

AN ATTEMPT TO IMPROVE THE PROCESSOR PERFORMANCE BY PROPER MEMORY MANAGEMENT FOR BRANCH HANDLING

Ms.Jisha P Abraham and Dr.Sheena Mathew

M A College of Engineering, Cochin University of Science and Technology

ABSTRACT

The performance of the processor is highly dependent on the regular 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 instructions supply to the processor. The technology developments in these fields indicates 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.

Prefetching can be done either with 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. The most significant source of lost performance when the process 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 in the pipeline, starting from fetch stage to execute 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. The most significant source of lost performance is, 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 in the pipeline, starting from fetch stage to execution stage. This paper we made an attempt to study the branch handling in a uniprocessor environment, whenever branching is identified instead of invoking the branch prediction the proper cache memory management is enabled inside the memory management unit.

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 indicates 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. 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 limit 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. 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 narrow 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. The only overhead in this case is the overhead caused while identifying the branch instruction.

2. BACK GROUND AND RELATED WORKS

Sair, et,al[1] survey several prefetchers, and introduce a method for classifying memory access behaviours in hardware: the branch access stream is matched against the behaviour specific table in parallel. We utilize such a similar technique in the listing the branches in the programming environment. Mahmut, Yuanrui [2]study shows that timing and scheduling of prefetch instructions is a critical issue in software data prefetching and prefetch instructions must be issued in a timely manner for them to be useful. If a prefetch is issued too early, there is a chance that the prefetched data will be replaced from the cache before its use or it may also lead to replacement of other useful data from the higher levels of the memory hierarchy. If the prefetch is issued too late, the requested data may not arrive before the actual memory reference is made, thereby introducing processor stall cycles. We make use of this concept will the pre-empting the current running sequence from the processor. I.K. Chen[3] Next-line prefetching tries to prefetch sequential cache lines before they are needed by the CPU's fetch unit. The next line is not resident in the cache; it will be prefetched when an instruction located some distance into the current line is accessed. This specified distance is measured from the end of the cache line and is called the fetch ahead distance.

Next-line prefetching predicts that execution will fall-through any conditional branches in the current line and continue along the sequential path. Next line guess will be incorrect except in the case of short branches and the correct execution path will not be prefetched. Performance of the scheme is dependent upon the choice of fetch ahead distance. For the calculation of the wastage of the processor cycles can be calculated using this fetch ahead distance.

Run ahead Execution [4]prefetches by speculatively executing the application, but ignoring dependences on long latency misses. This seems well-suited to our need to prefetch the branch instruction where the branching is identified.

3. BASELINE ARCHITECTURE

On a single processor system, if the processor executes only one instruction at a time, then multiprogramming in itself does not involve true parallelism. 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 some technical issues in common, related to the need for synchronisation between running programs. Figure 1 is somewhat detailed view of the running of the four programs-labelled P1, P2, P3, and P4-in multiprogramming mode[5]. The top part of the figure, all the four programs seems to be running in parallel. With time, each program makes progress in terms of both instruction and input/output performed. Due to single processor, in terms of actual execution of program execution, there is no parallelism in this system. Running program is such a crucial entity in a computer system that it is given a name of its own and referred to as a process. The behaviour of a running program, in terms it spend in compute phase and I/O phase.

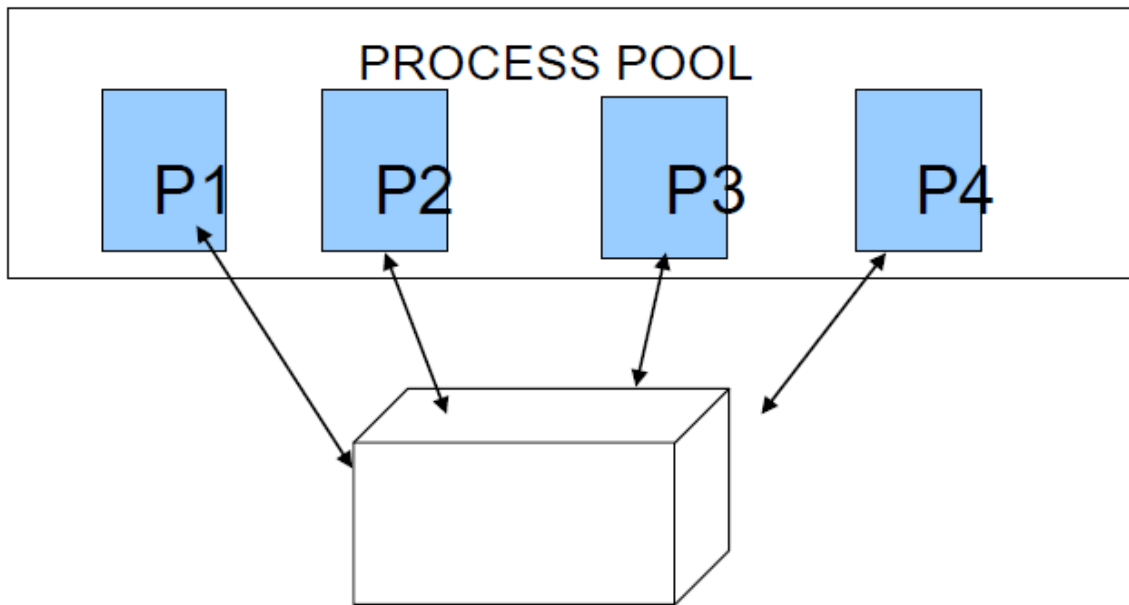


Figure 1: Multiprogramming Modeul

4. MOTIVATION

Parallelism in a uniprocessor environment can be achieved with pipelined concept. Today, pipelining is the key technique used to make fast CPUs[6]. 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:

Time per instruction on unpipelined machine

Number of pipe stages

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.

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 [7] shows that around 56% of instruction are conditional branching, 10% of instructions are unconditional and 8% of the instruction are comes under the call return pattern. Hence during the design of 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 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 early 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 with 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. The most significant source of lost performance when the process waiting for the availability of the next instruction. In both the cases we have to flush some instruction out from the pipe. Combining the software prefetching with the branch handling, we can overcome this problem. Whenever branching encountered the system will go for a context switching, pipe will be filled with instruction for the next process.

5. SYSTEM DESIGN

5.1 USER MODELING PROCESS

User Model life cycle is defined by a sequence of actions performed by an adaptive web-based system during user's interaction. The process is depicted in the Figure 2.

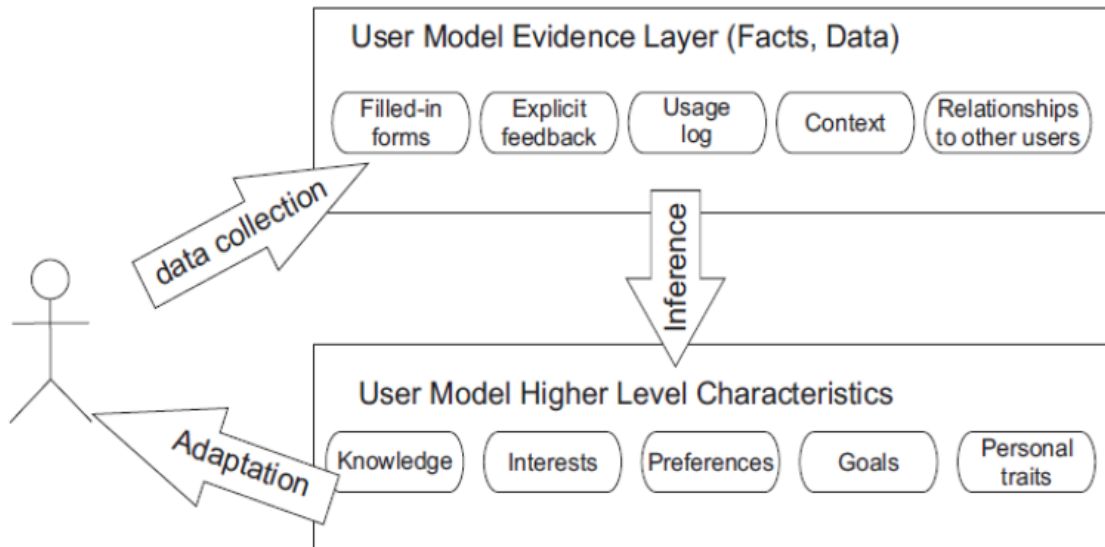


Figure 2: User Model Life Cycle

We separate the user modelling process into three distinct stages, as defined below.

1. Data collection, when an adaptive system collects various user-related data such as forms and questionnaires filled-in by the user, logs of user activity within the system, feedback given by the user to a particular domain item, information about user's context (device, browser, bandwidth, etc.). All these information represent observable facts and are stored in an evidence layer of the user model. The facts can be supplied also from external sources, for instance we can acquire information about user relationships with other users from an external application for social networking.

2. User model inference, when an adaptive system processes the data from the evidence layer into higher level user characteristics such as interests, preferences or personal traits. It is important to mention that these are only estimates of the real user characteristics.

3. Adaptation and personalization is the actual use of a user model (both evidence and higher level characteristics layer) to provide a user with personalized experience when using an adaptive system.

The success rate of the supplied adaptation effect gives us a feedback about an accurateness of the underlying user model.

The process has a cyclic character, as an adaptive system continuously acquires new data about the user (influenced by already performed adaptation and thus by a previous version of the user model) and continuously refines the user model to better reflect reality and thus to serve as a better base for personalization. An adaptive system must handle specific problems related to each step of the process. In data collection we must find a balance between user privacy and amount and nature of data which is required in order to deliver a successful personalization. We must find such sources of data, which do not pose additional extra burden on the user and design data collection process in such a way, which is unobtrusive and runs automatically in background. The main challenge of the actual user model inference is (apart from actual transformation of user

action history into user characteristics) maintenance of user characteristics, which should keep pace with user personal development, changing interests and knowledge.

6. ARCHITECTURAL SUPPORT

The miss stream is not sufficient to cover many branch related misses; we need to characterize the access stream[8]. 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. The following set of capture engines are required to perform the migration process. For the processing of each process it will enter to the pipeline, once it enters 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

6.1 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.

6.2 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.

6.3 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 summary 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*

6.4. 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.

7. ANALYSIS AND RESULT

The baselines for comparison in our simulation result figure 3 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 our experiment has the effects on the size of the cache line provided along with the number of cache blocks considered in the cache memory. The cache management is done with the first in first out method.

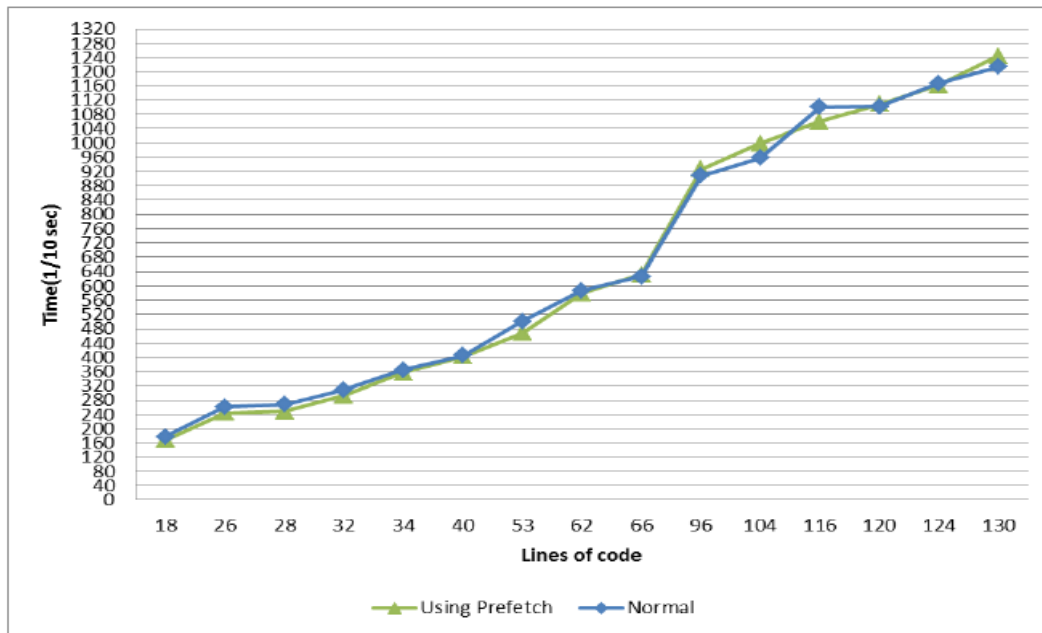


Figure 3 Lines of code Vs. Time Graph

8. 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. Whenever 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 mechanisms 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 processors. 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. But all the prefetching methods will only minimize the miss condition inside the system. No system is able to remove the miss condition 100% from the memory system. The proposed system will face the miss condition only at the starting of the processing of a particular process. We also tried to reduce the hardware requirement in the system which in turn will reduce the cost of the system.

REFERENCES

- [1] J. D. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In 35th International Symposium on Microarchitecture, pages 62-73, Nov. 2002.
- [2] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In 26th International Symposium on Computer Architecture, pages 186-195, M
- [3] I.K. Chen, C.C. Lee, and T.N. Mudge. Instruction prefetching using branch prediction information. In International Conference on Computer Design, pages 593-601, October 1997.
- [4] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In 9th International Symposium on High-Performance Computer Architecture, pages 129-140, Feb. 2003.
- [5] A. Brown and D. M. Tullsen. The shared-thread multiprocessor. In 21st International Conference on Supercomputing, pages 73-82, June 2008.
- [6] A.J. Smith, "Cache memories," ACM Computing Surveys, pp. 473-530, Sep. 1982
- [7] ECE 4100/6100 Advanced Computer Architecture, Lecture 5 Branch Prediction Prof. Hsien-Hsin Sean Lee School of Electrical and Computer Engineering Georgia Institute of Technology.
- [8] Lawrence Spracklen, Yuan chou and snathosh G Abraham "effective instruction prefetching in chip multiprocessor for modern commercial application" proceedings of 11th Intl. Symposium on HPCA, 2005