SELECTIVE CONTEXT BLOCKING AFTER BRANCH PERFORMANCE EVALUATION FOR IMPROVING SMT PERFORMANCE

Samir Talegaon and Wei-Ming Lin

Department of Electrical and Computer Engineering,The University of Texas San Antonio,San Antonio, TX 78249-0669, USA

ABSTRACT

Simultaneous Multi-Threading (SMT) is a tech-nique by which multiple independent threads can be simulta-neously executed and it increases the system throughput. For further improving the performance of the machine one more technique known as pipelining is implemented. Pipelining allows the SMT machine to execute more than one function in the same cycle. Pipelining optimizes several processes such as fetch, decode, dispatch, issue, execute and commit. Our focus is on the Dispatch Cycle in which instructions get Dispatched from the individual re-order buffers to the shared Issue Queue. Branch Miss-Prediction reduces the throughput of SMT machine due to the instruction flush-outs. The pipeline then has to be refilled starting from the branch instruction, after changing the branch direction. It causes a delay in the pipeline after which instructions will start to commit from that particular thread and this affects the overall system throughput adversely. Our approach is to implement a technique called as selective context blocking after context-branch evaluation. According to this technique the number of instructions a thread is allowed to dispatch is controlled, if it has a higher Branch Miss-Prediction percent-age within the past certain number of Branch Instructions. As a result of this we get an overall throughput increase of up to 4 % in the performance when compared to the default algorithm.

KEYWORDS

Pipelining, Branch Miss-predictions, Instruction Flush-outs

1. INTRODUCTION

Simultaneous Multi-Threading (SMT) provides us with an excellent improvement in Thread level Parallelism (TLP) and Instruction level Parallelism (ILP) [1] [2] [3]. SMT is basically multiple threads executing simultaneously, using a pipelined architecture. It does not allow the processor to be idle as multiple threads are executed simultaneously and also increases concurrency. Therefore improved resource utilization is obtained along with the help of pipelining. In the past work has been done to improve the efficiency of the SMT architecture by targeting the different Cycles of the pipeline such as Fetch Queue, Dispatch Queue, Issue Queue, Write-Back and Commit cycles.

Fetch Policies are designed to enhance the throughput and fairness by stalling or flushing of threads which experience cache misses [4] [5]. A new policy known as the ICOUNT Fetch Policy has been devised which in [6]. This is a tech-nique for the algorithm to selectively fetch instructions from those threads which have the least number of instructions in the entire pipeline. This provides a fair use of resources by all the threads. There is another Fetch Policy called the ICOUNT2.8-fairness Fetch Policy [7]. In this policy the system observes the fairness given to each thread and if unfairness is detected, it gives a higher priority to that thread. Unready Count Gating is a Fetch control policy defined in [4]. In this technique once the number of instructions which are dispatched into the IQ in an unready state exceeds a particular threshold, that thread is blocked from fetching any more instructions. A technique known as Fetch Halting is introduced in [8], which suspends instruction fetching when the processor is stalled by a critical long latency instruction.

In the Dispatch Cycle, research is targeted towards opti-mizing dispatch of the instructions, one such technique is IQ Capping. In this method, threads which are more "active" are given higher priority to dispatch into the IQ [9] by capping other threads which are relatively lesser "active". Instruction packaging, a method by which two instructions are packaged into one IQ entry, at the time of Dispatch, has been imple-mented [10]. IQ Instruction Recalling, a technique defined in [11] where stagnant instructions from IQ are recalled into the ROB and are thus removed from the IQ. This facilitates the Dispatch of instructions from other threads and maximizes the shared resource utilization. Speculative Capping [12] is a technique in which the dispatch is stalled after a particular number of speculative instructions dispatched by one thread exceed a pre-set threshold value. Capping per Thread Resource Utilization [13] defines a capping technique which enables fair usage of shared resource. In this method, the dispatch is capped at fixed/variable value; this value is the number of slots in IQ that thread is allowed to occupy.

Speculative techniques like branch prediction have been developed in an effort to reduce the branch miss prediction penalties. Many types of branch predictors have been devel-oped and each type has unique ways of making a prediction. Combining these different types of branch predictors and utilizing particular branch predictor which is most accurate for that branch helps to increase the prediction accuracy [14][15]. The cumulative accuracy for such branch predictors has reached almost up to 99 %. This accuracy is lesser for Integer type of instructions and is better for floating point type of branch instructions, which we discuss in detail in the topics below. In [16] Buyuktosunoglu et al. attempts to reduce the energy cost associated with the IQ by dynamically resizing the IQ.

Although these improvements are effective, there is still scope for further enhancing the system throughput by con-trolling how the threads are allowed to dispatch instructions in to the IQ. The SMT provides an inherent improvement of throughput (when compared to single threaded execution). Along with that, SMT technique itself reduces performance of the machine due of the bottlenecks in the shared resources which are compounded by Branch Miss Predictions. One such bottleneck is found in the Issue Queue. The IQ is a small shared resource which accepts instructions from the ROB and issues them to the functional units whenever the functional units become available. The problem is due to the fact that an IQ is not very big in order to accept all the instructions which are ready to be dispatched; hence the simulator uses a round robin technique to achieve fairness in the Dispatch Cycle. If we increase the size of the IQ beyond a certain point, as shown in graph, this provides very small improvement on the overall throughput and does not dramatically increase the performance of the simulator. And it has been observed, once the size of IQ is incremented beyond a certain point, the latency of the IQ increases and system throughput reduces, this is also discussed in sections below.

These shared resources have to be carefully managed else they cause the system throughput to dramatically reduce. Minimum stale instructions in the shared resources have to be maintained as these resources are very limited in quantity. Multiple instruction flush-outs due the Branch Miss Prediction hamper the performance of the machine; one such resource is the IQ. This queue represents point in the SMT pipeline after which the instructions from different threads cease to have independent storage and start to compete for the shared resources i.e.: a single IQ. Hence there is a need to continuously monitor the IQ usage and prevent instructions from a single thread over-occupying space in it. There is also a need to reduce the number of instruction which are allowed to be dispatched by threads which have a higher number of flush-outs, which causes unfairness in the IQ occupancy.

We propose a new technique which can improve the fair usage of IQ and achieve this by controlling the Dispatch of instructions from threads which have a higher count of Miss Predicted Branch instructions. Due to such Branch In-structions the pipeline has to be to be flushed out. Instruction flush-out is the worst case scenario because during the entire length of the time is wasted (i.e.: time from which the branch miss prediction occurs, until the time that the thread can start committing instructions again). Our paper aims to solve this particular problem with selective blocking of threads which show a comparatively higher number of Branch Miss Predictions.

2. ISSUE QUEUE

The Issue Queue in the SMT architecture is a data structure by which instructions can be executed out of order, and it also acts as a dispatch buffer. Before the Issue Queue, the instructions are fetched, decoded, renamed and dispatched in-order using independent data structures called as Instruction Fetch Queue (IFQ) and the Re Order Buffers (ROB). After these Queues, the instructions go into the IQ, and start to compete for this shared resource. This is why the size of the IQ directly impacts the execution dramatically, up-to a certain point. Also, only after the instructions pass through the IQ, the out of order executions are possible. Out of order execution is necessary because it greatly increases the speed of execution and gives us a very good throughput. It is inappropriate to compare two different executions which use different IQ sizes, hence we keep the size of IQ same for a particular comparison and compare between same IQ sizes only. Generally the size of IQ is 16/32/48 and we use these sizes to make a comparison.

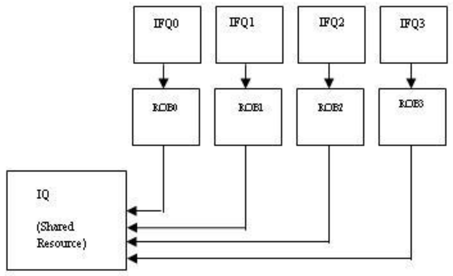


Fig. 1: Part of the structure of a typical pipeline

IQ occupancy is an important aspect which has to be taken into account in order to ensure fair usage of the IQ amongst all threads. Previous researchers have worked on obtaining a fair usage of the IQ by either blocking the Dispatch when one thread was overwhelmingly occupying the IQ, or, by Instruction recall wherein the instructions were recalled to the ROB from the IQ. These methods help to achieve the fairness and improve the throughput of the system however there is more scope to the improvement.

Branch miss predictions can cause unequal usage of the IQ and also reduce the throughput of the IQ. This happens when the instructions belonging to a certain context have to be flushed out because of a miss predicted branch. Also when Branches get miss predicted one after other in a successive manner, the penalty for IQ is compounded.

As IQ is located inside the micro-controller chip, having an extra-large IQ will be cost prohibitive by requiring extensive circuitry. A larger IQ will also cause higher delay when a particular instruction has to be searched and will impact the throughput of the system significantly; this is called as IQ latency. Therefore smart use of the IQ is required in order to achieve optimal efficiency. Our algorithm gives an idea of how this could be achieved by ensuring that all the threads get fair space in IQ, and one thread is not allowed to have a large number of instructions occupied in the IQ. Threads which have a higher branch miss prediction rate also have to be targeted, otherwise they hamper other thread's dispatching efficiency and ultimately the whole system throughput is reduced.

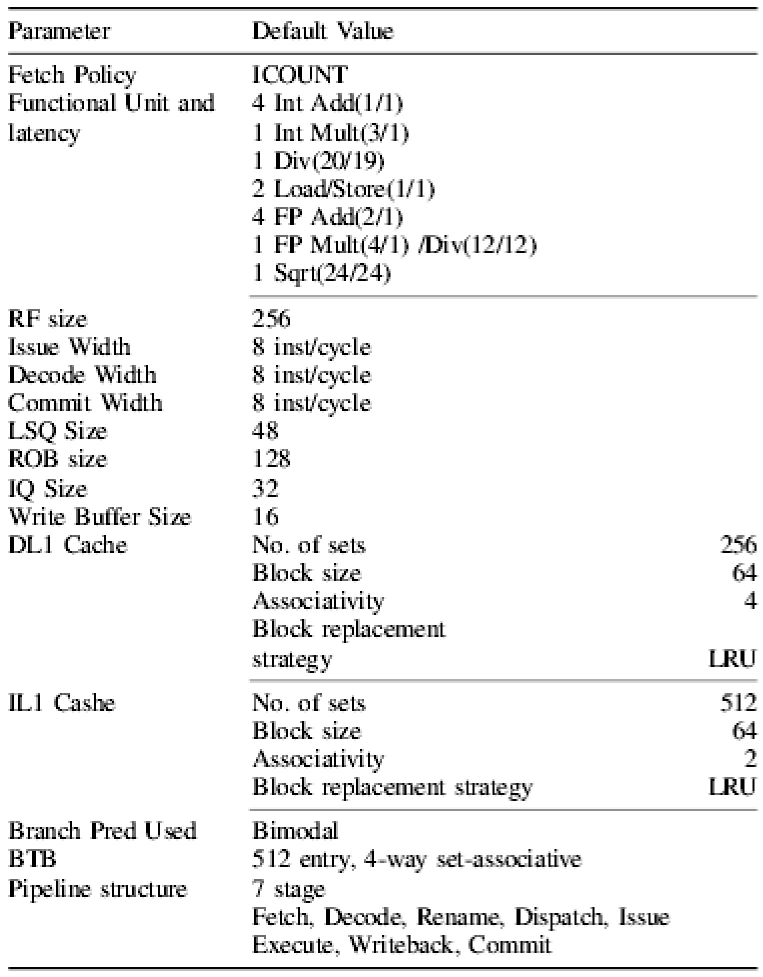
3. SIMULATION ENVIRONMENT

This section explains the environment and a few particu-lars which we use in order to simulate and show our research.

3.1 The Simulator - M-sim

M-sim a multi-threaded (SMT), cycle-accurate simulator is used in order to assess the usefulness of our algorithm [17]. Although m-sim supports both single threaded mode and multi-threaded mode, we only used the multi-threaded mode as perform simulations for the SMT architecture. The default parameters along with modified ones are shown in the above table.

Table 1: Default Parameters used for Simulations



Parameter	Default Value	
Fetch Policy	ICOUNT	
Functional Unit and latency	4 Int Add(1/1)	
	1 Int Mult(3/1)	
	1 Div(20/19)	
	2 Load/Store(1/1)	
	4 FP Add(2/1)	
	1 FP Mult(4/1) /Div(12/12)	
	1 Sqrt(24/24)	
RF size	256	
Issue Width	8 inst/cycle	
Decode Width	8 inst/cycle	
Commit Width	8 inst/cycle	
LSQ Size	48	
ROB size	128	
IQ Size	32	
Write Buffer Size	16	
DL1 Cache	No. of sets	256
	Block size	64
	Associativity	4
	Block replacement strategy	LRU
IL1 Cashe	No. of sets	512
	Block size	64
	Associativity	2
	Block replacement strategy	LRU
Branch Pred Used	Bimodal	
BTB	512 entry, 4-way set-associative	
Pipeline structure	7 stage	
	Fetch, Decode, Rename, Dispatch, Issue	
	Execute, Writeback, Commit	

3.2 Benchmark tools : SPEC2000 Suite

The SPCE2000 [18] suite which consists of INTEGER and FLOATING POINT Benchmarks was used to simulate and obtain the Instructions per Cycle (IPC) for the program. The different Benchmarks which are available with the suite were used with some permutations in order to achieve results which were not influenced by any one particular type of instructions i.e.: just the Integer or Just the Floating Point Type. Following table shows the combination in which they were used.

Table 2: SPEC2000 Benchmarks

Mix	Benchmarks Used
Mix1	vpr gcc mesa galgel
Mix2	vpr gcc galgel equake
Mix3	vpr gcc equake lucas
Mix4	vpr gcc lucas apsi
Mix5	gcc mcf mesa galgel
Mix6	gcc mcf galgel equake
Mix7	gcc mcf equake lucas
Mix8	gcc mcf lucas apsi
Mix9	mcf crafty mesa galgel
Mix10	mcf crafty galgel equake
Mix11	mcf crafty equake lucas
Mix12	mcf crafty lucas apsi

3.3 Metrics

When the simulator calculates the throughput IPC it uses the following formula. In the formula z is the total number of threads used for the simulation.

$$IPC_{(\text{throughput})} = \sum_{1}^{z} IPC_x \qquad (1)$$

The formula which is later used in the paper in order to calculate why we cannot increase the threshold beyond ancertain value is noted below. In the equation,

: is the percentage of instructions flushed,
f: is the number of instructions flushed and
C: is the number of instructions committed during the whole simulation.

$$\rho = \frac{f}{C} * 100 \qquad (2)$$

How we find out the number of instructions flushed out is as follows:

$$f = f_p + inst_R + inst_F - inst_C - 1 \qquad (3)$$

where f: denotes the total instructions flushed out during one whole simulation,
f_p: denotes the number of instructions flushed in previous clock cycles,
$inst_R$: denotes the number of instructions in the ROB,
$inst_F$: denotes the number of instructions in IFQ and
$inst_C$: denotes the number of instructions committed but still in ROB. The '1' being subtracted is due to the fact that the rollback algorithm leaves at-least one instruction in the ROB after flush-out (the branch instruction itself).

3.4 Legend

In our paper (Graphs) we have used several symbols in order to denote various notations. These are listed below along with their full meaning.

f = Percentage of instructions which are flushed
T = The Different Threshold values taken for simulation |IQ| = Size of the Issue Queue.

4. Motivation

The motivation for this paper is obtained from the fact that the IQ size is the first shared resource an instruction encounters in a pipeline and has a dramatic impact on the IPC. Also, even with the improvements done to the Branch Predictors, we still have a considerable number of instructions getting flushed from the pipeline; this is shown in the graph below. As we can see, for the mixes # 2, mix #10, mix #12 almost a third of the instructions fetched are flushed-out due to the Branch Miss Predictions and the average number of instructions flushed-out is 25 % in any given mix. This severely hampers the system throughput by causing pipeline-refills continuously and is a very serious flaw which needs to be addressed immediately which is what we attempt to do in our paper.

There is one more substantial observation and that is when the IQ size is increased the % of flushed instructions decrease, however this is counter-intuitive to our assumption that IQ size needs to be increased in order to improve performance.

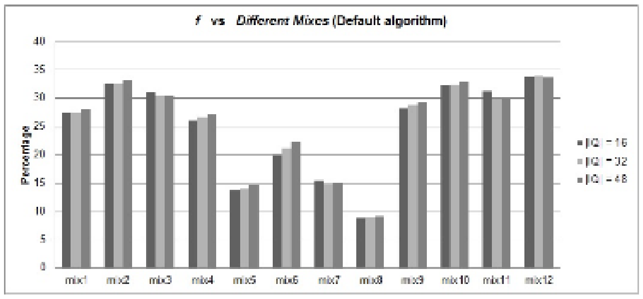


Fig. 2: Percentage of instructions getting flushed during entire simulation using the default algorithm

So according to the graph if IQ size is dropped the system throughput must rise. However as we know this is not true, when IQ size is reduced beyond a certain point the system throughput reduces. Therefore it becomes very important to address the problem that the IQ, as a shared resource needs to be monitored carefully and the threads which are allowed to dispatch instructions in it must be chosen similarly.

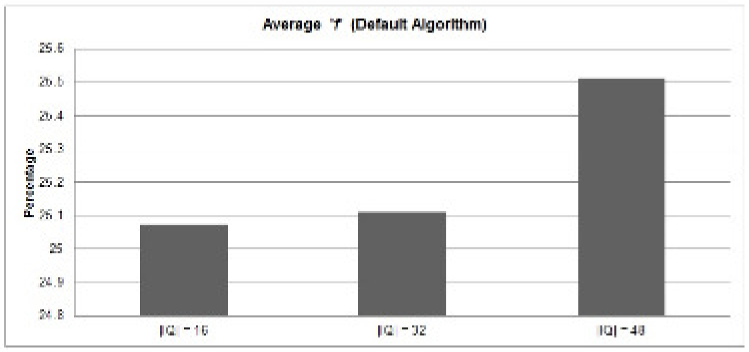


Fig. 3: Average Flush % for default algorithm

One solution to this problem would have been just to increase the size of IQ in order to allow for more instructions to occupy it. However the following graph shows us why this does not have the linear effect as desired. As we see from the graph, after a certain IQ size which is about 40, increasing the IQ size does not have the desired effect as compared to increasing it from 4 to 40 in the first place. We can therefore say that we cannot just increase the size of the IQ in order to improve the system throughput.

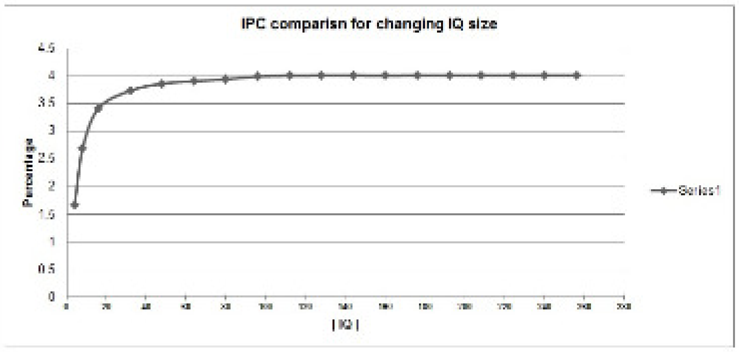


Fig. 4: Effect of varying IQ size to the IPC

The INTEGER type branch instructions have a higher miss prediction rate than the FLOATING POINT type. Hence we initially run the simulation for individual mixes of Integer and Floating point benchmarks. The simulation results are shown in section 6. We get a good 16 % improvement in the Integer benchmark simulations; however any general program consists of a mixture of INTEGER and FLOATING POINT type Instructions. Therefore we run the simulations for a mix of Integer and Floating point benchmarks. The original simulator which has been designed without taking this in account, suffers by having reduced throughput as a consequence.

Whenever a branch gets Miss Predicted there is a flush out of instructions from the ROB. Due to this flush, there is a penalty associated with every Branch Miss Prediction and this penalty is relative to the size of the pipeline of the machine. So if a particular thread has a higher number of Branch Miss Predictions, the penalty will be higher for that thread and consequently it will reduce the performance of the machine.

One more important point to note is in the Fetch Cycle, the simulator M-sim uses a Fetch Policy called as ICOUNT Policy. According to this policy the simulator will give higher priority for fetching instructions from those threads which have lowest number of instructions in the queues. Now as we can see the branch miss prediction problem is compounded as few threads which have higher number of branch miss predictions will be prioritized by the simulator in the fetch cycle and as a result the whole system throughput will reduce.

5. Methodology

Modifications to the Dispatch and WriteBack stages of the default algorithm are proposed in this paper. In the Dispatch stage a control algorithm is implemented which can limit any given thread's dispatch allowance into the IQ and in the WriteBack stage we make certain calculations based on the number of Miss Predicted Branch Instructions to decide whether to apply this control for the current simulation cycle.

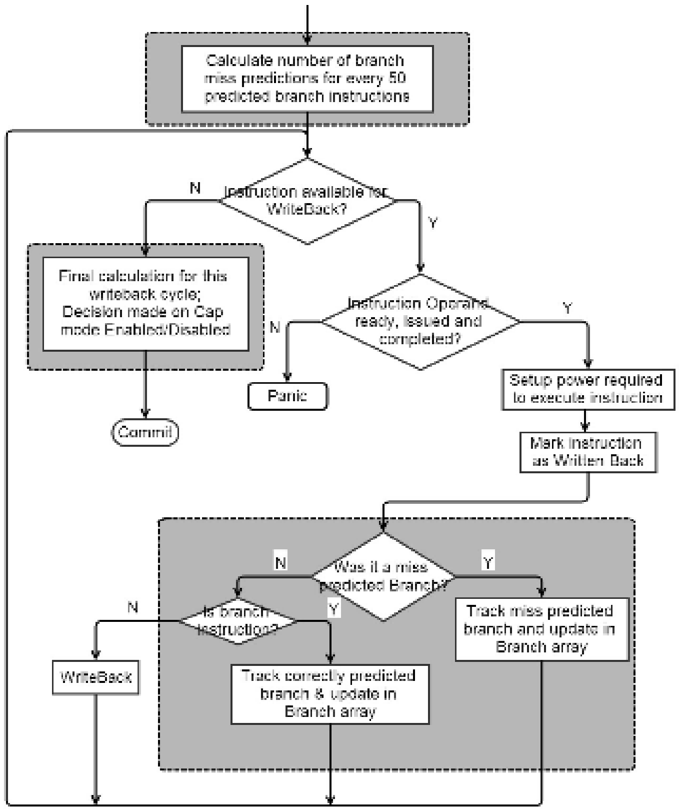

Calculate number of branch miss predictions for every 50 predicted branch instructions
Instruction available for WriteBack?
N
Y
Final calculation for this writeback cycle; Decision made on Cap mode Enabled/Disabled
Commit
Instruction Operands ready, issued and completed?
N
Y
Panic
Setup power required to execute instruction
Mark instruction as Written Back
Was it a miss predicted Branch?
N
Y
Is branch instruction?
N
Y
Track miss predicted branch and update in Branch array
WriteBack
Track correctly predicted branch & update in Branch array

Fig. 5: Writeback Algorithm (The shaded portion is the modification we have made to the Original Algorithm)

In the simulator, the writeback stage is where the branch miss predictions get dealt with. When the simulator finds out that a branch has been Miss Predicted, it simply Rolls-back to the branch which was incorrectly predicted and flushes out instructions from the ROB which lie before the branch. Such a flush-out has a penalty associated with it in terms of number of simulation cycles it takes to refill these Queues. Our algorithm aims to reduce such a branch miss prediction penalty by reducing the space available for the threads which cause these flush-outs more frequently. This is shown in the flowchart via the highlighted sections which mark our modifications made to the original code.

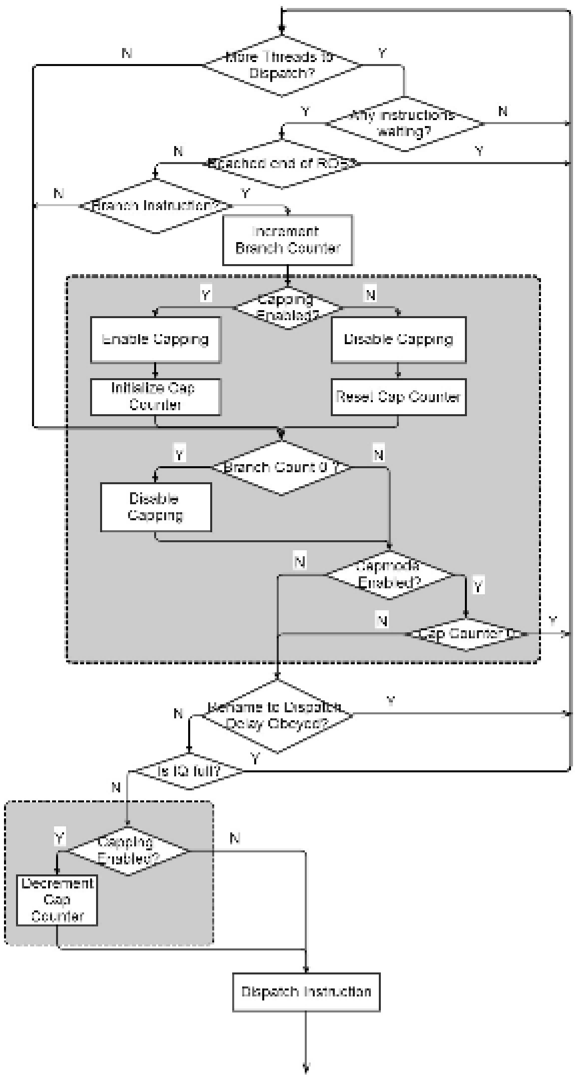


Fig. 6: Dispatch Algorithm (The shaded portion is the modification we have made to the Original Algorithm)

The way we achieve it is by taking into consideration the past 50 branch instructions for a particular thread. We count the number of Branch Miss Predictions which have been encountered. After this we choose a threshold (via testing we have found out that "4" is a good enough threshold) for that particular thread after which, if that thread has any more Branch Miss Predictions we temporarily stop that thread from dispatching instructions into the Issue Queue. This is to limit its occupancy in the shared resource and reduce the number of instructions which are flushed-out. This policy allows for other threads (which issue correctly predicted branch instructions) to occupy the IQ and thus results in a better shared resource utilization.

For the control algorithm we go to the Dispatch Stage where instructions are dispatched from their individual ROBs to the Shared IQ. This stage is very important because the instructions can compete for the shared resource, which was not the case in the individual ROB. Then we choose what instructions are allowed to occupy the ROB as this would dramatically impact the throughput of the system. A simple global variable remembers whether to limit the current thread being Dispatched or not, which is set in the Writeback Stage. When we find that the current thread has to be limited from Dispatching we CAP that thread. In this way the other threads automatically get a higher priority and continue to dispatch their instructions into the IQ.

6. Simulation Results

We run simulations based on our algorithm and plot graphs based on these simulations. The graph for those are shown below

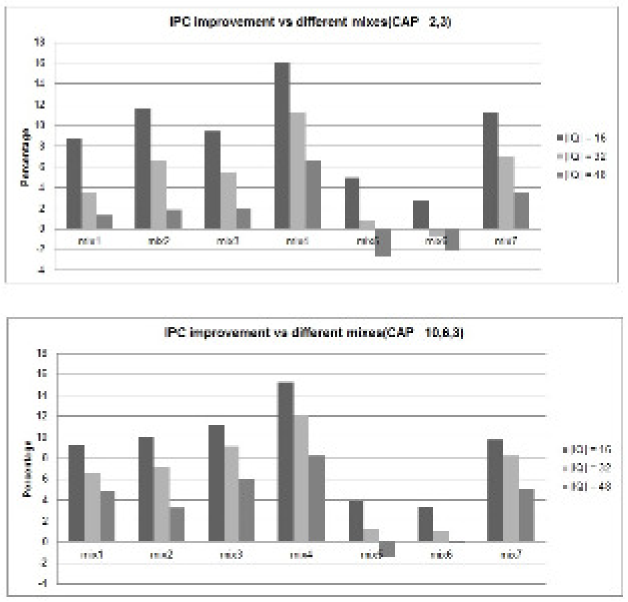


Fig. 7: Improvement for Integer Benchmarks alone compared to Default Algorithm;CAP values (10,6,3)

Primarily we ran the simulations for Integer and Floating Point Benchmarks individually. The improvement which was observed for Integer benchmarks was up to 16 %. The explanation for this improvement is in the fact that Integer benchmarks have a comparatively lesser branch prediction accuracy than floating point benchmarks. So when we selectively block the miss predicting threads from dispatching for a period of time until the branch is resolved, we get an improvement in the system throughput. Following are two examples of the improvement observed in the Integer Benchmarks compared to the Default Algorithm. The two different graphs are for two different values of CAP as shown in the label.

However due to the fact that in a regular program there would be no differentiation between integer and floating point instructions. For this we have to combine the two benchmarks and create 12 mixes which are listed in Section 3. For these mixes, when the original and the modified algorithm throughput are compared as we can see from the simulation graphs, the improvement in the IPC touches 9%.

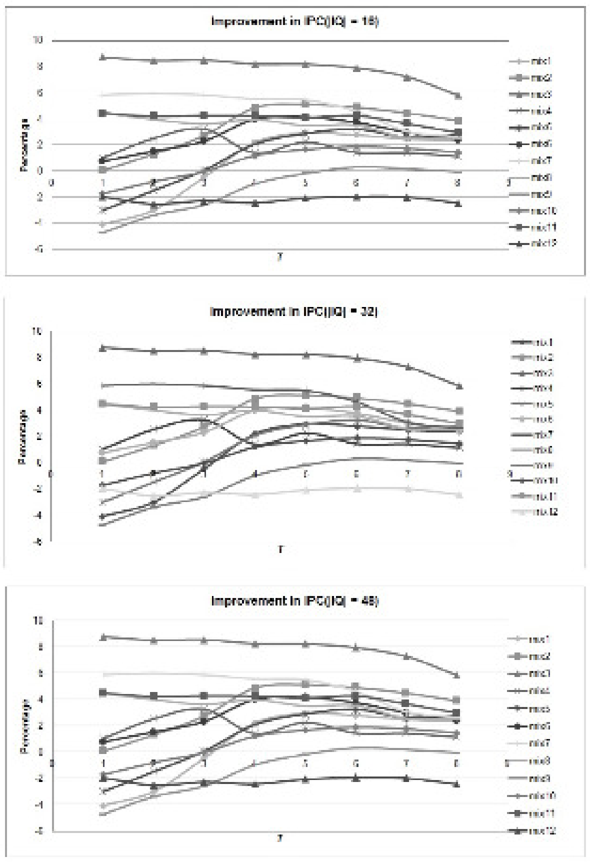


Fig. 8: Improvement in IPC when compared to default IPC , IQ = 16/32/48

The IPC improvement that is shown in following graph is calculated by taking an average of the improvement percentages in the different mixes used and it shows a 4 % improvement.

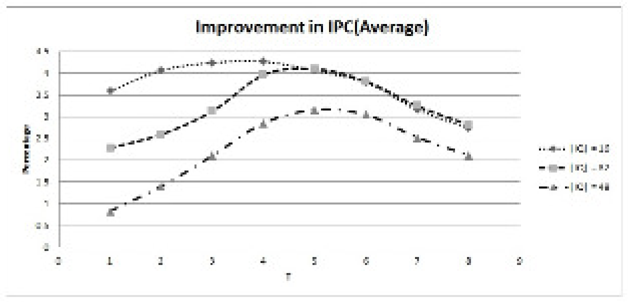


Fig. 9: Average Improvement of IPC when compared to Default IPC

As mentioned earlier this 4% increase is for the threshold value of "4" branches every 50 branch instructions and the CAP value is also fixed at 4 instructions. The CAP value is just the number of instructions to be dispatched after a branch instruction is encountered after which we CAP the Dispatch for that thread.

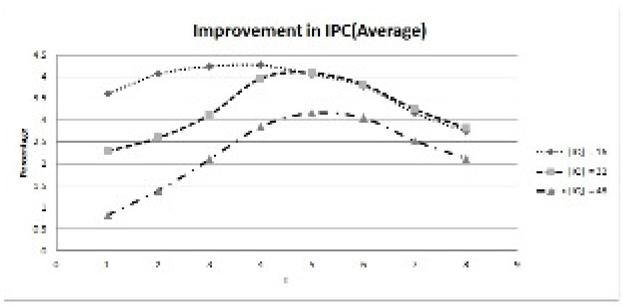


Fig. 10: Average Improvement of IPC when compared to Default IPC

This improvement is because we slow the Dispatch of such threads which have a higher branch miss prediction rate (more than 4 out of 50 past branch instructions miss predicted) and allow those threads which have a reduced branch miss prediction rate(less than 4 out of the past 50 branch instructions) . In that way the IQ utilization increases because of the fact that more branches which pass through the IQ are correctly predicted and hence lesser number of flushes take place.

6.1 Validity of the Proposed Algorithm

In order to confirm that our algorithm is valid and really improves the throughput we analyze the number of instructions flushed throughout the simulation.

Comparing this value with the number of instructions that are flushed in the original algorithm we can say that that our algorithm reduces the percentage of instructions flush overall when the threshold is above ’2’. That means that in all the mixes we use, our algorithm correctly predicts

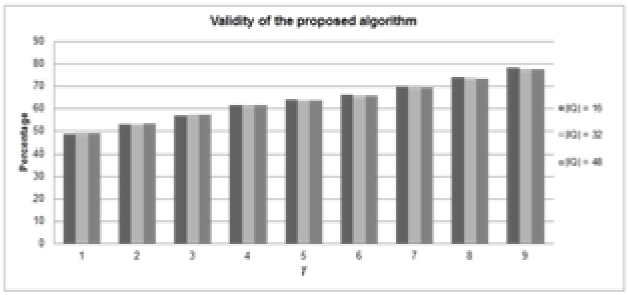


Fig. 11: Validity of the algorithm

when a branch miss prediction is going to occur and reduces its impact on the simulator by controlling the instructions that are allowed to go in to the IQ from that thread.

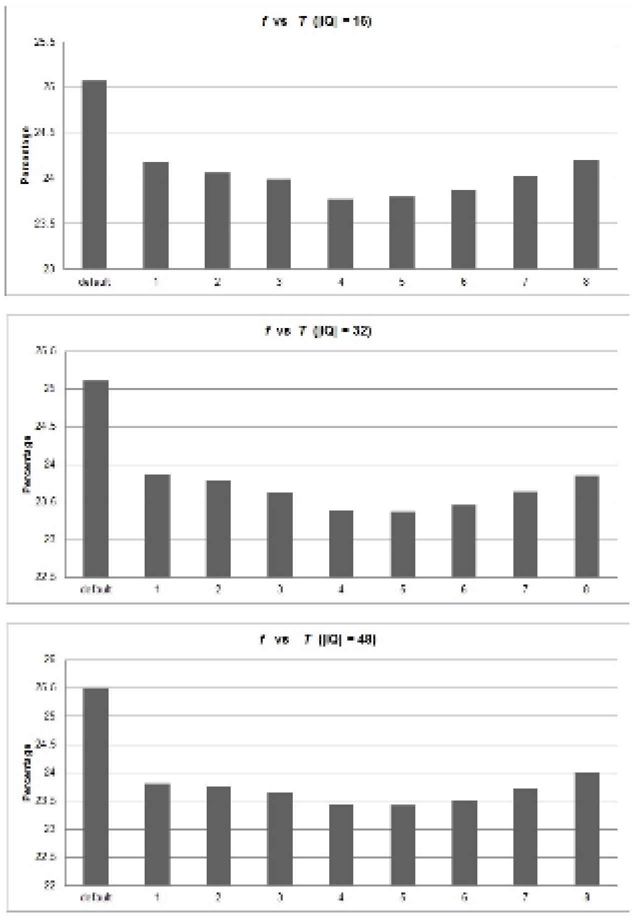


Fig. 12: Percentage of instructions flushed compared to different threshold values

According to the Validity Graph, we observe an important fact, that is if we go on increasing the threshold above 4 the graph shows higher percentage of validity. However we need to note that according to our calculations there are not many threads having, say, 9 branch miss predictions out of past 50 branch instructions and this is shown in the plots below.

As we can see increasing the threshold to that value will have a negative impact on the IPC because at that value of threshold our algorithm is unable to differentiate between the Integer and Floating Point Branch Instructions. Thus there has to be a trade-off between Validity of the Algorithm and throughput enhancement. To prove the above point we plot graphs for percentage of instruction flushed-out versus the threshold used. Note that in the graph the instructions flushed increases as we increase the threshold above 4. Thus we can say that we cannot go on increasing the threshold to improve the throughput of the program although this increases the validity of our algorithm.

7. Conclusion

We conclude that our selective Capping of threads whose branch prediction performance drop below a certain point, resulted in the improved usage of the IQ. This is because the amount of shared resource wasted i.e.:the IQ, by such a thread is limited. It increases the performance of the SMT machine by up-to 16 % (for Integer benchmarks) with an average improvement of 4 % . An even larger improve-ment in performance can be expected if the Floating Point benchmark performance for our algorithm was increased. We have shown via simulation, that the IQ plays a crucial role in the throughput obtained. It was also shown that just purely increasing the IQ size, above a certain threshold does not improve the performance of the machine linearly, and in fact it reduces the performance. We also conclude that improvement is achieved with minimal modifications to the original hardware and thus is cost effective .

References

[1] D. M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo and R.L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Imple-mentable Simultaneous MultiThreading Processor", In the proceedings of the 23rd Annual International Symposium on Computer Architecture, pp.191-200, May 1996.

[2] D.Tullsen, S.J. Eggars, H.M. Levy,"Simultaneous Multithreading: Max-imizing On-Chip Parallelism",22nd Annual International Symposium on Computer Architecture, 1995. Proceedings., pp 392-403, June 2995.

[3] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase and T. Nishiwaza,"An elementary processor architecture with simultaneous instruction issuing from multiple threads",ISCA '92 Proceedings of the 19th annual international symposium on Computer architecture, pp 136-145, NY, 1992.

[4] A. El-Moursy and D. H. Albonesi,"Front-End Policies for Improved Issue Efficiency in SMT Processors", HPCA '03 Proceedings of the 9th International Symposium on High-Performance Computer Architecture,pp 31-40,February 2003.

[5] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero,"Improving memory latency aware fetch policies for SMT processors",IEEE 13th International Symposium on High Performance Computer Architecture, 2007. HPCA 2007, pp 240-249, February 2007.

[6] J.L. Hennessey and D.A. Patterson, "Computer Architecture: A Quan-titative Approach", Morgan Kaufmann Publishers, 2007. [7]C. Sun, H. Tang, and M. Zhang, "Enhancing ICOUNT2.8 Fetch Policy with Better Fairness for SMT Processors",ACSAC'06 Proceedings of the 11th Asia-Pacific conference on Advances in Computer Systems Architecture,pp 459-465, 2006.

[8] N. Mehta, B. Singer, R.I. Bahar, M. Leuchtenburg, and R. Weiss,"Fetch Halting on Critical Load Misses",Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors,2004. ICCD 2004., pp 244-249, October 2004.

[9] J. Sharkey, D. Balkan, D. Ponomarev,"Adaptive reorder buffers for SMT processors",PACT '06 Proceedings of the 15th international conference on Parallel architectures and compilation techniques, pp 244-253, 2006.

[10] J. Sharkey, D. Ponomarev,"Exploiting Operand Availability for Effi-cient Simultaneous Multithreading",Computers, IEEE Transactions on (Volume:56 , Issue: 2),pp 208-223, February 2007.

[11] Y. Zhang, C. Douglas, W.-M. Lin, "On Maximizing Resource Utiliza-tion for Simultaneous Multi-Threading (SMT) Processors by Instruc-tion Recalling",The 18th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12),July 16-19,2012, Las Vegas, NV.

[12] Y Zhang, Wei-Ming Lin,"Capping Speculative Traces to Improve Per-formance in Simultaneous Multi-Threading CPUs",Parallel and Dis-tributed Processing Symposium Workshops & PhD Forum (IPDPSW),2013 IEEE 27th International , pp 1555-1564, May 2013.

[13] T. Nagaraju, C. Douglas, W.-M. Lin and E. John,"Effective Dispatch-ing for Simultaneous Multithreading (SMT) Processors by Capping Per Thread Resource Utilization",The Computing Science and Technology International Journal, Vol. 1, No. 2, pp 5-14, December 2011.

[14] S. McFarling,"Combining Branch Predictor",DEC Western Research Library Technical Note TN-36, June 1993.

[15] S. Pan, K. So and J. T. Rahmeh,"Improving the Accuracy of Dy-namic Branch Prediction Using Branch Correlation",5th International Conference on Architectural Support for Programming Languages and Operating System, pp 76-84, October 1992.

[16] A. Buyuktosunoglu, et al.,"A Circuit-Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors.â□AI

[17] J Sharkey, "M-SIM: A Flexible, Multithreaded Architectural Simulation Environment", Tech. Report CS-TR-05-DPI , Department of Computer Science,SUNY Binghamton, October 2005.

[18] Standard Performance Evaluation Corporation (SPEC) website, http://www.spec.org .

[19] J. P. Shen, M. H. Lipasti, "Modern Processor Design: Fundamentals of Superscalar Processors",Waveland Press, July 2013.