Fetch bottleneck and Branch penalty reduction using 2 instruction pre-fetch queues

Guru Prasadh V. Venkataramani, Hemanth Kumar Manoharan, Ranjani Parthasarathi
vvguruprasadh@yahoo.co.in, hemanth@cs.annauniv.edu, rp@cs.annauniv.edu

Abstract— **This paper proposes an idea to reduce branch penalty in a pipelined architecture by making novel use of two instruction pre-fetch queues, primary and auxiliary. When there is no branch instruction, both the queues together act as a single queue, referred to as an alternate mode of operation. Whenever a branch instruction is present in the current instruction mix, there is a switch over to the branch mode of operation in which the primary queue fetches from the address returned by the predictor while the auxiliary queue fetches from the other path until the branch address is resolved. Hence the next instruction (irrespective of whether the branch is taken or not) is always available. The idea has been implemented using simplescalar toolset 3.0 and tested on SPECINT2000 benchmarks, and a 25-55% improvement in IPC has been observed. A comparison of our idea with other architectures with two instruction fetch queues is also presented.**

Index Terms—**Instruction fetch queues, Instructions per cycle, simplescalar simulator, both path pre-fetching.**

I. INTRODUCTION

Today, pipelined processors are designed to exploit Instruction Level Parallelism in the program. Instruction misses in the fetch stage badly affect the performance of such microprocessors by introducing large delay cycles. Pre-fetching of instructions is one way of minimizing delays. This paper presents a pre-fetching technique wherein there is an additional instruction fetch queue apart from the normal fetch queue. The aim is to increase the fetch bandwidth and also to facilitate both-path pre-fetching when branches are encountered. The queue from which the dispatch stage consumes instructions at any point of time is labeled as the primary queue. The other one is called the auxiliary queue. Both the primary and the auxiliary queues are identical and operate in parallel. They have their own read ports and can fetch instructions simultaneously.

II. RELATED WORK

In the IBM 360/91 architecture [2], there was a separate branch target instruction queue in its design. When the processor decoded an instruction and detected a branch, it fetched the first four instructions of the branch target path and placed them into the branch target queue. Simultaneously, the processor determined the outcome of the branch. Depending on the outcome, the processor fetched from either inline instruction or branch target queue.

The Pentium [2] employs two instruction queues, namely, alternate inline instruction queue and a branch target buffer. Note, the second queue is used to store branch target addresses and hence is completely different in functionality from the other queue. The processor updates the history information in the branch target buffer. As the processor decodes instructions, it searches the buffer to search if there is a corresponding entry for that branch instruction. When there is a match, the processor determines to see if the branch should be taken. If so, it uses the target address previously stored in the branch target buffer to begin fetching and decoding instructions from the target path. The processor uses the inline instruction queue not being used, which begins to prefetch. If the branch is mispredicted, the processor flushes the instruction queue and the instruction prefetching starts over.

The IBM 370/165 [3][6] employs a branching strategy that fetches along two paths, the inline path and the target path. Two buffers are maintained, the main instruction buffer (MIB) and the auxiliary instruction buffer (AIB). When the effective address of the target instruction stream is known at stage n, the target stream is fetched into the AIB. Both streams are processed until the outcome is known.

The general organization of the rest of the paper is as follows. Our idea on instruction pre-fetching is discussed in section III, simulation details on the simplescalar toolset 3.0 are given in section IV, comparison with other architectures is outlined in section V and conclusion is presented in section VI.

III. OUR IDEA

Our technique was designed to integrate into the pipeline of a processor with support for speculation. The pipeline stages used for the purpose of discussion are fetch, dispatch, issue, execute, writeback and commit. The outcome of a branch is known in the writeback stage.

A. Branch Tables

A branch table is a small buffer, with around 8 entries, in which each entry contains the prediction direction for a branch, its taken and not-taken addresses. These tables keep track of branches encountered in the primary and auxiliary queues. While the primary queue branch table may have multiple entries, the auxiliary queue branch table, logically, has provision for just one entry. This is because, there is no provision for recursive both-path prefetching and hence there is no need remember more than one branch at any point of time. The idea becomes clearer on examining the technique presented here.

B. Instruction Pre-fetching

There are two modes of operation viz. the alternate mode and the branch mode. These modes are described in the following paragraphs.

i. Alternate Mode

The functionality of the queues varies with the current instruction mix. There are two modes of operation namely the branch mode and the alternate mode. The system, by default, starts off in alternate mode and remains in the alternate mode as long as the instruction mix consists of only Arithmetic and Logic Unit (ALU) instructions and load/store instructions. In the alternate mode, instructions 1 to N are fetched by the primary queue. Instructions N+1 to 2N are fetched by the auxiliary queue. As the first N instructions in the primary queue are consumed by the dispatch stage, instructions from 2N+1 onwards will be fetched by the primary queue. In short, when the first instruction leaves the primary queue, the process of fetching the 2N+1$^{\text{th}}$ instruction would have been initiated. Thus, we get a pre-fetch lookahead of 2N instructions in the alternate mode.

If a branch instruction appears in the primary queue when we are in the alternate mode, there is a switch over to the branch mode and the branch predictor is used to predict the target address. The next instruction fetched by the primary queue is the instruction at the address returned by the branch predictor. The auxiliary queue starts fetching in the other path associated with that branch. The term "other path" refers to the address, which contradicts the decision of the branch predictor.

If a branch instruction appears in the auxiliary queue while in the alternate mode, the auxiliary queue simply stops pre-fetching and waits till it becomes the primary queue. Meanwhile, it puts an entry into the auxiliary queue branch table.

Whenever there is a queue switch, the auxiliary queue branch table is consulted to check if there is any branch to be serviced; if yes, there is a switch over to branch mode for that branch.

ii. Branch Mode

Now when a branch instruction goes to the writeback stage and if it is correctly predicted, the instructions brought into the auxiliary queue are flushed, as they serve no purpose. But, since these instructions are in a different queue with a separate read port, this will not degrade the performance of the existing pipeline architecture.

If on the other hand, the branch is wrongly predicted, the auxiliary queue contains useful instructions or at least the process of fetching useful instructions would have already begun. Now the primary queue is flushed and a simple switch of queues will serve the purpose. Here the latency associated with the branch penalty will either not be incurred at all or the latency will be lesser.

In either case (branch being correctly predicted or incorrectly predicted), there is a switch back to alternate mode after the branch goes to completion because the branch mode of operation is with respect to a particular branch.

The complexity arises when branches appear on the speculated instruction path, where the primary queue has more than one branch instruction. In this case, if speculative execution proceeds, "recursive both path pre-fetching" will necessitate a new auxiliary queue for every branch in the speculated path. It is for this reason that we restrict the level of nesting to one. So while remaining in the branch mode for the original branch, we simply make an entry (for every branch instruction encountered in the primary queue) into a branch table and proceed. When the original branch runs to completion and is correctly predicted, the branch table is consulted if any branch remains to be serviced; if yes, there is a switch over to the branch mode for that branch and the process proceeds as explained before. In case the branch runs to completion and is wrongly predicted, the branch table is flushed, as the branches in the speculated path are meaningless.

If a branch instruction appears in the auxiliary queue when in branch mode, the fetch operation in the auxiliary queue is stalled and the auxiliary queue waits to become the primary queue. This is done because speculation is not supported in the auxiliary queue.

IV. SIMULATION DETAILS

The simplescalar simulator 3.0 was used to implement our idea in the pipeline of a superscalar processor with extensive support for speculation. The simulator was then run to gather statistics like number of branches mispredicted, number of branch penalties saved, number of cycles for which the auxiliary queue read port is used.

From the statistics gathered, the percentage of mis-predictions recovered without penalty and the percentage of utilization of the auxiliary read port has been calculated. The results obtained on running the SPECINT2000 benchmarks are shown in Fig. 1, Fig. 2 and Fig. 3 namely IPC comparison graph, penalty reduction percentage graph and auxiliary port utilization graph.

From Fig.3, it can be seen that the auxiliary port utilization is between 80-95%, which justifies the introduction of the new hardware port added. The percentage of mispredictions recovered without penalty varies between 40-80%. All this leads to an overall improvement in Instructions Per Cycle (IPC) by 25-55%.

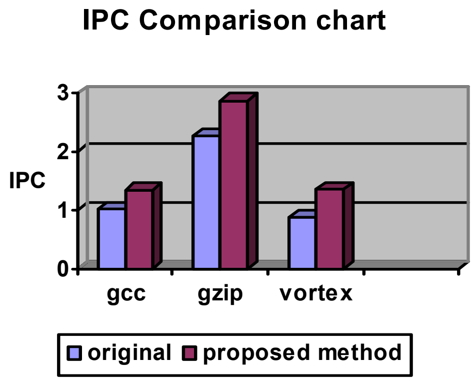

Figure 1

	Gcc	Gzip	Vortex
Original	1.0103	2.2674	0.8682
proposed method	1.3427	2.8639	1.355

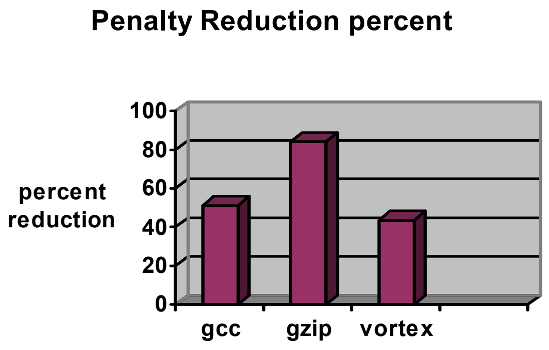

Figure 2

	Gcc	Gzip	Vortex
proposed method	50.86	83.98	43.3

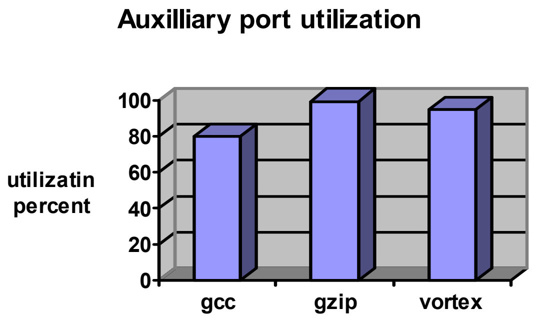

Figure 3

	Gcc	Gzip	Vortex
Proposed method	79.79	99.8	95.26

V. COMPARISON WITH OTHER ARCHITECTURES

In our idea, when branch instructions are not present in the current instruction mix, both the primary and the auxiliary queues together act as a single queue, that is, after consuming instructions 1 to N in the primary queue, the dispatch stage starts consuming instructions from N+1 to 2N from the other queue (which was previously the auxiliary queue and now has become the primary queue).

Thus, the main difference between our approach and those implemented in other architectures is that in other approaches, the auxiliary queue is utilized only when there are branches in the instruction queue whereas we make use of the auxiliary queue all the time.

The main advantage of our approach is that it merges seamlessly with the speculated mode of execution. The primary queue can continue in the speculated mode of execution and when the earliest speculated branch fails due to misprediction, the dispatch stage can shift over to the auxiliary queue. Hence, the existing speculated pipeline need not be modified much. This design hence addresses the power considerations arising out of the additional hardware needed for the new idea.

In contrast, the secondary queues in the IBM 370/165 and IBM 360/91 fetch only from the branch target i.e. branch taken address. If the branch were speculated to be taken, then both the queues would start fetching redundantly from the target address. This would not happen in our case.

VI. Conclusion

In this paper, we have presented a technique that would improve the execution time by reducing branch penalty. An analysis of the results after simulating the idea shows good promise. Although this technique requires an additional read port, high utilization of it can be taken as a justification for its presence. Its ability to merge seamlessly with today's speculated pipelined architectures is a major point in its favor.

References

[1] Jim Pierce, Trevor Mudge G. O. Young, “Wrong path instruction prefetching”, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor.
[2] Micheal K. Milligan, Harvey G. Cragon, “Processor implementations using queues”, University of Texas at Austin
[3] G. Alaghband, “Key elements of a computing system and their relationships”, Parallel computation and architectures
[4] John L Hennessy, David A Patterson, “Computer Architecture A Quantitative Approach”, Second Edition, Morgan Kaufmann Publishers, 1995
[5] SimpleScalar toolset http://www.simplescalar.org
[6] Eager execution –Mark Smotherman http://www.cs.clemson.edu/~mark/eager.html
[7] Harvey Cragon, ”Branch Strategy taxonomy and performance models”, Los Alamitos, CA, IEEE Computer Society Press, 1992
[8] I. Flores, "Lookahead Control in the IBM System 370 Model 165," IEEE Computer, November 1974

Guru Prasadh V. Venkataramani is currently doing B.E. computer science and engineering at College of Engineering, Anna University, Chennai, India. His fields of interest include computer architecture and compilers.

Hemanth Kumar Manoharan is currently doing B.E. computer science and engineering at College of Engineering, Anna University, Chennai, India. His fields of interest include computer architecture and embedded systems especially on aircrafts.

Ranjani Parthasarathi is currently assistant professor of computer science and engineering at the School of Computer Science and Engineering, Anna university, Chennai. She received her Ph.D. degree from the Indian Institute of Technology, Madras. Her fields of interest include computer architecture and reconfigurable computing.