Y-Pipe: A Conditional Branching Scheme Without Pipeline Delays

Michael J. Knieser and Christos A. Papachristou
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, OH 44106

Abstract. The "Y-Pipe" is a hardware conditional branching scheme that allows pipelined processors to perform conditional branches without any pipeline delays. The essence behind the Y-Pipe is the duplication of all pipeline stages before the execution stage of the pipeline. Given a scalar processor, under normal operations both paths are fetching and decoding the same instruction. After encountering a COMPARE-BRANCH instruction structure, the two paths in the Y-pipe will start fetching and decoding separate instructions. The scheme will work for consecutive COMPARE-BRANCH instruction structures. The Y-Pipe scheme has been simulated and compared against using the delayed branching scheme. The results show that the proposed Y-pipe scheme incurs no branch delays and provides 100% branch prediction.

1 INTRODUCTION

Many structural, control, and data hazards appear when assuming a pipelined design. One important control hazard is conditional branching. When an "if" conditional is encountered, the decision as to what to do after the "if" is known only after a few instructions have been fetched into the pipeline. These fetched instructions may be incorrect, thus causing the instructions to be flushed to allow for the correct instruction to enter the processor and be executed. Since 11%-17% of the time conditional branches are encountered in programs, it would be beneficial to keep the pipeline from being flushed every time a conditional branch is encountered [HenPat 90]. To solve this problem, many designers have implemented either a delayed branch approach or a statistical branch prediction approach. However, these approaches still encounter delays where the pipeline may have to be flushed or stalled.

This lead to the thinking that there must be a way to solve the unconditional branching dilemma without having to use delayed branching or statistical branching methods to eliminate pipeline delays. We came up with a hardware solution; the "Y-Pipe". The Y-Pipe is a hardware conditional branching scheme that allows pipelined processors to perform conditional branches without any pipeline delays. The essence behind the Y-Pipe is the duplication of all pipeline stages before the execution stage of the pipeline. Given a 5 stage pipeline with a fetch stage, a decode stage, an execution stage, a memory access stage, and a write back stage, the instruction fetch(IF) and the instruction decode(ID) stages would be duplicated. This would be like double fetching and double decoding instructions. Some years ago, the IBM 360/19 duplicated the IF stages, but treated each IF stage as a cache [HwaBri 84]. In our situation, the IF stages are merely instruction fetch stages. With the duplication of the pipeline stages, an instruction constraint is also necessary. The comparison instructions and the conditional branch instruction must be paired together. Given this software constraint and the additional hardware, no pipeline flushes or stalls will occur when a conditional branch in encountered.

2 ARCHITECTURE

2.1 Hardware

The basis of the hardware for the Y-Pipe is the duplication of all the pipeline stages before the execution stage. Assuming our 5 stage pipeline model, this would mean the instruction fetch and instruction decode stages would be duplicated. Figure 2.1 illustrates the processor's pipeline and the rationale behind the scheme's name. There are two paths in the pipeline through which instructions can be fetched and decoded. Under normal operation both paths are fetching and decoding the same instruction. After encountering a COMPARE-BRANCH instruction structure, the two paths in the Y-pipe will start fetching and decoding separate instructions. Path 1 will fetch the fall through instructions and path 2 will fetch the branch-target instructions. The result of the conditional branch will be determined before the instructions in path 1 and path 2 encounter the execution (EX) stage of the pipeline. This way the processor will operate without delays or stalls.

To make this happen, two extra bits are necessary. The bits are a Conditional Branch Bit (CBB) and an Active Path Bit (APB). The CBB functions as a flag to indicate when an encounter with a compare instruction happens. The APB's function is to indicate which IF and ID path is the correct path to be sent to the execution pipeline stage. The processor executes path 1 by default. Whenever encountering COMPARE-BRANCH structures in the pipe, this bit will be set accordingly to the outcome of the conditional branch. Afterwards, the APB is set back to the default condition.

To accomplish the Y-Pipe scheme, a two-phase system clock is necessary. This allows path 1 and path 2 to function independently of each other. Path 1 functions in phase 1 and path 2 in phase 2. A detailed description of the hardware architecture of the Y-pipe is in [KniPapa 92].
2.2 Software

It is also necessary to modify the relationship of certain instructions in a given processor. Normally, instructions are thought of as independent entities. We propose that the COMPARE instruction and the CONDITIONAL BRANCH instruction are not independent entities. We assume that the CONDITIONAL BRANCH is dependent on a COMPARE. This produces an ability to predict the future by a pipeline stage. When the processor encounters a COMPARE, CBB is set to flag this event. Having set CBB indicates to the processor that a CONDITIONAL BRANCH might follow and to be prepared for that instruction. The outcome of a CONDITIONAL BRANCH following a COMPARE is known in path 1 first, because it decodes itself first since all path 1 stages before the EX stage start functioning in phase 1. Given a CONDITIONAL BRANCH, the branch target address can be computed before the path 2 stages start functioning in phase 2. Therefore, the flow of instructions is not interrupted.

To make this happen the CONDITIONAL BRANCH must be modified. Current definitions of this instruction state that it performs an ALU comparison in the EX stage of a given pipeline. This is not possible given the Y-Pipe scheme because the decision comes too late. The Y-Pipe leaves the ALU comparison to the compare instruction making the branch instruction the carrier of the branch target address. Therefore the following changes occur to the processor’s COMPARE and CONDITIONAL BRANCH instructions.

- The COMPARE of a given processor set APB upon completion of the ALU compare in the EX stage.
- The CONDITIONAL BRANCH only holds the branch target address.

2.3 Interrupts

We have found precise interrupts to be the method of handling processor interruptions. Precise interrupts are defined as follows [SmiPle 88]:

1. All instructions before the save program counter have been executed.
2. All instructions after the saved program counter have not modified the process state.
3. If interrupt is caused by an exception condition raised by an instruction in the program, the saved program counter points to the interrupted instruction.

Under precise interrupts and the way interrupts appear, condition 3 may cause problems for the Y-Pipe scheme. If we receive a program interrupt which jumps to an interrupt service routine (ISR), precise interrupts say we are to return to the instruction which caused the interrupt. If this occurs, the compare-branch instruction structure may be broken causing difficulties when returning from an ISR. Therefore when any program returns from an interrupt, instead of returning to that instruction after the ISR, the processor would return to the last instruction that was to enter the WB stage of the pipe.

Due to the chance that an interrupt can disturb the compare-branch structure, it seemed necessary that we needed to keep a history of occurrences as to record the events of the compare-branch structure such that if an interrupt occurred, knowledge before that interrupt would be saved. We came up with the notion of Branch History Queues. The Branch History Queue would contain the both path 1 and path 2 program counters (PCI & PC), and APB and CBB. The depth of the Branch History Queue depends on the number of cycles necessary to complete a single compare-branch structure. In the 5 stage pipeline model, the depth of the Branch History Queue would be four. A compare-branch structure starts when CBB is set to when the correct instruction path is taken.

3 IMPLEMENTATION EXAMPLES

To show the functionality and feasibility of the Y-Pipe scheme, simulations and comparisons deemed necessary. We eventually choose to use the N.2 simulator, by TD Technologies, Inc., to test the Y-Pipe design. This simulator provides the ability for pipelining and parallel processing. The test strategy was to compare the Y-Pipe design to a delayed branching scheme. We used similar 5 stage pipelines as the model and created a compiler which would compile C programs into usable test code. We choose two examples to test the schemes.

3.1 Run-Length Coding

In image processing, it is useful to perform "run-length coding" on a binary image. Converting a binary image via run-length coding is a building block in shape recognition. The basics of run-length coding is converting horizontal scan lines of a binary image into a string of positions where a transition of '0'-to-'1' or '1'-to-'0' occur [Lee 90]. With the binary image in run-length coding form, the scanned image can now have the objects within itself separated for eventual shape identification of each object.

3.2 Lexical Analyzer

"Lexical analysis is the process of translating input into a form where input symbols are grouped into 'Grammatically significant' chunks call tokens, lexemes, or terminals" [Dietz 91]. In many cases compilers need to be fast, especially for those who are learning the language, due to the amount of time necessary to compile the program. The lexical analyzer we choose was quite simple and uses conditional branches to partition the input symbols.

3.3 Delayed Branching

Hennessy and Patterson have performed studies on delayed branching given a five stage pipe model. They have run traces on TeX, Spice, and GCC and found that on average 45% of the delay slots are filled with useful information [HenPat 90]. Given 11%-17% of the instructions in a program are conditional branch instructions, the best delayed branching can do is correctly predict 5%-8% of the conditional branches leaving 6%-9% to pipeline stalls or fetches.

When we ran our tests we kept this in mind.

4 RESULTS

We felt it necessary to isolate the conditional branches and test how many conditional branches were taken. We would record this result and compare this to the number of conditional branches encountered to produce a percent of
branches taken. This percentage of branches taken translates into the delay slot incorrectly filled.

Delayed Branching is nice in the sense that the performance losses are only accrued when a branch statement is taken in all of our cases. This is due to the compare instruction preceding all conditional branch statements. Therefore a branch penalty is encountered only when the conditional branch statement was true and had to branch to the target location. This is because the delay slot could not contain the instruction before the CONDITIONAL BRANCH because it was data dependent. Given 'while' and 'loop' statements in C, delayed branching only encountered branch penalties when exiting the 'while' and 'loop' statement. Therefore, the delay slot of the delayed branching scheme would optimally use the fall through instruction. The 'switch' statement was defined as a table loop-up. Only when encountering 'if-else' statements were the penalties the greatest.

The delay slot cannot be optimally filled because the context of the 'if' statement is not known. Therefore the programmer must optimally arrange 'if-else' statements. This is where the delayed branching scheme incurs the largest penalties.

Our results are based on probabilities of inputs ranging from .1 to .9. Comparison tests were run given each probability. For the Lexical analyzer, there were three possible inputs: character, digit, and punctuation. Each input was isolated and tested varying the probability. For the run-length coding, there was only one input; the picture. Again, we tested the input by varying the probability.

4.1 Lexical Analysis Results
The first graph, Fig. 4.1.1, shows the branch frequency of the lexical analyzer. The average frequency of conditional branches is about 7%. The next graph Figure 4.1.2, shows the branch penalty incurred given the two schemes. We observe that about 48% for the conditional branches were taken producing a 3.2% average performance degradation for delayed branching. The following graph, Fig. 4.1.3, illustrates the Y-Pipe performance, which is the ideal performance, to the the delayed branching performance, in terms of CPI. The worse case for delayed branching was a CPI of .942 producing a 6.2% performance degradation. The best case for delayed branching was a CPI of .983 producing a 2.45% performance degradation.

4.2 Run-Length Coding Results
The first graph, Fig. 4.2.1, shows the branch frequency of the run-length coder. The average frequency of conditional branches is about 6%. The next graph Figure 4.2.2, shows the branch penalty incurred given the two schemes. This shows about 41% for the conditional branches were taken producing a 2.48% average performance degradation for delayed branching. The following graph, Figure 4.2.3, illustrates the Y-Pipe performance to the the delayed branching performance, in terms of the CPI. The worse case for the delayed branching was a CPI of .963 producing a 3.8% performance degradation. The best case for delayed branching was a CPI of .980 producing a 2.0% performance degradation.

5 CONCLUSION
The paper proposes a conditional branching scheme, the Y-Pipe, which will incur no delays when encountering conditional branches given some hardware and software constraints. The software constraint is that the compare instruction must precede the conditional branch instruction. Also, the conditional branch instruction just contains the branch target address and only the compare instruction actually performs an ALU comparison and set the APB accordingly. The hardware requirement is the duplication of all pipeline stages before the execution(EX) stage of the pipeline in addition to the 2 bits controlling the compare branch structure.

The Y-pipe guarantees 100% branch prediction of conditional branches. Results comparing delayed branching and the Y-Pipe show the performance losses of delayed branching. The performance losses of delayed branching range from 2% to 6% given a five stage pipeline model.

Our future plan is to apply the Y-Pipe scheme to a superscalar design. Preliminary designs have shown a method to achieve a superscalar Y-Pipe design which will guarantee 100% conditional branch prediction.

References
[Dicta 91] D. Dicta, “Introduction To Compilers And Translation Engineering,” EE495D Course Notes, School of Electrical Engineering, Purdue University, August 1991 Revision, pp. 24-33.
Figure 2.1: Y-Pipe Pipeline Stages

Figure 4.1.3: CPI

Figure 2.1: Branch Frequency

Figure 4.1.1: Branch Frequency

Figure 4.1.2: Branch Penalty

Figure 4.2.1: Branch Frequency

Figure 4.2.2: Branch Penalty

Figure 4.2.3: CPI