計算機アーキテクチャ 125－1
（1997. 8. 20）

Data Dependence Speculation Combining Memory Disambiguation with Address Prediction

Toshinori Sato
Microelectronics Engineering Laboratory
Toshiba Corporation, Kawasaki 210, Japan
toshinori.sato@toshiba.co.jp

Data dependence speculation using effective address prediction is proposed. 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 instruction is speculatively resolved with store address prediction, and instructions which are dependent upon the store instruction are also speculatively executed. From the experimental evaluation, we have found that the data dependence speculation combining memory disambiguation with address prediction is effective on an out-of-order execution processor.

ロード・ストアアドレスの早期生成によるデータ依存解消手法

佐藤寿倫
株式会社東芝 マイクロエレクトロニクス技術研究所

ロード・ストアアドレスの予測を用いた投機的実行を提案する。ロードアドレスを予測することでロード命令を投機的に実行し、さらに該ロード命令とデータ依存関係にある命令も投機的に実行する。また、ストアアドレスを予測することでメモリ参照のあいまいさを投機的に解消し、さらに該ストア命令と潜在的に依存関係にあるロード命令も投機的に実行する。ロード・ストアアドレスの予測に失敗した場合にはペナルティを被るので、高精度の予測機構が必要である。従来プリフェッチに用いられていた簡易な予測機構を採用して、シミュレーションによりプロセッサの性能向上を確認した。

1 Introduction

Data dependences are the obstacles limiting the instruction level parallelism (ILP). There are two types of the dependences. One is the dependences through the registers, and the other is those through the memory. Among the dependences through the registers, the write after write (WAR) and write after read (WAR) hazards can be eliminated using register renaming[11, 22], but the read after write (RAW) hazard can not. For the dependences through the memory, however, there are not any techniques such as register renaming. This is because the effective address for a required datum can not be determined until the program is executed. Thus, any succeeding load and store instructions have to wait for that the preceding store instruction is resolved. This is the ambiguous memory dependences disturbing the exploitation of ILP. In summary, there are two serious data dependences between instructions. One is the RAW hazards through the registers, and the other is the ambiguous memory dependences.

So far, there are few proposals to remove these dependences. Moreover, most of which 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 instruction 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 method 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[5]. In many cases, its analysis is limited due to the lack of dynamic executing information. Dynamic memory disambiguation resolves the dependences during executing programs[8, 10, 16]. One of the problems included in the dynamic memory disambiguation is the code explosion due to correction codes for recovery action.

The address resolution buffer (ARB)[6, 7] can resolve the speculative memory references and ambiguous memory dependences by assuming that the addresses of succeeding memory instructions are different from that of the preceding store instruction. Thus, it is possible to execute memory instructions in out-of-order fashion. When the assumption is not true, the correcting sequence should be executed.

Moshovos et al.[14, 15] proposed the address conflict prediction by making use of the dynamic sequence history. Our proposal is similar to them. However, their hardware structure is very complicated.

There are many proposals predicting the effective address of load instructions [1, 4, 9, 17, 18, 20, 23]. Most of them[1, 4, 9, 17, 18] do not speculatively execute the instructions following the load instructions. Some of them[20, 23] execute speculatively the following instructions, but do not take advantage of the store address prediction.

Lipasti et al.[12, 13] proposed the value prediction based on value locality. They extended the branch prediction mechanisms to predict the values.

3 Data Speculation Combining Memory Disambiguation with Address Prediction

In this section, we propose a data speculation mechanism. In order to predict the effective address, we utilize the reference prediction table (RPT)[3]. For the preparation of our proposal, firstly in this section, we explain the RPT. Next, we describe the data speculation using address prediction. And lastly, we explain the data dependence speculation combining the memory disambiguation with address prediction.

3.1 Reference Prediction Table

The RPT, which has a similar structure with 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 reference. 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 1 shows the RPT structure. The stride is the difference between last two data addresses generated by an instruction. The state information encodes the past history and indicates whether next prefetching is initiated.

The state information is decided according to Figure 2. Note that the state transition described in Figure 2 is different from the original one proposed in [3]. There are four states, which are *predict*, *weakly predict*, *no-predict*, and *weakly no-predict*.

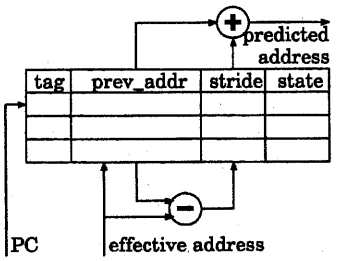

Figure 1: Reference Prediction Table

The prediction address is generated as follows. The program counter (PC) indexes the RPT. The previous effective address and the stride value are acquired from an entry of the RPT indexed by the PC, if the tag field is matched. The prediction address is the sum of *pred_addr* and *stride*. The state information is also

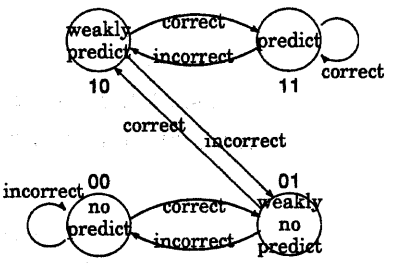

Figure 2: State Transition of RPT

acquired. If the state is *predict* or *weakly predict*, the prediction address is valid. Otherwise, the address is invalidated.

Next, we explain the state transition of the RPT. It is quite similar to the two bit saturated counter (2bC) used by the branch predictors. If the 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 that it is simple to be implemented.

3.2 Data Speculation with Address Prediction

Firstly, we explain the speculative execution using load address prediction. By speculatively resolving load instructions, the length of the data dependence path can be reduced. In order to predict the effective address, the RPT is accessed during the decode stage. This is different from the previous proposals[1, 4, 9, 17] which predicts the effective address during the fetch stage. The difference is due to the purpose of the address prediction. 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. Due to the policy that the address prediction is performed during the decode stage, the useless prediction is prohibited when a load instruction is about to be issued. The pipeline diagram is shown in Figure 3[1].

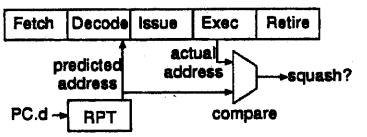

Figure 3: Pipeline with RPT

The RPT indexed by the PC is accessed at the beginning of the decode stage. The predicted address is corresponded with the load instruction and held until the exec stage. When the actual address is gen-

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

erated in the exec stage and it is same to the predicted address, the prediction is succeeded. In such a case, the load instruction need not perform any memory access. If the actual address is different from the predicted address, the misprediction occurs and the recovery process has to be performed. The load instruction must be executed completely, i.e. memory access is occurs, and the instructions dependent upon the mispredicted load instruction are squashed as shown in Figure 4. In the figure, the instructions marked with * have to be squashed. There are two strategies to squash dependent instructions. One is squashing all instructions following the mispredicted load instruction (Figure 4a), and the other is squashing selectively the instructions which are dependent upon the mispredicted load instruction (Figure 4b). Even though the former has the penalty to re-execute independent instructions, it is simpler and easier to implement than the latter. Actually, it is possible for the former scheme to utilize the recovery mechanism of the branch misprediction. In Section 4, we will choose the former for evaluation.

```
ld    r1 <- r10(0)          ld    r1 <- r10(0)
add   r5 <- r3 + r4*        add   r5 <- r3 + r4
add   r2 <- r1 + r5*        add   r2 <- r1 + r5*
fadd  f1 <- f1 + f2 *       fadd  f1 <- f1 + f2
st    r2, r10(0)    *       st    r2, r10(0)    *
        (a)                         (b)

            * squashed instruction
```

Figure 4: Mispredicted Load Instruction

3.3 Combining Memory Disambiguation with Address Prediction

Next, we propose the data dependence speculation by combining the memory disambiguation with the address prediction. The effective address of an unresolved store instruction is predicted and the store instruction is speculatively resolved. Thus, load instructions which probably cause the conflict of memory reference can be executed before the store address is generated. Note that any store instructions are not speculatively executed.

Similar to the 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 resolving the store instruction speculatively and prevent it from disturbing the following load instructions which is dependent upon the store instruction due to the ambiguous memory reference. If a load address is different from the predicted store address, the load instruction can be executed speculatively. If a store instruction is about to be issued, the store address prediction is not performed. The diagram is as same as that of the load address prediction shown in Figure 3.

When a store address prediction fails, the recovery process must be performed. The probable dependent instructions has to be squashed. For this purpose, there are three schemes as shown in Figure 5.

In the figure, the instructions marked with * are the squashed instructions, and those marked with @ refer to same memory location. The schemes are (i) the scheme squashing all instructions following the mispredicted store instruction (Figure 5a), (ii) the scheme squashing all instructions following the load instruction whose address conflicts with mispredicted store address (Figure 5b), and (iii) the scheme squashing selectively the instructions which is truly dependent upon the store instruction (Figure 5c). Even though the first one has the penalty to re-execute independent instructions, it is simplest and easiest to implement among them. Actually, it is possible to utilize the recovery mechanism of the branch misprediction. In Section 4, we will choose the first scheme for evaluation.

```
@ st   r11, r10(0)          @ st   r11, r10(0)
  add  r5 <- r3 + r4 *        add  r5 <- r3 + r4
  ld   r6 <- r1(0)  *         ld   r6 <- r1(0)
@ ld   r7 <- r5(0)  *       @ ld   r7 <- r5(0)  *
  fadd f1 <- f1 + f2 *        fadd f1 <- f1 + f2 *
  add  r2 <- r1 + r7 *        add  r2 <- r1 + r7 *
         (a)                         (b)

@ st   r11, r10(0)
  add  r5 <- r3 + r4
  ld   r6 <- r1(0)
@ ld   r7 <- r5(0)  *     @ address conflict
  fadd f1 <- f1 + f2      * squashed instruction
  add  r2 <- r1 + r7 *
         (c)
```

Figure 5: Mispredicted Store Instruction

3.4 Pipelined Reference Prediction Table

Since the RPT has a similar structure to cache, the access time of the RPT may dominate the cycle time of the processor. Furthermore, in order to predict the effective address, the addition must be performed. Thus, the access to the RPT may be on the critical path. In such situation, the RPT can be pipelined and the access to the RPT is started in the fetch stage as shown in Figure 6. For example, it is possible to implement the RPT which accesses the data array during the fetch stage and calculates the effective address during the decode stage.

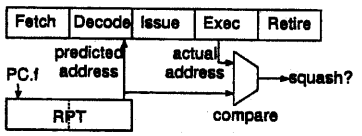

Figure 6: Pipelined RPT

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 (version 1.0.2)[2]. The SimpleScalar architecture is based on the MIPS architecture, and the cycle-by-cycle simulator is executed on a SPARCstation. The baseline model is an out-of-order issue superscalar processor based on the register update unit (RUU)[21]. The configuration of the baseline processor is summarized on Table 1. The penalty for the address misprediction is assumed to be 3 cycles.

Table 1: Baseline Processor Configuration

Fetch Width	8 instructions, interleaved sequential
Branch Predictor	512 set, 2way set-associative BTB, gshare scheme, 12-bit BHR, 4096 entry PHT, 3 cycle miss penalty
Instruction Windows	64 entry instruction queue, 8 entry load/store queue
Issue Width	8 instructions
Functional Units	5 iALU's, 1 iMUL/DIV, 2 ld/st units, 2 fALU's, 2 fMULs, 2 fDIV/SQRT's
FU Latency (total/issue)	iALU 1/1, iMUL 3/1, iDIV 35/35, Ld/St 2/1, fADD 2/1, fMUL 3/1, fDIV 6/6, fSQRT 6/6
Register Files	32 32-bit fixed point registers, 32 32-bit floating point registers
I-Cache	64K 4way set-associative, 32 byte blocks, 2-port, 6 cycle miss penalty
D-Cache	64K 4way set-associative, 32 byte blocks, 2-port, write-back, non-blocking load, hit under miss, 6 cycle miss penalty
L2 Cache	ideal
RPT	1024 entry, direct-mapped

4.2 Workload

The SPEC92 benchmark suite is used for this study. The reference input files which are provided by SPEC are used with slight modifications. Table 2 shows the summary. The Fortran programs were converted to C programs using AT&T F2C (version 1994.11.03), and then all programs were compiled by GNU GCC (version 2.6.3) with the optimization option, `-O3`. Each program was executed to completion or for the first 1 billion instructions.

5 Experimental Results

This section presents the experimental results. Firstly, we show the improvement of processor performance. For measuring performance, we use the committed instruction per cycle (IPC). And next, the prediction accuracy is discussed.

Table 2: Benchmark Programs

Benchmark	Input	Modification
008.espresso	bca.in	
022.li	li-input.lsp	short input
023.eqntott	int_pri_3.eqn	
026.compress	in	
072.sc	loada3	
085.gcc	cexp.i	
013.spice2g6	greycode.in	short input
015.doduc	doducin	
034.mdljdp2	mdlj2.dat	MAX_STEPS=250
039.wave5		
047.tomcatv		N=129
048.ora	params	ITER=15200
052.alvinn		NUM_EPOCHS=50
056.ear	args.short	
077.mdljsp2	mdlj2.dat	MAX_STEPS=250
078.swm256	swm256.in	ITMAX=120
089.su2cor	su2cor.in	short input
090.hydro2d	hydro2d.in	short input
093.nasa7		
094.fpppp	natoms	short input

5.1 Performance Improvement

Figure 7 shows the improvement of processor performance. The IPC 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 useful instructions are considered for counting the IPC. Next bar indicates the performance of the model performing the load address prediction. Next presents the performance of the model performing the store address prediction and speculative memory disambiguation. The remaining bar shows the performance of the model performing the load and store address prediction and speculative memory disambiguation.

As can be seen in Figure 7, the data dependence speculation with load address prediction is very effective. The performance improvement is up to 23.6%. This explains that the data dependence path including load instructions is one of the critical paths limiting the exploitation of the ILP.

On the other hand, the speculative memory disambiguation using store address prediction has little contribution. The improvement of the ILP is less than 5%. Furthermore, in the cases of `023.eqntott`, `052.alvinn`, `078.swm256`, and `094.fpppp`, the processor performance is slightly degraded. The reason why the speculative store resolution is not effective is as follows. The recovery process is performed whenever the predicted store address is not correct. However, it is not always necessary. When the store address prediction fails, there are four situations according that there are address reference conflicts or not. 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

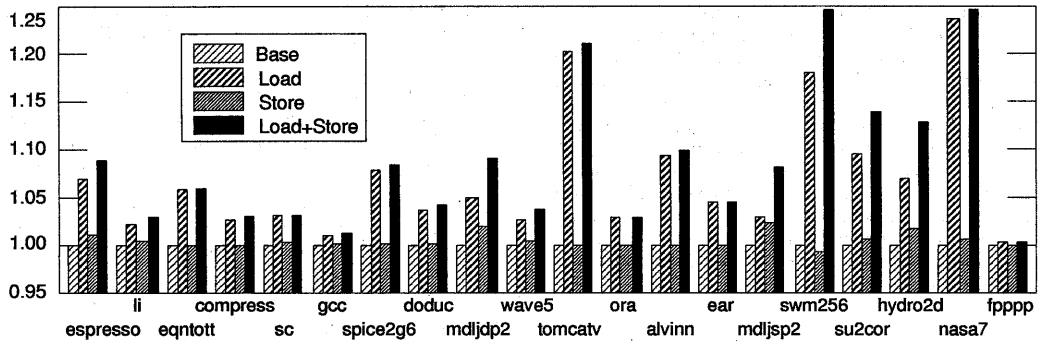

Figure 7: Performance Improvement

conflicts, and (iv) the misprediction results in no conflict and actually there at least one conflict. Only 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 store address prediction is not effective when it is performed alone, however, the effectiveness is increased when it is combined with the load address prediction. Especially, `078.swm256` and `089.su2cor` are improved dramatically by combining the load and store predictions. The performance improvement when the speculative memory disambiguation is combined with the load/store address prediction is up to 24.7%.

5.2 Prediction Accuracy

Table 3 shows the prediction accuracy for each programs. The first column shows the name of each benchmark program. Next two columns indicate the address prediction accuracies when only load instructions are predicted and when only store instructions are predicted. The last two columns show the accuracy when load and store address predictions are combined. The prediction accuracy is significantly high, even for the simple prediction mechanism of the RPT described in Section3. In general, the prediction for the floating point programs are more accurate than that for the integer programs. However, there seem to be no correlation between the performance improvement and the prediction accuracy. In order to investigate the relation between the performance improvement and the prediction accuracy, further study evaluating the ratio of the useful speculative execution per all memory operations is needed.

Table 3: Prediction Accuracy

Benchmark	Load	Store	Load/Store	
			Load	Store
008.espresso	91.34	88.99	91.33	89.03
022.li	86.74	82.50	86.55	93.93
023.eqntott	92.69	88.29	92.70	86.70
026.compress	74.26	86.04	74.96	84.78
072.sc	86.69	83.74	86.78	84.35
085.gcc	84.78	89.67	84.89	89.60
013.spice2g6	97.57	96.17	97.56	96.37
015.doduc	92.78	92.31	92.81	92.51
034.mdljdp2	91.70	94.03	91.70	94.08
039.wave5	98.13	99.23	98.13	99.23
047.tomcatv	98.49	99.33	98.49	99.34
048.ora	99.15	99.99	99.15	99.99
052.alvinn	99.51	99.51	99.51	99.53
056.ear	97.11	96.58	97.11	96.68
077.mdljsp2	89.97	93.33	90.04	93.57
078.swm256	99.33	99.38	99.34	99.38
089.su2cor	99.09	99.03	99.07	98.95
090.hydro2d	88.24	91.83	88.31	91.81
093.nasa7	99.28	99.96	99.28	99.96
094.fpppp	96.24	97.54	96.23	97.17

6 Concluding Remarks

We have proposed the speculative execution for data dependences. By combining the memory disambiguation and the 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 RPT has enough accuracy to improve the processor performance. When only load instructions are speculatively executed, the improvement is up to 23.6%. This explains that the data dependence path including load instructions is one of the critical paths limiting the exploitation of the ILP. On the other hand, the speculative memory disambiguation using store address prediction has little contribution. The improvement of the ILP is less than 5%. In order to make speculative store address resolution to be effective, it is necessary to study the precise detecting scheme for mispredicted memory disambiguation. When the load and store predictions are combined with each other, that is when the memory disambiguation and the address prediction are combined with each other, the improvement of processor performance is up to 24.7%. It confirms that the proposed method is useful.

One of the future study dealing with the data de-

pendence speculation is the speculative execution of store instructions. For that purpose, special hardware support such as ARB[7] is required. The study of the trade-off between the hardware cost of the ARB and the performance improvement is necessary. From the experimental results, the prediction accuracy for integer programs are relatively low. It should be investigated alternative address prediction methods such as the scheme based on the register specifier buffer[19]. The selective squashing scheme is also under investigation. If the instruction which must not be squashed are reused, the effectiveness of the data dependence speculation can be increased.

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, S.Bennett, "Evaluating Future Microprocessors: the SimpleScalar Tool Set", Technical Report CS-TR-96-1308, Computer Science Department, University of Wisconsin Madison, July 1996.

[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] R.J.Eickemeyer, S.Vassiliadis, "A Load-Instruction Unit for Pipelined Processors", IBM Journal of Research and Development, vol.37, no.4, pp.547-564, July 1993.

[5] J.R.Ellis, "Bulldog: A Compiler for VLIW Architectures", The MIT Press, 1986.

[6] M.Franklin, "The Multiscalar Architecture", Ph.D. Dissertation, Technical Report CS-TR-93-1196, Computer Science Department, University of Wisconsin Madison, Nov. 1993.

[7] 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, 1996.

[8] D.M.Gallagher, W.Y.Chen, S.A.Mahhlke, J.C.Gyllenhaal, W.W.Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer", Proc. of Architectural Support for Programming Languages and Operation Systems VI, pp.183-195, 1994.

[9] M.Golden, T.N.Mudge, "Hardware Support for Hiding Cache Latency", Technical Report CSE-TR-152-93, Department of Electrical Engineering and Computer Science, University of Michigan, Feb. 1993.

[10] A.S.Huang, G.S.Slavenburg, J.P.Shen, "Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation", Proc. of 21st Ann. Int'l Symp. on Computer Architecture, pp.200-210, 1994.

[11] R.M.Keller, "Look-Ahead Processors", ACM Computing Surveys, vol.7, No.4, pp.177-195, Dec. 1975.

[12] 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.

[13] 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.

[14] A.I.Moshovos, S.E.Breach, T.N.Vijakumar, G.S.Sohi, "A Dynamic Approach to Improve the Accuracy of Data Speculation", Technical Report CS-TR-1316, Computer Science Department, University of Wisconsin Madison, March 1996.

[15] 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, 1997.

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

[17] H.Nishimoto, A.Katsuno, Y.Kimura, "Improving Instruction Level Parallelism using Load Address Predicting Techniques", IPSJ SIG Notes, 96-ARC-119, pp.49-54, 1996.

[18] 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.

[19] T.Sato, "Data Dependence Path Reduction with Tunneling Load Instructions", Submitted to Int'l Symp. on High Performance Computing, Nov. 1997.

[20] 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.

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

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

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