Speculative Resolution of Ambiguous Memory Aliasing

Toshinori Sato
Toshiba Microelectronics Engineering Laboratory
580-1, Horikawa-Cho, Saiwai-Ku, Kawasaki 210, Japan
toshinori.sato@toshiba.co.jp

Abstract

The ambiguous memory aliasing is proposed to be speculatively resolved. A load instruction is speculatively executed with load address prediction, and its dependent instructions are speculatively executed. A store instruction is also speculatively resolved with store address prediction, and its dependent instructions are speculatively executed. From the experimental evaluation, we have found that this data speculation combining memory disambiguation with address prediction improves the performance of coming-generation superscalar processors by up to 21.0%.
Keywords: instruction level parallelism, out-of-order execution, dynamic speculation of data dependence, data address prediction, dynamic memory disambiguation

1. Introduction

In order to achieve high performance system, uniprocessors of which a multi-processor system consists should be considered. As a consequence of the trend toward shrinking the transistor size and enlarging the die area, it will be possible to integrate as many functional units as we expect. However, they are not always utilized effectively and instruction level parallelism (ILP) does not increase with the same rate of the increasing number of functional units. It is generally known that dependences including control and data dependences are the obstacles disturbing the processor performance. The former is closely concerned with the instruction supply mechanism, and the latter is tightly connected with the data supply mechanism. Each mechanism is the basic element constructing processors. In order to improve the instruction fetch efficiency, it is one of the promising candidates to fetch multiple basic blocks in a single cycle. We have already evaluated such a mechanism that caches dynamic instruction stream and provides grouped instructions[15]. In this paper, we will investigate the remaining mechanism: the data supply mechanism.

Copyright 1998 IEEE. Published in the Proceedings of IWIA'97, 22-24 October, 1997 in Maui, Hawaii. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

Data dependences are the obstacles limiting the ILP. There are two types of the data dependences. One is the dependences through registers, and the other is those through memory. Among the dependences through registers, the write after write (WAW) and write after read (WAR) hazards can be eliminated using register renaming[8, 19], but the read after write (RAW) hazard can not. For the dependences through memory, however, there are not any techniques such as register renaming. This is because any effective addresses for required data can not be determined until the program is executed. Thus, any succeeding load and store instructions have to wait for the time that a preceding store instruction is resolved. This is the ambiguous memory dependences disturbing the exploitation of ILP. To sum up, there are two serious data dependences between instructions. One is the RAW hazard through registers, and the other is the ambiguous memory dependence.

So far, there are few proposals to remove these dependences. Moreover, most of them are too complicated to be implemented. In this paper, we propose a simple mechanism for data dependence speculation using effective address prediction. A load instruction is speculatively executed with load address prediction, and instructions which are dependent upon the load instruction are also speculatively executed. A store in-

struction is speculatively resolved with store address prediction, and instructions which are dependent upon the store instruction are also speculatively executed.

The organization of the rest of this paper is as follows. Section 2 surveys previously proposed related works. Section 3 explains a data speculation scheme combining address prediction with memory disambiguation. In Section 4, the evaluation methodology is presented and the effect of the data speculation is evaluated in Section 5. Finally, our conclusions are presented in Section 6.

2. Related Work

The problem of the ambiguous memory dependences can be addressed by static dependence analysis and by dynamic memory disambiguation. Static dependence analysis is conducted during compilation. In many cases, its analysis is limited due to the lack of dynamic executing information. Dynamic memory disambiguation resolves the dependences during executing programs[5, 13]. One of the problems included in the dynamic memory disambiguation is the code explosion due to correction codes for recovery actions.

The address resolution buffer (ARB)[4] can resolve the speculative memory references and ambiguous memory dependences by assuming that all addresses of succeeding memory instructions are different from that of a preceding store instruction. Thus, it is possible to execute memory instructions in an out-of-order fashion. When the assumption is found to be not true, the correcting sequence should be executed.

Moshovos et al.[11] proposed address conflict prediction by making use of dynamic sequence histories. If a pair of the memory references by a store and a load instructions is predicted not to conflict, the load instruction can be executed beyond the store instruction in an out-of-order fashion. Our proposal is similar to them. However, their hardware structure is more complicated than ours.

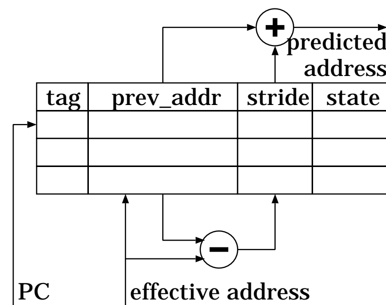
There are many proposals predicting effective addresses of load instructions [1, 14, 17, 20]. Some of them[1, 14] do not speculatively execute instructions following the load instructions. The others [17, 20] execute speculatively the following instructions, but do not take advantage of store address prediction.

Lipasti et al.[9, 10] proposed value prediction based on value locality. They extended branch prediction mechanism to predict the values. The scheme is based on the last outcome. That is, an instruction uses the same value which is used at the last time, when the same instruction is emerged in the future. Therefore, if the value changes frequently, the scheme does not work effectively.

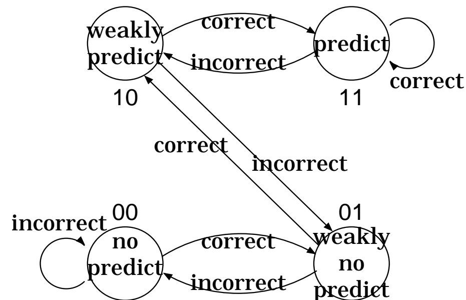
3. Data Speculation Scheme

In this section, we propose a data speculation mechanism[16]. In order to predict effective addresses, we utilize reference prediction table (RPT)[3]. First, we explain the RPT. Next, we describe the data speculation using address prediction. And lastly, we explain how to speculatively resolve the ambiguous memory aliasing.

3.1. Reference Prediction Table



(a) Reference Prediction Table



(b) State Transition

Figure 1. Reference Address Prediction

The RPT, which has a similar structure to instruction cache, is proposed by Chen et al. for hardware prefetching[3]. We apply the RPT to predict the effective address because of its simplicity. The RPT keeps track of previous memory references. An entry of the RPT is indexed by the instruction address and holds the previous effective address (*pred_addr*), the stride value (*stride*), and the state information (*state*). Figure 1a shows the RPT structure. The stride is the difference between the last two data addresses generated

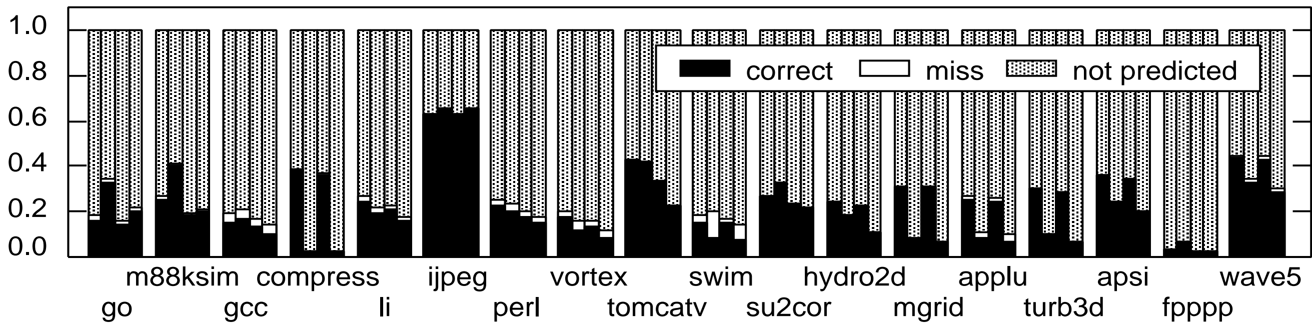


Figure 3. Chen's State Machine

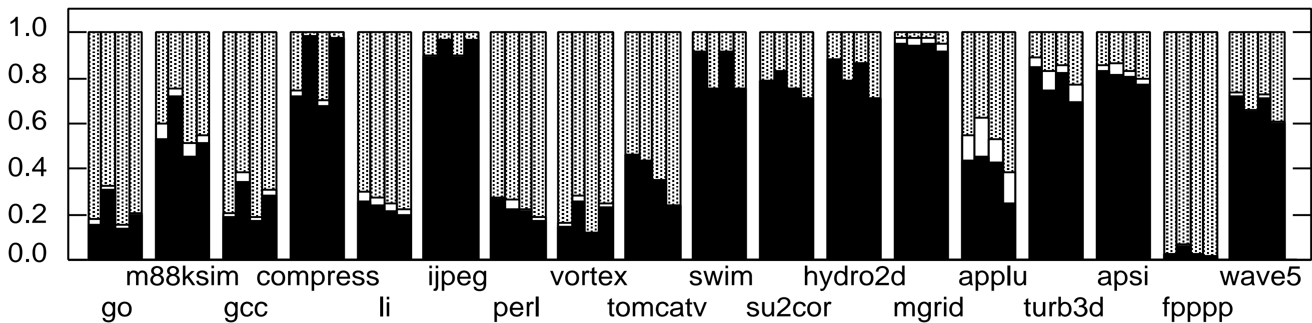


Figure 4. 2-bit Counter State Machine

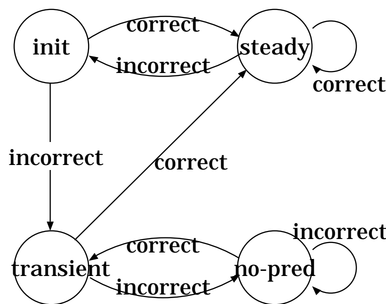


Figure 2. Chen's State Machine

by an instruction. The state information encodes the past history and indicates whether the next prefetching is initiated. The state information is decided according to Figure 1b. Note that the state transition described in Figure 1b is different from the original one proposed in [3]. There are four states, which are *predict*, *weakly predict*, *no-predict*, and *weakly no-predict*.

The predicted address is generated as follows. The program counter (PC) indexes the RPT. The previous effective address and the stride value are supplied from an entry of the RPT indexed by the PC, if the tag field is matched. The predicted address is the sum of *pred_addr* and *stride*. The state information is also provided. If the state is *predict* or *weakly predict*, the predicted address is valid. Otherwise, the prediction is not initiated. Next, we explain the state transition of the RPT. It is quite similar to the two bit saturated counter (2bC) used by branch predictors. If a prediction is correct, the counter is incremented. Otherwise, it is decremented. When the most significant bit is `1`, the state is (*weakly*) *predict* and thus the predicted address is valid.

The reason why we choose the 2bC scheme is as follows. Figure 2 shows Chen's state machine[3]. When this machine is used, the prediction is performed only in *steady* state. In order to compare Chen's and 2bC counter state machines, we executed SPEC95 benchmark suite using a functional simulator[2]. The RPT is direct-mapped and the number of its entries is 1024. We counted the number of load and store instructions, and calculated the percentage of time that the RPT could provide a predicted address and that of time that the prediction was correct. Figures 3 and 4 summarize the results for both state machines. For each group of four bars, the first bar (see from left to right) indicates the statistics of the RPT predicting only load ad-

dresses. Next bar presents those of the RPT predicting only store addresses. The remaining bars show those the RPT predicting both load and store addresses. The third bar is for the load address prediction, and the last bar is for the store address prediction. Each bar is divided into three parts. The bottom part (black) indicates the percentage of the instruction whose data address is correctly predicted ($RPT_correct$). The middle part (white) indicates the percentage that is mispredicted (RPT_miss). And the top part (gray) indicates the percentage that is not predicted by the RPT. We define the prediction accuracy as the $RPT_correct$ over the $RPT_hit (= RPT_correct + RPT_miss)$. It can be seen that the difference between the prediction accuracy of two machines is small. However, the difference between the RPT_hit is large and thus that between the $RPT_correct$ is also large. This implies that the RPT using the 2bC state machine can more aggressively speculate the data dependences through memory references. From this investigation, we chose the RPT using the 2bC state machine.

3.2. Data Speculation with Address Prediction

First, we explain the speculative execution using load address prediction. By speculatively executing load instructions, the length of data dependence path can be reduced. In order to predict the effective address, the RPT is accessed during an instruction is issued into the instruction window. This is different from the previous proposals[1, 14] which predict the effective address during the earlier stages. The difference is due to the purposes of address predictions. Our goal is not to execute a load instruction in the earlier pipeline stages, but to prevent a unresolved load instruction from disturbing the following dependent instructions. For the purpose of the aggressive speculation, the address prediction is performed for every load instruction if the predicted address is validated by the RPT state machine. The pipeline diagram is shown in Figure 5[1].

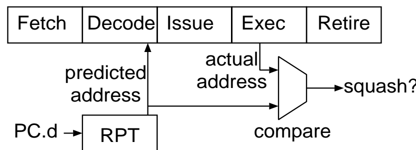


Figure 5. Pipeline with RPT

The RPT indexed by the PC is accessed during the decode stage. The predicted address corresponds with the load instruction and reserved until the Exec stage. It is assumed that the instruction window could hold not only operand values but also predicted values. A load instruction is speculatively executed using the predicted address. When the actual address is generated in the Exec stage, it must be compared with the predicted one. The comparison is performed between the actual address and the predicted value held in the instruction window. If two addresses are equal with each other, the prediction is succeeded and the load instruction needs not perform a memory access using the actual address. Otherwise, the misprediction occurs and the recovery process has to be performed. The load instruction must be executed completely, i.e. a memory access occurs, and the instructions dependent upon the mispredicted load instruction are squashed as shown in Figure 6a. In the figure, the instructions marked with * have to be squashed. All instructions following the mispredicted load instruction are squashed. This scheme is simple and easy to implement. Actually, it is possible for the squashing process to utilize the recovery mechanism of branch misprediction.

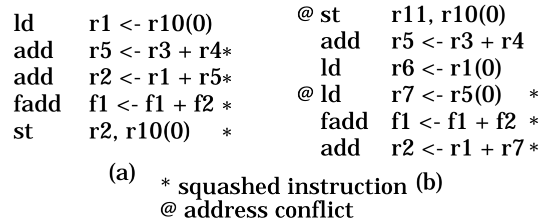


Figure 6. Misprediction Handling

3.3. Speculative Resolution of Ambiguous Memory Aliasing

Next, we propose the data dependence speculation by combining memory disambiguation with address prediction. We assume that the address reorder buffer implemented inside HP PA-8000[6] is used. The effective address of an unresolved store instruction is predicted and the store instruction is speculatively resolved. Thus, load instructions which probably cause the conflicts of memory references can be executed before the store address is generated. Note that any store instructions are not speculatively executed.

Similar to load address prediction, a store address is predicted during the decode stage. It is not the purpose to execute the store instruction speculatively. Our goal is to resolve the store instruction speculatively and to prevent it from disturbing the following load instructions which are dependent upon the store instruction

[1] The pipeline is a simple structure for easy understanding. Actually, some instructions might be executed in multiple stages.

Table 1. Baseline Processor Configuration

Fetch Width	8 instructions
Branch Predictor	512 set, 2way set-associative BTB, gshare scheme, 12-bit BHR, 4096 entry PHT, speculatively updated in ID stage, 8 entry return address stack, 3 cycle miss penalty
Insn. Windows	64 entry instruction queue, 8 entry load/store queue
Issue Width	8 instructions
Commit Width	8 instructions
Functional Units	5 iALU's, 1 iMUL/DIV, 4 Ld/St, 2 fALU's, 2 fMULs, 2 fDIV/SQRT's
Latency(total/issue)	iALU 1/1, iMUL 3/1, iDIV 35/35, Ld/St 2/1, fADD 2/1, fMUL 3/1, fDIV/SQRT 6/6
Register Files	32 32-bit fixed point registers, 32 32-bit floating point registers
Insn. Cache	64K 4way set-associative, 32 byte blocks, 4-port, 6 cycle miss penalty
Data Cache	64K 4way set-associative, 32 byte blocks, 4-port, write-back, non-blocking load, hit under miss, 6 cycle miss penalty
L2 Cache	unified, 256K 4way set-associative, 64 byte blocks, 32 cycle miss penalty
RPT	1024 entry, direct-mapped

due to the ambiguous memory reference. If a load address is different from the predicted store address, the load instruction can be executed speculatively. The diagram is the same as that of load address prediction shown in Figure 5.

When a speculative resolution fails, a recovery process must be performed. The probable dependent instructions have to be squashed. Let us see Figure 6b. The instructions marked with `*` are the squashed instructions, and those marked with `@` refer to the same memory location. The squashing scheme is different from that for load address mispredictions. It squashes all instructions following the load instruction whose address conflicts with the actual store address. Store address misprediction does not always cause the squashing process. This can be implemented using the address reorder buffer. Owing to this strategy, useless squashing process is eliminated. Let us imagine the recovery process performed whenever the predicted store address is not correct. When the store address prediction fails, there are four situations according to the status of address reference conflicts. The situations are (i) the misprediction results in the address conflict and actually there are not any conflicts, (ii) the misprediction results in the conflict and actually there is at least one conflict, (iii) the misprediction results in no conflict and actually there are not any conflicts, and (iv) the misprediction results in no conflict and actually there is at least one conflict. Only the case (iv) needs the recovery process. In the cases of other situations, it is not necessary to squash instructions following the store instruction whose address is mispredicted. The squashing is not only unnecessary but also harmful. The address reorder buffer solves this problem.

4. Evaluation Methodology

In this section, we describe the evaluation methodology by explaining a processor model and benchmark programs.

4.1. Processor Model

We evaluated the effect of the proposed mechanism by using the SimpleScalar tool set[2]. The SimpleScalar architecture is based on the MIPS architecture, and the fully execution-driven and cycle-by-cycle simulator is executed on a SPARCstation. The baseline model is an out-of-order execute superscalar processor based on the register update unit[18]. Following the discussion explained in [7], we decided the configuration of the baseline processor summarized in Table 1. The penalty for the data dependence misprediction is assumed to be 3 cycles. We evaluated three models. The first is the model performing load address prediction. The second is the model performing store address prediction and speculative memory disambiguation. And the last one is the model performing load and store address predictions and speculative memory disambiguation. In the remainder of this paper, we call these models `load`, `store`, and `ldst` models, respectively.

4.2. Workload

The SPEC95 benchmark suite is used for this study. We used the `test` input files provided by SPEC, and the object files provided by University of Wisconsin Madison[2]. Each program was executed to completion or for the first 100 million instructions.

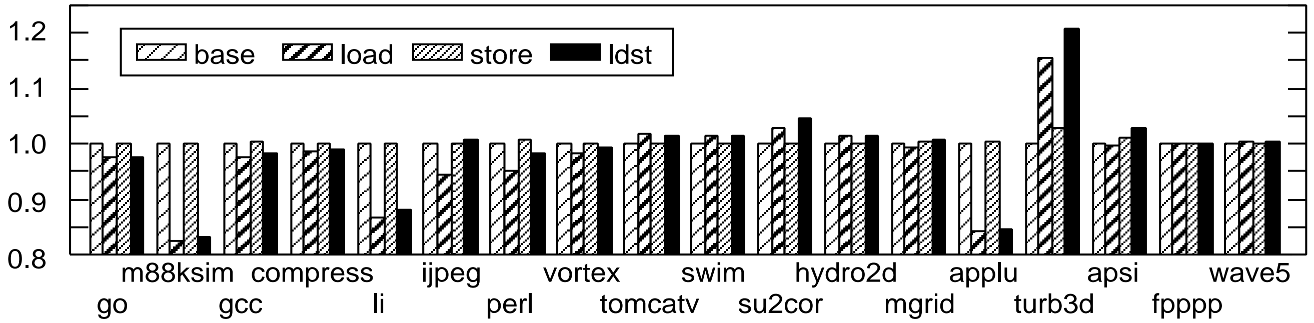


Figure 7. Performance Improvement

Table 2. Address Prediction Accuracy (%) and L1 Cache Miss Rate (%)

program	address prediction				cache miss rate							
	load	store	ldst		base		load		store		lsdt	
			load	store	insn	data	insn	data	insn	data	insn	data
go	88.25	95.37	87.93	97.36	0.50	1.05	-0.02	-0.14	0.00	0.00	-0.02	-0.14
m88ksim	89.20	96.31	88.33	95.41	0.00	0.11	0.00	0.00	0.00	0.00	0.00	0.01
gcc	93.73	93.83	94.10	94.80	0.92	1.54	-0.05	-0.08	0.00	0.00	-0.04	-0.06
compress	96.19	99.80	96.47	99.86	0.03	10.88	0.00	0.00	0.00	0.00	0.00	0.01
li	86.62	89.95	85.60	92.49	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.01
ijpeg	99.55	99.58	99.59	99.66	0.24	0.72	0.00	0.07	0.00	0.00	0.00	0.01
perl	95.65	83.69	96.47	86.83	0.14	0.04	-0.01	0.00	0.00	0.00	-0.01	0.00
vortex	94.38	92.79	95.27	94.27	1.08	0.68	-0.04	0.01	0.00	0.00	-0.03	0.01
tomcatv	99.53	100.0	99.78	100.0	0.00	0.12	0.00	0.00	0.00	0.00	0.00	0.00
swim	100.0	100.0	100.0	100.0	0.00	0.13	0.00	0.00	0.00	0.00	0.00	0.00
su2cor	99.08	99.08	99.41	99.69	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00
hydro2d	99.27	99.90	99.76	99.91	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00
mgrid	97.53	97.18	97.72	97.13	0.00	3.92	0.00	0.05	0.00	0.00	0.00	0.05
applu	83.05	81.19	83.02	78.98	0.00	3.44	0.00	0.03	0.00	0.00	0.00	0.02
turb3d	97.75	96.22	97.83	96.61	0.00	12.44	0.00	0.03	0.00	0.00	0.00	0.01
apsi	96.50	94.86	97.01	96.34	0.04	3.39	0.00	-0.03	0.00	0.01	0.00	-0.03
fpppp	98.74	98.93	98.48	96.57	10.89	0.03	-0.02	0.00	0.00	0.00	-0.02	0.00
wave5	98.99	99.93	98.96	99.93	0.00	0.64	0.00	0.18	0.00	0.00	0.00	0.00

5. Experimental Results

This section presents the experimental results. First, we show the improvement of processor performance. For measuring performance, we use the committed instruction per cycle (IPC). And next, we discuss some factors influencing the performance improvement.

5.1. Performance Improvement

Figure 7 shows the improvement of processor performance. The IPCs of the evaluated models are normalized by that of the baseline model. For each group of four bars, the first bar (see from left to right) indicates the performance of the baseline model. Only committed instructions are considered for counting the IPC. Remaining three bars indicate the performance of the `load`, `store`, and `ldst` models, respectively.

As can be seen in Figure 7, the data dependence speculation with load address prediction hardly contributes processor performance. For eleven of eighteen programs, processor performance is degraded. The sole exception is `turb3d`. The performance is significantly improved by 15.7%. This encourages us, since it implies there is a possibility to exploit ILP using data

Table 3. Branch Prediction Accuracy (%)

program	base			load			store			ldst		
	addr	dir	jr	addr	dir	jr	addr	dir	jr	addr	dir	jr
go	78.80	80.50	97.31	-0.27	0.04	-4.74	0.01	0.00	0.00	-0.25	0.04	-4.34
m88ksim	95.78	96.51	90.08	-1.13	0.00	-14.66	0.00	0.00	0.00	-1.11	0.00	-14.36
gcc	87.34	91.13	79.80	-0.34	-0.02	-4.02	-0.01	-0.01	0.00	-0.28	-0.01	-3.29
compress	96.61	97.02	96.47	-0.12	-0.02	-1.43	0.01	0.01	0.00	-0.22	0.00	-3.15
li	93.61	95.60	86.31	-1.55	0.54	-14.59	0.00	-0.01	0.00	-1.31	0.57	-13.10
ijpeg	98.40	98.80	99.74	-0.02	0.00	-0.11	0.00	0.00	0.00	-0.02	0.00	-0.11
perl	91.64	96.92	64.86	-0.89	-0.13	-5.57	0.06	0.06	0.00	-0.28	-0.02	-1.99
vortex	92.48	94.72	97.48	-0.78	-0.01	-6.25	-0.01	-0.01	0.00	-0.61	-0.03	-4.72
tomcatv	96.80	98.16	91.03	0.04	0.10	-0.57	0.00	0.00	0.00	0.08	0.18	0.00
swim	98.36	98.36	99.97	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
su2cor	96.69	99.07	80.76	-0.20	-0.20	-0.01	0.00	0 00	0.00	0.20	0.20	-0.01
hydro2d	95.93	97.91	87.02	0.21	0.21	0.01	0.00	0.00	0.00	0.07	0.08	0.02
mgrid	98.07	98.17	64.42	-0.01	0.00	-1.81	0.00	0.00	0.00	-0.01	0.00	-2.22
applu	97.76	97.80	96.60	-0.01	0.00	-2.12	0.00	0.00	0.00	0.00	0.00	-1.52
turb3d	98.19	98.21	99.83	-0.09	-0.08	-0.12	0.00	0.00	0.00	-0.05	-0.04	-0.09
apsi	96.04	97.22	96.46	-0.15	0.03	-2.55	0.02	0.02	0.00	-0.11	0.03	-1.87
fpppp	93.17	93.31	80.43	-0.19	0.00	-2.45	0.00	0.00	0.00	-0.19	0.02	-1.45
wave5	98.42	98.43	99.99	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

dependence speculation. On the other hand, the speculative resolution of ambiguous memory aliasing using store address prediction does not contribute much to processor performance. The improvement of ILP is less than 3%. However, it always improves processor performance, except `tomcatv`. Thus, we have confirmed the speculative resolution is useful for exploiting ILP. The performance improvement when the speculative resolution is combined with load and store address predictions is up to 21.0%.

5.2. Discussion

In this section we investigate some factors damaging the performance: address prediction accuracy, instruction and data cache misses, and branch prediction accuracy.

5.2.1 Address Prediction Accuracy

The left side of Table 2 shows the address prediction accuracy. The first column shows the name of each benchmark program. Next two columns indicate the address prediction accuracy for the `load` and `store` models, respectively. The next two columns show the load and store address prediction accuracy for the `ldst` model, respectively.

For the three programs (`m88ksim`, `li`, and `applu`) whose performance is significantly degraded in the `load` and `ldst` models, the load address prediction accuracy is relatively low. This implies that the poor prediction accuracy is one of the harm characteristics for the performance improvement using the data dependence speculation. The misprediction is very severe because it can not be detected until the actual address is calculated and thus misprediction penalty is very high. Moreover, the performance is not always increased for the programs of which prediction accuracy is high. Therefore, only prediction accuracy does not dominate the performance improvement.

5.2.2 Instruction and Data Cache Misses

The miss rates of primary caches are presented in the right side of Table 2. First two columns indicate the instruction and data cache miss rates for the baseline model. The next three groups of two columns indicate the absolute changes of the miss rates from that for the baseline model for the `load`, `store`, and `ldst` models, respectively. The changes of the miss rates are small and it seems that there are no relation between the performance degradation and the changes of miss rates.

Table 4. Prediction Accuracy (%)

program	branch prediction accuracy						address prediction		
	load			ldst			load	ldst	
	addr	dir	jr	addr	dir	jr		load	store
go	-0.20	0.02	-3.40	-0.19	0.02	-3.28	73.81	73.95	97.36
m88ksim	-0.82	-0.02	-10.43	-0.80	-0.01	-10.18	88.14	88.16	95.41
gcc	-0.11	-0.03	-1.02	-0.09	-0.02	-0.87	92.10	92.53	94.80
compress	-0.12	-0.02	-1.39	-0.22	0.00	-3.15	91.00	91.59	99.86
li	-0.41	0.73	-8.24	-0.28	0.74	-7.13	79.19	79.39	92.49
ijpeg	-0.01	0.00	-0.05	0.00	0.00	-0.05	99.65	99.69	99.66
perl	-0.12	-0.03	-0.69	-0.15	-0.06	-0.69	97.21	96.97	86.83
vortex	-0.14	0.00	-1.16	-0.13	0.00	-1.00	93.80	94.66	94,27
tomcatv	-0.01	0.06	-0.56	0.06	0.06	0.00	98.38	99.20	100.0
swim	0.00	0.00	0.00	0.00	0.00	0.00	98.41	98.71	100.0
su2cor	0.00	0.00	0.00	0.20	0.20	0.00	97.65	98.41	99.69
hydro2d	0.08	0.08	0.00	0.08	0.08	0.00	97.82	99.27	99.91
mgrid	0.00	0.00	-0.43	0.00	0.00	-0.41	96.69	96.60	97.13
applu	0.00	0.00	-0.77	0.00	0.00	-0.77	79.07	79.31	78.98
turb3d	-0.08	-0.08	-0.03	-0.05	-0.04	-0.03	96.76	96.91	96.61
apsi	-0.08	0.00	-1.14	-0.06	0.00	-0.88	95.72	96.37	96.34
fpppp	-0.08	0.01	-0.88	-0.07	0.02	-0.88	94.32	94.39	96.57
wave5	0.00	0.00	0.00	0.00	0.00	0.00	95.97	95.97	99.93

5.2.3 Branch Prediction Accuracy

Table 3 shows the branch prediction accuracy. The four groups of three columns indicate the results for the baseline, `load`, `store`, and `ldst` models, respectively. For each group, each column presents the accuracy of target address prediction, that of branch direction prediction, and that of the indirect jump address prediction, respectively. For the `load`, `store`, and `ldst` models, the absolute change from the baseline model is presented. In general, target address prediction accuracy is degraded in the cases of the `load` and `ldst` models. The main reason is the diminution of the prediction accuracy for indirect jump addresses. It becomes worse by up to 16.9% (from 86.31% to 71.72% for `li`). One of the reasons why the branch direction prediction accuracy becomes worse is that it is harmful to update registers by useless speculative load instructions. It is said that even slight decrease of branch prediction accuracy affects severely processor performance. The examples are `m88ksim` and `li`, whose accuracy of target address decreases over 1%, as can be seen in Table 3. There are two solutions. One is to reduce jump instructions using sophisticated compilers[12]. The other is to make the speculative scheme more conservative. All load instructions are not speculatively executed, but only instructions whose operands are not ready are speculatively executed. We have evaluated the latter strategy. The left side of Table 4 shows the branch prediction accuracy using the conservative strategy. Each column shows the absolute change from the baseline results. The significant improvement is found, especially for the indirect jump address prediction. The right side of the table indicates the address prediction accuracy, and shows the address prediction accuracy is decreased for several programs. Figure 8 shows the performance improvement using the conservative strategy. The IPC of the evaluated models are normalized by that of the baseline model shown in Figure 7. Next, we compare the conservative scheme with the aggressive scheme. Using the aggressive scheme, the performance of integer programs is degraded by 6.1% and 4.4% on average for the `load` and `ldst` models, respectively. For floating-point programs, it is improved by 0.7% and 1.8% on average for the `load` and `ldst` models, respectively. On the other hand, the conservative scheme degrades the performance of integer programs by 3.2% and 2.5% on average for the `load` and `ldst` models, respectively. For floating-point programs, the performance is improved by 1.4%and 2.4% on average for the `load` and `ldst` models, respectively. In general, the conservative scheme is effective for the programs whose performance degradation is large.

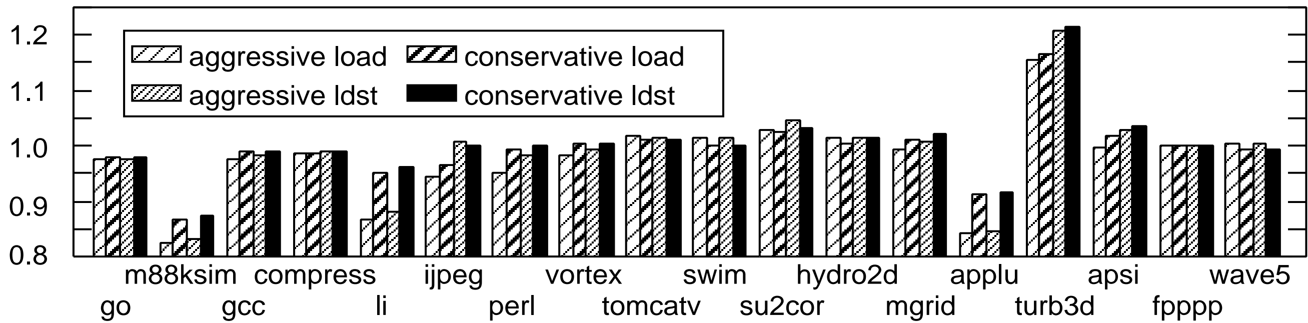


Figure 8. Performance Improvement

6. Concluding Remarks

We have proposed the speculative execution for data dependences. By combining memory disambiguation and address prediction, the unresolved store instructions are speculatively disambiguated and the probable dependent load instructions are speculatively executed. In order to predict the effective address, we utilize the RPT proposed for the hardware prefetching. From the experimental results, the data dependence speculation using the RPT has little contribution to processor performance. The sole exception is `turb3d`. The performance is significantly improved by 15.7%. On the other hand, the speculative memory disambiguation using store address prediction does not much but always contributes to processor performance. When load and store predictions are combined with each other, that is when memory disambiguation and address prediction are combined with each other, the improvement of processor performance is up to 21.0% using the aggressive scheme.

One of the future studies dealing with the data dependence speculation is to reduce the miss-speculation penalty. We are now investigating the selective squashing scheme. If the instructions which are not data-dependent upon speculatively executed instructions and must not be squashed are reused, the effectiveness of the data dependence speculation may be increased. Since data dependences are kept in instruction window, it may be easy to select instructions depending upon a mispredicted load instruction. The instruction reissue is also being evaluated. The independent instructions upon the mispredicted instruction should not be fetched again from memory but issued again inside the instruction window. This scheme eliminates the penalty of refetching, and thus the performance degradation caused by misprediction can be reduced. The other is to increase the effect of speculative resolution of ambiguous memory aliasing. We are now evaluating an another memory dependence prediction scheme which predicts load values using store instruction addresses.

Acknowledgement

The author thanks to Prof. Hironori Nakajo and Prof. Shigeru Kusakabe whose comments and suggestions helped to improve the quality of this paper. The author also thanks to the IWIA organizing committee members for inviting him to the workshop. Lastly, the author is grateful to Dr. Mitsuo Saito and Dr. Shigeru Tanaka for their continuous encouragements.

References

[1] T.M.Austin, G.S.Sohi, "Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency", Proc. of 28th Ann. Int'l Symp. on Microarchitecture, pp.82-92, 1995.

[2] D.Burger, T.M.Austin, "The SimpleScalar Tool Set, Version 2.0", ACM SIGARCH Computer Architecture News, vol.25, no.3, pp.13-25, June 1997.

[3] T-F.Chen, J-L.Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors", IEEE Trans. on Computers, vol.44, no.5, pp.609-623, May 1995.

[4] M.Franklin, G.S.Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References", IEEE Trans. on Computers, vol.45, no.5, pp.552-571, May 1996.

[5] D.M.Gallagher, W.Y.Chen, S.A.Mahlke, J.C.Gyllenhaal, W.W.Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer", Proc. of Architectural Sup-

port for Programming Languages and Operation Systems VI, pp.183-195, 1994.

[6] D.Hunt, “Advanced Performance Features of the 64-bit PA-8000”, Proc. of COMPCON'95, pp.123-128, 1995.

[7] S.Jourdan, P.Sainrat, D.Litaize, “Exploring Configuration of Functional Units in an Out-of-Order Superscalar Processor”, Proc. of 22nd Ann. Int'l Symp. on Computer Architecture, pp.117-125, 1995.

[8] R.M.Keller, “Look-Ahead Processors”, ACM Computing Surveys, vol.7, no.4, pp.177-195, Dec. 1975.

[9] M.H.Lipasti, C.B.Wilkerson, J.P.Shen, “Value Locality and Load Value Prediction”, Proc. of Architectural Support for Programming Languages and Operation Systems VII, pp.138-147, 1996.

[10] M.H.Lipasti, J.P.Shen, “Exceeding the Dataflow Limit via Value Prediction”, Proc. of the 29th Ann. Int'l Symp. on Microarchitecture, pp.226-237, 1996.

[11] A.I.Moshovos, S.E.Breach, T.N.Vijakumar, G.S.Sohi, “Dynamic Speculation and Synchronization of Data Dependences”, Proc. of 24th Ann. Int'l Symp. on Computer Architecture, pp.181-193, 1997.

[12] F.Mueller, D.B.Whalley, “Avoiding Unconditional Jumps by Code Replication”, Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp.322-330, 1992.

[13] A.Nicolau, “Run-Time Memory Disambiguation: Coping with Statically Unpredictable Dependencies”, IEEE Trans. on Computers, vol.38, no.5, pp.663-678, May 1989.

[14] T.Sato, H.Fujii, S.Suzuki, “Hiding Data Cache Latency with Load Address Prediction”, IEICE Trans. on Information and Systems, vol.E79-D, no.11, pp.1523-1532, Nov. 1996.

[15] T.Sato, “NCB: A Mechanism for Improving Instruction Fetching Efficiency”, Proc. of 9th Joint Symp. on Parallel Processing, pp.221-228, May 1997.

[16] T.Sato, “Data Dependence Speculation Combining Memory Disambiguation with Address Prediction”, Proc. of 10th Summer United Workshop on Parallel, Distributed, and Cooperative Processing (IPSJ SIG Notes 97-ARC-125-1), pp.1-6, Aug. 1997.

[17] Y.Sazeides, S.Vassiliadis, J.E.Smith, “The Performance Potential of Data Dependence Speculation & Collapsing”, Proc. of 29th Ann. Int'l Symp. on Microarchitecture, pp.238-247, 1996.

[18] G.S.Sohi, “Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers”, IEEE Trans. on Computers, vol.39, no.3, pp.349-359, Mar. 1990.

[19] R.M.Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”, IBM Journal, vol.11, pp.25-33, Jan. 1967.

[20] L.Widigen, E.Sowadsky, K.McGrath, “Eliminating Operand Read Latency”, ACM SIGARCH Computer Architecture News, vol.24, no.5, pp.18-22, Dec. 1996.