An Effective Bypass Mechanism to Enhance Branch Predictor for SMT Processors

Yongfeng Pan[1], Xiaoya Fan[1], Liqiang He[2], and Deli Wang[1]

[1] Department of Computer Sciences, Northwestern Polytechnical University, Xi'an, China,
pan_yong_feng@hotmail.com
fanxy@nwpu.edu.cn
wdl900@mail.nwpu.edu.cn
[2] Department of Computer Sciences, University of Cyprus, Cyprus,
liqiang@cs.ucy.ac.cy

Abstract. Unlike traditional superscalar processors, Simultaneous Multithreaded processor can explore both instruction level parallelism and thread level parallelism at the same time. With a same fetch width, SMT fetches instructions from a single thread not so deeply as in traditional superscalar processor. Meanwhile, all the instructions from different threads share the same Function Unites in SMT. All the characteristics make it possible to enhance the performance of SMT through reducing the branch mis-prediction. Based on the fact that about 15% of branch instructions whose directions can be definitely known at predicting cycle, a simple and effective bypass mechanism is proposed. This scheme doesn't depend on any existed branch predictor, and can be used as an effective enhancement to them. Execution-driven simulation results show that the branch prediction miss rates of our predictor decrease by more than 15% on average compared with a simple base line (g-share) predictor and improve the instruction throughput by about 2.5%.

1 Introduction

Simultaneous Multithreaded processors [1, 2] increase the instruction throughput by allowing fetching and running instructions from several threads simultaneously at a single cycle. In SMT processors, functional units that would be idle due to instruction level parallelism (ILP) limitations of a single thread are dynamically filled with useful instructions from other threads, an SMT processor can hide both long latency operations and data dependencies in one thread effectively. These advantages increase both processor utilization and instructions throughput.

With the pipeline deepening and issue widths increasing, the branch predictor plays an important role in improving the performance of an SMT processor [3]. At the same time, according to Matt Ramsay et al. [4], high accuracy of branch predictor is not always needed for a SMT processor because the SMT processors can hide the penalty effectively. So a simple and effective predictor is more suitable for SMT processors.

It is well known that a conditional branch instruction uses the result from previous instructions to make a branch decision. In our experiment, we observe that at most time the distance between the instructions that produce results and a branch instruction which use the results is not too long. In some programs, there are a high percentage of conditional branches whose direction decisions can be definitely known at predicting cycle, and for these branch instructions the predictor should not predict wrongly, and thus the wrong path instruction fetching should also be avoided for them. When this feature is considered in SMT processors, the saved fetch slots can be used to fetch more useful and correct instructions in the threads, and improve the overall performance as a result.

In this paper, we proposed a simple and effective bypass mechanism which exploits the above feature through combining a Writing-Register-Table (WRT) and a base line (g-share) branch predictor into together. It is easy to be implemented, and needs less hardware than many dynamic predictors existed. Compared with our base line, execution-driven simulation results show that the branch prediction miss rates are reduced. Although we use a simple branch predictor as our base line predictor, our scheme doesn't depend on it, and can be used as an effective enhancement to any existed branch predictors.

The paper is organized as follows. First, we introduce the related works about branch predictor. Then, we study the characteristics of branch instruction and their relied registers. and gives our proposed bypass mechanism. In the third part, we give the simulation results and analysis, and finally we conclude our paper.

2 Related works

Till now, many proposals for branch predictors have been put forward. Yeh and Patt [5] show that a two-level branch predictor can achieve high levels of branch prediction accuracy. And S. McFarling [6] proposed g-share branch predictor. To solve the problem of branch interference, Chih-Chieh Lee et al [7], introduced Bi-mode Predictor, Eric Sprangle et al. introduced the Agree Predictor [8]. Additionally, Skewed Branch Predictor [9], the Filter Mechanism [10],YAGS predictor [11] are also introduced in traditional superscalar processors. Moreover, value predictors using the concept of value locality to improve branch predictor [12, 13, 14, 15, 16] are always been use as a component of modern combined predictor.

These years, some new methods are introduced such as Lucian N. Vintan's pre-computed branches [17] which compute the destination of conditional branches as early as the first operand is ready for superscalar processors, Robert S. Chappell's Difficult-path branch prediction using subordinate micro-threads [18], Craig Zilles's Execution-based prediction using speculative slices [19], Daniel A. Jimnez introduced the Neural Branch Prediction [20, 21], Renju Thomas et al studied dynamic dataflow-based identification of correlated branches from a large global history [22], Abhas Kumar et al. evaluated the importance of branches in modern deep pipelined processors [23], and David Tarjan introduced the hashed perceptron predictor, which merges the concepts behind the g-share, path-based and perceptron branch predictors [24].

Though these methods are suitable for superscalar processors, they do not take advantage of the characteristics of SMT processors. In this paper, we proposed a simple and effective bypass scheme for SMT, which try to decrease the wrong path instruction fetching when a branch can be predicted correctly for sure. Our scheme outperforms traditional simple predictors and can be used with any other traditional predictors. Although it is based on the previous work [27], however, there is a big difference between them. Our scheme does not predict all conditional branches; instead, those branches whose operands are not being wrote by in flight instructions will not go through the branch predictor. And in next section we will show this in detail.

3 An effective bypass mechanism to improve branch predictor

In this section we present our basic idea, and give the proposed bypass scheme in detail.

3.1 Basic idea

Traditionally, high performance processors always employ complex predictors and fetch deeply to find more independent instructions to increase ILP. However, in SMT processors, as there are enough instructions can be fetched from different threads, it is not necessary for SMT to fetch as deeply as traditional processors. Besides, the SMT processors can hide the mis-predict penalty effectively, a simple predictor becomes more suitable for it.

Every conditional branch relies on the operands produced by former instructions. If the relied operands have been produced before the branch using (here we assume the operands was not changed), the branch decision can be known or predicted correctly for sure, and thus the wrong path instruction fetching should also be avoided for them. To know the probability of this case, we develop experiment to explore the distance between the last produced operands and the branch who use them. When the distance is long enough, we can use them to make a correct prediction. In this paper, we focus this research on SMT processor, the experiment shows a high percentage (15%) of branches in SMT processors whose distances are long enough to be made a definite correct prediction. Thus the saved fetch slots can be used to fetch more useful and correct instructions in the threads, and improve the overall performance as a result.

To take advantage of this fact, we proposed a simple and effective bypass mechanism in this section. The architecture is shown in next subsection.

3.2 Architecture of our scheme

The architecture of our scheme is shown in Figure 1. It includes three components: a Writing Register Table (WRT), a base line predictor, and an update engine.

The WRT Using Alpha ISA as an example, The WRT have 64 entries that represent 64 physical registers. Each entry includes three fields: the first one is a 9 bits counter to record the number of instructions in flight that will write the register (here we set the maximum number in flight is 512), "0" means there is no instruction in flight will write the register. Whenever there is an instruction will write register, the corresponding counter will increase by one, and when the instruction finishes the counter decreases by one. The second field is a one-bit dirty-or-clean field, "1" means there are some instructions in flight that will write new value the register. The third field is a flag which indicates the flag of the register, that is branch on equal to, not equal to, greater than, greater than or equal to, less than, less than or not equal to zero, and the low bits of the register.

The Base Line Predictor The base line predictor component can be any predictor. Here we use a simple g-share/bi-mod predictor as an example. As an enhancement of branch predictor, our scheme must be used together with an existed branch predictor. When the fetch engine meets a branch instruction whose source register is not clean (the dirty bit in WRT is set), it looks up the base line predictor to get a branch prediction.

The Update Engine The update engine is a bypass-logic, when one instruction is executed and its destination is a register, the result of this instruction will update the flag field and the counter of WRT.

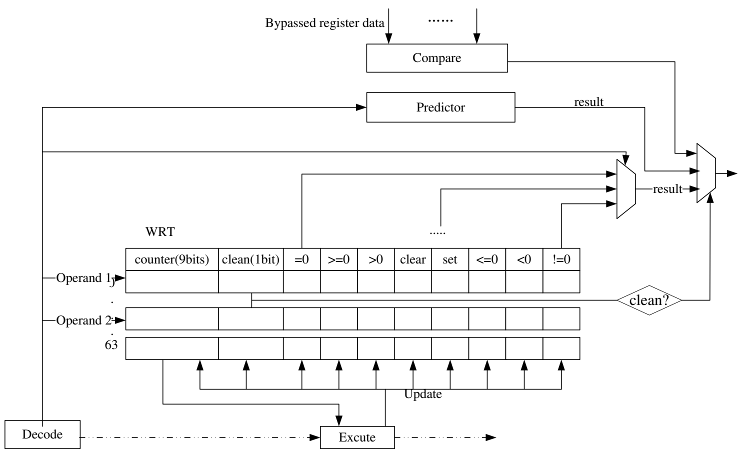

Fig. 1. The scheme of our predictor

3.3 How it works?

To enhance branch predictor using our scheme, we need the detail information (branch or non-branch instruction, source register numbers if it is branch) about instructions. Generally, the information can be known at Decode stage. In Alpha ev6 processor, a line predictor is used to help to determine next fetch block at fetch stage, and is updated by a branch predictor at Decode stage if they are disagree. If a SMT processor has similar architecture as Alpha processor, we can get the information of instruction without problem. But if the branch predictor is accessed in fetch stage, our scheme needs some pre-decode information from other structure, here we propose using pre-decode Icache to provide the needed information.

With the provided instruction details, when the SMT processor meets a conditional branch instruction, it will look up the entry corresponding to the destination-register in the WRT, and check if the dirty-or-clean bit is set. If it is clean (the bit is not set), it means that this register has not been updated by the in flight instructions. In this case, in next cycle, the Instruction Fetch (IF) stage will fetch instructions according to the value (target address) stored in the register. Otherwise the destination of next instruction will be decided by the predictor which can be any traditional predictor. In this paper, we use g-share/bi-mode predictor as an example.

If an instruction in flight needs write a new value to a register which will be used as a referenced register by a branch in the future, the dirty-or-clean bit in the WRT entry corresponding to the register will be set (if it is not set before) and the "counter" will increase until the instruction is finished in the pipeline and then the "counter" decrease and if the "counter" is 0, then the dirty-or-clean bit will be 0.

Additionally, we does not treat mis-predict instructions particularly, that is because, such instructions will also go through the pipeline, and when they finish, we update our scheme just like other instructions.

In super-scalar processors, the fetch depth can be as far as 20 to be able to find enough instruction to issue, and in such situations, there is no need to use our scheme because few conditional instructions whose distance from its relied registers are more than 20.

However, in SMT processors, the fetch depth in a particular running thread is not so deep because of the paralleling running mechanism in it, so there are more chances that conditional branches can get their operands and make decision than in superscalar processors. And these chances give us enough space to improve the accuracy of branch prediction and thus improve the overall performance. In next section, we will give the experiment framework and the simulation results.

3.4 The cost of our scheme

We assume that every thread has its own such bypass structure. Each structure needs 64*18bits and the extra bypass logic. In some architectures, the clean bit has been implemented in many modern superscalar processors. Therefore, the overall cost of such bypass scheme is quite little.

4 Experiment framework and simulation results

In this section, We present our experiment framework and the results.

4.1 Experiment framework

We modify the sim-safe tool of simplescalar3.0 [25] simulator to calculate the distance between the branch instructions and the relied instructions. Firstly, we measure the length both double-operand-branch ISA (PISA) and single-operand-branch ISA (Alpha). Then, based on the fact that double-operand-branch ISA and single-operand-branch have similar results, we do our further experiment on SMTSIM [2] to test our scheme in SMT processor.

The configuration of SMTSIM is given in Table 1. According to [26], the private simple predictors for each thread can achieve higher performance when they have the same cost as shared predictors. In our experiment, to compare with our predict scheme we modify the simulator to let every thread has its own predictor. We implement our scheme in the simulator, and combine the g-share predictor and the WRT into together.

We select some of SPEC2000 benchmarks (8 integers and 5 floats) to test our schemes. To get the different data of the same benchmarks, we test the same benchmark in different threads environment, for instance, we test the data of "gzip" in 1, 2, 4, 6, 8 threads. Consequently, we know that the different clean rate of condition-branch in the same benchmark. From the thirteen benchmarks, we created eight two-thread, four four-thread, two six-thread and one eight-thread work-loads randomly and the combination of these benchmarks with their running instructions are listed in the Table 2 and Table 3. We use ref inputs for these benchmarks and and fast forward 10 billion instructions before starting detailed simulation.

Table 1. Configuration of SMT processor

Parameter	*Value*
Functional Unites	3 FP, 6 integer (including branch), 4 load/store, 2 synchronization
Pipeline	8 stages
Branch miss penalty	6 cycles
Instruction Queue	32-entry FP, 32 entry Int
Inst./Data Cache	64KB/64KB, 2-way, 64 byte
L2/L3 Cache	512KB/4MB, 2-way, 64 byte
I/D TLB, miss penalty	48/128 entry, 160 cycles
Latency (to CPU)	L2 6, L3 18, Mem 80 cycles
Fetch Police	ICOUNT.2.8 [2]
Fetch/Rename/Issue/ Commit Width	8 instructions/cycle

Table 2. Information of Individual benchmarks

Order	*Benchmarks*	*NumberofInstructionsinmillion*	*NumberofBranchesinmillion*
1	Mgrid	281	0.05
2	Crafty	498	42
3	Equake	640	88
4	Gcc	263	31
5	Gzip	614	36
6	Mesa	500	49
7	Art	213	29
8	Ammp	477	11
9	Mcf	543	78
10	Vortex	337	38
11	Bzip2	477	51
12	Twolf	445	59
13	Parser	200	36

Table 3. The combinations of different benchmarks

NumberofThreads	*CombinationsofBenchmarks*	*TotalNumberofInstructionsinmillion*
2	1+5	437
	3+4	744
	1+2	470
	3+5	706
	6+9	690
	7+10	376
	8+11	123
	12+13	382
4	1+2+3+4	743
	6+7+9+10	726
	2+8+5+11	587
	9+11+12+13	712
6	1+2+3+4+5+6	725
	7+8+9+10+11+12	685
8	5+6+7+9+10+11+12+13	703

One assumption needs to be pointed out. When we had more than one result of the same benchmarks in the same thread work-loads, we use the average data of them to make it more accurate.

4.2 Simulation Results

In our experiments, we first measured the distance which is the basic of our bypass scheme, when the branch instruction has one operand (Alpha), we record the distance between the instruction produce this operand and the branch use this operand. When the branch instruction has two operands (PISA), we record the distance between the instruction produce the last operand and the branch. And the results are illustrated in Figure 2 and Figure 3.

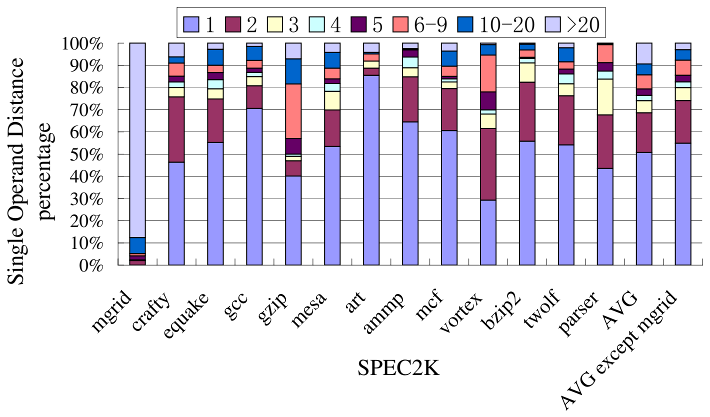

Fig. 2. The distance between a conditional branch and its relied instruction of Alpha

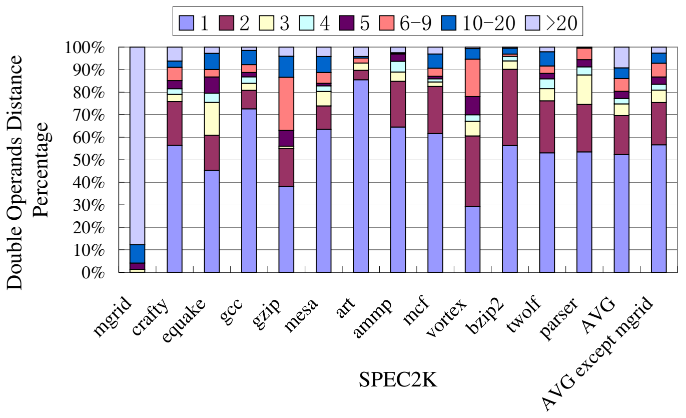

Fig. 3. The distance between a conditional branch and its relied instruction of PISA

Both of the figures (Figure 2 and Figure 3) have similar results. That is because (1) both Alpha and PISA ISA are RISC, their branch destinations are determined by one or two registers, and (2) the distance has a close relationship with the characteristic of Benchmarks instead of the instruction sets. Therefore, to make things simple, in the rest of this paper, we focus on the study of Alpha ISA and conduct our other experiments on SMTSIM simulator.

In the Figure 2 and Figure 3 we can find that the distance of about 15% of branch instruction and the instruction writing the register which will be used by this branch is more than 6. To illustrate this point more clearly, let us look a fragment of a program:

A. LDL R1, [R2+100]

B. MUL R1, R1, R3

C. MOV R3, R2

D. BEQ R1, Label

Instruction B writes the register R1, and the branch instruction D use the sign of R1 to determine whether it will go to Label. The distance mentioned above stand for the number of instructions between B and D. In this example, the distance is the 2.

In a SMT processor, all threads use the same FU; there are many chances that some instructions from other threads fit in the slots between instruction B and D, so the distance of B and D will be more than 2 in this example.

Therefore, in a fetch width is 8 instruction per cycle and 4 threads environments, if other threads can provide one independent instruction to insert into the branch and its relied instruction, then when the distance is more than 5 the destination of this branch can be sure before it is predict. Similarly, when there are 8 running threads, when the distance is more than 2 then this branch is sure.

To get the information precisely, we modify the SMTSIM simulator to record the clean rate in different thread work-loads. When a branch instruction in the decode stage of pipeline, and if the clean bit in WRT is 0, then this branch instruction is called clean branch. The clean rate means the percentage of clean branch instructions in the total branches. We use ICOUNT 2.8 [2] fetch police and the results are illustrated in Figure 4.

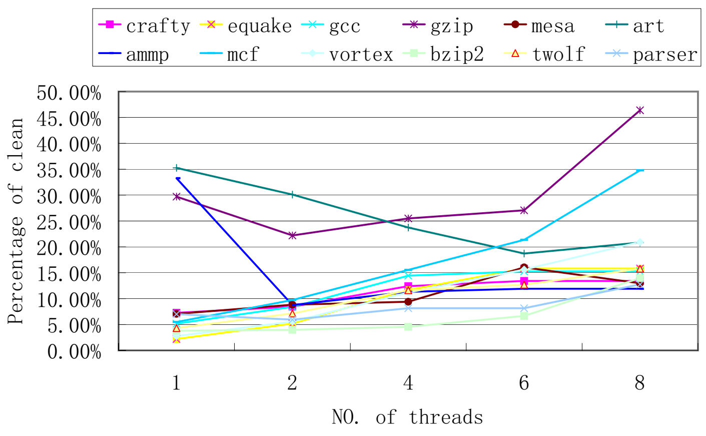

Fig. 4. The relation between clean rate and number of threads

As the increasing of the number of threads, the percentage of clean branches becomes higher and higher. That is because the in the SMT processor, some instructions from different threads which do not affect the operands of the current branch are inserted into the issue queue. So the distance between the branch and its relied instructions is enlarged. And there are more chances that the operands are ready when the branch needs to use them. And the Figure 4 shows that the percentage is quite optimistic.

There is an exception program, mgrid. Because there are no enough branches in this program, we get a very positive result (Figure 5). However, to be more accuracy, we elimination this program and the average result is illustrated in Figure 6.

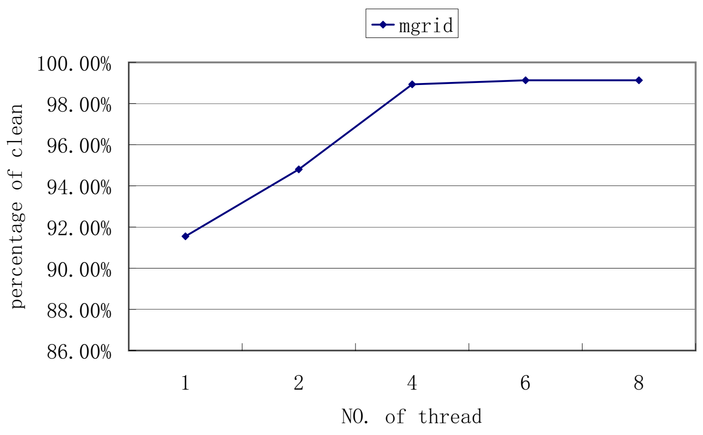

Fig. 5. The relation between clean rate and number of threads of program "mgrid"

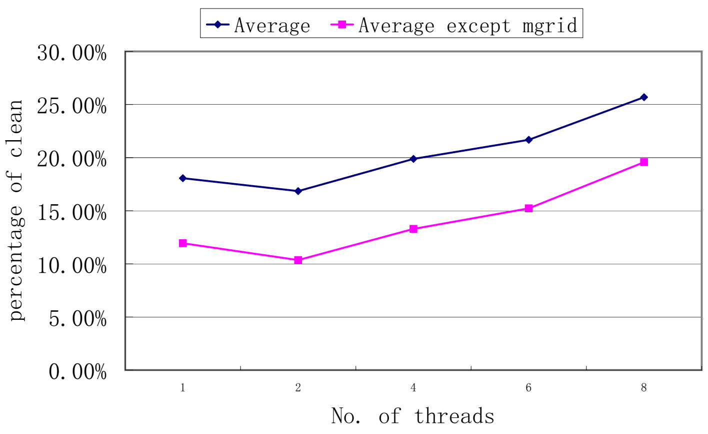

Fig. 6. The relation between clean rate and number of threads on average

Figure 6 shows that when there are 2 running threads, the clean rate is lowest, which because the clean rates of some float benchmarks decline significantly. And when the number of running threads is more than 2, the clean rates higher and higher as the increasing of thread number. Particularly, when there are eight threads, the clean rate is 19.57%. Considerably, if we eliminate such branch instruction from mis-prediction, then the rate can be higher. For instance, when we use a predictor which correct rate is 90%, by using this scheme in Figure 1, we can elevate the correct rate of prediction to 91.96% (19.57%*(1-90%)+90%).

4.3 Branch prediction miss rate

We implement our scheme with g-share and bi-mode and called g-share WRT and bi-mode WRT respectively. And we compare the mis-prediction rate among 4K g-share, 4K g-share with WRT, and 4.5K bi-mode, and the results are as illustrated in Figure 7.

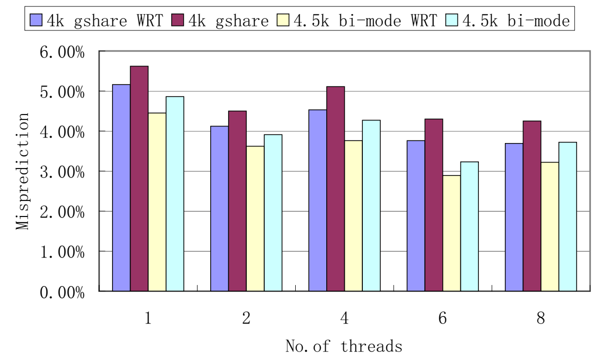

Fig. 7. Branch prediction miss rate

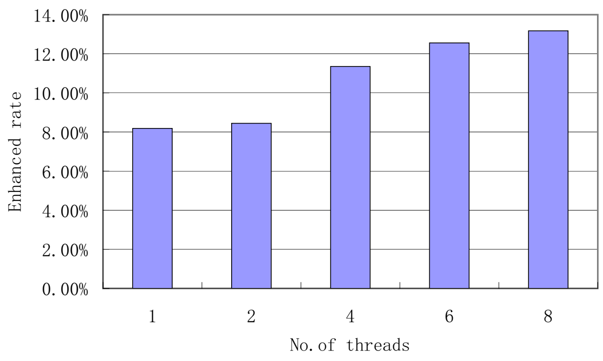

Fig. 8. Enhanced rate of prediction

As mentioned before, WRT is not a predictor. It can be used as a complementary of any predictor to improve the predict accuracy. In the Figure 7, we know that our scheme can improve the original predictor effectively no matter what the predictor is.

In Figure 8, we show the relationship between the number of threads and the enhancement of prediction. It is clear that the more running threads, the more benefit we can get from our bypass scheme. Furthermore, when the threads number increases from 2 to 4, the enhancement is higher than others. And this illustrates that when there are only 2 running threads, the independent instructions are insufficient to insert between the conditional branch and the relied instruction. And when there are 4 or 6 threads, the independent instructions from other threads are enough to show the efficiency of our scheme. Noticeably, when there are 8 threads, although the enhancement is higher than 6 threads, the mis-prediction rates are also slightly higher than 6 threads. This shows that too many threads become helpless to further reduce the mis-prediction rates.

4.4 Wrong path instruction fetch rate

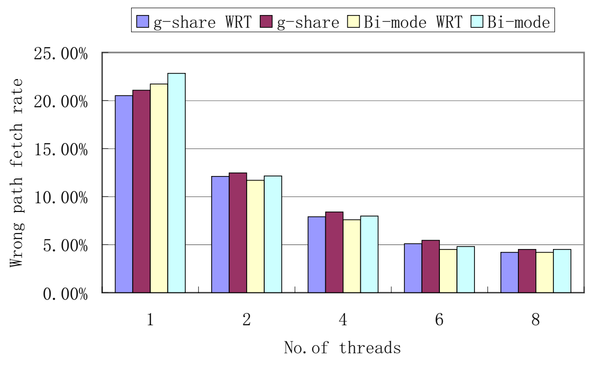

Fig. 9. Wrong Path instruction fetch rates of the predictors on average

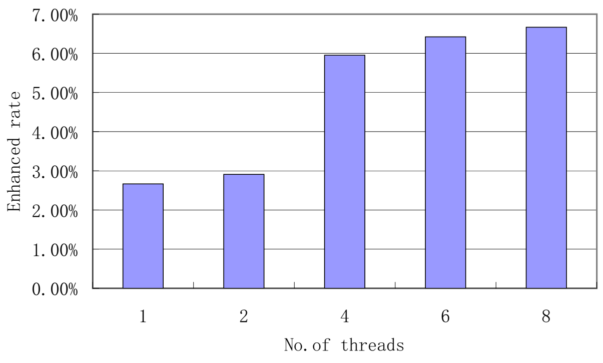

Fig. 10. Enhanced rate of wrong path fetch

SMT processor can reduce the wrong path fetch rate effectively; in addition, our scheme can enhance this rate by 6% in 4 threads and 6.7% in 8 threads situations.

Similarly, there is a milestone in both Figure 8 and Figure 10. In the 4 threads situation, the improvement is most obvious. And the further increase of thread number has little influence on the enhancement. This hints us that we can obtain higher benefit by increase 2 threads to 4.

4.5 Processor Performance

Although the SMT can tolerance the degradation of performance caused by mis-prediction, our scheme still improve the overall performance by 2.55% on average. Figure 11 shows the instruction throughput of our scheme compared with the original g-share predictor.

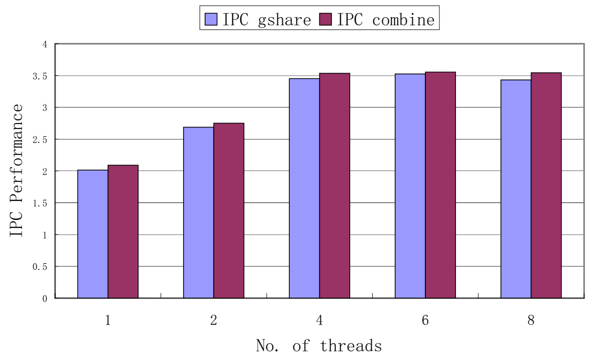

Fig. 11. Instruction throughput of our scheme and g-share predictor

5 Conclusion

In this paper, we study the distance of conditional branches and its relied instructions, and present a bypass mechanism for SMT processors. With the help of WRT, the accurate of original prediction can be improved by 15% percent on average. The implementation of our predictor is simple, and the hardware cost is little. Execution-driven simulation results show that our predictor can be more effective as the increasing of the number of threads.

6 Future Works

As the fetch policy become the bottleneck of IPC performance, we will do research on the utilizing some unique characteristics of SMT to enhance the performance overall.

7 Acknowledgement

This work was supported by National Natural Science Foundation of China (NSFC) Grant No.60573107.

References

1. D. M. Tullsen, S. J. Eggers, H. M. Levy et al, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in Proc.22nd Annual International Symposium on Computer Architecture, June 1995.
2. D. M. Tullsen, S. J. Eggers, H. M. Levy et al, "Exploiting Choice: Instruction Fetch and Issure on an Implementable Simultaneous Multithreading Processor," In Proc. 23rd Annual International Symposium on Computer Architecture, June 1996.
3. Steven Swanson, Luke K. McDowell, Michael M. Swift, "An evaluation of speculative instruction execution on simultaneous multithreaded processors," ACM Transactions on Computer Systems (TOCS) archive Volume 21 , Issue 3 (August 2003) Pages: 314 - 340, 2003
4. Ramsay M, Feucht Ch, Lipasti M H, "Exploring Efficient SMT Branch Predictor DesignUniversity of Wisconsin-Madsin," Department of Electrical and Computer Engineering, 2003.
5. T-Y.Yeh and Y. N. Patt, "Two Level Adaptive Training Branch Prediction", 24th ACM/IEEE International Symposium on Micro architecture, (November 1991), pp.51-61.
6. S. McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.
7. Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N.Mudge, "The Bi-Mode Branch Predictor," In Proc. MICRO 30, Dec. 1997.
8. E. Sprangle, R. Chappell, M. Alsup, and Y. Patt, "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference," In Proc. 24th Ann. Int. Symp. on Computer Architecture, May 1997.
9. P. Michaud, A. Seznec, and R. Uhlig, "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors," In Proc. 24th Ann. Int. Symp. on Computer Architecture, May 1997.
10. P.-Y. Chang, M. Evers, and Y. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," In Proceeding of International Conference on Parallel Architectures and Compilation Techniques, Oct. 1996.
11. A. N. Eden and T. Mudge, "The YAGS Branch Prediction Scheme," In Proceedings of the 31st Annual International Symposium on Micro-architecture, Dec 1998.
12. Timothy H. Heil, Zak Smith, J. E. Smith, "Improving Branch Predictors by Correlating on Data Values," In Proceedings of the 25th ISCA, June 1998.
13. Nathan Tuck D. M. Tullsen, "Multithreaded Value Prediction," In Proc. of the 11th International Symposirm on High Performance Computer Architecture, Feb. 2005.
14. Brad Calder, Glenn Reinman, D. M. Tullsen, "Selective Value Prediction," In Proc. of the 26th International Symposium on Computer Architecture, May 1999.
15. M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and data speculation," in Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems. pp.138-147, Oct. 1996.
16. Y. Sazeides and J. E. Smith, "The predictability of data values," in Proceedings of the 30th Annual ACM/IEEE International Symposium on Micro-architecture, December 1997.
17. Lucian N. Vintan, Marius Sbera, Ioan Z. Mihu, Adrian Florea, "An alternative to branch prediction: pre-computed branches," ACM SIGARCH Computer Architecture News archive Volume 31 Pages: 20 - 29 2003.
18. Robert S. Chappell, Francis Tseng, "Difficult-path branch prediction using subordinate micro-threads," In International Conference on Computer Architecture Proceedings of the 29th annual international symposium on Computer architecture, 2003.
19. Craig Zilles, Gurindar Sohi, "Execution-based prediction using speculative slices," In International Conference on Computer Architecture Proceedings of the 28th annual international symposium on Computer architecture, 2001.

20. Daniel A. Jiménez, "Fast Path-Based Neural Branch Prediction", In International Symposium on Micro-architecture, 2003.
21. Daniel A. Jiménez, Calvin Lin, "Neural methods for dynamic branch prediction", ACM Transactions on Computer Systems, Volume 20, November 2002.
22. Renju Thomas et al, "Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history," International Conference on Computer Architecture, Pages: 314 - 323, 2003.
23. Steven Swanson, et al, "An evaluation of speculative instruction execution on simultaneous multithreaded processors," ACM Transactions on Computer Systems, Volume 21, 2003.
24. David Tarjan, Kevin Skadron, "Merging path and g-share indexing in perceptron branch prediction," ACM Transactions on Computer Systems, Volume 2, September 2005.
25. Austin T, Larson E, Ernst D. "SimpleScalar: An infrastructure for computer system modeling," IEEE Computer200235(2)59-67
26. Ren Jian, "Dynamic Branch Predictor Evaluation on Simultaneous Multithreading Processor," Computer Science China, 2006.
27. Liqiang He and Zhiyong Liu, "A new value based branch predictor for SMT processors," In Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems, 2004