Branch Classification to Control Instruction Fetch in Simultaneous Multithreaded Architectures*

P.M.W. Knijnenburg
LIACS, Leiden University, the Netherlands,
peterk@liacs.nl

A. Ramirez, F. Latorre, J. Larriba, M. Valero
Department d'Arquitectura de Computadors, UPC, Spain,
{aramirez,flatorre,larri,mateo}@ac.upc.es

Abstract

In Simultaneous Multithreaded architectures many separate threads are running concurrently, sharing processor resources, thereby realizing a high utilization rate of the available hardware. However, this also implies that threads are competing for resources and in many cases this competition can actually degrade overall performance. There are two major causes for this: first, instructions that, because of a long latency data cache miss, cause dependent instructions not to proceed for many cycles thereby wasting space in the instruction queues, and second, execution of instructions that belong to a mispredicted path. Both of these have a harmful effect on throughput and the second moreover wastes energy.

In this paper we propose a fetch policy that avoids issuing instructions to the pipeline if we are not confident that the instruction belongs to the correct execution path. In this way, we avoid using resources for instructions that will not contribute to performance. This fetch policy, called agstall, is based on a dynamic branch classification mechanism. Branch instances are classified as either strongly biased or not strongly biased. We consider all strongly biased branches as easy to predict, and we stall the thread on not strongly biased branches to avoid mispredicting them. Our results show that agstall achieves similar or better performance than icount, and reduces by up to 86% the number of wrong-path instructions executed.

*This research was supported by the EC IST programme (contract HPRI-CT-1999-00071), by the Ministry of Science and Technology of Spain (contract TIC2001-0995), and by CEPBA.

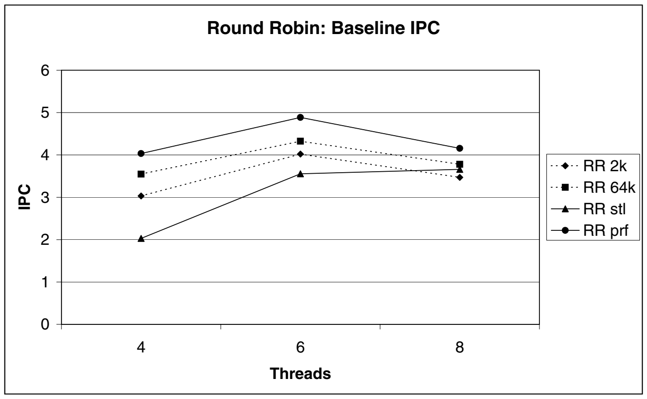
1. Introduction

In Simultaneous Multithreaded architectures (SMTs) many separate threads are running concurrently, sharing the resources of a single processor, thereby realizing a high utilization rate of the available hardware. In this way both Instruction Level Parallelism (ILP) within a single thread, and Thread Level Parallelism (TLP) between different threads can be exploited, giving rise to low horizontal and vertical waste [9]. However, in an SMT architecture threads are competing for resources and in many cases this competition can actually degrade performance. This effect is illustrated in Figure 1(a) where we show, for several branch predictors, the throughput for 4, 6, and 8 threads for the processor configuration given in Section 3. We observe that throughput decreases for 8 threads because too many threads are competing for scarce resources, exept for the stall-on-branch scheme. The present SMT configuration is better suited to support 6 than 8 threads.

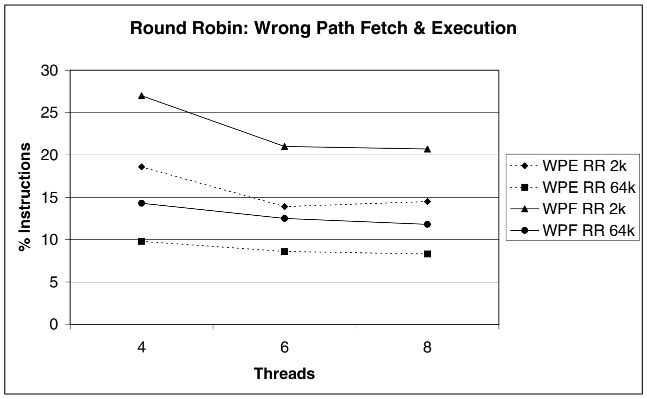
An important shared resource in an SMT architecture are the instruction queues where instructions are waiting to be dispatched to the functional units. We would like to prevent that these queues are polluted which can happen because of

1. chains of instructions that are dependent on a long latency instruction like a load that misses in the L2 cache or a divide instruction, and
2. instructions from mispredicted paths.

The most serious challenge for first objective are loads that miss in the L2 cache since this can take 100 cycles or more. The *icount* instruction fetch mechanism [16] partially deals with this problem by fetching from threads with least instructions in the frontend, thereby enforcing short depen-



(a) Throughput



(b) Waste caused by misprediction

Figure 1. SMT performance for Round Robin instruction fetch using a 2k-entry and a 64k-entry gshare predictor (RR 2k and RR 64k, respectively), perfect branch prediction (RR prf), and stall on branch (RR stl).

dence chains in the queue. Next, adaptations of the instruction fetch engine have been proposed that take into account cache behavior. Limousin et al. [8] predict whether a load gives rise to an L2 miss and subsequently restrict instruction fetch from this thread. Tullsen and Brown [15] detect L2 misses and flush the frontend of instructions that are dependent on this load.

The second objective can obviously be tackled by stalling on a branch and fetching the next instruction when the branch is resolved. However, as can readily be seen in Figure 1(a), stalling on a branch reaches far lower throughput than when a branch predictor is used for 4 and 6 threads. In these cases there is not enough parallelism available to keep all resources occupied in case we stall on branches. For 8 threads, however, there is enough parallelism and IPC for stall-on-branch is higher for 8 than for 6 threads. IPCs for 8 threads is less than IPCs for 6 threads for the other schemes because 8 threads gives too much competiton: even for perfect branch prediction (where there are only correct path instructions) IPC for 8 threads is less than IPC for 6 threads because of too much competition. In Figure 1(b) we have plotted the waste caused by imperfect branch prediction, where we show the instructions that are fetched along a mispredicted path as a percentage of the total number of instructions fetched and the instructions executed along a wrong path, as a percentage of the total number of instructions executed. Wrong path fetching of instructions wastes bandwidth to the instruction cache and energy. Instructions that are executed along a wrong path waste resources, degrading the performance, and energy.

Taken together, Figures 1(a) and (b) show a dilemma. If we do not want to execute along a mispredicted path thereby minimizing waste and using all resources for useful work, we need to stall upon encountering a branch. Stalling, however, degrades performance to unacceptable levels and therefore we need to incorporate some kind of prediction mechanism. However, by doing this we inevitably create mispredicted paths and hence waste resources and energy.

The goal of the present work is to reduce the number of wrong path instructions in the instruction queue, thereby reducing waste, without reducing performance. We achieve this by only fetching instructions past a branch when we are highly confident that the branch is predicted correctly.

There are two ways to ensure that a prediction will probably be correct. First, we can use a *confidence estimator* [7] and stall a thread if a branch is likely to be predicted wrong. Second, we can use a *branch classifier* to classify branches as easy or hard to predict. It has been shown [4] that about 60% of all dynamic branches is heavily biased: they are either taken or not taken for over 95% of the times they are encountered. This observation has been exploited previously to enhance branch prediction accuracy [1, 12]. Also, branches can have a low transition rate, that is, they do not change direction often. In [5] it has been shown that over 60% of the dynamic branches have a transition rate of less than 5%. Hence, branches tend to be 'stable' during the execution of a program and are therefore easy to predict. In order to decide between branch classification and confidence estimation as the mechanism to stall on branches, we conducted some initial experiments and found that branch classification outperforms confidence estimation considerably, both in resulting throughput and wrong path execution. The reason for this seems to be that confidence estimation is too aggressive in speculating past branches. Therefore we employ a branch classification mechanism in this paper. We call the resulting instruction fetch policy *agstall*.

By using the *agstall* branch classifier and instruction fetch policy, we achieve the best of both worlds:

- good prediction accuracy for easy branches (up to 99.2% accurate) to ensure high throughput of the entire machine and high IPC for individual threads, and
- a low percentage of instructions executed along a wrong path and hence low energy waste in program execution.

We show that *agstall* in many cases outperforms conventional instruction fetch policies in terms of throughput and always outperforms them with respect to wrong path fetch and execute. We study the design space of *agstall* by considering two implementations, a small and a big one. Moreover, we discuss the parameter that determines how aggressive branches are classified as easy and hence the aggressiveness of speculation in *agstall*.

The paper is organized as follows. In section 2 we discuss related work. In section 3 we discuss our experimental setup. In section 4 we discuss our proposed branch classifier and instruction fetch mechanism *agstall*. We present experimental results in section 5 and an analysis of *agstall* in section 6. Finally, we draw some conclusions in section 7.

2. Related work

It is well known that accurate branch prediction is crucial for harvesting high levels of Instruction Level Parallelism in conventional superscalar processors. In the past few decades, many proposals for branch predictors have been put forward, ranging from simple one bit last direction prediction, to sophisticated two level adaptive schemes [4, 13, 17]. An important proposal has been the gshare branch predictor by McFarling [11] in which access to the pattern history tables is hashed by XORing branch address with global branch history. Gshare is the base predictor used in this study. Hily and Seznec study branch prediction for multithreaded architectures [6].

In this paper we employ a dynamic branch classifier that classifies branches as either easy or hard. In [1, 3], dynamic branch filter mechanisms are proposed to filter out these biased branches using simple counters and only using the dynamic predictor to predict the other branches. Alternatively, profiling can be used to classify branches as easy or hard to predict [2, 12]. In this paper, we employ a dynamic branch classifier resembling the classifier from [1]. The main difference is that we do not reset the classifier counters to zero when a branch does not comply with its bias, but decrement the counter instead. This introduces some slack in the classification mechanism. Furthermore, like in [12], we use a small agree-type predictor [14] to predict easy branches, rather than using their bias as prediction. It has been shown [12] that this arrangement improves branch prediction for easy branches significantly. However, unlike [1, 12], we do not try to predict the hard branches but stall the instruction fetch for this thread instead.

Pipeline gating [10] uses a branch prediction confidence estimator [7] to control instruction fetch: on a low confidence branch the pipeline is stalled. They show a slight decrease of performance together with a large decrease of wrong path instructions. This technique strongly resembles our technique. The main difference is that we propose a branch classifier that filters out biased branches instead of using confidence estimation. The main advantage is that non-biased branches do not pollute the branch predictor in our approach and a small dynamic predictor can be used with a high prediction accuracy (up to 99% accurate for 8 threads).

Instruction fetch policy is another important factor for SMT performance. In [16] it has been shown that the *icount* fetch policy performs best over a number of fetch policies. In this policy, priority is given to threads with few instructions in the front end of the processor. The *agstall* mechanism resembles *icount* to a certain extend in that threads that are less likely to be mispredicted are favored. Ties can be broken by *icount*, harvesting the strong points of this mechanism. The BRCOUNT fetch mechanism from [16] that favors threads that are least likely to be on a wrong path resembles *agstall*, but it differs in that it decides this based on the number of branches in the frontend, instead of classifying branches as in our approach.

3. Experimental setup

We use the SMT simulator provided by Dean Tullsen [16], called SMTSIM, which is based on the Alpha ISA. The characteristics of the simulator used are listed in Table 1. The simulator stops whenever one of the threads finishes. Since rename registers are an obvious bottleneck in this type of architecture, we have varied the number of rename registers from 32 to 256 and ran a number of experiments to decide the optimal number. We found that in general IPC values were severely degraded for 32 or 64 rename registers. For 192 or 256 rename registers there was a slight increase in IPC compared to 128 rename registers. However, this increase is so little that we did not consider the extra cost justified. Therefore, we employed 128 rename registers for the remainder of our experiments. We use both Round Robin instruction fetch and the *icount* fetch policy.

We use all SPECint95 benchmarks, except `m88ksim` that did not execute properly in the original simulator. We use

Parameter	Value
Threads	8 thread contexts
Fetch width	8 instructions from up to 2 threads
Instr. queue	32 deep integer and FP
Instr. latencies	Based on Alpha 21164
BTB	256 entry, 4-way set associative
FUs	6 integer (including branch), 3 FP, 4 load/store, 2 synchronization
Pipeline	8 stages
Branch	6 cycles mispredict penalty
I-Cache	64 KB, 2-way, 64b lines
D-Cache	64 KB, 2-way, 64b lines
L2	1 MB, 2-way, 64b lines
L3	4 MB, 2-way, 64b lines
Latencies from previous level	L2 10 cycles, L3 20 cycles Memory 100 cycles

Table 1. Processor configuration

the train input data, except for `compress` where we use reference data since the train data proved to run for too few cycles. In order to be able to use 8 threads, we use `vortex` twice, once with train data and one with reference data.

We study the situation for 4, 6, and 8 threads. For 4 threads we use `perl`, `vortex`, `li`, and `gcc`. For 6 threads we add `ijpeg` and `compress`. For 8 threads we add `go` and the second version of `vortex`. We have chosen these collections of benchmarks in order to have a balanced mix of benchmarks [4].

4. *agstall* instruction fetch policy

In this section we discuss the *agstall* branch classifier and instruction fetch policy which is based on the AGBIAS branch predictor [12] in which branches are statically classified by profiling. In *agstall* we classify branches dynamically as easy or hard. We predict easy branches while hard branches cause the thread to stall. If there are enough easy branches, then single thread performance will not be degraded too much by stalling on hard branches. Easy branches can be accurately predicted (up to 99.2% accurate) and hence the number of instructions fetched and executed along a mispredicted path will decrease. At the same time, in a simultaneous multithreaded environment, there are enough instructions from other threads available to keep the processor busy and the overall performance will not degrade by stalling. In fact, we show that in many cases performance will increase, due to improved sharing of resources and less destructive competition. The result is an SMT processor organization that has high throughput, but wastes less resources and hence consumes less energy.

The *agstall* scheme is shown in Figure 2 and consists of two structures: a branch classifier and a branch predictor.

Branch classifier consists of a direct mapped table of 8 bit counters. The most significant bit indicates the branch direction. Each time a branch is executed, the classifier is updated. If the classifier and the actual direction of a branch coincide, the counter is incremented by 1. If the counter reaches 64, the branch is classified as *easy*. Hence the 7th bit in the counter indicates whether the branch is easy or not. The counters count further until they saturate at 127. In case the classification and the actual direction of a branch do not coincide, the counter is decreased by 16.

Branch predictor For easy branches, we use an agree predictor [14] to predict its outcome. The agree predictor consists of a gshare-type [11] Pattern History Table (PHT) indexed by XORing branch address with the BHR. The PHT consists of 2 bit saturating counters that are interpreted as *agree* or *disagree* with a direction bit from the direction table. The direction table is indexed by the branch address and the direction bit is set upon first encounter of a branch.

We employ two configurations of different size. The small implementation is slightly larger than the small gshare predictor (512B), and the large implementation uses as many bits as the large gshare predictor.

agstall **2k** uses a 2k-entry classifier, a 2k-entry agree predictor, and a 8k-entry direction table. We use an 11 bit BHR in this configuration. It uses 2KB + 512B + 1KB = 3.5 KB.

agstall **8k** uses a 8k-entry classifier, a 16k-entry agree predictor, and a 16k-entry direction table. We use a 14 bit BHR. It uses 8KB + 4KB + 4KB = 16KB.

5. Results

In this section we present the results obtained by the *agstall* instruction fetch policy and compare them to the conventional instruction fetch policies Round Robin and *icount* [16].

5.1. Round Robin instruction fetch

In this subsection we show results of adding the *agstall* scheme to a Round Robin instruction fetch mechanism. We

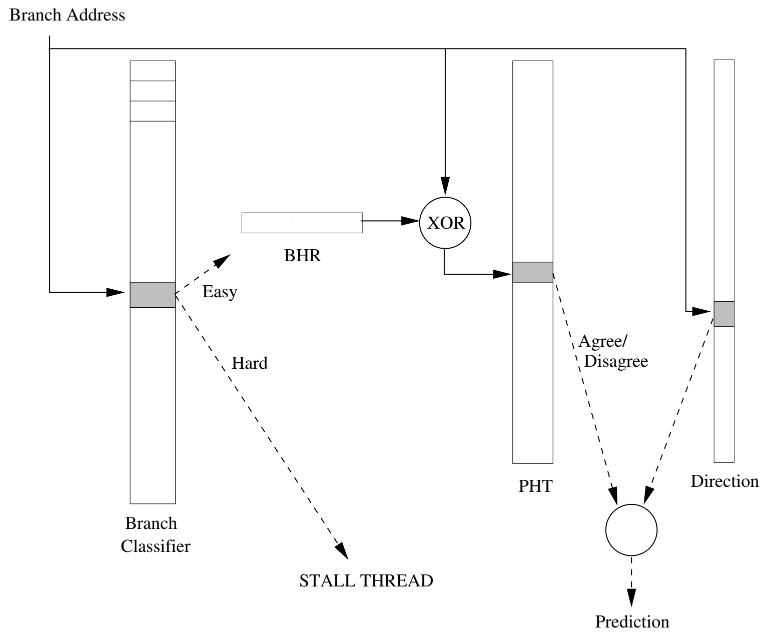


Figure 2. *agstall* **Branch Classification and Instruction Fetch Scheme**

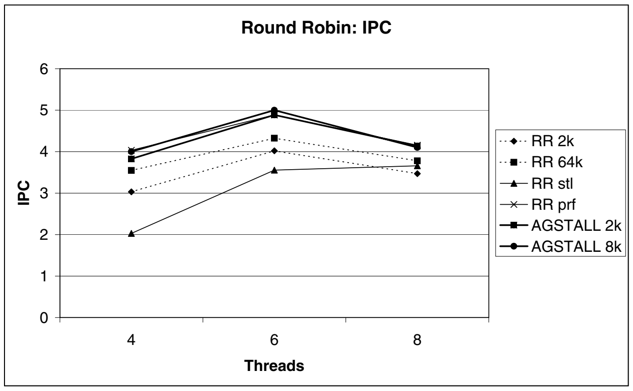
agstall 2k			
	4 threads	6 threads	8 threads
WPF 2k	59.3	62.6	72.5
WPF 64k	23.1	47.2	51.7
WPE 2k	60.2	71.2	73.8
WPE 64k	24.5	54.7	60.2
agstall 8k			
WPF 2k	77.0	86.2	87.9
WPF 64k	56.6	76.8	78.8
WPE 2k	77.4	87.1	88.3
WPE 64k	57.1	79.1	79.5

Table 2. Relative improvement of *agstall* **over Round Robin 2k and 64k: Wrong Path Fetch (WPF) and Wrong Path Execution (WPE) (%).**

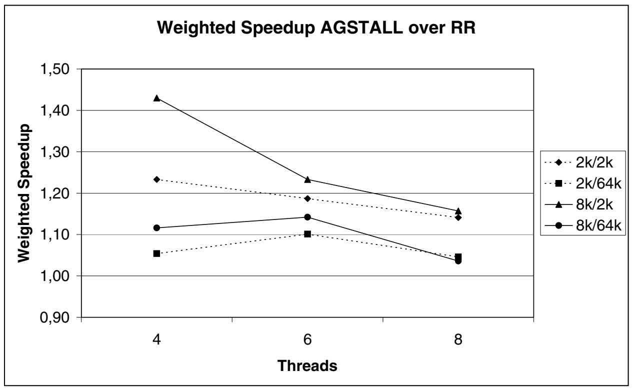
use both a 2k-entry and a 64k-entry gshare branch predictor in the conventional SMT. The resulting architectures are denoted as RR 2k and RR 64k, respectively.

First, in Figure 3(a) we show the performance for all organizations for 4, 6, and 8 threads. We show throughput for Round Robin with perfect branch prediction (RR prf), and stall-on-branch (RR stl). We observe that the IPC is highest for 6 threads, outperforming 8 threads by 20%. This figure can be explained as follows. For 4 threads, there is not enough parallelism available to exploit the processor resources. Hence, when going to 6 threads, IPC increases. When going further to 8 threads, IPC drops again for all organizations, except for stall on branch. The competition has become too high so that even for perfect branch prediction the overall IPC drops. Only for stall on branch there is a slight increase in IPC when going from 6 to 8 threads because of better exploitation of available resources. This can happen because at any given time there are several contexts stalled and hence there is less competition than for the other organizations. We also observe that *agstall* reaches an IPC that is close to the IPC reached by perfect branch prediction. In particular, *agstall* 8k outperforms perfect branch prediction slightly for 6 threads which is the optimal number of threads for this configuration.

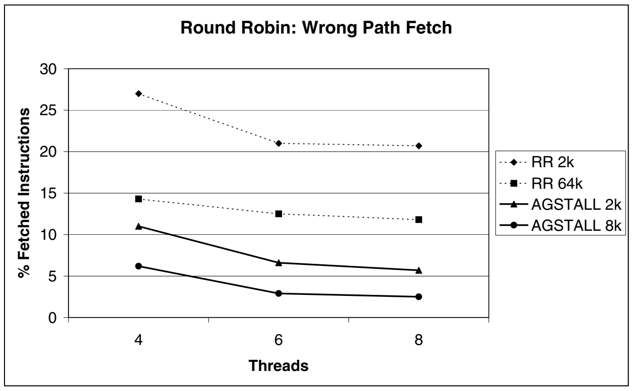
In order to compare our approach to conventional approaches we employ weighted speedup, a metric proposed in [15]. The reason for introducing this metric is because it is difficult to compare two policies in a multithreaded environment by using overall IPC. Any policy that gives priority to high IPC threads will boost overall IPC of a workload, but low IPC threads will not proceed in this policy. However, speedup based on overall IPC is high and this obscures the fact that some threads are not running very well. Therefore, we should look at the speedup of the individual threads in the same workload for two different policies in order to give a fair comparison. For this purpose weighted speedup



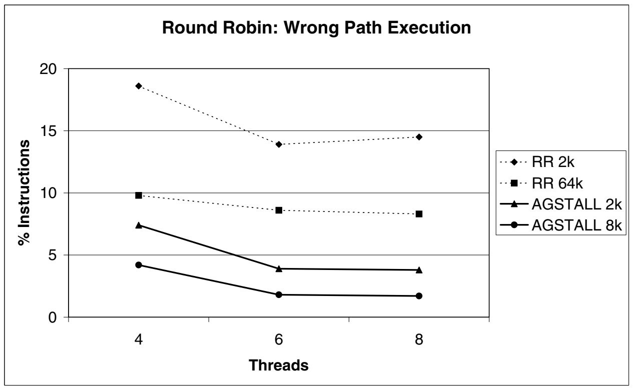
(a) IPC Values



(b) Weighted Speedup



(c) Wrong Path Fetch



(d) Wrong Path Execution

Figure 3. Round Robin instruction fetch

(WSP) is defined as

$$WSP = \frac{1}{|Threads|} \sum_{t \in Threads} \frac{IPC_{new}(t)}{IPC_{old}(t)}$$

With this definition, if the new policy slows down certain threads much compared to the old policy and favors other threads a lot, then this causes speedup terms less than 1 to appear in the summation. Hence the WSP number will be not be artificially inflated. A WSP value of 1 means that all threads run more or less at the same speed in both old and new policies. A WSP value greater than 1 means that several threads are running faster in the new policy and the other threads are not running too much slower. Therefore weighted speedup is a fair comparison of performance and the higher its value the better. In our experiments, all individual speedup values are close to 1.

We have shown the weighted speedup of *agstall* over Round Robin in Figure 3(b). In this figure, the entry 2k/2k denotes the weighted speedup of *agstall* 2k over RR 2k, etc. We see that the weighted speedup of *agstall* with Round Robin over Round Robin alone is quite high. In particular, for the case of 6 threads where the architecture performs best, *agstall* 2k has an 18% and 10% speedup over a 2k-entry or a 64k-entry gshare predictor, respectively. For this situation, *agstall* 8k has 23% and 14% speedup, respectively.

In Figure 3(c) we show the wrong path fetches. We see that *agstall* has significantly less wrong path fetches than the Round Robin fetch mechanisms. In Figure 3(d) we have shown the instructions executed along a wrong path as the percentage of the total number of executed instructions. We see that *agstall* improves Round Robin considerably for all number of threads and for both branch predictors. A larger implementation of *agstall* has less wrong path fetches and executions, since the classifier and the predictor are less polluted by interference. In Table 2 we show relative improvements which increase with the number of threads, for both branch predictors in the baseline configuration.

We conclude that the *agstall* fetch mechanism on top of Round Robin improves on Round Robin with a gshare branch predictor considerably. We improve both on throughput and on wrong path fetch and execution. This last means that *agstall* wastes considerably less energy than a conventional implementation. Since *agstall* 2k is much smaller than 64k-entry gshare, we also reduce the energy

	4 Threads	6 Threads	8 Threads
agstall 2k			
WPF 2k	64.2	72.7	76.6
WPF 64k	31.5	53.1	59.8
WPE 2k	63.8	73.0	74.5
WPE 64k	31.0	55.1	56.9
agstall 8k			
WPF 2k	77.4	85.6	88.0
WPF 64k	56.7	75.2	79.4
WPE 2k	77.5	85.2	86.4
WPE 64k	57.1	75.4	76.9

Table 3. Relative improvement of *agstall* over *icount* 2k and 64k: Wrong Path Fetch (WPF) and Wrong Path Execution (WPE) (%).

spent in the branch predictor itself.

5.2. *icount* instruction fetch

In this section we show the results when we use the *icount* fetch mechanism proposed in [16] that gives priority to threads with least instructions in the frontend of the processor. In order to exploit the best of both worlds, we implement *agstall* on top of *icount*: we stall on branches that are likely to be predicted wrongly and for the threads that are likely to be predicted correctly, we prefer those threads that have few instructions in the pipeline and hence we both balance threads and give preference to threads that can be executed fast.

We again employ a 2k-entry and a 64k-entry gshare branch predictor for *icount*. The resulting architectures are denoted as *icount* 2k and *icount* 64k, respectively. We show the throughput for all configurations for 4, 6, and 8 threads in Figure 4(a). Moreover, we show *icount* with stall-on-branch (*icount* stl) and perfect branch prediction (*icount* prf). We show the weighted speedup of *agstall* over *icount* in Figure 4(b), where 2k/2k denotes the speedup of *agstall* 2k over *icount* 2k, etc. Relative improvement is given in Table 3. We observe that *icount* 64k and both configurations of *agstall* reach almost the same performance. In particular, for 6 threads where the architecture reaches its highest throughput and thus is the preferred number of threads for this configuration, *agstall* 2k improves on *icount* 64k by 1.4%, and *agstall* 8k improves by 2.7%. In both other cases, there is a slight decrease in throughput. Note, however, that *agstall* 2k uses considerably less hardware for this purpose. Both *agstall*s outperform *icount* 2k always.

	4 Threads	6 Threads	8 Threads
agstall 2k	3.14	4.33	5.78
agstall 8k	3.23	4.59	6.17

Table 4. Average number of running contexts.

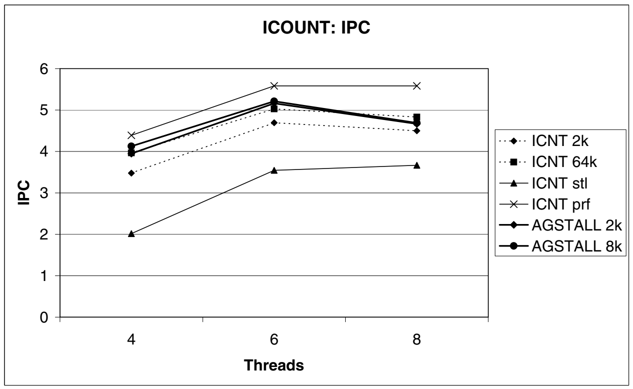
Next, we look at wrong path fetch and wrong path execution in Figures 4(c) and 4(d), respectively. The relative improvements are given in Table 3. We observe that *agstall* on top of *icount* improves on *icount* alone in all cases significantly, up to 86%. Again, improvements in Wrong Path Fetch and Execution increase with for more threads.

We conclude that *agstall* on top of *icount* is an effective way of reducing the energy consumption of an SMT architecture. When *icount* employs a small branch predictor we also improve significantly on throughput. We reach a slightly higher throughput for 6 threads than the large branch predictor and lose only a few percent throughput for 4 or 8 threads. However, in both cases we reduce energy consumption considerably. Moreover, by employing only a small branch prediction structure for *agstall* 2k, we also consume less energy in the branch predictor than *icount* 64k.

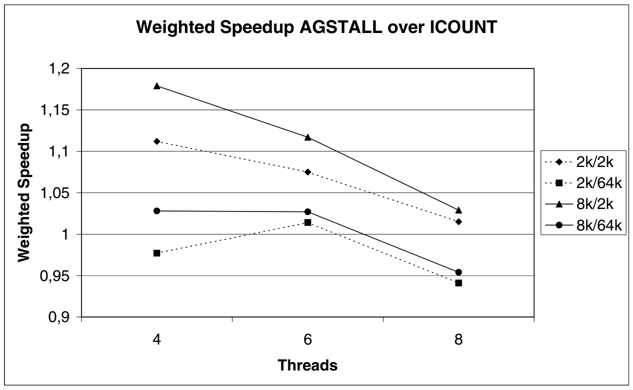
6. Analysis of *agstall*

In this section we discuss a number of aspects of the *agstall* sheme. First, we discuss the amount of stalling that is induced by *agstall*. We use three metrics. In Figure 5(a) we show the percentage of the total number of cycles that contexts are stalled, as a function of number of contexts that are stalled per cycle for *agstall* on top of *icount* for 6 threads. In Figure 5(b) we show the percentage of dynamic branches that are classified as hard per benchmark. In Table 4 we show the average number of contexts alive during the execution of a workload. From these results we see that the large implementation classifies less branches as hard than the small implementation, due to less interference in the classifier. Therefore, the average number of contexts alive at any given time is higher for *agstall* 8k than for *agstall* 2k. This holds in particular for 8 threads where much interference in *agstall* 2k occurs. However, we see in Figure 4(a) that throughput decreases for 8 threads, showing again that increased competition may degrade overall performance.

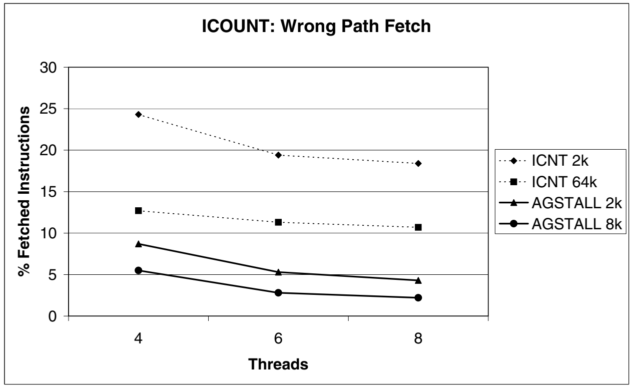
Next, we discuss branch prediction accurracy. In Table 5 we show the hit rate for the easy branches. We see that this hit rate is very high, in particular for *agstall* 8k (99% or higher). We also see that more threads means a slightly lower hit rate, due to interference. This is less pronounced in the large predictor, as expected. This means that the easy



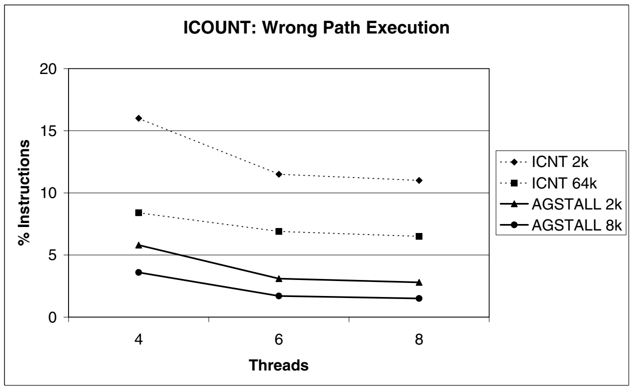
(a) IPC Values



(b) Weighted Speedup



(c) Wrong Path Fetch



(d) Wrong Path Execution

Figure 4. ***icount*** **instruction fetch**

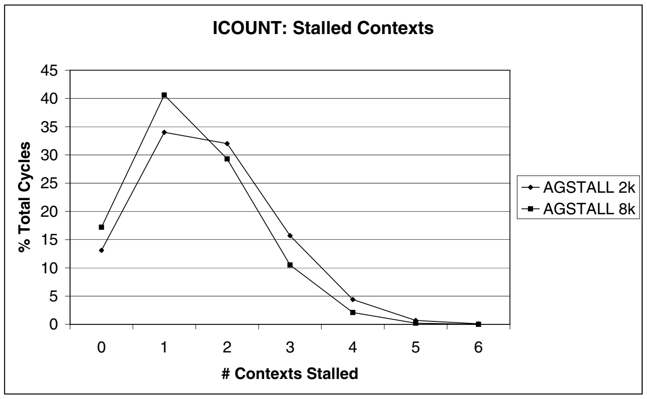
	4 Threads	6 Threads	8 Threads
agstall 2k	96.0%	95.2%	94.7%
agstall 8k	99.2%	99.2%	99.0%

Table 5. Hit rate easy branch prediction.

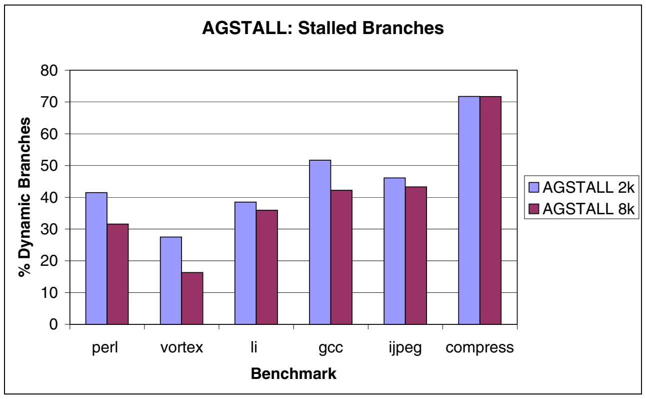
branches are very well predicted and therefore we obtain a low percentage wrong path fetches and executions.

Next, in Figure 6, we show IPCs for the individual benchmarks for the four most important configurations. IPCs are generally higher for *icount* 64k than for *icount* 2k as its prediction accuracy is higher. For 6 threads, *agstall* yields higher IPC for `vortex` and `li` than *icount* does. In these benchmarks there are many easy branches with high prediction accuracy and hence these threads do not stall often. Furthermore, an important routine in `ijpeg` is `idct`, that has a long straightline code segment. Hence, this routine can be serviced at high speed with no stalls and `ijpeg` reaches a higher IPC than for *icount*. `Compress` has a lower IPC, since most of the dynamic branches in it are hard as shown in Figure 5(b). Nevertheless, `compress` does not suffer as much as we would expect based on this large number of hard branches. This is because these branches are often mispredicted in *icount* and hence little gain is harvested by speculation. Note that if we would leave `compress` out of consideration, *agstall* would give much higher IPCs than *icount*. Nevertheless, for `compress` branch prediction and speculation yields higher throughput than stalling on hard branches. `Perl` and `gcc` have almost the same IPC as for *icount*. These benchmarks have a large number of hard branches that are mispredicted often in *icount*, causing waste in the pipeline, and are classified as hard in *agstall*, causing the thread to stall. We also see that IPC values for 6 threads are higher for *agstall* 8k than for *agstall* 2k. However, if we look at the breakdown for 8 threads in Figure 6(b), we see that this is no longer the case. This is because in the 8k implementation more threads are alive at any given time and hence more threads are competing for the resources. The amount of resources in the current configuration is too small to accommodate 8 threads and hence IPCs suffer.

Finally, we analyze another dimension in the design space of *agstall*, namely, how aggressive branches are classified

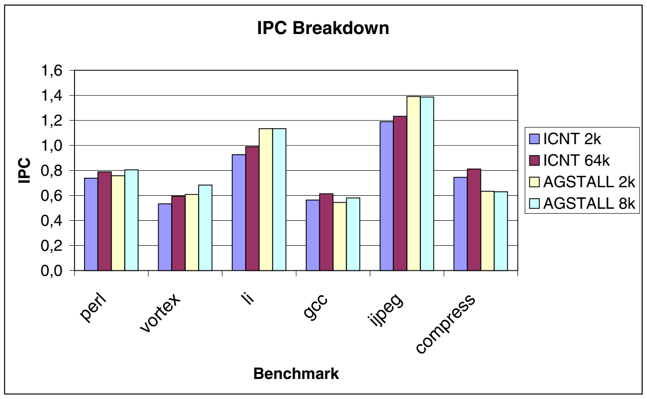


(a) Percentage of time that contexts are stalled

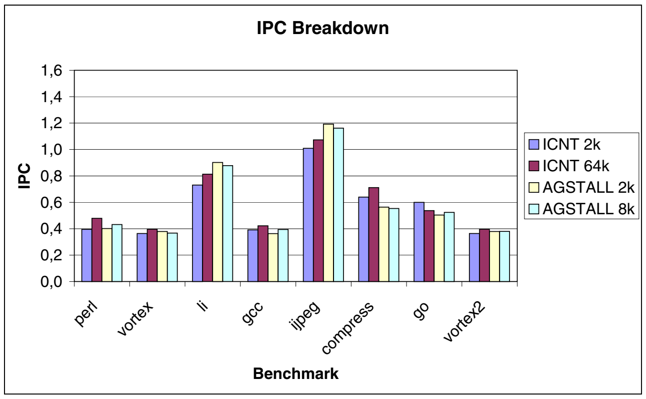


(b) Percentage dynamic branches stalled

Figure 5. Effects of stalling for 6 threads.



(a) 6 threads



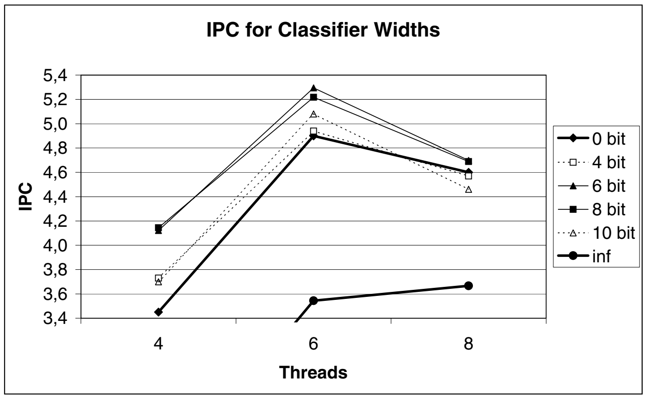
(b) 8 threads

Figure 6. Breakdown of IPC for individual benchmarks.

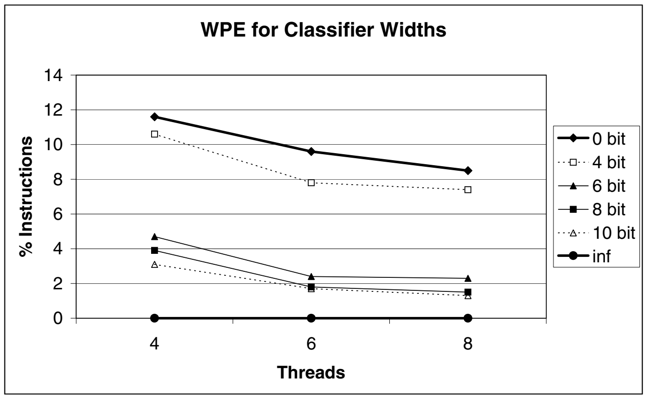
as easy. This corresponds to value of the threshold, or the width of the branch classifier. At the same time, the narrower the classifier, the less expensive it is to build and the less energy it consumes itself. If the width is zero, all branches are classified as easy and they are predicted by the agree predictor. *agstall* on top of *icount* behaves like *icount* employing a 16k-entry agree predictor. For the other limit, if the width is large, all branches are classified as hard, and *agstall* on top of *icount* behaves like *icount* that stalls on branches. We have furthermore implemented classifiers of width 4, 6, 8, and 10 bit. Obviously, the more narrow the classifier, the more branches are classified as easy and the architecture speculates more aggressively. However, increased speculation does not necessarily give more throughput, as can be seen in Figure 7(a). IPC increases with classifier width until it reaches its best value at 6 bit and decreases for wider classifiers. Hence, decreasing the amount of speculation at first increases IPC: there are less threads alive at any given moment and hence less destructive competition. Then, as we decrease speculation further, there is too few parallelism available to fully exploit all resources and IPC drops. In this paper, we use 8 bit that has a slightly lower IPC than 6 bit. On the other hand, wrong path execution drops with increasing classifier width, as can be seen in Figure 7(b). This is to be expected since less speculation implies less wrong paths. In particular, 8 bit has lower wrong path execution than 6 bit. Hence there is a tradeoff between IPC and wrong path fetch and execution. In this paper we have chosen for lower number of wrong path instructions and hence we have implemented an 8 bit wide classifier. Note that IPC values for 6 and 8 bit wide classifiers are substantially better than for *icount* with 16k-entry agree predictor (0 bit) and, at the same time, there is much less waste due to wrong paths.

7. Conclusion

In this paper we have described a novel instruction fetch policy for Simultaneous Multithreaded architectures, called *agstall*, with the goal of freeing the instruction queue of useless wrong path instructions. This saves both energy and



(a) Throughput



(b) Wrong Path Execution

Figure 7. IPC and WPE for different prediction aggression levels.

resources, and can moreover improve performance. This policy is based on branch classification: the processor stalls a thread when a hard branch is encountered and speculates on easy branches. The result is that less contexts are alive at any given moment in time, thereby improving the sharing of resources and diminishing negative competition. We have shown that employing this mechanism on top of a simple Round Robin fetch mechanism, both improves the performance (IPC) to that of perfect branch prediction, and significantly reduces the number of instructions that are executed along a mispredicted path by up to 86%. If we put *agstall* on top of *icount*, we outperform *icount* slightly for 6 threads in which the current configuration performs best. However, mispredicted path execution can be reduced by up to 85%. *agstall* can be implemented with little or no extra cost compared to a conventional branch predictor.

References

[1] P.-Y. Chang, M. Evers, and Y.N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *Proc. PACT*, pages 48–57, 1996.

[2] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y.N. Patt. Branch classification: A new mechanism for improving branch predictor performance. In *Proc. MICRO27*, pages 22–31, 1994.

[3] A.N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proc. PACT*, pages 69–77, 1998.

[4] M. Evers and T.-Y. Yeh. Understanding branches and designing branch predictors for high-performance microprocessors. *Proceedings of the IEEE*, 89(11):1610–1620, 2001.

[5] M. Haungs, P. Sallee, and M. Farrens. Branch transition rate: A new metric for improved branch classification analysis. In *Proc. HPCA*, pages 241–250, 2000.

[6] S. Hily and A. Seznec. Branch prediction and simultaneous multithreading. Technical Report RR-2843, INRIA, 1996.

[7] E. Jacobson, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *Proc. MICRO29*, pages 142–152, 1996.

[8] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor. In *Proc. ICS*, pages 236–245, 2001.

[9] J.L. Lo, S.J. Eggers, J.S. Emer, H.M. Levy, R.L. Stamm, and D.M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997.

[10] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proc. ISCA*, pages 132–141, 1998.

[11] S. McFarling. Combining branch predictors. *WRL Technical Note TN-36*, 1993.

[12] A. Ramirez, J.L. Larriba-Pey, and M. Valero. Branch prediction using profile data. In *Proc. Euro-Par*, pages 386–393, 2001.

[13] J.E. Smith. A study of branch predictor strategies. In *Proc. ISCA*, pages 135–147, 1981.

[14] E. Sprangle, R.S. Chappell, M. Alsup, and Y.N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *Proc. ISCA*, pages 284–291, 1997.

[15] D.M. Tullsen and J.A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proc. MICRO34*, 2001.

[16] 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 implementable simultaneous multithreading processor. In *Proc. ISCA*, pages 191–202, 1996.

[17] T.-Y. Yeh and Y.N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. ISCA*, pages 257–266, 1993.