Analysing a Multistreamed Superscalar Speculative Instruction Fetch Mechanism

Rafael R. dos Santos[*] and Philippe O. A. Navaux[**]

Informatics Institute
CPGCC/Federal University of Rio Grande do Sul,
P.O. Box 15064 91501-970 Porto Alegre - RS, Brazil

Abstract. This work presents a new model for multistream speculative instruction fetch in superscalar architectures. The performance evaluation of a superscalar architecture with this feature is presented in order to validate the model and to compare its performance with a real superscalar architecture. This model intends to eliminate the instruction fetch latency introduced by branch instructions in superscalar pipelines. Finally, some considerations about the model are presented as well as suggestions and remarks to future works.

1 Introduction

Even using accurated branch prediction mechanisms, current superscalar architectures offer less performance than an ideal architecture.

When a branch is encountered, the branch predictor can predict it as to be taken or not taken. In the first case, contiguous instructions are already into the fetch stage when the prediction is made, and these instructions do not take part of the predicted path. For the second case, nothing occurs if the prediction is correct. But for both cases, we are assuming that the branch prediction mechanism is efficient.

The problem usually is that branches are predicted as to be taken and each time this occur a flow interruption also occurs. The time requiered to refill the fetch buffer and put the correct instructions into the instruction queue is greater than the necessary time to execute these instructions. If there are many functional units and flow interruptions occur frequently, it should be expected that the instruction queue will be empty for many cycles.

This means that the instruction fetch mechanism must be designed to keep the instruction queue with instructions that can be scheduled for execution on free functional units.

A superscalar architecture could not execute instructions faster than it can fetch instructions from the instruction cache. So, design an efficient fetch mechanism is more important than increase the number of resources, because it is not

[*] Ph.D. Student – E-mail: rrsantos@inf.ufrgs.br - UNISC - University of Santa Cruz do Sul - Santa Cruz - RS - Brazil

[**] Ph.D. – E-mail: navaux@inf.ufrgs.br - CPGCC/Federal University of Rio Grande do Sul

possible to increase the performance only by increasing the number of functional units.

The question is how many instructions could be fetched per cycle? Where are these instructions comming from? An alternative approach to branch prediction is instead of predict whether the branch is taken or not taken, is execute speculatively both the taken and not-taken paths and cancel the execution of the incorrect path as soon as the branch result is known [3].

2 Fetching instructions from multiple streams

Talcott [9] reffers a technique called *Fetch Taken and not-Taken Paths* to reduce the branch problem. In [1] was presented a new model based on this scheme. In this model, both paths of a conditional branch are fetched and put into the fetch buffer.

When the branch prediction stage transfers instructions, from the fetch buffer to the instruction queue, and reaches a branch into the fetch buffer, both possible paths of this branch and its instructions are already into the fetch buffer. In this case, no delay is introduced when the branch is predicted as taken. The transfer is redirected to the predicted path whithout delay.

To enable this operation, the fetch stage must detect the branch instruction just when it is fetched. In the next cycle, it must also start to fetch from both paths. When the branch is predicted, the instructions transfer to instruction queue is not interrupted.

In the multistreamed superscalar architecture proposed in [6], the fetch stage was modified to enable to fetch both paths of a branch instruction.

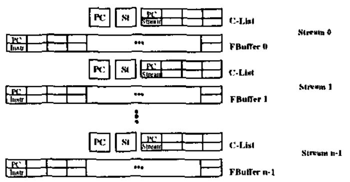

Fig. 1. Fetch Buffer Structure (e.g. fetch depth equal to 4 streams)

The figure 1 suggests a new buffer structure in the multistream pipeline. The number of stream buffers defines the fetch depth. Each stream has four independent elements: Program Counter, Status Bit, Children List and Fetch Buffer.

In the *Fetch Buffer* are stored the fetched instructions. The *PC* is used to point the instructions that are in the stream. The *status bit* indicates whether the stream structure is busy and the *Children-List* stores the identification of the children streams.

The *Children-List* also stores the branch address and the identifier of its children streams allowing the predict stage to start the transfer of instructions of the new stream when a branch is predicted, without any delay.

2.1 Multistream Architecture Operation

The *Fetch* stage fetches instructions to put them into the fetch buffer. When conditional branch instruction is detected, a new stream is generated and initialized, as if there were available resources.

The stream generation consist of *Children-List* updating operation with the branch address and the identification of a new stream structure that will store the instructions related to the new stream. For each branch detected there are two possible paths. The instructions that are in the not taken path are fetched and stored into the same stream structure where is stored the conditional branch. But, the taken path and their instructions will be stored in this new stream structure.

The stream structure initialization consists of the *PC* initialization with the target address and the setting of the *status bit*, to indicate that the structure was allocated.

The predict stage transfers instructions, from the fetch buffer to the instruction queue (like conventional superscalar architectures) looking for branch instructions. However, when it finds a branch instruction, it also makes a prediction. When the prediction is to be taken, this stage just concatenates the instructions which are in the children stream of this branch, discarding the closest instructions.

When the prediction is not taken, the children stream is discarded and the neightbouring instructions continue to be transfered to the instruction queue. When a stream is discarded, all children streams originated by this stream are also discarded, through a recursive operation.

3 Experimental Framework

For this work, we used 4 benchmarks from the SPECint95 suite (compress, go, ijpeg, li). Also, we used a execution-driven simulator to generate traces. This simulators acomplish with the SPARCV7 instruction set and simulates the execution in a scalar pipelined fashion. This is the reference machine.

For the superscalar simulations, we used 2 trace-driven simulators which executed the choosed benchmarks based on the traces generated by the first simulator. These 2 simulators are respectively called *Real* and *Mulflux* as we will describe below:

- Real Simulator: Simulates the execution of programs using two-level branch prediction [4, 5].
- Mulflux Simulator: Simulates the execution of programs using the proposed speculative instruction fetch mechanism [6].

4 Performance Analysis

In this section, we analize the performance of the multistreamed model based on the data extracted through several simulations. The experiments simulate different configurations of the real machine and the multistreamed machine. The tests started with a fetchwidth equal to 2 up to 8 instructions per cycle.

We can point out that in all situations the cycles with no dispatch are less than for the real machine. The no dispatch decreases with the increase of fetchwidth in the real machine. The percentage decreases from 40.50%, for fetchwidth equal to 2, to 39.30% in configurations with a fetchwidth equal to 8 instructions, as showed in the figure 2.

In the Mulflux machine, this percentage increases join to the increase of fetchwidth. This occur because the mispredictions and resources conflict increase too. Other studies have been developed to research more accurated branch predictors and the ideal balancing of the architectures resources to support a new and more powerfull parallelism through the pipeline. The sum of the three components, which causes no dispatch in both machines, correspond to in the percentage of cycles with no dispatch.

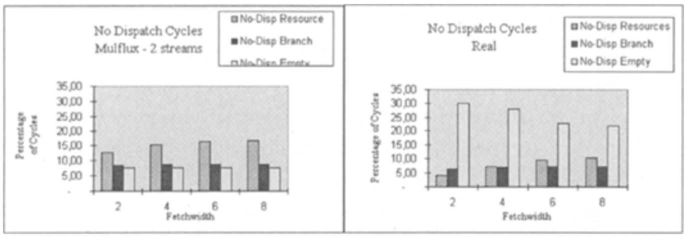

Fig. 2. No Dispatch in Mulflux with 2 Streams and Real Architecture

The divergency between the components is important in the Real machine. The occurency of an empty queue is the critical component that contributed to the existency of no dispatch cycles. This is not the case for the Mulflux machine. It is predictable that when there are more instructions ready to dispatch, the number of functional units becomes the main problem. This is true in the Mulflux architecture.

The occurency of empty queue in the Real machine decrease from 30.05%, with fetchwidth equal to 2 to 21.79%, with fetchwith equal to 8 instructions per cycle. In the Mulflux, the results obtained are 7.73% and 7.80% for the same configurations perhaps applying multiple streams.

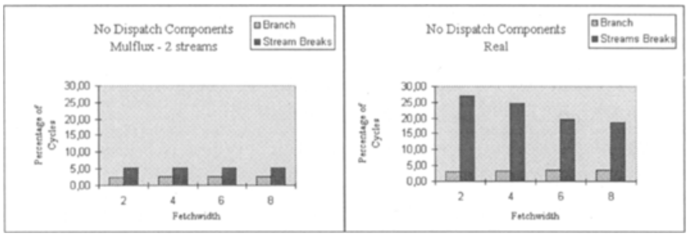

Fig. 3. Components that Causes Emptying Queue in Mulflux with 2 Streams and Real Architecture

The figure 3 presents the efficiency of the multistream model to reduce the empty queue occurrency. In the Mulflux, the stream interruptions are up to 5.24%, but in the Real architecure this percentage reach 27.07%.

The multistreamed model enables a reduction around 74.24% of empty queue occurrency. The effect of stream interruptions were reduced by 80.64% in the Mulflux machine. Another important aspect is that resource conflict and speculation depth must be considered with more attention when the multistream model is used.

4.1 Impact Analysis of Multistream Implementation

Before the implementation of the multistream model, we must consider the impact in the close blocks, and also the fetchwidth, resource conflicts and speculation depth.

In the previous results we did not consider the instruction cache misses. The use of the multistream model could generate more cache misses and the fetch latency can be important. So, the potential of the mechanism can be reduced. Then, we performed some experiments using a instruction cache with the same delay cycles as those of the Intel Pentium to observe the performance under real conditions. In this cache, the miss latency is equal to 3 cycles for the L1 cache, and 13 cycles when the miss causes an access to the L2 cache.

Also, for the last results we consider that the fetchwidth was multiplied by the number of valid stream structures. If the fetchwidth is equal to 8 instructions and the number of valid streams (initialized structures) is equal to 4 then the total and the real fetchwidth is equal to 32 instructions per cycle. This was made in the last experiments.

To avoid problems with the cache size and its configuration we have proposed a new strategy called *Dynamic Split Fetch - (DSF)* which consists in spliting the total fetchwidth between the valid stream structures [6]. In this case, if there are 4 valid streams and the fetchwidth is equal to 8 instructions per cycle, will be fetched 2 instructions for each valid stream and the fetchwidth is kept up to 8 instructions.

4.2 Overall Performance

In this section we discusse the overall performance delivered by the multistreamed mechanism and compare it with real architectures performance. Thus, we could observe the aspects discussed in the last section.

Increasing the number of functional units delivers more speed up for the multistreamed architecture as show in the figure 4.

However, we can observe that no dispatch cycles increases even when the number of functional units increases. This is due to the speculation depth that was kept for all configuration with a single branch unit. The machine considered has n generic functional units and only one branch unit that could stall the dispatch of instructions when saturated. This is showed in the graphic 5.

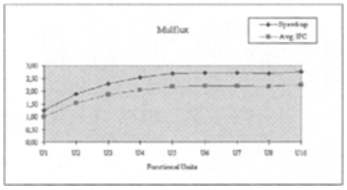

Fig. 4. Multistream Speed up when the Number of Functional Units Increase

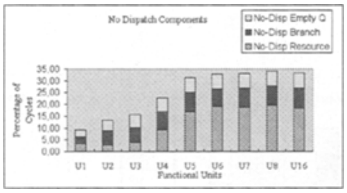

Fig. 5. Components of No Dispatch when Functional Units Increases

We made several simulations variating the number of functional units for the multistreamed architecture. We wanted to show that the resource conflict is the main factor in the limitation of the performance in this machine like in an ideal machine.

We performed experiments with 5 machines consisting of: a Multistream with perfect icache, a Multistream with normal icache, a Multistream with normal icache and DSF (*Dynamic Split Fecth*), a Real machine with perfect icache and a Real machine with normal icache.

The results that will be presented comes from the same configurations for each machine [6]. We used 2 streams structures to the multistream machines. All machines have a fetchwidth equal to 8 instructions per cycle, and a fetch buffer and instruction queue with 16 and 32 entries respectively; a dispatchwidth equal to 8 instructions; 8 functional units, each one with 8 reservation stations; 1 branch unit with 8 reservation stations; 8 bus results and reorder buffer with 64 entries.

The figure 6 shows the no dispatch cycles expended for each machine. The best case is the multistream machine with perfect icache and using a total fetchwidth that consists in multiplying the fetchwidth by the number of valid stream structures. We could observe that the Mulflux with normal icache has a percentage of no dispatch cycles similar to the Mulflux with normal icache using DSF. The division of the fetchwidth by each valid stream do not harm the performance of the considered cases. The Real machines expend more cycles with no dispatch.

The figure 7 shows the components that cause no dispatch cycles in each machine. In the mulstistream machines, the worst component is the resource conflict (around 45.00% of the no dispatch cycles). In the Real machines the emptying queue result around 60% of the occurency of no dispatch. In the second

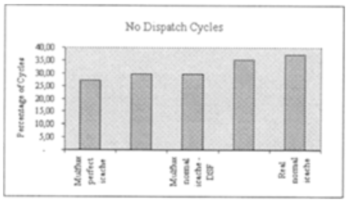

Fig. 6. Percentage of No Dispatch

figure 7, we could notice the reduction of the stream interruptions in multistream machines.

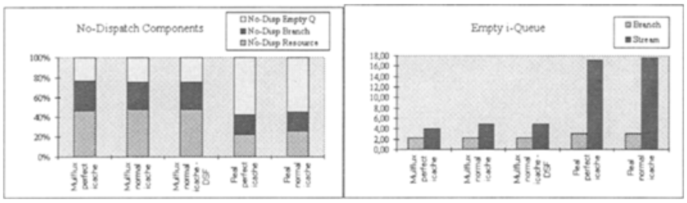

Fig. 7. No Dispatch Components and Emptying Queue Components

5 Conclusions and Future Works

The superscalar speed up is directly proportional to its IPC (*Instructions per Cycle*). It is desirable to obtain a speed up proportional to the number of functional units present in the architecture. However, the constant flow interruptions flush the instruction queue and decrease the number of ready instructions which could be dispatched. Thus, the IPC is reduced drastically because of the instruction queue flushing and a desirable speed up could not be achieved.

The multistreamed model allow a reduction around 74.24% of empty queue occurrency. The effect of stream breaks was reduced by 80.64% in the Mulflux machine. Another important aspect is that resource conflict (structural hazards) and speculation depth must be considered carefully when the multistream model is used.

In our experiments, the instruction cache performed similar performance in both cases: multistream and real architectures. The use of *Dynamic Split Fetch* brings a worthwhile strategy that allows to keep the number of icache buses.

Even if we reduce the occurency of empty instruction queue, we have verified that the decrease of no dispatch cycles do not decrease as we wanted. In the *Mulflux* machine the no dispatch cycles is between 28.99% and 33.11%, while in the *Real* machine it is between 39.30% and 40.50%. The increase of instructions flow in the instruction queue generate a major resource conflict like in the ideal architecture. Increasing the number of functional units but keeping the speculation depth did not allow good results in our experiments. Because of this, we are looking for resource balancing and ideal speculation depth in multistreamed architectures. The saturation of branch unit could come late the instructions execution.

Remarks to the next step could be pointed here. We are looking to introduce new instruction cache mechanisms in our simulations. Such mechanisms have been proposed and we believe that they will be used in next generations of superscalar microprocessors. Also, after to get a good configuration of our architecture we plan to compare it with other alternatives like trace processors [8], simultaneous multithreading [2], multiscalar [7] and other alternatives which certainly will be suggested.

Such comparison is very important to get an idea about the potential of such architecture and its complexity of implementation. The fourth generation of microprocessors can not be predicted at the moment but many new schemes have been proposed. The question is how to increase the performance of current microprocessors and how to obtain more machine parallelism using efficiently the increasingly chip density ?

References

1. Chaves Filho,Eliseu M. et al. A Superscalar Architecture with Multiple Instruction Streams. In: SBAC-PAD, VIII. Recife, Agosto 1996. **Proceedings....** SBC/UFPE, 1996, pp 67-77 (in portuguese).
2. Eggers, Susan J. et al. Simultaneous Multithreading: A Plataform for Next-Generation Processors. *IEEE Micro*, V.17, n.5, Sep/Oct 1997.
3. Fromm; Richard. Branching on Superscalar Machines: Speculative Execution of Multiple Branch Paths Project Final Report. December 11, 1995. CS 252:Graduate Computer Architecture.
4. Yeh, Tse-Yu; Patt, Yale N. Two-Level Adaptive Training Branch Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 24., 1991. **Proceedings...** New York: ACM, 1991. p. 51-61.
5. Yeh, Tse-Yu; Patt, Yale N. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 20., 1993. **Proceedings...** New York: ACM, 1993. p. 257-266.
6. Santos, Rafael R. dos. *A Mechanism for Multistreamed Speculative Instruction Fetch.* Porto Alegre: CPGCC/UFRGS, 1997 (M.Sc. Thesis - in portuguese).
7. Sohi, G. S.; Breach, S. E.; Vijaykumar, T. N. Multiscalar Processors. *Computer Architetcure News*, New York, v.23, n.2, p. 414-425, 1995.
8. Smith, James E,; Vajapeyam, Sriram. Trace Processors: Moving to Fourth-Generation. **Computer**, Los Alamitos, v.30, n.9, p.68-74, Sep. 1997.
9. Talcott, Adam R. *Reducing the Impact of the Branch Problem in Superpipelined and Superscalar Processors.* Santa Barbara: University of California, 1995 (Ph. D. Thesis).