ISSN (Online) : 2319 - 8753
ISSN (Print) : 2347 - 6710

International Journal of Innovative Research in Science, Engineering and Technology

Volume 3, Special Issue 3, March 2014

2014 International Conference on Innovations in Engineering and Technology (ICIET'14)
On 21st & 22nd March Organized by

K.L.N. College of Engineering, Madurai, Tamil Nadu, India

Design of Out-Of-Order Superscalar Processor with Speculative Thread Level Parallelism

A.Kamaraj[#1], X.Sharly Monica[#2]

[#1]Department of ECE, Mepco Schlenk Engineering College, Sivakasi,

Virudhu Nagar District, Tamilnadu, India.

[#2]Department of ECE, Mepco Schlenk Engineering College, Sivakasi,

Virudhu Nagar District, Tamilnadu, India.

Abstract**—** The Complexity of handling the complex flow logic has become the major impact in parallel programming. The two main problems associated with the Scheduling of Superscalar Processor are interrupt precision and implementing multiple levels of Branch Prediction. The proposed work implements the Speculative Thread Level parallelism Technique on superscalar Processor, as an alternative source of parallelism which can boost the performance for applications, by overcoming the causes using cache coherence protocols and thus prevent the collision due to dependencies. To address this critical need, a Register transfer level (RTL) model of a superscalar micro architecture have been developed with similar complexity of a current generation processor. The RTL model is written in Verilog and is fully synthesizable. The RTL model is tightly integrated with a C functional simulator to assist and accelerate verification. The dissertation also proposes novel architecture and compiler techniques to efficiently extract speculative parallelism from multiple loop levels.

Keywords**—** Branch Prediction, Thread Level Parallelism, Speculative Execution, Cache coherence Protocol

I. INTRODUCTION

Superscalar Processors exploit Instruction level parallelism (ILP) depending on the instruction processing and the dependencies between those instructions and in its issue process to different processing units [7]. The scheduling of those instructions can be done either statically or dynamically. Static approach that rely much more on software. Dynamic approach that depend on hardware to locate parallelism. In dynamic scheduling the dynamic stream of instructions is analyzed and the dependencies are found from the instructions [7]. The out-of-order execution of multiple instructions in issue and execution unit is the most general form [1].

The implementation has been done in such a manner to perform operation at faster clock cycle and to maximize the number of instructions issued per cycle. The proposed work implements the coarse-grain unit of parallelism and the resultant Instruction per cycle (IPC) is eventually much lower than the processor performance level. Thus the hurdles can be handled efficiently based on the dependencies between read and write process [1]. Data value speculation has been proposed to relieve the penalties due to data dependencies and minimize their impact on the performance of the processor by means of predicting the input/output operands of instructions [8].In recent studies [7] [6], it has been shown that the performance impact of this technique for superscalar processors is reasonable and its potential improvement approaches a linear function of the prediction accuracy. On the other hand, its potential is much higher for speculative multithreaded architectures. Other critical components are the issue and rename logic. In order to increase the performance, some alternative micro architecture has been proposed by means of exploiting coarse-grain parallelism in addition to the instruction-level (fine-grain) parallelism [5]. These micro architectures split programs into threads and then, they speculatively execute them concurrently. This kind of parallelism is referred as Speculative Thread-Level Parallelism [3]. Threads are speculatively executed because they are both control and data dependent on previous threads, since independent threads are hard to find in many non-numeric applications.

These micro architectures include support to roll-back the execution in case of either a control or a data dependence misspeculation occurrence.

A. Overview

All micro architectures provide support for multiple contexts and using appropriate mechanisms to predict values produced by one thread which is consumed by another thread. The difference lies in splitting of program into threads. The compiler is responsible for splitting the program into threads. Speculative thread level parallelism has significant potential to boost performance have been shown. However, most of them use different heuristics to partition a sequential instruction stream into speculative threads.

B. Speculative Thread Level Parallelism

Thread-Level Speculation (TLS) is an aggressive parallelization technique that is applied to regions of code which although contain a good amount of parallelism, cannot be statically proven to conserve the sequential semantics when it is executed in parallel [3]. With TLS, threads concurrently execute iteration of a loop out of sequential order even in the presence of true dependences. They use software/hardware structures that are speculative storage, to trace the dependencies information by storing it and to regress to a safe point and restart the computation upon the occurrence of a dependency violation (rollback recovery) [6].

In order to guarantee correct sequence of execution, threads update their changes into the global non speculative storage only when it is determined that the locations it read-from and wrote-to do not generate a dependency-violation. The usual implementation is to have the threads buffer their writes and commit them sequentially when they become master. Hardware approaches employ a modified cache coherency protocol to detect the occurrence of inter-thread data dependencies and initiate a rollback. In servicing a rollback the speculative state needs to be cleared and the threads affected by the violation are restarted to carry out the cancelled iterations.

II. Design Of Speculative Out-Of-Order Superscalar Processor

In out-of-order execution, the instruction stream is not executed in the same order as that of original program sequence and it gets executed based on the availability of source code operands. In this project as a part the design and implementation of canonical pipeline stages of a superscalar micro architecture was developed, by inheriting from commercial superscalar based designs [8]. Although, the RTL model of the individual pipeline stage is parameterized by the width of the stage and the sizes of specialized memory structures within the stage, a specific micro architecture configuration has been chosen as a starting point to understand the design complexity involved. The Table-1 depicts the architectural configuration as an overview of the complexity model of the processor. Instruction Fetch is will get the instruction stream as input sequentially. Every cycle, the program counter (PC) is incremented sequentially, till the end of the control instruction in the instruction stream [7].

TABLE I

Configuration for Superscalar processor Architecture

STAGE	DESCRIPTION
Fetch	4-Wide 128-Entry Bimodal Branch predictor 16-Instruction Fetch Buffer
Decode	4-Wide ISA(Similar to MIPS)
Rename	4-Wide 32-Entry Rename map table
Dispatch	4-Wide
Issue	4-Wide issue 32-Entry Issue Queue
Register Read	4-Wide 128-Physical Register File
Execute	1-Simple ALU, 1-Branch ALU, 1-Complex ALU
Load-Store Unit	16-Entry Load Queue 16-Entry Store Queue
Write Back	4-Wide
Retire	4-Wide 128-Entry Active List

A. Stages of Superscalar processor

The major components of the architecture are given and there are major units, that is Fetch Unit, Rename Unit, Issue Unit, and Back end Unit. In Fig-1, the architecture of the proposed superscalar processor is described.

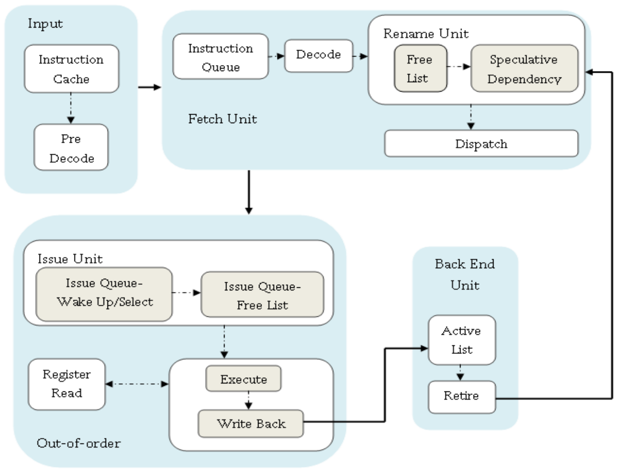

Fig-1: High Level Block Diagram of Superscalar Processor

B. Instruction Fetch

In a program, conditional branches tend to occur more frequently than other control instructions [5]. The branch prediction mechanism has three major structures with random logic, branch target address (BTA), branch predictor (BP), and return address stack (RAS) [8]. On considering the cycle time, the fetch stage has two important timing paths, accessing the interleaved instruction cache for reading two aligned cache blocks. The complexity of accessing the cache would increase with increasing the size and the set associativity of the cache [3].

Generating next PC using information from the BTA, BP, and RAS for a group of instructions that have been fetched. The complexity of the next PC logic would increase with a larger BTA, a more complicated or larger branch predictor, or wider fetch bandwidth. Moreover, it is important to generate the next PC in one cycle to avoid losing cycles on every predicted-taken branch.

If the BTA misses for the control instruction in a cycle, the next cycle of process generates a selection recovery signal and selection recovery target address for the previous stage [8]. If an instruction happens to be a predicted-taken branch in the fetch block, subsequent instructions are discarded. Each stage has the FIFO Buffer to update the control instructions in the proper sequential order and its status which is called Control Transfer queue (CT queue).After a control instruction at the head of the FIFO retires, the CT queue updates the BP with the computed direction [7].

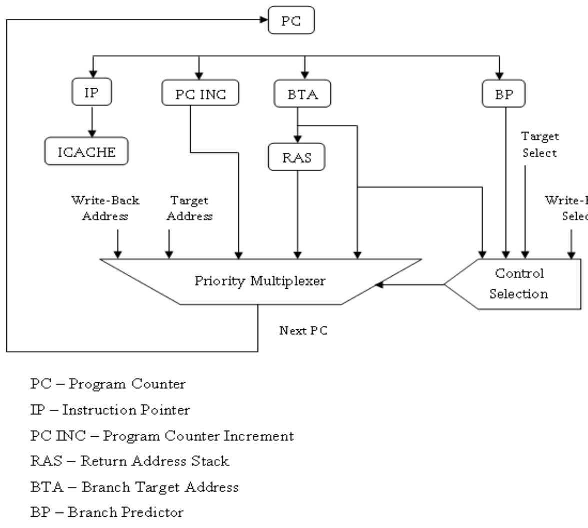

Fig-2: Instruction Fetch stage

This leads to in-order update of the branch prediction structure. Instruction alignments, extracting the fetch block, pre-decoding, and generating the recovery signal are serialized logic and this working is explained in Fig-2. The instruction queue serves two purposes:

- It allows instruction fetching, even though the rest of the front-end is stalled because of a hardware resource limitation
- It simplifies decode, rename and dispatch logic by always providing a fixed number of instructions.

C. Instruction Decode

Decode unit performs the mapping of parallel instructions and send it to the instruction queue based on allocation for the particular instruction [7].

Decoding stops if the active list or a queue becomes full, but there are very few decode restrictions that depend on the type of instruction being decoded. The principal expectation involves integer multiply and divide instructions. Their results go into two special registers. No other instructions have more than one result register.

D. Register Renaming

Register renaming [2] removes the false dependencies among instructions which are limited architectural registers. Fig-3 depicts it’s function. Fundamentally, the dependencies between instructions are analyzed and process is done, a dynamic instruction stream has three types of data dependencies:

- True dependency, where the source register of a younger instruction depends on the outcome of another, which is the older instruction in the dynamic instruction stream.
- Output dependency, where the destination registers of a younger instruction is the same as the destination register of another, which is the older instruction in the dynamic instruction stream.
- Anti-dependency, where the destination register of a younger instruction is the same as the source register of another, which is the older instruction in the dynamic instruction stream.

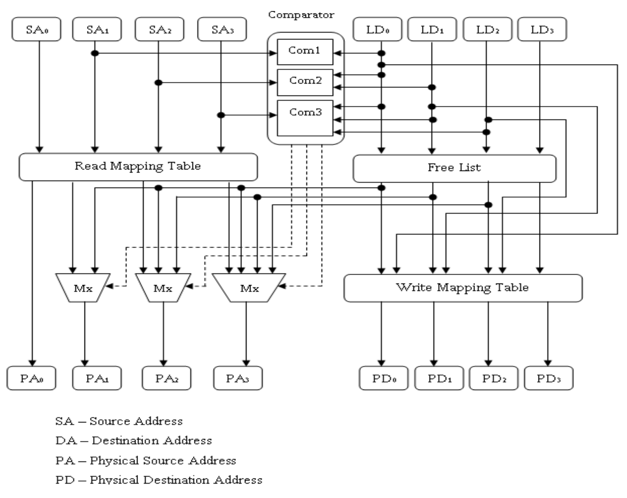

Fig-3: Register renaming Logic

E. Issue

The Issue stage buffers the renamed instructions and selects instructions for execution based on the availability of their source operands [4]. The maximum buffer size is referred to as the issue window, and the maximum number of instructions selected for parallel execution in a cycle is referred to as the issue width. The issue window and issue width are the fundamental characteristics of the issue stage, and determine its logic complexity [2]. In summary, an Issue stage consists of two major operations: wakeup and select [3]. The wakeup operation is dependence resolution performed in the issue window, and the select operation is arbitrating among ready-to-execute instructions in the issue window. In our design, the issue window is centralized, and the Issue stage is pipelined between wakeup and select logic.

Each functional unit executes a different type of integer instruction, and instructions are associated with their functional unit type during the decode stage [5]. The complexity of the wakeup operation grows with issue window size and the number of wakeup ports [6].

F. Execute

The source register specifier of an issued instruction, index into the PRF [5] to read the corresponding values, At the same time, source register specifier are also compared with the Write back destination register specifier to detect the scenario whereby a producer instruction's result needs to be directly updated to a consumer instruction .The functional unit in the execute stage performs an arithmetic or logic operation on the source operands of an instruction [4].

G. Write Back

The Write back stage contains the latches holding the results from the execute stage, which serve as the source for feeding the logic path. The Logic path forwards the result values from the executed instructions to the dependent instructions, to support optimal execution of the producer and its dependent instructions in consecutive cycles [6]. The Write back stage also acts as the source for branch misprediction signals [5].

H. Retire

Although instructions execute out-of-order, they update the architectural processor state in the correct program order to maintain the sequential execution model [7].The Retire stage also maintains an Architectural Map Table, containing mappings between architectural registers and physical registers for committed versions of architectural registers. When an instruction commits, the Active List updates the map table with the instruction's physical destination register mapping and releases the previously mapped physical register [6].

III. Experimental Result And Discussion

A complex micro architecture might enhance IPC, but at the same time could increase the propagation delay. For instance, increasing the size of the issue window can boost IPC for applications with abundant ILP, but at the same time, clock rate may decrease to accommodate the larger content addressable memory. In general, any attempt to increase micro architectural complexity to get better IPC has a direct impact on the propagation delay.

The applications are experimentally analyzed in the superscalar processor using Modelsim as the simulation tool and Xilinx PlanAhead 14.4 is used for its resource estimation and virtex-5 is used for implementation. The applications used in this study are compress performs data compression and decompression and it carries certain loop dependencies occur frequently. The jpeg and mpeg performs various algorithms on images.

In nqueen complexity level for the computation is very large and it has complex dependencies that either hoist them outside of the loop or else explicitly forward them using wait-signal synchronization. This chapter contains the results for the execution of processor results with out-of-order execution.

A. Performance Measure

The Performance measure of the speculative Execution of the out of order superscalar processor, which uses the better branch prediction mechanism in order to overcome the complexities in the dependencies due to branch and jump instruction within and outside the loop, is analyzed hereby using the comparison graphs. The comparison Table-2 describes the Branch prediction accuracy by taking the average between the correctly predicted branch and the mispredicted branch from the series of execution. In the Prediction accuracy graph Fig-4, the rate of branch prediction accuracy increased by 15% than the static and voting method stated in earlier works.

TABLE II
Comparison Table for Branch Prediction Accuracy

TASKS	**STATIC PREDICTION**	**VOTING METHOD**	**CACHE COHERENCE METHOD**
compress	0.81	0.87	0.4
jpeg	0.89	0.91	0.5
mpeg	0.75	0.80	0.37
nqueen	0.91	1	0.46

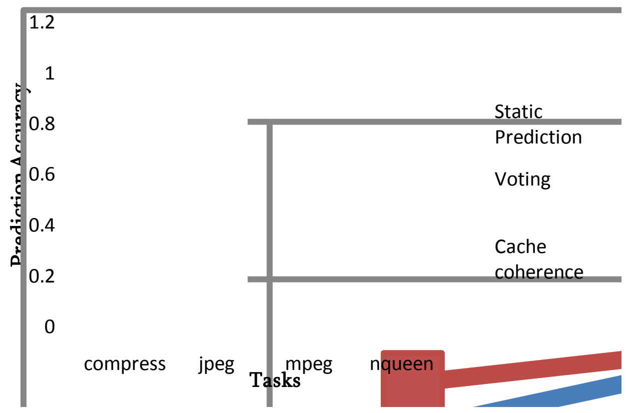

Fig-4: Comparison graph - Branch Prediction Accuracy

In this the difference between the prediction rate decreases in the cache coherence method and it is clearly explained in the graph, that is if the difference rate decrease, the number of misprediction among the predicted branch and jump calls is less, so the prediction accuracy will increased automatically and leads to less miss match. In the following comparison graph Fig-5 the performance can be calculated in terms of the processing cycle. The comparison of performance done with the baseline stages of this complex effective superscalar processor, that is the average IPC of the pipeline and simple superscalar processor is taken and its performance is calculated and it is put side by side with this out-of-order execution processor.

The drop in IPC does show the same trend as it shows in the C simulator. The average drop in IPC among these 4 tasks was seen to be around 3%. Thus, the effects of pipelining on accuracy by out-order speculative execution results in minimization of IPC and thus decrease the delay to complete the processor execution. Thus the performance efficiency of processor execution increased. Table-3 summarizes the performance of each application on baseline architecture that implements our coherence scheme.

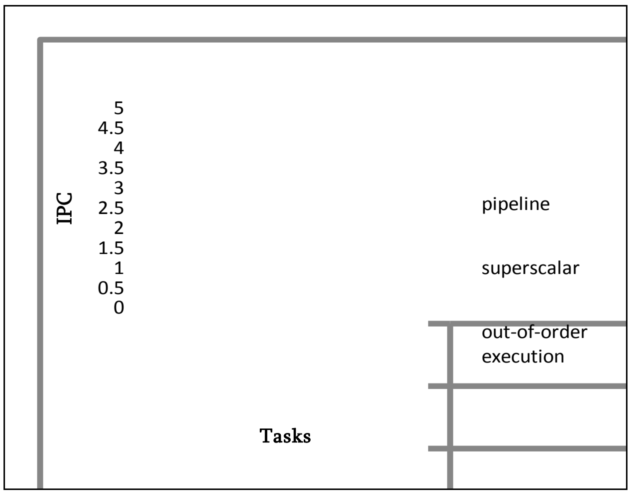

Fig-5: Comparison graph - performance of the processor

TABLE III

Performance impact of Speculative Out-of-Order Processor

APPLICATION	OVERALL SPEEDUP	PARALLEL EXECUTION COVERAGE
compress	2.98	66.72%
jpeg	1.99	52.37%
mpeg	2.66	46.63%
nqueen	2.84	45.55%

All the speedups and the other statistics related to the processor are with respect to the original executables without any overheads running on the processor. Hence our speedups are absolute speedups and not relative speedups. The range of increase in the overall speed up and the parallel execution coverage percentage of increase is given and stated below. The overall speed up limited by the execution coverage that is the fraction of original execution time was parallelized. It is stated from the table 5.2 that the speedup of the process increased up to the rate of 2 to 3 and the parallel execution coverage rate shows that the processor execution speedup will be considerably not affected by the complex loop and branch dependencies in the program.

Thus it is concluded that the cache coherence scheme which is used to handle dependencies among the tasks perform the better work in combination with the out-of-order execution of the program sequence, Individually and as a whole the methods used in this speculative out-of-order superscalar processor increases the overall speed up, parallel coverage and decreases the delay and thus the performance efficiency is potentially effective.

IV. CONCLUSION

Speculative out-of-order execution of the superscalar processor, the proposed work uses the cache coherence mechanism to predict and to manage the branch occurrence while processing the instructions and the results of this execution increased the efficiency than the previous methods used for the superscalar execution by considerable performance. The results are furnished for different task programs and their resource utilization, power efficiency and the overhead is also limited using this out-of-order execution.

In our proposed method, the performance of the processor increased by 15% and the processing speed also increases. The main impact of handling the complex instruction which was done by the branch prediction mechanism achieved 75% of accuracy in predicting the branch thus the speed up of the processor increases. The Future work is to extend the micro architecture of the superscalar processor in order to perform task level parallelism implemented with the soft processors and to increase the scalability to perform the parallel processing with the deterministic number of processor and to attain the reach out performance.

REFERENCES

[1]. Y. Etsion et al., “Task Superscalar: An Out-of-Order Task Pipeline,” Proc. IEEE/ACM 43rd Ann. Int’l Symp. Microarchitecture (MICRO), pp. 89-100, 2012.

[2]. J.C. Jenista et al., “OoOJava: An Out-of-Order Approach to Parallel Programming,” Proc. Second USENIX Conf, Hot Topics in Parallelism (Hot Par ’10), 2010.

[3]. C.H Chen and K.S.Hsiao, “Scalable Dynamic Instruction Scheduler through Wakw-Up Spatial Locality,” IEEE Trans. Computer, vol. 56, no. 11, pp. 1534- 1548, Nov-2007.

[4]. M. Labrecque and G. Steffan, “Improving Pipelined Soft Processors with Multithreading,” Proc. Int’l Conf. Field Programmable Logic and Applications (FPL ’07), 2007.

[5] T.N. Buti et al., “Organization and Implementation of the Register-Renaming Mapper for Out-of-Order IBM POWER4 Processors,” IBM J. Research and Development, vol. 49, no. 1, pp. 167-188, 2005.

[6]. M.A Ramirez et al., “Direct Instruction Wakeup for Out-of-Order Processors, “Proc. Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA ’04), pp. 2-9, 2004.

[7]. J. Shen and M. Lipasti, Modern Processor Design: Fundamentals of Superscalar Processors. McGraw-Hill, 2004.