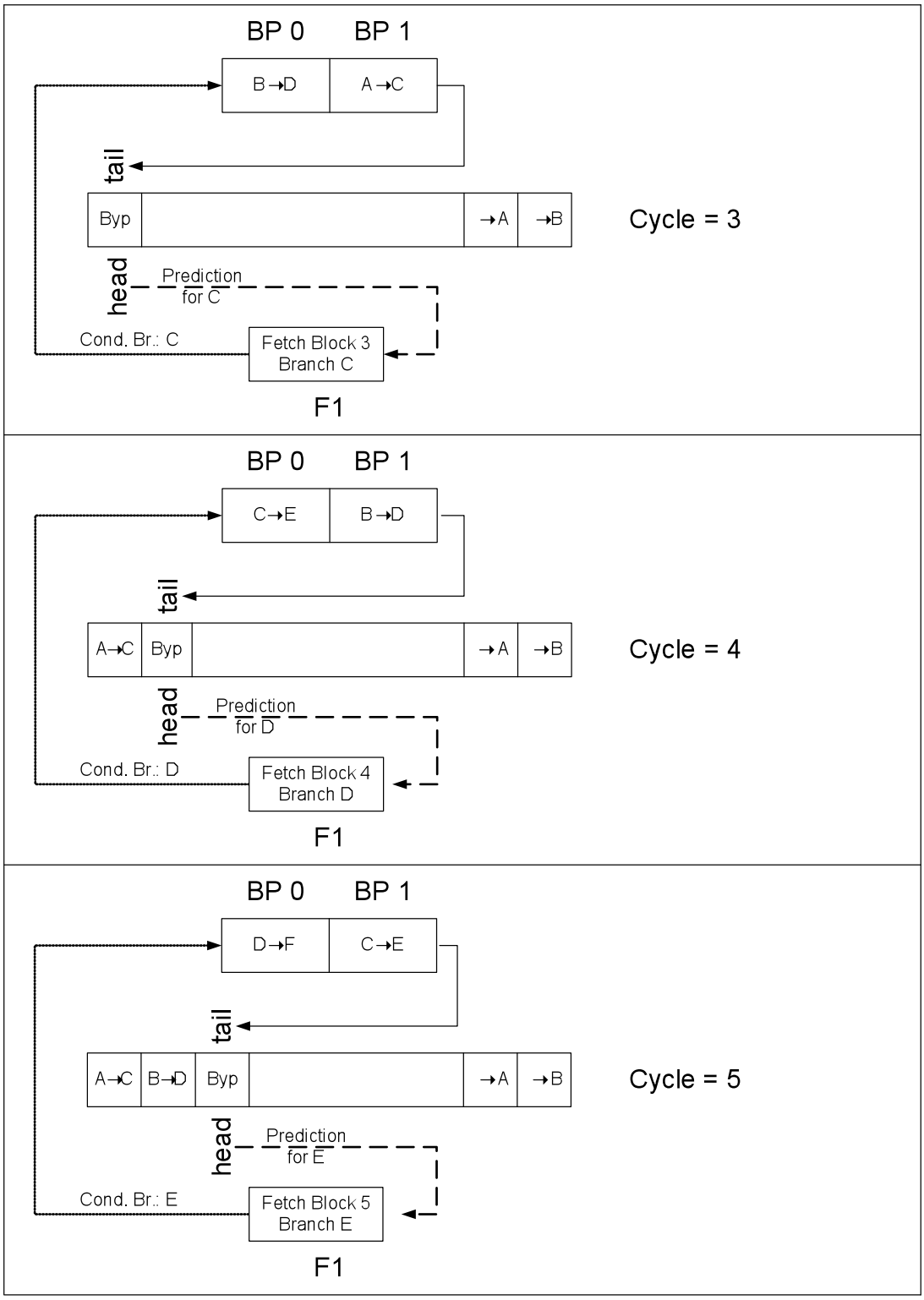

Figure 6- 9: Cycle by cycle change in state of the pipeline for decoupled effective ahead pipeline (Part 2).

Cycle 3: F1 fetches block 3. It will have a BTB hit on branch C (see Figure 6-7). The queue is empty. We will use the prediction that is given by the branch predictor. The access of the predictor for address A completes and will be ready to give a prediction. Since we used the prediction we will increment the head of the queue. We will write this prediction to the queue at the tail. An important item to notice is that we used the address of A to predict branch C. F1 will produce the next PC to fetch from block 4. We will also trigger a read from the branch predictor in the next cycle with C as the address. See the state of the pipeline during cycle 3 in Figure 6-9.

Cycle 4: F1 fetches block 4. F1 will have a BTB hit on D (see Figure 6-7). As in the previous cycle, we will use the prediction from the predictor since the queue is still empty. An important item to notice is that we used the address of B to predict branch D. We will write the prediction at the tail of the queue and increment the head like in the previous cycle. F1 will produce the next PC to fetch from block 5. We will also trigger a read from the branch predictor in the next cycle with D as the address. See the state of the pipeline during cycle 4 in Figure 6-9.

Cycle 5: F1 fetches block 5. F1 will have a BTB hit on E (see Figure 6-7). As in the previous cycle, we will use the prediction from the predictor since the queue is still empty. An important item to notice is that we used the address of C to predict branch E. We will write the prediction at the tail of the queue and increment the head like in the previous cycle. F1 will produce the next PC to fetch from block 6. We will also

trigger a read from the branch predictor in the next cycle with E as the address. See the state of the pipeline during cycle 5 in Figure 6-9.

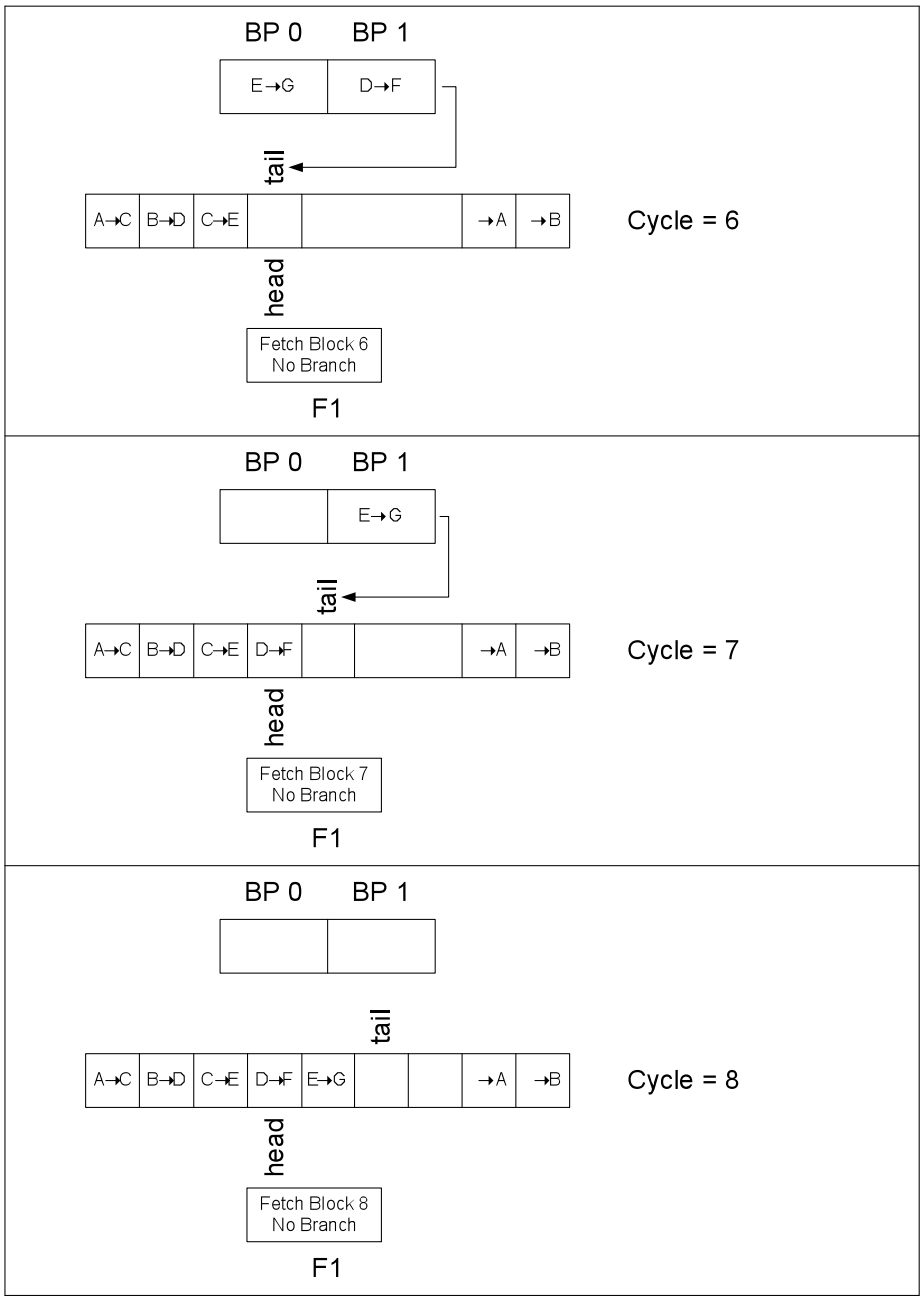

Figure 6- 10: Cycle by cycle change in state of the pipeline for decoupled effective ahead pipeline (Part 3).

Cycle 6: F1 fetches block 6. F1 will have no BTB hit since there are no branches in the fetch block. We will write the prediction at the tail of the queue. This prediction by D will be used whenever F will have a BTB hit. We are pre-reading the predictions into the queue. F1 will produce the next PC to fetch from block 7. We will not trigger a read in the next cycle since we do not have an address to read from the predictor. See the state of the pipeline during cycle 6 in Figure 6-10.

Cycle 7: F1 fetches block 7. F1 will have no BTB hit since there are no branches in the fetch block. We will write the prediction at the tail of the queue. This prediction by E will be used whenever G will have a BTB hit. We are pre-reading the predictions into the queue. F1 will produce the next PC to fetch from block 8. We will not trigger a read in the next cycle since we do not have an address to read from the predictor. See the state of the pipeline during cycle 7 in Figure 6-10.

Cycle 8: F1 fetches block 8. There is no valid prediction that is coming out of the predictor in this cycle. So we will keep the queue as it is. We will read the head of the queue in case we get a BTB hit for this cycle. F1 will have no BTB hit since there are no branches in the fetch block. F1 will produce the next PC to fetch from block 9. We will not trigger a read in the next cycle since we do not have an address to read from the predictor. See the state of the pipeline during cycle 8 in Figure 6-10.

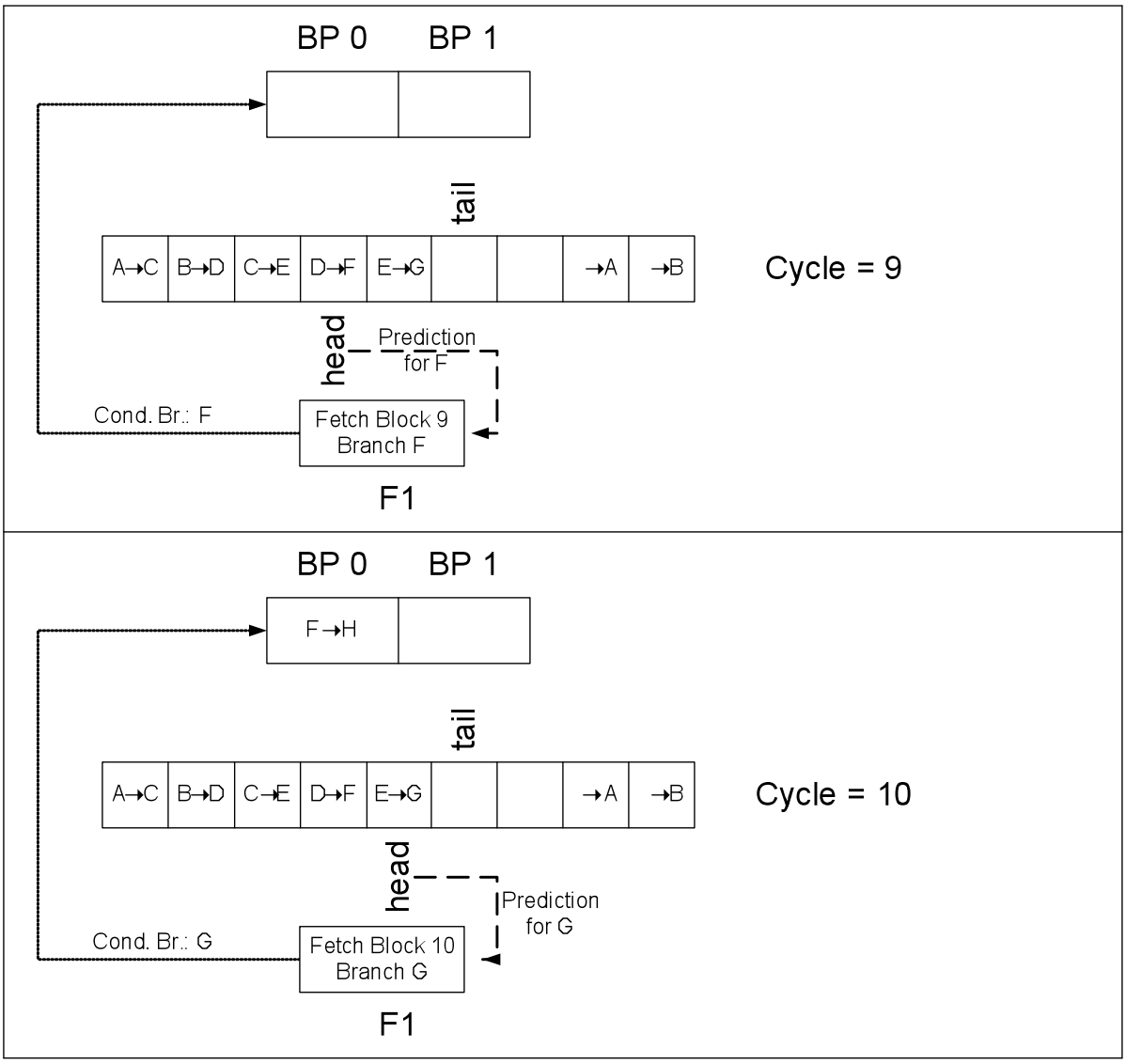

Figure 6- 11: Cycle by cycle change in state of the pipeline for decoupled effective ahead pipeline (Part 4).

Cycle 9: F1 fetches block 9. F1 will have a BTB hit on branch F. We will use the prediction from the queue and increment the head (prediction made by D for F). There is no valid prediction that is coming out of the predictor in this cycle. F1 will produce the next PC to fetch from block 10. We will also trigger a read from the branch

predictor in the next cycle with F as the address. See the state of the pipeline during cycle 9 in Figure 6-11.

Cycle 10: F1 fetches block 10. F1 will have a BTB hit on branch G. We will use the prediction from the queue and increment the head (prediction made by E for G). There is no valid prediction that is coming out of the predictor in this cycle. F1 will produce the next PC to fetch from block 11. We will also trigger a read from the branch predictor in the next cycle with G as the address. See the state of the pipeline during cycle 10 in Figure 6-11.

Thus, we saw how the queue is used to pre-read entries. This buffer helps us to decouple the branch predictor from the rest of the pipeline.

6.2.4 Handling BTB misses

BTB misses are handled by keeping delayed versions of the head and tail of the queue. By keeping these, we can recover the head and tail of the queue. For a N-deep I-cache, the delayed head and tail will be delayed by N cycles. When there is a BTB miss, we flush Fetch Stage 1 to start fetching from the correct address. We need to see how the branch predictor pipeline should be flushed in the case of a BTB miss. Figure 6-12 shows the stages of the branch predictor to be flushed for a 3-deep branch predictor with 1/2/3/4 deep I-cache.

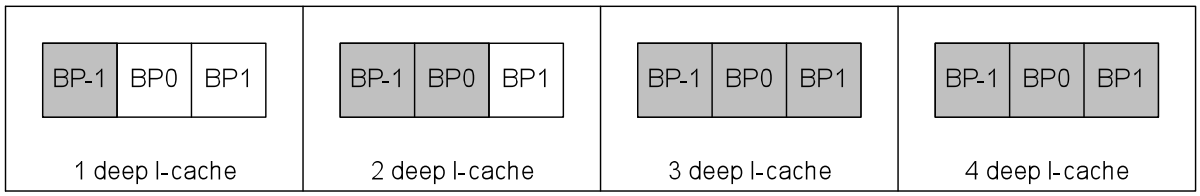

Figure 6- 12: Stages of 3-deep branch predictor to be flushed for different depths of I-cache.

6.2.5 Handling Branch Mispredictions

To recover from a misprediction, we need to checkpoint the head of the queue. This is the entry that predicted the branch. There are two types of control instructions that can mispredict: conditional branches and indirect branches. In case a conditional branch mispredicts, we need to restore the head to the <checkpointed_head>+1 entry. If an indirect branch mispredicts, then we need to restore the head to <checkpointed_head>. As we had seen, when we restart the pipeline, we need entries in the queue to get predictions. Since we have pre-read the entries from the branch predictor, we restore the tail to <restored_head>-depth of the branch predictor.

One corner case appears in the case of an I-cache miss since it causes a long stall in fetch. In this case, it is possible that there are fewer than <depth> entries present in the predictor queue after the <restored_head>. The branch predictor is stalled because of the I-cache miss. The required predictions are still getting read from the queue. When the misprediction signal arrives, we need to find out how many entries are present in the queue after the restored head. In case (<restored_head>-tail) < depth of the branch predictor, then we cannot flush the branch predictor fully. We

need to save the registers of the branch predictor, so that the number of entries in the queue will become depth once the branch predictor drains those entries into the predictor queue.

6.2.6 Handling Load Violations

We have seen in the earlier example that popping predictions from the queue does not actually remove them from the queue. We remove a prediction from the queue once the corresponding branch retires. So, we keep track of a retirement head in the queue. Whenever there is a load violation or other exception, we restore the head to the retirement head and restore the tail as we restore it for branch mispredictions (along with the corner case).

6.2.7 Final Design of the Predictor Queue

Let us put together all the parts of the predictor queue to handle all the cases. Figure 6-13 shows the parts of the queue with all the mini-queues and their respective heads and tails. This queue is implemented as a circular FIFO.

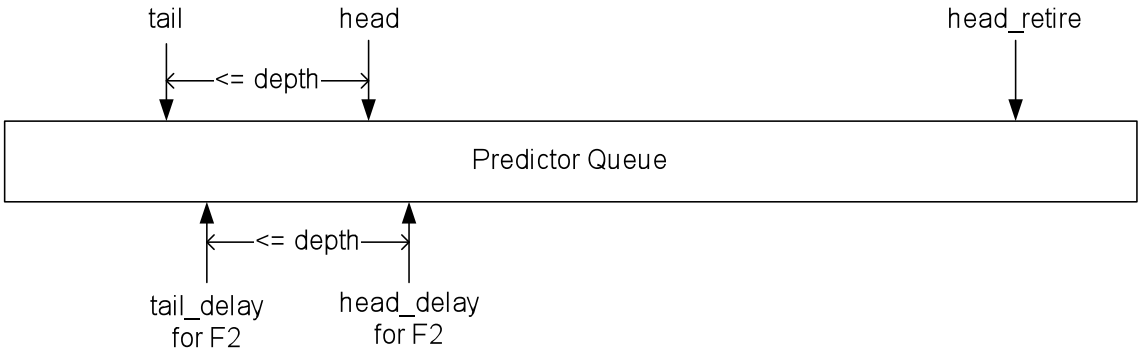

Figure 6- 13: Design of the decoupling queue.

To update the BP with the correct index, we need to keep the lower bits of the PC that was used to predict the branch. Thus, in this queue, we also write the lower bits of the PC that was used to predict the branch. The ROB retires branches at a rate of less than or equal to <retire_width>. But we need to retire branches from the predictor queue for updates at a rate of 1 per cycle. Thus, the retire head splits into 2 heads: commit and retire. The retire head keeps track of branches retired from the ROB and the commit head slowly drains updates to the predictor at the rate of 1 per cycle. The queue structure requires 2 read ports (1 for head read and 1 for update) and 1 write port (to write predictions at the tail).

6.2.8 Introduction of Global History Register

The global history register is introduced in Fetch Stage 1. Each time Fetch-1 triggers a read from the predictor, it sends the address of the branch as well as the global history register to the branch predictor. Since the branch in Fetch-1 is being used to predict a future branch (<depth> branches away), a fixed number of future bits (<depth> number of bits) is missing in the global history register for the prediction. These missing bits will all be available by the time the prediction is read out, but none are available at the time the predictor is indexed.

To handle the problem of N missing history bits for a N-deep branch predictor, we read out 2^N candidate predictions from the predictor table and store them together

in the predictor queue. The desired prediction is post-selected from among the candidate predictions using the N most recent bits of the global history register.

Note that the post-selection is based on the N most recent bits of the global history register XORed with the low N bits of the PC of the branch that was used to index the predictor. To speedup the post-selection logic, we can swizzle the N candidate predictions in the predictor queue according to the bits of the PC and then post-select the prediction using just the N bits of history. Table 6-2 shows how swizzling is done for a 2-deep predictor: the value of the low 2 bits of the PC ("PC bits" column) determines the order of the four predictions in the predictor queue (indicated by the "Index" column).

Table 6- 2: Order of swizzling of predictions.

PC bits	History bits	Index
00	00	00
	01	01
	10	10
	11	11
01	00	01
	01	00
	10	11
	11	10
10	00	10
	01	11
	10	00
	11	01
11	00	11
	01	10
	10	01
	11	00

6.3 Evaluation of the Decoupled Effective Ahead Pipeline in the C++ Simulator

We first implemented DEAP in the C++ cycle-level simulator. Performance results from this simulator are presented in this section. The branch predictor is a 2^{16}-entry gshare predictor with 16 bits of global history. (The BTB has 1,024 entries and is 4-way set-associative.)

Table 6-3 and Figure 6-14 show misprediction rates of the baseline BP and the pipelined BP with depths of 1 through 4. The misprediction rates are quite close to those projected by the limit study (see Table 6-1).

Table 6- 3: Misprediction rates of a 2^{16}-entry gshare predictor for different depths.

Benchmarks	Baseline	Depth =1	Depth =2	Depth =3	Depth =4
bzip	10.05	10.43	10.67	10.71	10.83
crafty	5.25	5.16	5.30	5.60	5.65
gap	2.34	2.64	2.93	2.98	2.99
gcc	4.17	4.50	4.79	5.01	5.14
gzip	8.39	8.49	8.59	8.82	8.97
mcf	2.41	2.36	2.39	2.41	2.44
parser	5.50	5.49	5.58	5.59	5.67
perl	2.10	2.19	2.44	2.63	2.91
twolf	10.61	10.72	11.04	11.20	11.38
vortex	0.54	0.46	0.87	0.92	1.67
vpr	10.06	10.17	10.24	10.20	10.27
Average	**5.58**	**5.69**	**5.89**	**6.01**	**6.17**

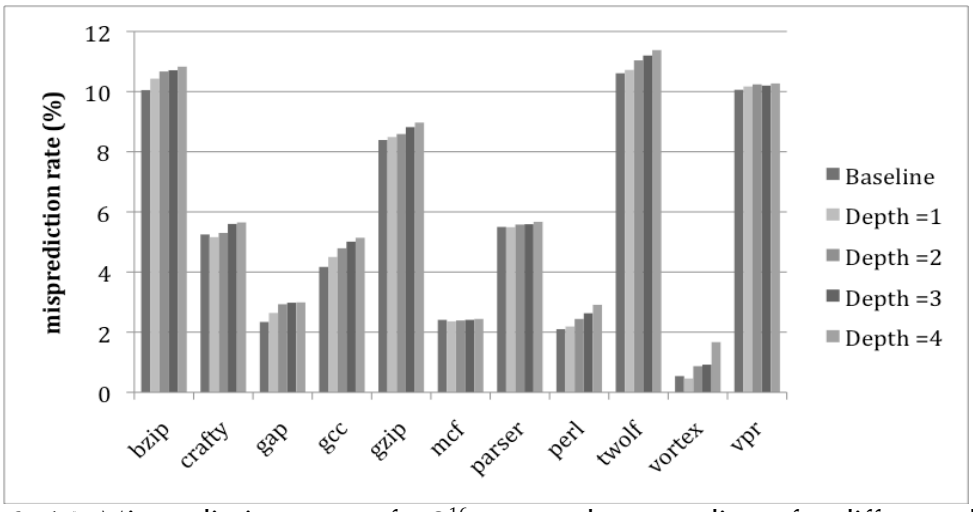

Figure 6- 14: Misprediction rates of a 2^{16}-entry gshare predictor for different depths.

Since the accuracy goes down by a small amount we need to see how much it affects the performance of the processor. Table 6-4 and Figure 6-15 show performance in terms of instructions per cycle (IPC) for different depths of the branch predictor.

Table 6- 4: IPC of processor with a 2^{16}-entry gshare predictor for different depths of the branch predictor.

Benchmarks	Baseline	Depth =1	Depth =2	Depth =3	Depth =4
bzip	0.86	0.85	0.85	0.85	0.84
crafty	0.87	0.87	0.87	0.86	0.86
gap	0.81	0.81	0.80	0.80	0.80
gcc	0.93	0.93	0.92	0.91	0.91
gzip	0.64	0.64	0.64	0.63	0.63
mcf	0.15	0.15	0.15	0.15	0.15
parser	0.98	0.98	0.98	0.98	0.97
perl	0.87	0.86	0.86	0.86	0.85
twolf	0.82	0.82	0.81	0.81	0.80
vortex	1.14	1.15	1.13	1.13	1.11
vpr	0.62	0.62	0.60	0.60	0.60
Average	**0.79**	**0.79**	**0.78**	**0.78**	**0.77**

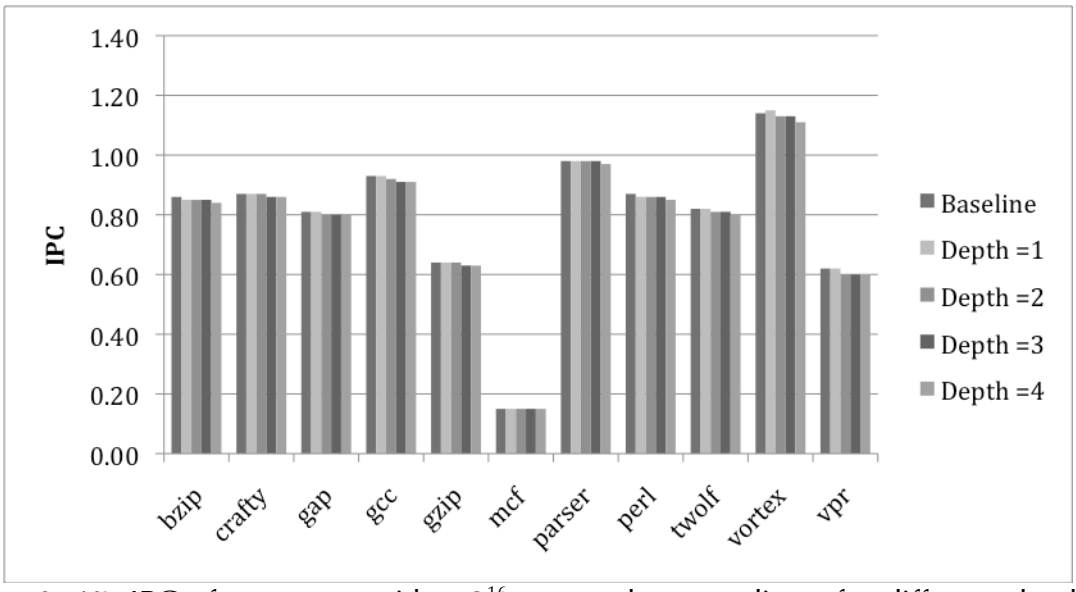

Figure 6- 15: IPC of processor with a 2^{16}-entry gshare predictor for different depths of the branch predictor.

The maximum performance drop is 3.3%, and 1.8% on average. Thus, we see that we can decouple the predictor from the rest of the pipeline and still achieve high performance. This study shows that we can use large and complex predictors and achieve high performance. Pipelining is required since we cannot fit the large predictor in 1 cycle for an aggressive processor. But now it can be pipelined with DEAP and the processor can get the performance it needs from the predictor.

6.4 Evaluation of the Decoupled Effective Ahead Pipeline in the RTL Simulator

I next designed and verified the RTL model of a fetch unit with DEAP and integrated it with the RTL model of the whole superscalar processor. This section evaluates the RTL implementation of DEAP. As in the previous section, the branch predictor is a 2^{16}-entry gshare predictor with 16 bits of global history.

Table 6-5 and Figure 6-16 show the misprediction rates of the baseline BP and pipelined BP of different depths.

Table 6- 5: Misprediction rates of the 2^{16}-entry gshare predictor for different depths.

Benchmarks	1-cycle gshare	BP Depth			
		1	2	3	4
bzip	10.13	10.49	10.65	10.82	10.89
gap	2.35	2.62	2.95	2.99	2.99
gzip	8.34	8.54	8.65	8.81	9.01
mcf	2.43	2.34	2.39	2.42	2.44
parser	5.49	5.47	5.55	5.63	5.69
vortex	0.56	0.43	0.90	0.95	1.60

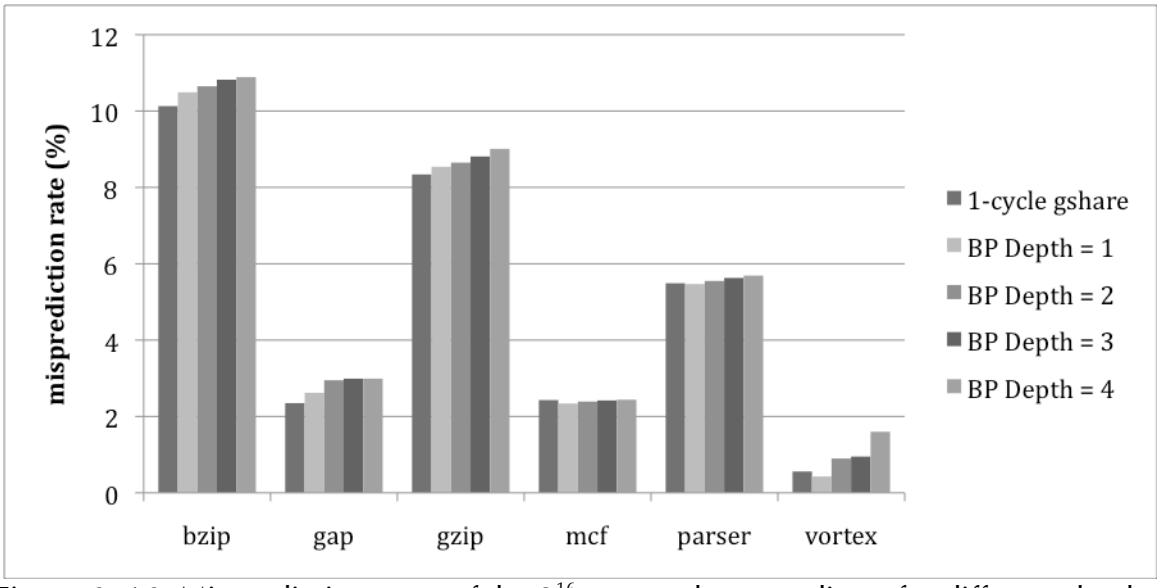

Figure 6- 16: Misprediction rates of the 2^{16}-entry gshare predictor for different depths.

We can see that even in the RTL simulator, the accuracy of the branch predictor goes down by around 0.6% on average as in the case of the C++ simulator. Let us look at the IPC of the processor. Table 6-6 and Figure 6-17 show the IPCs of the processor in the RTL simulator.

Table 6- 6: IPC of the processor with a 2^{16}-entry gshare predictor for different depths of the branch predictor.

Benchmarks	1-cycle gshare	BP Depth			
		1	2	3	4
bzip	0.81	0.81	0.81	0.81	0.8
gap	0.82	0.82	0.81	0.81	0.81
gzip	0.65	0.65	0.65	0.65	0.64
mcf	0.89	0.89	0.89	0.89	0.89
parser	0.76	0.76	0.76	0.75	0.75
vortex	0.75	0.76	0.75	0.74	0.73

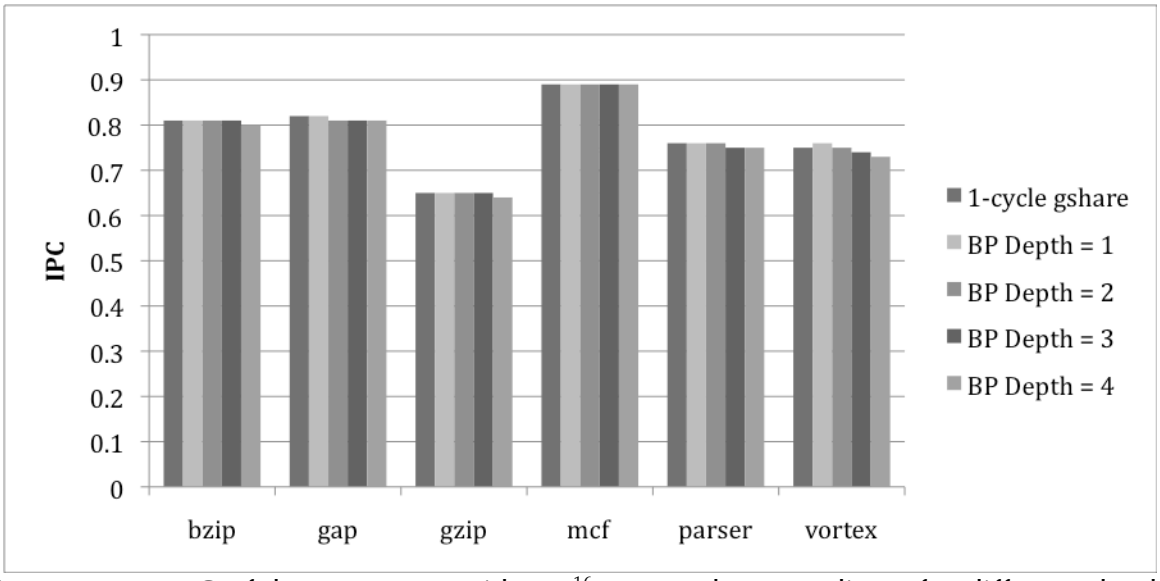

Figure 6- 17: IPC of the processor with a 2^{16}-entry gshare predictor for different depths of the branch predictor.

The drop in IPC does show the same trend as it shows in the C++ simulator. Since we cannot run the simulations on all the benchmarks, the average drop in IPC among these six benchmarks was seen to be around 1.3%. Thus, we see that the effect of pipelining on accuracy and IPC is minimal.

Let us see the improvement of pipelining on frequency of the processor. For this experiment, I will assume that the size of Branch Predictor (BP) will always be the

bottleneck and that the BTB size will never come into the delay of Fetch Stage 1. Figure 6-18 shows how the delay of Fetch Stage 1 varies with different depths of the branch predictor. The first bar in the graph is for the baseline 1-cycle gshare predictor and the rest of the bars are for the pipelined gshare predictor of depths 1 through 5. I am fixing the size of branch predictor table to be 16KB (2^{16} entries).

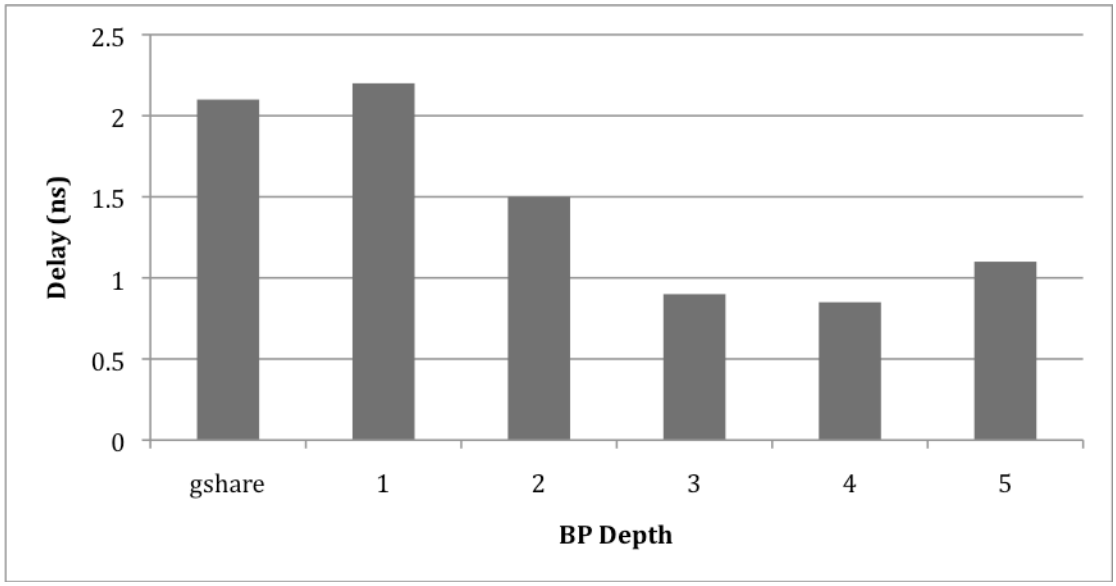

Figure 6- 18: Variation in delay of Fetch Stage 1 with different BP depths.

We can see from the figure the benefit of pipelining the branch predictor. If the branch predictor dictates the frequency of the processor, our technique can increase the frequency of the processor by almost 2.5 times with minimal impact on IPC. Pipelining the branch predictor beyond three deep yields diminishing returns and even a reversal in the delay trend, because the cost of post-selection increases exponentially. This is due to the fact that the number of missing bits of global history

increases with the depth of the branch predictor. Overall, we see that our simple design pipelines the branch predictor effectively.

CHAPTER 7

Summary

Throughout the last few chapters, I described RTL designs of two different pipelined instruction fetch units and discussed their advantages and disadvantages. Figure 7-1 shows a model of a pipelined fetch unit that improves performance of the processor through the combination of a fast cycle time, high accuracy and 1 prediction per cycle. The variation can be in the depth of the I-cache and depth of the branch predictor.

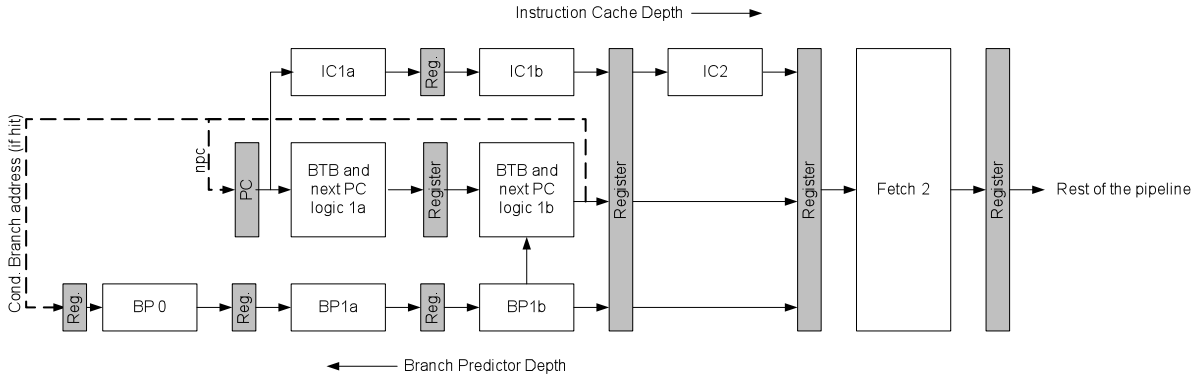

Figure 7- 1: Model of instruction fetch unit with I-cache depth and decoupled effective ahead pipeline.

In this thesis, I have designed, verified and evaluated RTL models of two different fetch unit designs.

1. The first fetch unit design implements the original form of Block Ahead Prediction [2]. In the original form of Block Ahead Prediction, the BTB and BP are pipelined together.

2. The second fetch unit design separates the BP pipeline from the BTB and I-cache. This enables the BP to scale arbitrarily large while maintaining a fast cycle time. Seznec et al. described the basic approach to separately pipelining the BP and BTB [5]. Our specific design is distinguished in that our BP is *decoupled* from instruction fetching. The decoupling is facilitated by a queue between the BP and the Fetch 1 stage.

For both fetch unit designs, I did the design in two phases. In the first phase, I familiarized myself with the overall approach by implementing it in a C++ cycle-level simulator. In addition to gaining familiarity with the approach, positive results from the first phase – a negligible drop in accuracy and a bandwidth of one prediction per cycle – motivated moving to the second phase. In the second phase, I implemented the detailed synthesizable RTL model using the Verilog hardware description language. All RTL models have been successfully integrated with FabScalar-generated cores to facilitate whole-processor simulation of standard benchmarks. Thus, the RTL models of the two fetch units have been verified in the context of whole pipelines.

To summarize, the contributions of this thesis are as follows:

1. Design, verification and evaluation of a synthesizable RTL model of a 2-cycle pipelined fetch unit employing the original form of Block Ahead Prediction [2], in which the BTB and BP are pipelined together.
2. Design, verification and evaluation of a synthesizable RTL model of a fetch unit employing Decoupled Effective Ahead Pipelining (DEAP), in which there is a

separate N-cycle BP pipeline. A novel aspect is the decoupling of the N-cycle BP pipeline from instruction fetching, as explained above. The N-cycle BP pipeline could conceivably be combined with either a 1-cycle or 2-cycle BTB pipeline. The current design only implements the former and the latter is left for future work.

3. FabFetch, a novel toolset within the FabScalar framework [6][7]. With FabFetch, we can automatically generate synthesizable RTL designs of arbitrary fetch units, which differ in their fetch width, pipeline depth and sizes of structures. FabFetch adds multiple dimensions to the fetch stage in the FabScalar framework. The FabFetch-generated designs have been verified by simulating them with FabScalar-generated cores for 100 million instruction Simpoints [1] of 6 SPEC 2000 integer benchmarks.

For the 2-cycle pipelined fetch unit employing Block Ahead Prediction [2], a 40% reduction in cycle time was observed for all fetch widths with a meager 1.6% drop in instructions per cycles (IPC) with respect to the baseline 1-cycle predictor. In the fetch unit featuring DEAP, accuracy decreases by less than 0.6% and IPC decreases by only 3.3% due to pipelining the BP for 4 cycles. DEAP can be used to pipeline the BP for many cycles without much loss in IPC and accuracy. Moreover, due to pipelining, we get a many-fold increase in frequency. Both of the above implementations are able to sustain a bandwidth of 1 prediction per cycle.

7.1 Future Work

We need to study the power and area implications of the logic involved in the pipelining the branch predictor. We need to see whether putting more power in the branch predictor (by increasing its size and pipelining it) does improve EDP of the processor as suggested by [12].

It would be useful to see the power and area used by the L-Tage [13] and Perceptron [14] predictors with decoupled effective ahead pipelining to make a strong case for using them in current processors to improve performance of the processor.

We have seen that the accuracy of return and other indirect branches is not good, leading to drop in performance in certain benchmarks. To improve the performance of returns, we need to implement different RAS schemes like top-of-stack checkpointing [15]. For predicting the targets of indirect branches, we need to implement history-based indirect predictors [16]. Both schemes combined together will bring down the misprediction rate of other branch types.

REFERENCES

[1] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. "Automatically Characterizing Large Scale Program Behavior", *In the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, October 2002, San Jose, California

[2] A. Seznec, S.Jourdan, P. Sainrat, P. Michaud, "Multiple-Block Ahead Branch Predictors", *Proceedings of the 7th conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, October 1996

[3] Daniel A. Jiménez, "Reconsidering Complex Branch Predictors", *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Anaheim, CA, February 2003

[4] Scott McFarling, "Combining branch predictors", *Technical Report TN-36m, Digital Western Research Laboratory, pages 292-303,* June 1997

[5] A. Seznec, A. Fraboulet, "Effective ahead pipelining of instruction block address generation", *Proceedings of the 30th International Symposium on Computer Architecture (IEEE-ACM),* San Diego, June 2003

[6] Niket K. Choudhary, "A Synthesizable HDL Model for Out-of-Order Superscalar Processors." *M.S. Thesis, Department of Electrical and Computer Engineering, North Carolina State University,* August 2009

[7] Niket K. Choudhary, Salil Wadhavkar, Tanmay Shah, Sandeep Navada, Hashem H. Najaf-abadi, and Eric Rotenberg. "FabScalar", *in the Workshop on Architecture Research Prototyping (WARP), in conjunction with ISCA-36*, June 2009

[8] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Mi-chael Bucher, Sunil Basavarajaiah, Julie Oh, Ravi Jenkal, "FreePDK: An Open-Source Variation-Aware Design Kit," *pp.173-174, 2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, 2007

[9] Tanmay A. Shah, "FabMem: A Multiported RAM and CAM compiler for Superscalar Design Space Exploration." *M.S. Thesis, Department of Electrical and Computer Engineering, North Carolina State University,* August 2010

[10] J. F. K. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer, vol. 17, no. 1, pp. 6-22,* Jan. 1984

[11] Dong Ye, David Kaeli, "A reliable return address stack: microarchitectural features to defeat stack smashing", *ACM SIGARCH Computer Architecture News, v.33 n.1*, March 2005

[12] Dharmesh Parikh, Kevin Skadron, Yan Zhang, Mircea Stan, "Power-Aware Branch Prediction: Characterization and Design", *IEEE Transactions on Computers, v.53 n.2, p.168-186,* February 2004

[13] André Seznec, "The L-Tage Predictor", *Journal of Instruction Level Parallelism,* May 2007

[14] Daniel A. Jiménez and Calvin Lin, "Dynamic Branch Prediction with Perceptrons", *Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, Monterrey, NL, Mexico 2001

[15] Skadron K., Ahuja P. S., Martonosi M., Clark D. W, "Improving prediction for procedure returns with return-address-stack repair mechanisms", *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture. 259–271*

[16] Karel Driesen and Urs Hölzle, "Accurate Indirect Branch Prediction", *Proceedings of the 25th annual International Symposium on Computer architecture (ISCA '98). 167-178.*