A NEW VALUE BASED BRANCH PREDICTOR FOR SMT PROCESSORS

LIQIANG HE and ZHIYONG LIU
Institute of Computing Technology, Chinese Academy of Sciences
P.O.BOX 2704-25, Beijing, P.R.CHINA 100080
email: hlq@ict.ac.cn and zliu@nsfc.gov.cn

ABSTRACT
Simultaneous Multithreaded (SMT) processors improve the instruction throughput by allowing fetching and running instructions from several threads simultaneously at a single cycle. With the pipeline deepening and issue widths increasing, the branch predictor plays a more important role in improving the performance of an SMT processor. Although the existing branch predictors have presented a high prediction accuracy, the complexity of their implementation and the hardware overhead of them are very large for SMT processors. In this paper, we present a simple and effective value based branch predictor for SMT processors. The predictor is easy to be implemented, and needs less hardware than that of all the other dynamic predictors we consider. Execution-driven simulation results show that our predictor outperforms many predictors in literature. The performance speedup of our predictor over the best predictor considered is 13% on average. The branch prediction miss rates and the instruction rates fetched along the wrong path are decreased.

KEY WORDS
SMT, branch predictor, value prediction

1 Introduction

Simultaneous Multithreaded (SMT) processors [1, 2] improve 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 running threads. By allowing fetching instructions from other threads, an SMT processor can hide both long latency operations and data dependencies in a thread. These advantages increase both processor utilization and instructions throughput.

With the pipeline deepening and issue widths increasing, the branch predictor plays a more important role in improving the performance of an SMT processor. Although the traditional branch predictors have presented a high prediction accuracy, the complexity of their implementation and the hardware overhead of them are very large for SMT processors.

In this paper, we present a simple and effective value based branch predictor for SMT processors. The rationale behind our scheme is that the outcome of a branch instruction is determined by and only by its referenced values. Although the branch outcome shows some locality in branch history which we used in traditional predictors, it has no relationship with its history in fact when it is predicted at a special fetch pipeline stage. Locality of the branch outcome does not come from the branch itself but its referenced values of such as the values of registers or that of memory locations, and the locality of values is also the base of the success of value predictor. Our new branch predictor exploits the value locality of the branch instruction referenced (architecture registers in this paper, not the memory locations for the load instruction always puts the value into a register before the value can be used in today RISC based processors), and predicts the outcome of the branch with a higher accuracy.

The implementation of our predictor is simple, and the hardware needed is less than that of all the dynamic predictors we consider. Execution-driven simulation results show that our predictor outperforms most predictors in literature. The performance speedup of our predictor over the best predictor considered is 13% on average. The branch prediction miss rates and the instruction rates fetched along the wrong path are decreased.

This paper is organized as follows. Section 2 reviews the related works. Section 3 describes our value based branch predictor, and discusses two related issues about it. Section 4 and 5 describe the methodology and analyze the simulation results. Finally, section 6 concludes the paper.

2 Related Work

Accurate branch prediction is crucial for obtaining high levels of ILP in conventional superscalar and SMT processors. Till now, many proposals for branch predictors have been put forward, ranging from simple static prediction schemes to sophisticated dynamic predictors. With dynamic predictor, Yeh and Patt have shown that a two-level branch predictor can achieve high levels of branch prediction accuracy [10]. Some techniques have been proposed to solve the problem of branch interference, which is caused by two unrelated branches mapping to the same entry of Pattern History Table (PHT) through the predictor's indexing function in the two-level predictor. One technique is to change the index functions. *gshare* branch predictor proposed by McFarling [8] belongs to this class. It XORs the

Branch History Register (BHR) with the lower bits of the branch address to generate the index into the PHT. Since BHR histories are not uniformly distributed, the introduction of address bits into the index gives a more useful distribution across all PHT entries. *gshare* predictor is generally considered as the most effective "aliased" single component branch predictor.

In addition, other index functions may also be used. On Alpha 21264 [9], the branch predictor is composed of three units: the local, global, and choice predictors. The local predictor uses a two-level table that holds the history of individual branches. The global predictor is indexed by a global history of all recent branches. The global predictor correlates the local history of the current branch with all recent branches. The choice predictor monitors the history of the local and global predictors and chooses the best of the two predictors for a particular branch.

"De-aliased" global history predictors -the *skewed* branch predictor, the *bimodal* predictor and the *agree* predictor- belong to another technique to address the branch interference. The *agree* predictor [6] records a direction (taken or not taken) of a branch, and the predictor itself is interpreted as 'agree with this direction' or not. In this way, destructive interference in the PHT can be avoided since an 'agree' value in the PHT for two distinct branches may mean 'taken' for one and 'not taken' for the other. *2Bc-gskew* [7] is a "de-aliased" hybrid predictor which combines *e-gskew* [11] and a *bimodal* [12] branch predictor. It consists in four identical predictor-table banks, that is the three banks from the *e-gskew* plus a meta-predictor. The *e-gskew* predictor can effectively remove conflict aliasing impact, and the *bimodal* predictor can decrease the number of destructive aliasing instances and replace them by harmless aliasing. As a hybrid predictor, *2Bc-gskew* may deliver a higher prediction accuracy than a single component predictor.

Besides the traditional branch predictors, there is a new class of predictors which are built on neural networks in recent years [13, 14]. Although their implementation complexities are still high, they show a new promising way in the branch prediction research area.

3 A Value Based Branch Predictor

In this section, we will present our Value Based Branch Predictor (VBBP), and discuss two related issues about it.

3.1 Overview

The particular implementation of a value based branch predictor depends on the instruction set architecture (ISA). The implementation presented in this work is based on the ISA of the Alpha architecture [9]. Conditional branches of the ISA have only one source register operand, and this operand is used by the predictor to predict the outcome of a branch. The outcomes of unconditional branches are always taken and need not to be predicted. Special modifications would be needed to extend our prediction mechanism to different ISAs which we do not discuss in the this paper.

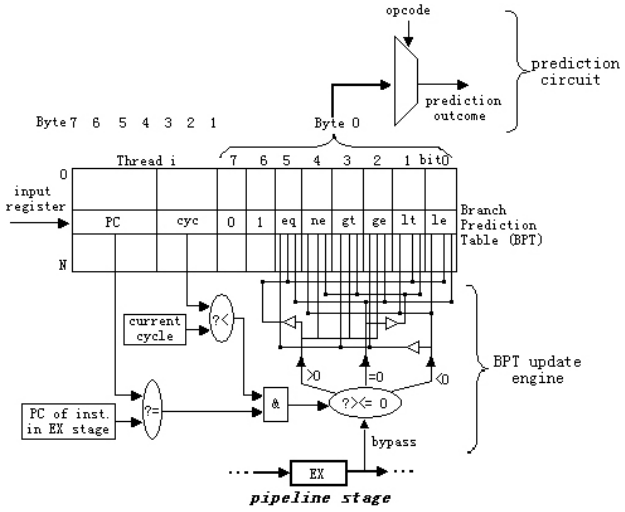

Figure 1. Value based branch predictor for SMT processors.

The architecture of our VBBP predictor is shown in Figure 1. It includes three components: Branch Prediction Table (BPT), prediction circuit (above the BPT in Fig.1), and BPT update engine (under the BPT in Fig.1).

BPT is indexed by the source logical register identifier of the branch instructions of a thread. In SMT processors, there are several threads running simultaneously at a single cycle. BPT has a group of entries for every thread. In Fig.1, it only depicts the entries for thread *i*. In addition, for the Alpha ISA, there are thirty-two integer registers and thirty-two float-point registers, such that the total number of entries in BPT is sixty-four multiplied by the number of running threads.

Each entry of the BPT has eight bytes, and includes three fields:

- **Prediction outcome information:** the lowest byte (byte 0) in Fig.1, every bit of it is a flag used by a class of branch instructions from the left to the right: unused (or reserved), unconditional branch, branch on equal to, not equal to, greater than, greater than or equal to, less than, and less than or equal to zero. The highest two bits are not modified by any operations, and the other bits are set if the corresponding condition is true, otherwise are cleared.

- **Cyc:** byte one to three in Fig.1. It is a saturating counter that records the relative fetch cycle of the latest instruction that had that logical register as destination belonging to thread *i*. This information is used by the BPT update engine to avoid using an outdated value to modify the flags of prediction outcome information.

- **PC:** The highest four bytes which store the PC of the latest instruction of thread *i* that had that logical register as destination. It is also used by BPT update engine when it is modifying the prediction flags.

When a branch instruction of a running thread is fetched and decoded, the opcode and the source register identifier of it are sent to the prediction circuit. The opcode is used to select a bit flag from the prediction outcome information byte, and the register identifier is used to index a corresponding entry of the BPT belonging to the thread. The output flag is the predicted outcome of the branch instruction, and the control-flow can speculatively execute along the predicted path. This operation can be completed in a cycle.

The information of BPT is maintained by the BPT update engine. In addition, it needs the fetching and decoding components of a processor to help recording the PC and the decoding cycle of the latest instruction which uses a register as its operation destination.

The BPT update engine includes a zero determining logic which determines whether the input value is equal to, larger than, or less than zero. The input of the logic is a bypassing value from the EX (execute) pipeline stage of an SMT processor which will modify the value of the corresponding register belonging to a thread. For the instructions that do not modify any value of registers, the BPT update engine does nothing at that cycle.

The bit flags in prediction outcome information byte are updated by the BPT update engine according to the output of the zero determining logic. Before making decision by the logic, the update engine checks if the *PC* of the instruction in the EX stage is equal to that in the corresponding entry of BPT, and if the value of current cycle is larger than the value of *Cyc* in the entry plus two (the shortest latency of an instruction). If the two conditions are held, the engine will modify the flags, otherwise it does nothing.

Using the *Cyc* as an updating condition is to avoid modifying the flags in the cases that a new instruction enters the pipeline, whose destination register is the same as that of the instruction in the EX stage, in that cases the outcome of a branch will depend on the new instruction but not the old one in the EX stage.

The *PC* of the instruction of a thread in EX stage used by the update engine can be obtained from the reorder-buffers. The *Cyc* is the value of a counter which records the relative cycles from the beginning of the program execution.

Figure 1 only presents the branch prediction mechanism (taken/not taken) of our predictor. To be executed successfully for a branch instruction, a BTB (Branch Target Buffer) table and a RAS (Return Address Stack) table similar as that in the traditional branch predictors are also needed for our predictor that we do not discuss in this paper.

3.2 Recovering from a misprediction

With our VBBP predictor, if a branch misprediction occurs, the processor flushes the instructions in the pipeline along the wrong path and reexecutes from the right path, and the predictor itself will update the flags of prediction outcome information according to the result of the branch. This similar to the existing predictors. This part of circuit does not shown in Fig.1.

3.3 Why it works-the theory

A conditional branch is to make a branch decision (taken/not taken) according to the values of it referenced.

To be said in detail, the success of our predictor is based on the analysis of the action of branch prediction and the ISA that we consider. Due to only one operand in the conditional branch instruction, the predictor can use a simple zero determining logic to predict the outcome of a branch. In addition, as described in section 1, the outcome of a particular branch is determined by and only by the values it referenced (a register or a memory location) at that cycle, not the history of itself. Although the existing branch predictors are successful when using the branch history to predict the future outcome of the branch, they exploit the value locality of the branch referenced in fact, not the history locality of the branch itself which has been seen the core idea of the existing predictors. In our value based branch predictor, it tries its best to keep the newest value information through bypassing from the pipeline, and uses the information to predict the outcomes of all the branch instructions that refer to the value. With the timely updating mechanism, our predictor obtains a high prediction accuracy.

It does have some similarities between the organization of BPT in our predictor and the condition code bits stored in the program status word of some ISAs such as SPARC and PowerPC [17], but the differences between them are also significant. First, there is one copy of the flags for each logical register in our predictor, whereas the condition code bits are only one copy or several copies but not indexed by the logical registers. Second, the condition code bits are set only by the correspondent *setting* instructions that always be in the last place before the branch instruction. When the branch instructions are predicted, the *setting* instructions can not set the bits timely for the prediction. However, the BPT in our predictor can be set by any instructions that modify the content of registers, such that the prediction flags contain the information timely enough to predict the outcome of a coming branch.

3.4 Implementation of our predictor

The basic idea of our predictor seems similar to a last value prediction based branch predictor in the superscalar out-of-order processors [15, 16], but the difference between them are as the following. First, we do not keep the predicted

value in a structure as that in a value predictor. It only uses the value to set the branch flags. Second, the predictor sets the branch flags without considering whether there are branch instructions that use the flags. Although it seems to be blindfold, it does save time when a branch prediction needs to access the flags.

The prediction of a particular branch can be completed in a cycle. In a processor with high frequency, the updating of prediction flags by the BPT update engine can be pipelined into two cycles: zero determining is done in the first cycle, and the flags are modified in the second cycle. In some cases, flag modification and branch prediction need to be done in a cycle. If the two operations accessed different entries of the BPT, they can be done in parallel. Otherwise, the flag modification operation should be executed after the branch prediction operation.

The budget of our predictor is less than that of all the dynamic predictors we consider. The entries in BPT of every thread needs only 512 bytes (eight bytes per entry multiplied by sixty-four entries). The other hardware include a *Cyc* counter, the selector used by the predicting circuit, and the zero determining logic in the BPT update engine.

4 Methodology

In this study, we focus on the heterogeneous multitasking mode of SMT processor. We modify the SMT simulator (SMTSIM) [3] to implement our value based branch predictor and gather detailed statistics of the experiments. We then compare the results of our predictor with that of four other existing branch predictors (*gshare*, *agree*, *2bcgskew*, and predictor in Alpha 21264). This execution-driven simulator emulates unaltered Alpha executables, and models all typical sources of processor latency and conflict. The major simulator parameters are given in Table 1.

Table 1. Simulator parameters.

Parameter	Value
Functional Units	3 FP,6 Int(4 LD/ST)
Pipeline depth	9 stages
Instruction Queue	32-entry FP,32-entry Int
Latency	Based on Alpha 21264
Inst./Data Cache	64KB/64KB,2-way,64 byte
L2/L3 Cache	512KB/4MB,2-way,64 byte
I/DTLB,miss pen	48/128 entry,160 cycles
Latency(to CPU)	L2 6,L3 18,Mem 80 cyc
branch miss pen	7 cycles

Our workload consists of eight integer and ten floating point programs from the SPEC CPU 2000 benchmark suite [4]. We compiled each program with GCC with the -O4 optimization and produced statically linked executables. Each program runs with the reference (expect for *bzip2*, which uses the train) input set. From these eighteen benchmarks, we created nine two-thread, and three four-thread workloads. The running instructions each of the program in the workloads are listed in Table 3 according to the method in [5]. All the combinations are otherwise arbitrary. We run the workload in the simulator with each predictor listed in Table 2. In all the simulators, predictors are shared by all the threads running simultaneously.

Table 2. Parameters of branch predictor.

Predictor	Entries
gshare	pht:2K
21264 Based	lht,lpt:8K, gpt,cpt:16K
agree	pht:2K
2Bc-gskew	meta,g0,g1:64K, bimodal:16K
VBBP	$64 \times number\_of\_thread$

Table 3. Number of running instructions of each program in every workload.

No.	Workload	Int (billion)
1	art,perlbmk	0.3
2	crafty,mcf	1.5
3	equake,mesa	3
4	mgrid,ammp	4
5	bzip2,lucas	5.3
6	parser,twolf	5.3
7	applu,sixtrack	3
8	gcc,facerec	0.5
9	gzip,swim	2
10	art,perlbmk,crafty,mcf	1.5
11	applu,sixtrack,gzip,swim	3
12	bzip2,lucas,parser,twolf	5.3

5 Simulation Results

This section presents the simulation results of our experiments. It compares the performance of different branch predictors with our value based branch predictor in SMT processors, and also presents the branch prediction miss rates and the wrong path instruction fetch rates in all experiments. It uses the serial numbers listed in Table 3 to denote workloads in the figures of this section.

5.1 Processor performance

Figure 2 shows the instruction throughput of our branch predictor and the other predictors considered in the experiments.

At most cases, the performance of our VBBP predictor is better than that of other predictors. The predictor whose performance is closest to our predictor is that in Alpha 21264. Over the predictor of Alpha 21264, the best speedup of our predictor is obtained for the *equake-mesa* workload, which is up to 84.5%, and the average speedup is 13%. Table 4 lists the speedups of our predictor over the other predictors considered.

Table 4. Speedups of VBBP predictor over other predictors of our considered.

gshare	agree	2Bcgskew	21264 based
14.89%	16.09%	13.93%	13.11%

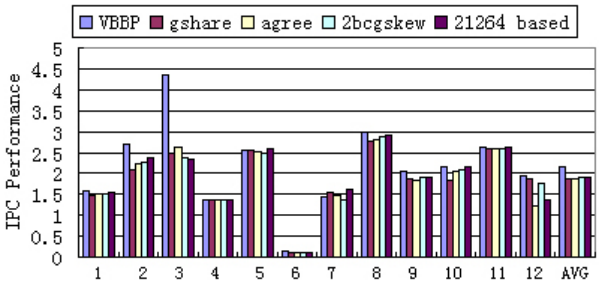

Figure 2. Instruction throughput of our predictor and other dynamic predictors in the experiments.

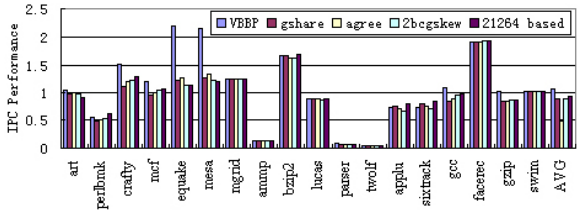

Figure 3. Individual IPC performances of the programs in the two threads running experiments with our predictor and other dynamic predictors.

Figure 3 shows the individual IPC performances of the programs in the two-threads experiments. From the Fig. 3, we can see that our predictor improves all the performances of the two programs running together and that shows the fairness of our predictor to the programs in a workload.

Figure 4 shows the individual IPC results of the programs in the four-threads experiments. Similar as in the Fig. 3, it also shows the fairness of our predictor.

5.2 Branch prediction miss rate

Figure 5 shows the branch prediction miss rates of our predictor and the other predictors considered in the experiments. Our predictor decreases the prediction miss rates largely at some cases. For the *art-perlbmk* and *gzip-swim* workloads, the prediction miss rates of our predictor are one-sixth and half respectively of that of predictor in Alpha 21264. However, it also increases the prediction miss rates largely at other cases such as for the *mgrid-ammp*, *bzip2-lucas*, and *applu-sixtrack* workloads, but the performances of the them do not suffer with the prediction miss rates increasing. Table 5 lists the branch prediction miss rates on average of the predictors.

Table 5. Branch prediction miss rates of the predictors on average.

VBBP	gshare	agree	2Bcgskew	21264
4.57%	6.71%	5.90%	3.33%	3.89%

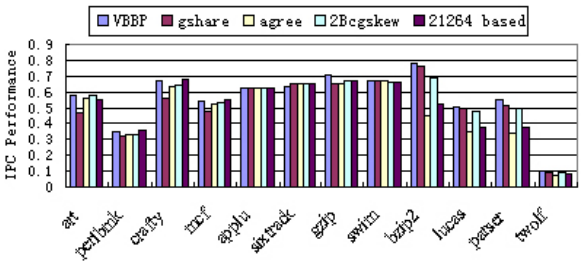

Figure 4. Individual IPC performances of the programs in the four threads running experiments with our predictor and other dynamic predictors.

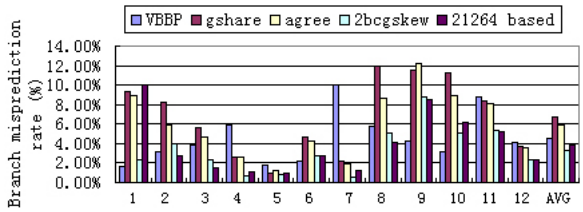

Figure 5. Branch misprediction rates of our predictor and other dynamic predictors in the experiments.

5.3 Wrong path instruction fetch rate

Figure 6 shows the rates of instruction fetched along the wrong path (WP) in our experiments, and Table 6 lists the WP fetched rates on average of the predictors. The WP fetch rates of *equake-mesa*, *mgrid-ammp*, *bzip2-lucas*, and *applu-sixtrack* are increased dramatically with our VBBP predictor compared with that in Alpha 21264, and the WP fetch rate of *gzip-swim* is decreased largely in our predictor. For the other cases, the WP fetch rates of our predictor and other predictors are similar.

Table 6. WP instruction fetch rates of the predictors on average.

VBBP	gshare	agree	2Bcgskew	21264
8.62%	11.56%	10.99%	7.77%	8.11%

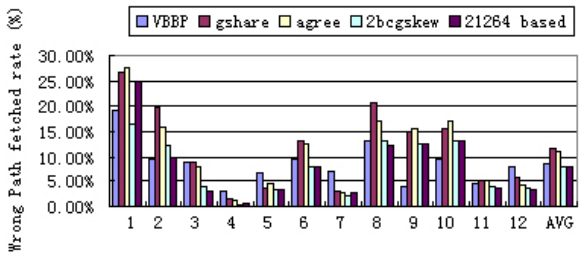

Figure 6. WP fetch rates of our predictor and other dynamic predictors in the experiments.

6 Conclusion

Branch prediction is important to SMT processors. In this paper, we present a simple and effective value based branch predictor for SMT processors. The rationale behind our scheme is that the outcome of a branch instruction is determined by and only by its referenced values of . Our new branch predictor exploits the value locality of the branch instruction, and predicts the outcome of the branch with a higher accuracy. The implementation of our predictor is simple, and the hardware needed is less than that of all the dynamic predictors. Execution-driven simulation results show that high instruction throughput can be obtained using our predictor. The performance speedup of our predictor over the best predictor considered here is 13% on average. The branch prediction miss rates and the wrong path instruction fetch rates are decreased.

7 Acknowledgement

This work was supported by National Natural Science Foundation of China (NSFC) Grant No. 60325205, 60303033, and 60373043.

References

[1] D. M. Tullsen, S. J. Eggers, H. M. Levy et al. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proc. 22nd Annual International Symposium on Computer Architecture*, pp.392-403, June 1995.

[2] D. M. Tullsen. Exploiting Choice: Instruction Fetch and Issue on a Implementable Simultaneous Multithreading Processor. *Proc. 23nd Annual International Symposium on Computer Architecture*, pp.191-202, May 1996.

[3] D. M. Tullsen. Simulation and Modeling of a Simultaneous Multithreading Processor. *Proc. 22nd Annul Computer Measurement Group Conference*, Dec. 1996.

[4] J. L. Henning. SPEC CPU 2000: Measuring CPU Performance in the new millennium. *IEEE Computer*, Volume 33, pp.28-35, July 2000.

[5] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[6] E. Sprangle, R.S. Chappell, M. Alsup et al. The agree predictor: A mechanism for reducing negative branch history interference. *Proc. 24nd Annual International Symposium on Computer Architecture*, pp.284-291, June 1997.

[7] A. Seznec and P. Michaud. De-aliased hybrid branch predictors. Tec. Rep. RR-3618, Inria, Feb 1999.

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

[9] Alpha 21264/EV6 Microprocessor Hardware Reference Manual. ftp://ftp.compaq.com/pub/products/alphaCPUdocs.

[10] T-Y. Yeh and Y. N. Patt. Two-Level Adaptive Training Branch Prediction. *Proc. 24th ACM/IEEE Micro*, pp.51-61, Nov. 1991.

[11] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. *Proc. 24nd Annual International Symposium on Computer Architecture*, pp.292-303, June 1997.

[12] C-C. Lee, I-C.K. Chen, and T.N. Mudge. The bi-mode branch predictor. *Proc. 30th ACM/IEEE Micro*, pp.4-13, Dec 1997.

[13] Daniel A. Jimnez, Fast Path-Based Neural Branch Prediction, *Proc. 36th Annual International Symposium on Microarchitecture (MICRO-36)*, San Diego, CA, December 2003.

[14] Egan C., Steven G., Quick P., Anguera R., Vintan L. Two-Level Branch Prediction using Neural Networks, *Journal of Systems Architecture*, vol. 49, issues 12-15, pg.557-570, ISSN: 1383-7621, Elsevier, December 2003.

[15] T. Sato. First Step to Combining Control and Data Speculation. *Proc. IWIA'98*, pp.53-60, Oct. 1998.

[16] J. Gonzalez, A. Gonzalez. Control-Flow Speculation through Value Prediction. *IEEE Transaction on Computer*, Vol.50, No.12, pp.1362-1376, Dec 2001.

[17] D.A.Patterson and J.L.Hennessy. Web Extension I: Survey of RISC Architectures. http://www.cp.eng.chula.ac.th/faculty/pjw/teaching/cs152/berkeley/index-mkp.htm