Data Dependence Speculation using Data Address Prediction and its Enhancement with Instruction Reissue

Toshinori Sato
Toshiba Microelectronics Engineering Laboratory
580-1, Horikawa-Cho, Saiwai-Ku, Kawasaki 210-8520, Japan
toshinori.sato@toshiba.co.jp

Abstract

In this paper, we introduce an instruction reissue mechanism in order to enhance dynamic data dependence speculation using data address prediction. Since instructions which are not data-dependent upon speculatively executed instructions are not squashed, the effect of data dependence speculation is enhanced. We extend register update unit to reissue misspeculated instructions. The overhead caused by the extension is small, and thus it does not have any impact on processor cycle time. From the experimental evaluation, we have found that the instruction reissue with dynamic data dependence speculation improves processor performance even for the application programs whose performance is degraded when instruction squashing is used.
Keywords: instruction level parallelism, out-of-order execution, dynamic speculation of data dependence, address prediction, instruction reissue

1 Introduction

Dependences are the obstacles disturbing the processor performance. They include control, name, and data dependences. The control dependence attracts many researchers and is attacked by the techniques such as branch prediction and speculative execution. The name dependence is caused by the resource shortage and can be eliminated using register renaming. However, the data dependence can not be removed by such techniques, as it is called *true dependence*. Hence, the data dependence is the serious obstacle limiting instruction level parallelism (ILP).

Recently, several dynamic data dependence speculation schemes have been proposed[7, 8, 16, 17, 19, 22]. They predict data addresses of load and store instructions, and speculate these instructions and their dependent instructions. If a speculation is mispredicted, a processor has to recover its state. A straightforward implementation of the recovery mechanism is instruction squashing which is currently used to correct processor state when a control dependence is misspeculated. The instruction squashing is not adequate for the data dependence speculation, because it throws away execution results of instructions which are independent of the misspeculated instructions and because these instructions should be fetched again from instruction memory. This wastes the useful computations. Moreover, the overhead including the instruction squashing and refetching is very serious since penalty caused by data dependence misprediction is quite larger than that caused by control dependence misprediction. We have evaluated data dependence speculation using the instruction squashing, and found that processor performance is degraded for several programs[17]. Therefore, the penalty caused by a misprediction of data dependence speculation should be reduced, and the instruction reissue is expected to be one of the promising solutions for the problem. In this paper, we present a practical implementation of the instruction reissue. We extend register update unit (RUU)[20] to perform the instruction reissue[18]. The modification is very slight and thus hardware overhead is small.

The organization of the rest of this paper is as follows. Section 2 surveys previously proposed related works. Section 3 explains a data dependence speculation scheme using data address prediction. Section 4 describes an instruction reissue mechanism extending the RUU. In Section 5, the evaluation methodology is presented and simulation results are shown in Section 6. Finally, our conclusions are presented in Section 7.

2 Related Work

There are many proposals predicting data addresses of load instructions [1, 4, 6, 15, 19, 22]. The goal of these studies is not to execute the load instructions speculatively but to execute load instruction at earlier pipeline stage. Thus, most of them[1, 4, 6, 15] do not speculatively execute instructions following the load instructions. The others [19, 22] execute speculatively the following instructions, but do not take advantage of store address prediction. Recent studies propose the data address prediction of store instructions [7, 8, 16, 17]. Gonzalez et al.[7, 8] have evaluated the speculative execution of both load and store instructions using the data address prediction. The store operations are speculated with the help of the address resolution buffer[5]. In [16, 17], the predicted data addresses are used for resolving ambiguous memory dependences speculatively. If the (predicted) data address of a preceding store instruction is different from that of a succeeding load instruction, the load instruction can be executed beyond the store instruction in out-of-order fashion. The data address prediction method utilized in this paper is based on [16, 17].

Moshovos et al.[13] 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.

Instruction reissue is one of the promising solutions for the problems caused by mispredicted data dependence speculation. Lipasti et al.[11, 12] introduced the instruction reissue scheme. The instructions dependent on a predicted instruction are forced to retain in the reservation station. When the predicted instruction produces an actual value, the predicted value must be compared with the actual one. If they match, the prediction is correct and the dependent instruction release the reservation station. If the prediction fails, the dependent instructions are invalidated and reissued. However, they only have proposed the concept of the instruction reissue. They have not presented any practical implementation of the scheme. If the scheme were implemented, the processor cycle time would increase since it would be very difficult to find all dependent instructions in parallel. The dependent instructions would have to be searched serially. This causes either the increase of the cycle time or that of the miss penalty, each of which degrades the processor performance. Gonzalez et al.[7] also have proposed the instruction reissue scheme, which is very similar to [11, 12]. When a prediction is verified, all dependent instructions are committed if the prediction is correct. Otherwise, they are reissued. Gonzalez et al. also have not presented any practical mechanism but only described the concept. Tyson et al.[21] have evaluated the usefulness of the instruction reissue. They have proposed a streamlining load value scheme and found that the instruction reissue proposed by Lipasti et al.[11, 12] can improve processor performance even for the applications whose performance is degraded when the instruction squashing is used. However, the practical implementation of instruction reissue has not been discussed. Rotenberg et al.[14] have investigated an instruction reissue scheme in Trace Processor architecture and found it is useful for dataflow speculation. However, they do not evaluated it in superscalar processors which are currently in mainstream.

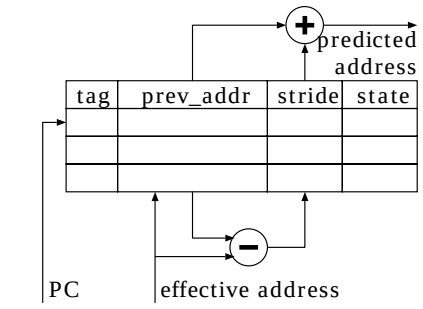
In this paper, we present a practical implementation of the instruction reissue. The mechanism introduces only small hardware overhead and does not suffer from the increase of the processor cycle time.

3 Data Dependence Speculation Mechanism

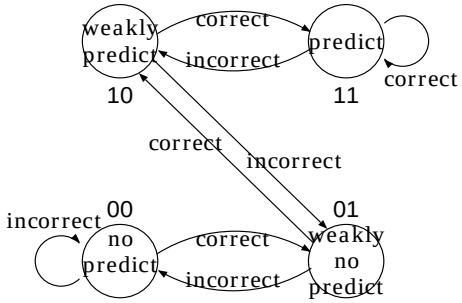
In this section, we describe a data dependence speculation mechanism[16, 17]. In order to predict effective addresses, we utilize the reference prediction table (RPT)[3]. First, we explain the RPT. Next, we describe the data speculation using address prediction. And last, we explain how to speculatively resolve the ambiguous memory aliasing.

3.1 Reference Prediction Table

The RPT, which has a similar structure with the 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. Figure 1a shows the RPT structure. 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*). 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. An example of 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]. It is quite similar to the two bit saturated counter (2bC) used by branch predictors. The reason why we choose the 2bC scheme is that the RPT using the 2bC state machine can more aggressively speculate the data dependences than that using the original state machine[17]. There are four states, which are *predict*, *weakly predict*, *no-predict*, and *weakly no-predict*.



(a) Reference Prediction Table



(b) State Transition

Figure 1: Reference Address Prediction

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. 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.

3.2 Data Speculation with Address Prediction

Next, 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

when an instruction is issued into the instruction window. For the purpose of the aggressive speculation, 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 2[1].

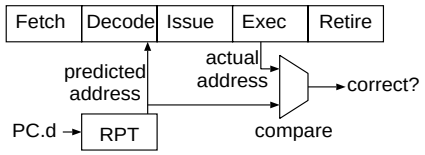


Figure 2: Pipeline with RPT

The RPT indexed by the PC is accessed during the decode stage. The predicted address corresponds with the load instruction and reserved until the execution stage. It is assumed that instruction window could hold not only operand values but also predicted values. The structure of the instruction window will be described in Section 4.

A load instruction is speculatively executed using the predicted address. When the actual address is generated in the execution stage, it must be compared with the predicted one. The comparison is performed between the actual address and the predicted one held in the instruction window. In the case that two addresses match, the prediction succeeds. If the actual address is different from the predicted one, the misprediction occurs and the recovery action has to be performed. The instructions dependent upon the mispredicted load instruction are squashed as shown in Figure 3. 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 3a), and the other is invalidating selectively the instructions which are dependent upon the mispredicted load instruction and reissuing them (Figure 3b). We call the former scheme instruction squashing, and the latter one instruction reissue. Even though the instruction squashing has the penalty to redundantly execute independent instructions, it is simpler and easier to implement than the instruction reissue. Actually, it is possible for the instruction squashing to utilize the recovery mechanism of the branch misprediction. The practical implementation of the instruction reissue will be explained in Section 4.

```
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 3: Mispredicted Load Instruction

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

3.3 Speculation of Ambiguous Memory Aliasing

Lastly, we explain the data dependence speculation by combining the memory disambiguation with address prediction. The effective address of an unresolved store instruction is predicted and the ambiguous memory dependences are 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 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 ambiguous memory dependences speculatively and to execute the following load instructions dependent upon the store instruction. If a load address is different from the predicted store address, the load instruction can be executed speculatively. The diagram is as same as that of load address prediction shown in Figure 2.

```
@ 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 4: Mispredicted Store Instruction

When a speculative resolution fails, the recovery action must be performed. The probable dependent instructions have to be squashed. Let us see Figure 4. The instructions marked with @ refer to same memory location, and those marked with * are the squashed instructions. The schemes are (i) the scheme squashing all instructions following the mispredicted store instruction (Figure 4a), (ii) the scheme squashing all instructions following the load instruction whose data address conflicts with mispredicted store address (Figure 4b), and (iii) the scheme invalidating selectively the instructions which is truly dependent upon the store instruction (Figure 4c). Even though the first one has the penalty to redundantly 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. The second squashing scheme is as follows. It squashes all instructions following the load instruction whose address conflicts with the actual store address. The store address misprediction does not always cause the squashing action. Owing to this strategy, useless squashing action is eliminated. Let us compare the scheme with the first one. When a store address prediction fails, there are four situations according that the address reference conflicts are caused 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 conflicts, and (iv) the misprediction results in no conflict and actually there is at least one conflict. Only case (iv) needs the recovery action. 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 proposed scheme solves this problem. The second scheme can be implemented by utilizing address reorder buffer of HP PA-8000[9]. The address reorder buffer detects if a pair of a store and a load instructions refer the same memory location. And if so, all instructions following the load instruction which speculates beyond the store instruction are squashed. The third scheme is implemented by modifying the address reorder buffer. The address reorder buffer only detects a load instruction which fails a speculation. After detecting the load instruction, the instruction reissue explained in Section3.2 is performed. The load instruction and its dependent instructions are selectively invalidated and reissued.

4 Instruction Reissue Mechanism

In this section, we describe the instruction window extending the RUU in order to perform instruction reissue[18].

4.1 Extended Register Update Unit

When an address prediction is performed, the predicted address and the load value read from data memory using the predicted address should be kept in the instruction window. When an actual address is produced, it should be compared with the predicted one. If they match, the prediction is correct. Otherwise, the prediction fails and the processor state must be corrected. All instructions dependent upon the mispredicted instruction should be reissued. In order to support the instruction reissue, we propose to extend the RUU. The RUU is very suitable for the instruction reissue, since it forces each instruction to retain until the instruction has been committed. The extended RUU for instruction reissue is depicted in Figure 5. An entry of the instruction window consists of two source operand fields, a destination field, a dispatched bit, a functional unit field, an executed bit, a data address field, a predicted bit, a reissued bit, and a program counter field. If a source operand is not ready, the ready bit is reset to indicate that the source operand is not available[2] and a tag for the operand is obtained. When the operand is ready, the content of the source register is held in the source operand field and the ready bit is set. The destination register number with renaming tag is held in the destination field, and an execution result is held in the destination field when the execution completes. The destination field is also used to keep the load value read from data memory using the predicted address. The dispatched bit indicates if the instruction is dispatched into a functional unit which is specified by the functional unit field. The executed bit is set when the execution finishes. If the executed bits is set, the following instructions which are dependent upon the instruction can obtain a source operand. The data address field holds a data address when the instruction is a load/store instruction. Note that the overhead caused by the data address field can be reduced if the instruction window is divided into two structures, one of which holds load/store instructions, and the other of which holds the remaining instructions. The data address field is necessary only for the former one, i.e. load/store queue. The predicted bit indicates if the corresponding instruction speculates data dependence. When a address prediction is initiated, the predicted bit is set. The data value obtained from data memory using the predicted address is held in the content slot of the destination field. The reissued bit indicates if the corresponding instruction was reissued. The reissued bit is also set when an address prediction is incorrect. Lastly, the program counter field is used for correction of mispredictions and for precise interrupts.

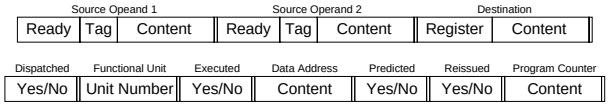


Figure 5: Extended RUU Entry

When an instruction produces an execution result, its destination tag is broadcasted as same with the original RUU. Simultaneously, the signal indicating if a reissued bit is set, is also broadcasted. Here, we call this signal the misprediction signal. When the prediction is not correct the misprediction signal is asserted. The correctness of the prediction is decided by comparing the predicted and actual addresses. In the case that the misprediction signal is not asserted, the following instructions obtains their source operands if their source and the broadcasted destination tags match. The instructions whose source operands are obtained are ready to be dispatched. In the case that the misprediction signal is asserted, the following instructions should be invalidated and reissued if their source and the broadcasted destination tags match and if the corresponding dispatched bit is set. These instructions have been already dispatched using a wrong operand value. The corresponding dispatched and executed bits should be reset, and the reissued bit is set. The mispredicted instruction has produced the actual value, and thus the reissued instructions have obtained the correct source operands and are ready to be dispatched. When a reissued instruction finishes, the mispredicted signal is asserted. Namely, the mispredicted signal indicates if the mispredicted or reissued instructions have finished their execution. The following process is as same as above. The instructions dependent on the reissued instruction should also be reissued. In this manner, all instructions dependent upon a mispredicted instruction are searched and reissued gradually.

This scheme introduce only one signal into the dynamic instruction scheduling mechanism. The signal decides the action resulted from an associative search which examines if the source and destination tags match. The signal is asserted when an address is mispredicted or when a reissued instruction finishes. If the signal is not asserted, the dynamic instruction scheduling mechanism conduct usual process as same to the original RUU. Otherwise, it detects instructions which should be reissued. This mechanism is very simple, and hence it only requires slight hardware overhead and it does not increase the processor cycle time. It is logically possible to search and detect all instructions dependent on a mispredicted instruction concurrently. However, it is very difficult to implement such a mechanism without causing the increase of the processor cycle time, which degrades the

[2] In [20], the ready bit is set if a source operand is not ready. However, in this paper, we set the ready bit when the source operand is obtained in order to make explanations clear.

processor performance.

It is true that an instruction might be dispatched before it is detected to be reissued even if a misprediction is detected and hence that the effectiveness of functional units usage might be reduced. However, we expect that instructions which are ready to be dispatched into a load and store unit using predicted addresses are not always dispatched before the actual addresses are calculated. Therefore, we do not worry about the performance degradation caused by the redundant dispatching.

4.2 Dynamic Scheduling Example

An example is useful to understand the process of the instruction reissue. Figure 6 shows an instruction sequence. For easy understanding, followings are assumed. First, each operation of $f_1 - f_6$ refers only one source operand. Second, instruction I3 is a load instruction and its execution latency is two cycles, which consists of one cycle for address generation and an another cycle for memory access. And last, the execution latency of the remaining operations is one. The instruction sequence has to be executed serially due to data dependences, and hence it takes eight cycles to execute the sequence after decoding instruction I1 when any data dependences are not speculated.

$$\begin{aligned}
&\texttt{I1: } r11 \leftarrow f_1(r10)\\
&\texttt{I2: } r12 \leftarrow f_2(r11)\\
&\texttt{I3: } r13 \leftarrow f_3(r12)\\
&\texttt{I4: } r14 \leftarrow f_4(r13)\\
&\texttt{I5: } r15 \leftarrow f_5(r14)\\
&\texttt{I6: } r16 \leftarrow f_6(r15)
\end{aligned}$$

Figure 6: Instruction Sequence Example

Figure 7 illustrates an example of instruction reissue. The processor is assumed as follows. First, its issue width is three and it has the unlimited number of functional units. Second, the commit process is omitted. Third, instruction I3 predicts a data address and its dependent instructions speculate data dependences. Forth, it is also assumed that the destination register number with the renaming tag is same to the architectural register number. And last, $r10$ is assumed to be ready. Now let us explain the example. First, instructions I1 – I3 are issued to the RUU. I1 is dispatched to a functional unit since $r10$ is ready, and it sets the ready (r) and dispatched (d) bits. I3 predicts the data address and sets the predicted (p) bit. The predicted address is held in the data address field. At next cycle, instructions I4 – I6 are issued. I1 has finished its execution and $r11$ becomes ready, thereby I2 is dispatched. The thick arc indicates that the destination and source tags match. I3 obtains $r13$ from the data memory which is kept in the corresponding destination field, and I4 is dispatched using $r13$. I1 and I3 set their corresponding executed (e) bits. At third cycle, I2 and I4 have finished and $r12$ and $r14$ become ready, thereby I3 and I5 are dispatched. At next cycle, there are two situations. One is when the predicted address of I3 is correct, and the other is when I3 fails in the prediction. If the prediction is correct, I3 completes and resets the predicted bit. I5 has finished and I6 is dispatched. And finally, the execution of the sequence finishes in five cycles. Comparing with the execution without data dependence speculation, the execution cycle is reduced by three cycles. On the other hand, if the prediction fails, the execution is as follows. The predicted and reissued (i) bits corresponding

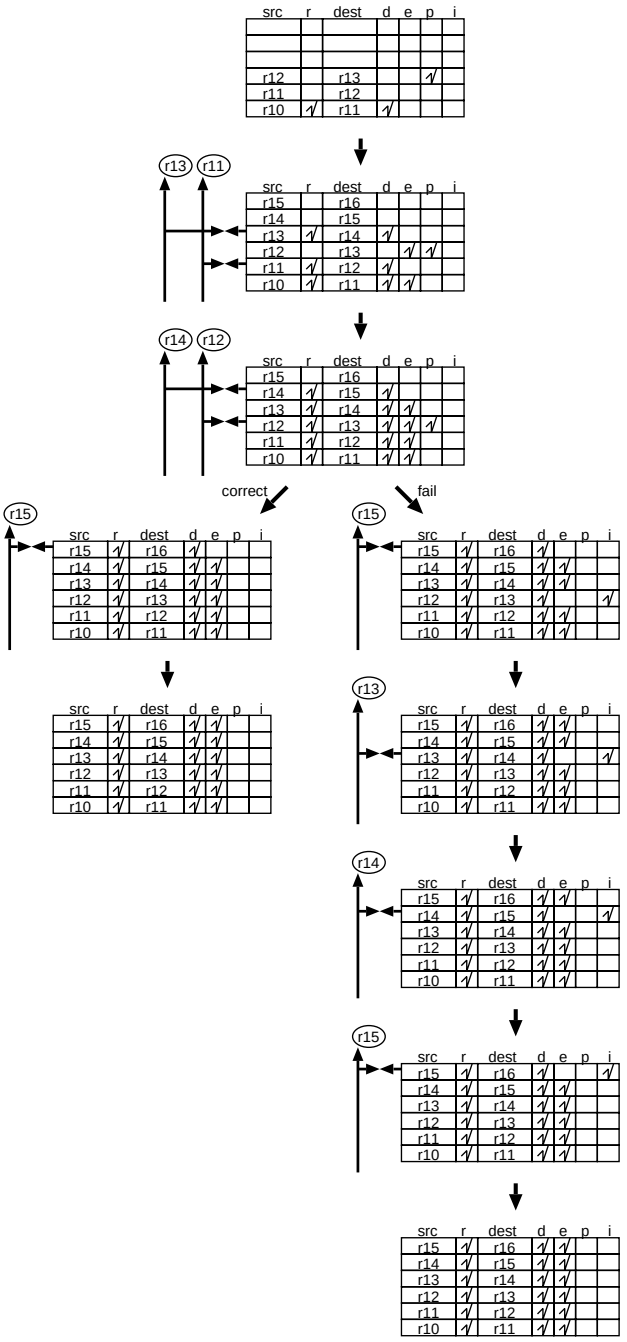


Figure 7: Instruction Reissue with RUU

to I3 are reset and set, respectively. I5 has finished and I6 is dispatched. At next cycle, I3 completes and the actual $r13$ becomes ready. I3 broadcasts the misprediction signal, since its associated reissued bit is set. It is detected that I4 has been dispatched using a wrong operand, because its source register tag matches with the broadcasted destination tag and because its dispatched bit is set. Therefore, I4 should be reissued and its reissued bit is set. At sixth cycle, I4 has finished and I5 is reissued due to the same reason why I4 was reissued. Next, I5 has finished and I6 is reissued. And lastly at next cycle, I6 finishes. The number of

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

the execution cycle when the misprediction occurs is eight, which is as same as that in the case when the original RUU is used. Fortunately in this example, misprediction does not occur any miss penalties. In actual computing, there should be cases when reissued instructions can not be dispatched immediately after detected to be reissued, and hence the processor may suffer some miss penalties.

From the explanation above, it can be seen that implementing the instruction reissue scheme is possible with slight overhead and that the mechanism works effectively.

5 Evaluation Methodology

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

5.1 Processor Model

We evaluated the effect of the instruction reissue mechanism by using SimpleScalar tool set[2]. The SimpleScalar architecture is based on MIPS architecture, and a fully execution-driven and cycle-by-cycle simulator is executed on a SPARC-station. In order to consider the effect of speculations, an execution-driven simulation is more desirable than a trace-based simulation, since trace-based simulators can not simulate misspeculated instructions. The baseline model is an out-of-order execution superscalar processor based on the RUU[20]. Following the discussion explained in [10], we decide the configuration of the baseline processor summarized in Table 1. The RPT is assumed to be direct-mapped and to have 1024 entries. If the instruction squashing scheme is used, the penalty for squashing the instructions 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.

5.2 Workload

The SPEC95 benchmark suite is used for this study. The `test` input files which are provided by SPEC are used. We used the object files provided by University of Wisconsin Madison[2]. Each program was executed to completion or for the first 100 million instructions. For measuring performance, we use the committed instruction per cycle (IPC).

6 Experimental Results

This section presents the experimental results. First, we present address prediction accuracy. Next, we show the effect of the instruction squashing. Last, we present performance improvement when the instruction reissue is utilized.

6.1 Address Prediction Accuracy

Table 2 shows the address prediction accuracy. We define the prediction accuracy as the cases when the prediction succeeds over the cases when the speculation is initiated. 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 remaining columns show the load and store address prediction accuracy for the `ldst` model, respectively. The prediction accuracy is significantly high, even for the simple prediction mechanism of the RPT. In general, the prediction for the floating point programs are more accurate than that for the integer programs.

Table 2: (%)Address Prediction Accuracy

program	load	store	ldst	
			load	store
099.go	88.25	95.37	87.93	97.36
124.m88ksim	89.20	96.31	88.33	95.41
126.gcc	93.73	93.83	94.10	94.80
129.compress	96.19	99.80	96.47	99.86
130.li	86.62	89.95	85.60	92.49
132.ijpeg	99.55	99.58	99.59	99.66
134.perl	95.65	83.69	96.47	86.83
147.vortex	94.38	92.79	95.27	94.27
101.tomcatv	99.53	99.99	99.78	99.99
102.swim	99.99	99.99	99.99	99.99
103.su2cor	99.08	99.08	99.41	99.69
104.hydro2d	99.27	99.90	99.76	99.91
107.mgrid	97.53	97.18	97.72	97.13
110.applu	83.05	81.19	83.02	78.98
125.turb3d	97.75	96.22	97.83	96.61
141.apsi	96.50	94.86	97.01	96.34
145.fpppp	98.74	98.93	98.48	96.57
146.wave5	98.99	99.93	98.96	99.93

6.2 Performance Impact of Instruction Squashing

Figure 8 shows the processor performance when the instruction squashing is used[17]. The IPCs of the evaluated models are normalized by that of the baseline model. For each group

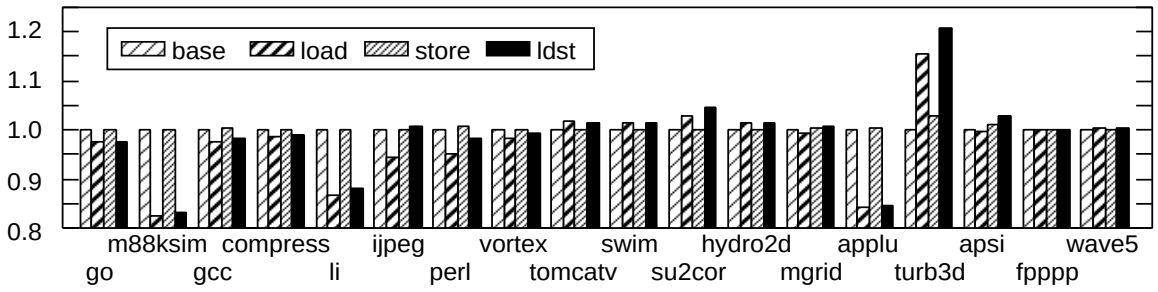


Figure 8: Instruction Squashing

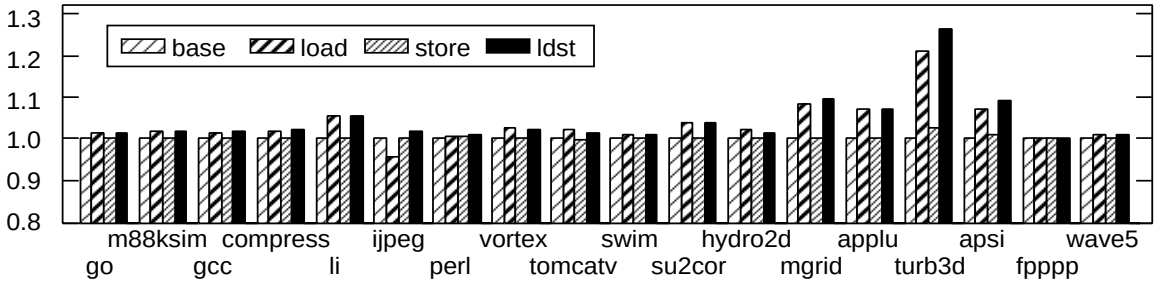


Figure 9: Instruction Reissue

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. Table 3 in Appendix shows the detailed data.

As can be seen in Figure 8, the data dependence speculation with load address prediction hardly contributes to processor performance. For eleven of eighteen programs, processor performance is degraded. On the other hand, the speculative resolution of ambiguous memory aliasing using store address prediction slightly contribute to processor performance. While the improvement of ILP is less than 3%, it always improves processor performance except `tomcatv`. And lastly, the simulation results of the `ldst` model has a similarity with those of the `load` model.

6.3 Performance Impact of Instruction Reissue

Figure 9 shows the processor performance when the instruction reissue is used. Layout and subject matter of Figure 9 are the same as for Figure 8. Table 3 in Appendix shows the detailed data.

Different from Figure 8, most programs increase processor performance in `load` model case, except for `ijpeg`. This means that speculatively executing load instructions using address prediction becomes effective with the help of the instruction reissue. Processor performance is increased by up to 21.0%. The significant improvement from the evaluation using the instruction squashing is seen in `m88ksim`, `li`, and `applu`. In these programs, processor performance is severely decreased using the instruction squashing as can be seen from Figure 8. However, if the instruction reissue is utilized, it is improved by up to 7.2%. In `store` model case, the improvement from the evaluation using the instruction squashing can not be seen. This is because the cases that the instruction squashing (reissue) occurs due to misspeculated resolution of ambiguous address aliasing are rare. It is generally known that the ambiguous address aliasing rarely happen to true address conflict. In `ldst` model case, processor performance increases for all programs, and its improvement rate is up to 26.5%. This also confirms that the instruction reissue enhances data dependence speculation. For several programs, the improvement rate is smaller than that of `load` model. This is because the RPT entries assigned to load instruction is reduced due to their assignments to store instructions. The speculative resolution of ambiguous memory aliasing using store address predictions has little contribution to processor performance, and thus the reduction of the RPT entries assigned to load instruction decreases the performance improvement.

In summary, it is confirmed that the instruction reissue is effective to enhance the data dependence speculation.

7 Concluding Remarks

In this paper, we have described a practical implementation of the instruction reissue contributing to data dependence speculation. We have evaluated data dependence speculation using address prediction, and found that the instruction squashing scheme diminishes the effect of data dependence speculation but the instruction reissue does not. Utilizing the instruction reissue, processor performance is improved for almost all the cases, and its improvement rate is up to 26.5%. The results confirms the usefulness of the proposed instruction reissue mechanism.

Currently we are evaluating a load value predictor[18] attached with the data address predictor. Combining the address and value predictors increases the number of predicted instructions, thereby data speculation is further enhanced.

Acknowledgment

The author is grateful to Dr. Mitsuo Saito and Dr. Shigeru Tanaka for their continuous encouragements. He also thanks the anonymous reviewers whose comments and suggestions helped to improve the quality of this paper.

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, 1997.

[3] T-F.Chen, J-L.Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors", IEEE Trans. Comput., vol.44, no.5, pp.609-623, 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, 1993.

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

[6] 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, 1993.

[7] J.Gonzalez, A.Gonzalez, "Speculative Execution via Address Prediction and Data Prefetching", Proc. of 11th Int'l Conf. on Supercomputing, pp.196-203, 1997.

[8] J.Gonzalez, A.Gonzalez, "Memory Address Prediction for Data Speculation", Proc. of EuroPar Conf., 1997.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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, June 1997.

[14] E.Rotenberg, Q.Jacobson, Y.Sazeidas, J.Smith, "Trace Processors", Proc. of 30th Ann. Int'l Symp. on Microarchitecture, pp.138-148, 1997.

[15] T.Sato, H.Fujii, S.Suzuki, "Hiding Data Cache Latency with Load Address Prediction", IEICE Trans. Inf. & Syst., vol.E79-D, no.11, pp.1523-1532, Nov. 1996.

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

[17] T.Sato, "Speculative Resolution of Ambiguous Memory Aliasing", Proc. of Int'l Workshop on Innovative Architecture for Future Generation Parallel Processors and Systems(IEEE Computer Society Press), pp.17-26, Oct. 1997.

[18] T.Sato, "Load Value Prediction using Reference Address Renaming", Proc. of 10th Joint Symp. on Parallel Processing, pp.15-22, June 1998 (In Japanese).

[19] 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.

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

[21] G.Tyson, T.M.Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming", Proc. of 30th Ann. Int'l Symp. on Microarchitecture, pp.218-227, 1997.

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

Appendix

Table 3 presents performance improvement rate. We define the performance improvement rate as the increased IPC over the IPC of the baseline model. The first column shows the name of each benchmark program. Next three columns indicate the improvement rate of the squashing model, and the remaining three columns indicate that of the reissued model. For each group of three columns, the columns from left to right show the improvement rate of the `load`, `store`, and `ldst` models, respectively.

Table 3: (%)Performance Improvement

program	squash			reissue		
	load	store	ldst	load	store	ldst
099.go	-2.59	0.01	-2.34	1.54	0.01	1.50
124.m88ksim	-17.5	0.10	-16.8	2.07	0.10	1.79
126.gcc	-2.68	0.30	-1.85	1.61	0.29	2.03
129.compress	-1.33	0.07	-0.84	2.15	0.07	2.37
130.li	-13.3	0.17	-11.8	5.39	0.17	5.36
132.ijpeg	-5.70	0.15	0.88	-4.48	0.15	2.19
134.perl	-4.67	0.80	-1.58	0.82	0.80	1.28
147.vortex	-1.63	0.05	-0.63	2.82	0.05	2.43
101.tomcatv	1.66	-0.01	1.56	2.50	-0.01	1.50
102.swim	1.32	0.00	1.32	1.32	0.00	1.32
103.su2cor	2.78	0.01	4.37	3.96	0.01	4.09
104.hydro2d	1.43	0.04	1.47	2.41	0.04	1.40
107.mgrid	-0.60	0.46	0.92	8.41	0.46	9.61
110.applu	-15.6	0.43	-15.4	7.16	0.43	7.25
125.turb3d	15.7	2.79	21.0	21.0	2.79	26.5
141.apsi	-0.25	1.14	2.95	7.01	1.13	9.14
145.fpppp	0.02	0.00	0.01	0.08	0.00	0.06
146.wave5	0.22	0.00	0.22	1.20	0.00	1.16