CACHELINE OVERRIDE IN BRANCH TARGET BUFFER USED IN BRANCH HANDLING

Jisha P Abraham¹ and Dr. Sheena Mathew²

¹ M. A. College of Engineering, Kothamangalam, Kerala, India
² Department of Computer Science, Cochin University of Science and Technology, Cochin, Kerala, India

Abstract: The performance of the processor is highly dependent on the supply of correct instruction at the right time. Whenever a data miss is occurring in the cache memory the processor has to spend more cycles for the fetching operation. One of the methods used to reduce instruction cache miss is the instruction prefetching, which in turn will increase the number of instruction to be supplied to the processor. Even modern processors use branch target buffer to predict the target address of branches such that they can fetch ahead in the instruction stream increasing the concurrency and performance of the processor[1]. All the developments in these fields indicate that in future the gap between processing speeds of processor and data transfer speed of memory is likely to be increased. Branch Predictor play a critical role in achieving effective performance in many modern pipelined microprocessor architecture. Commonly used methods for branch predictions are: a) In software prefetching the compiler will insert a prefetch code in the program. In this case as actual memory capacity is not known to the compiler and it will lead to some harmful prefetches. b) In hardware prefetching instead of inserting prefetch code it will make use of extra hardware which is utilized during the execution. The most significant source of lost performance when the process is waiting for the availability of the next instruction. The time that is wasted in case of branch misprediction is equal to the number of stages i the pipeline, starting from fetch stage to execution stage. All the prefetching methods are given stress only to the fetching of the instruction for the execution, not to the overall performance of the processor. In this paper we made an attempt to study the branch handling in a uniprocessing environment, where, whenever branching is identified instead of invoking the branch prediction the proper cache memory management is enabled inside the memory management unit. The simulation results which we obtained will give an icebreak to the improvement in the system performance with out any extra hardware. By invoking the memory management unit at the correct time we are able to achieve this result.

Keywords: Branch target buffer, prefetch, branch handling, processor performance.

1. INTRODUCTION

The performance of superscalar processors and high speed sequential machines are degraded by the instruction cache misses. Instruction prefetch algorithms attempt to reduce the performance degradation by bringing prefetch code into the instruction cache before they are needed by the CPU fetch unit. The technology developments in these fields indicate that in future the gap between processing speeds of processor and data transfer speed of memory is likely to be increased. A traditional solution for this problem is to introduce multi-level cache hierarchies[2]. To reduce memory access latency by fetching lines into cache before a demand reference is called prefetching. Each missing cache line should be prefetched so that it arrives in the cache just before its next reference. Each prefetching techniques should be able to predict the prefetch addresses. This will lead to following issues; Some addresses which cannot accurately predict the limits will reduce the effectiveness of speculative prefetching. With accurate predictions the prefetch issued early enough to cover the nominal latency. The full memory latency may not be hidden due to additional delays caused by limited available memory bandwidth. Prefetched lines, whether correctly predicted or not, may be issued too early will pollute the cache by replacing the desirable cache blocks. There are two main techniques to attack memory latency: (a) the first set of techniques attempt to reduce memory latency and (b) the second set attempt to hide it. These prefetching can be either done with software or with hardware technology. Save memory bandwidth due to useless prefetch, improve prefetch performance. Our study on the branch prediction will lead to the following conclusion. Domain-fetching is not an effective mechanism for filling the required instruction to the cache memory. Conventional hardware prefetches are not useful over the time intervals in which performance loss is the most dire. Bulk transferring the private cache is surprisingly ineffective, and in many cases is worse than doing nothing[3]. The only overhead in this case is the problem caused while identifying the branch instruction.
2. BASELINE ARCHITECTURE AGE

On a single processor system, if the processor executes only one instruction at a time, then multiprogramming in itself does not involve true parallelism [4]. Of course, if the single main processor in the system happens to be pipelined, then that system feature leads to pseudo-parallelism[]. Pseudo-parallelism share the same technical issues in common, related to the need for synchronisation between running programs. In the case of pseudo-parallelism the system will under go all types of hazards problems. In order to overcome the control hazard problem the system will invoke the branch prediction methods.

3. MOTIVATION

Parallelism in a uniprocessor environment can be achieved with pipelined concept. Today, pipelining is the key technique used to make fast CPUs. The time required between moving an instruction one step down the pipeline is a processor cycle. The length of processor cycle is determined by the time required for the slowest pipe stage. If the stages are perfectly balanced, then the time per instruction on the pipelined processor is equal to:

\[ \text{Time per instruction on pipelined machine} = \text{Time per instruction on unpipelined machine} \]

\[ \text{Number of pipe stages} \]

\[ \text{equ}(1) \]

\[ \text{Figure 1} \]

Pipelining yields a reduction in the average execution time per instruction. It is an implementation technique that exploits parallelism among the instruction in the sequential instruction stream. In the case of a RISC processor, it basically consisting of five pipeline stages. Figure 1 deals with the various stages [im- instruction fetch, reg- decoding, alu- execution, dm-store] associated with it.

Pipeline overhead arises from the combination of pipeline register delay and clock skew. The pipeline registers add setup time plus propagation delay to the clock cycle. The major hurdle of pipelining is structural, data and control hazards. We are mainly concentrated on the control hazards; arise from the pipelining of branches and other instructions that changes the PC. The structure of a standard program [5] shows that around 56% of instruction are conditional branching, 10% of instructions are unconditional and 8% of the instructions are comming under the call return pattern. Hence during the design pipeline structure we should take care of the non-sequential execution of the program.

Control hazards can cause a greater performance loss for MIPS pipeline than do data hazards. When a branch (conditional/unconditional) is executed, it may or may not change the PC to something other than its current value. The simplest scheme to handle branches is to freeze or flush the pipeline, holding or detecting any instructions after the branch until the branch destination is known.

It will destroy the entire parallelism achieved in the system. If the branching can be predicted earlier to some extent we can overcome the delay. That is, the required instruction can be brought to the cache region early. Prefetching can be done either using software or hardware. In software prefetching the compiler will insert a prefetch code in the program. In this case as actual memory capacity is not known to the compiler and it will lead to some harmful prefetches. In hardware prefetching instead of inserting prefetch code it will make use of extra hardware and which is utilized during the execution[6,7]. The most significant source of lost performance is when the out from the process waiting for the availability of the next instruction. In both the cases we have to flush some instruction out from the pipe. In this paper we made an attempt to combining the software prefetching with the branch handling inorder to overcome the delay problem generated inside the system this problem[8]. But we are not able to overcome the flushing of instruction out from the pipe.

4. ARCHITECTURAL SUPPORT

The miss stream is not sufficient to cover many branch related misses; we need to characterize the access stream. For that we construct a working set predictor which works in different stages. First, we observe the access stream of a process and capture the patterns and behaviour. Second, we summarize this behaviour and transfer the summary data to the stack region. Third we apply the summary via a prefetch engine to the target process, to rapidly fill the caches in advance of the migrated process[9] The following set of capture engines are required to perform the migration process Figure 2.

For the processing of each process it will enter the pipeline, once it enter to the decoding stage the processor is able to find the branching condition. By that time the next instruction will be in the fetch cycle.

\[ \text{Figure 2} \]

The block diagram of Architecture
4.1 Memory locker
For each process we add a memory locker, a specialized unit which records the selected details of the each committed memory instruction. This unit passively observes the current processes it executes, but does not directly affect execution. If a conservatively implemented memory locker becomes swamped with input records may safely be discarded; this will degrade the accuracy of later prefetching. Memory locker can be implemented with small memory tables. Each table with in the memory locker targets a specific type of access patterns.

4.2 Next-block
These detect sequential block accesses; entries are advanced by one cache block on a hit.

4.3 Target PC
This tracks individual instructions which walk through memory in fixed sized steps. The value for the target PC is loaded from the memory locker.

4.4 Target generator
Whenever the core is signalled to branching target generator activates. Our baseline core design assumes hardware support for branching; at halt-time, the core collect and stores the register state of the process being halted. While register state is being transferred, the target generator reads through the tables populated by the memory locker and prepare a compact summary of the process which is moving towards the running state. This summery is transmitted after the architected process state, and is used to prefetch the ready working set when it resume on the running state. During summarization each table entry is inspected to determine its usefulness by observing whether the process comes to ready state is whether from start or waiting state. Table entries are summarized for transfer by generating a sequence of block addresses from each, following the behaviour pattern captured by that table We encode each sequence with a simple linear-range encoding, which tells the cache management unit, starting at start-address.

4.5 Target-driven prefetcher
When a previously preempted branch is activated on a core, its summary records are read by the prefetcher. Each record is expanded to a sequence of cache block addresses, which are submitted for prefetching as bandwidth allows. While the main execution pipeline reloads register values and resumes execution, the prefetcher independently begins to prefetch a likely working set. Prefetches search the entire memory hierarchy, and contend for the same resources as demand requests. Once the register state has been loaded, the process resumes execution and competes with the prefetchers for memory resources. Communication overlap between the process are modelled among transfers of register state, table summaries, prefetches, and the service of demand-misses. We model the prefetch engine as submitting virtually addressed memory requests at the existing ports of the caches, utilizing the existing memory hierarchy for service.

5. ANALYSIS AND RESULT
The baselines for comparison in our simulation result are the prediction rate of each benchmark when executed as a single process and run to completion with no intervening process. The program is executed for different size of instruction for experiment. The simulation is done in a uniprocessor environment. The branch handling is done using the various cache management schemes such as fifo, lifo, mru etc. all the experiment results lead to the conclusion that if the number of miss is less the overall system performance is going to get increased. The various branch prediction methods are definitely reducing the miss rate inside the system. But no method there with 100% hit rate. Figure 3,4,5 shows the simulation results. And the Figure6 shows the comparison of these various methods. The x-axis will show the size of codes and the y-axis shows the time period. In this method the system will under go miss condition only during the starting of the execution. But this will occur in all the execution methods. The variation in the performance is due to the basic working principles of the various schemes. And this factor will depend on the order of the pages required by program for execution. In this method the only over head which we will find is the only one cache line. If we are able to implement the memory locker with the help of an CAM memory component the speed of comparison can be increase a lot.

6. CONCLUSIONS
The report here sheds light on the applicability of instruction cache prefetching schemes in current and next-generation microprocessor designs where memory...
latencies are likely to be longer. When ever , a new prefetching algorithm is examined that was inspired by previous studies of the effect of speculation down miss predicted not correct paths. Data prefetching has been used in the past as one of the mechanism to hide memory access latencies. But prefetching requires accurate timing to be effective in practice. This timing problem will affect more in the case of multiple processor. Which is handled with the help of target-driven prefetcher. The designing of an effective prefetching algorithm which will minimize the prefetching overhead, and it is a big challenge and needs more thoughts and effort on it

REFERENCES


AUTHOR

Jisha P Abraham received the B.Tech. in Computer Science and Engineering from M.G University,Kottayam in 1997 and M.E. degrees in Computer Science and Engineering from Anna University, Tamilnadu in 2006. From 1998 onwards working at M. A. College of Engineering, Kothamangalam, Kerala, India. Presently working as Associate Professor in Computer Science and Engineering.