IEICE TRANS. INF. & SYST., VOL.E93–D, NO.12 DECEMBER 2010

PAPER

Register File Size Reduction through Instruction Pre-Execution Incorporating Value Prediction

Yusuke TANAKA[†*], *Nonmember and* **Hideki ANDO**[††a)], *Member*

SUMMARY *Two-step physical register deallocation* (TSD) is an architectural scheme that enhances memory-level parallelism (MLP) by pre-executing instructions. Ideally, TSD allows exploitation of MLP under an unlimited number of physical registers, and consequently only a small register file is needed for MLP. In practice, however, the amount of MLP exploitable is limited, because there are cases where either 1) pre-execution is not performed; or 2) the timing of pre-execution is delayed. Both are due to data dependencies among the pre-executed instructions. This paper proposes the use of value prediction to solve these problems. This paper proposes the use of value prediction to solve these problems. Evaluation results using the SPECfp2000 benchmark confirm that the proposed scheme with value prediction for predicting addresses achieves equivalent IPC, with a smaller register file, to the previous TSD scheme. The reduction rate of the register file size is 21%.

key words: *microarchitecture, microprocessor, instruction pre-execution, value prediction, register file*

1. Introduction

Supporting many in-flight instructions allows aggressive exploitation of instruction-level parallelism (ILP) and memory-level parallelism (MLP), leading to enhanced performance. The exploitation of MLP is especially effective in memory-intensive programs. Providing support for many in-flight instructions, however, requires a large register file, which, in turn, adversely affects the clock cycle time due to the longer access time. Although this adverse effect can be alleviated by pipelining, this complicates the bypass logic. In addition, having a deep pipeline increases the branch misprediction penalty, lowering IPC. Since it is difficult to remove the adverse effect of a large register file completely, it is important to reduce the register file size without degrading the performance.

Two-step physical register deallocation (TSD) is a novel register renaming scheme [1], [2], that allows the pre-execution of instructions that cannot be executed due to the lack of a physical register in the conventional renaming scheme, thereby exploiting MLP aggressively. The TSD can exploit a large amount of MLP under an infinite number of physical registers, independently of the actual physical register count. Thus, a large register file is not required for exploiting MLP, keeping a high clock frequency.

The TSD deallocates physical registers in two phases: 1) temporal deallocation, which allows the physical register to be allocated to another instruction; and 2) final deallocation, which allows the result write to be granted. The TSD completely removes the pipeline stall in the rename stage, caused by a shortage of physical registers, by deallocating a physical register temporarily and allocating it to an instruction. Such instructions are inserted into the instruction window. While waiting for the temporarily allocated physical register to become available, these instructions are executed (*pre-execution*) if their source operands are available; the result is not written to the physical register. Instead, the result of a pre-executed instruction is passed to its dependent instructions via the bypass logic. Pre-execution can, therefore, be performed continuously, which ensures as early memory accesses as possible in the case of an infinite number of physical registers. Thus, many memory accesses can be overlapped. Later, when the temporarily allocated physical register actually becomes available, the instruction is notified, and executed again (*main execution*). At this time, the result is written to the physical register, as in a normal processor. During the main execution, a load obtains data from the cache, which, without pre-execution, would instead have resided in main memory.

Although the TSD is effective, in some cases instructions are either not pre-executed or not pre-executed early enough, as a result of the following two problems. First, because the result of a pre-executed instruction is passed only by the bypass logic, a consumer cannot be pre-executed unless the producer and its consumer are issued back-to-back. Second, if multiple cache misses occur successively in a particular data dependence chain, the latency of the later misses cannot be avoided sufficiently in the main execution. This is because pre-execution of later loads is delayed.

Both problems stem from data dependences. We introduce value prediction to solve these problems [3]–[5]. We apply value prediction to pre-execution candidate instructions whose physical register is only temporarily allocated. This allows instructions to be pre-executed independently of any precedent instructions. Loads, that would not have been pre-executed or not pre-executed early enough in the conventional TSD, can be pre-executed early enough, thereby improving MLP. As a result, we can further reduce the register file size. Note that, although our approach requires ex-

Manuscript received March 12, 2010.
Manuscript revised July 12, 2010.

[†]The author is with the Department of Computational Science and Engineering, Nagoya University, Nagoya-shi, 464–8603 Japan.
[††]The author is with the Department of Electrical Engineering and Computer Science, Nagoya University, Nagoya-shi, 464–8603 Japan.
[*]Presently, with Denso Corp.
a) E-mail: ando@nuee.nagoya-u.ac.jp

DOI: 10.1587/transinf.E93.D.3294

tra hardware, we aim to reduce critical path length by investing the hardware to off-critical paths.

The study of value prediction was originally aimed at enhancing ILP. Our study differs in that our aim is to enhance MLP. Since pre-execution does not update the state, the proposed scheme does not require recovery from misprediction, unlike the ILP-oriented schemes.

The remainder of this paper is organized as follows. Section 2 describes related work. In Sect. 3, the previously proposed TSD scheme is explained. Section 4 discusses the effect of value prediction when used for instruction pre-execution in the TSD. Evaluation results are presented in Sect. 5, and finally our conclusions are given in Sect. 6.

2. Related Work

2.1 Instruction Pre-Execution

A number of studies focusing on pre-execution have been carried out [6]–[12]. These schemes extract, either statically or dynamically, the instructions necessary to generate memory accesses as a thread. Then they spawn this thread at a certain point in the program execution to a different context of the processor. Unlike these schemes, the TSD does not require a multithreaded environment such as simultaneous multithreading [13] or chip multiprocessors.

To the best of our knowledge, the only pre-execution scheme that does not need a multithreaded environment is *runahead execution* [14]. This scheme enters a special mode called a runahead when an L2 cache miss occurs. In this mode, the architectural state is checkpointed, and instructions following the missed load are executed until the triggered miss is resolved. If another L2 cache miss occurs while in the runahead mode, the missed line is prefetched.

2.2 Enhancing ILP with Value Prediction

Originally, value prediction was employed to enhance ILP by breaking data dependences and executing instructions dependent on the predicted instruction speculatively [15]–[17]. If the prediction is found to be correct, the speculation increases ILP, resulting in speedup. If not, the state must be recovered, and the penalty thereof degrades performance. A basic recovery method squashes all instructions succeeding to the mispredicted instruction, and restarts execution from the mispredicted instruction. More elaborately, an alternative method reissues only those instructions dependent on the mispredicted instruction from the instruction window. Although the squashing recovery is simple, the penalty is large. On the other hand, the penalty for reissue is small, but the mechanism complicates the instruction window. This is because those instructions that depend on the predicted instruction must be selectively controlled. That is, the mechanism must ensure that they are not removed from the instruction window after issue, but are removed after the prediction has been validated.

Reinman *et al.* examined in detail the effectiveness of value prediction (load-result and load-address predictions) with the squash and reissue recovery schemes [18]. Their evaluation results indicate that, because the prediction accuracy is not very high, simple squash recovery achieves only limited speedup; reissue is necessary for solid speedup by reducing the misprediction penalty.

Basically, ILP-oriented value prediction and our scheme are orthogonal, since our scheme is applied only to the instruction pre-execution. However, compared with ILP-oriented value prediction, our scheme has an advantage in that it does not require recovery from misprediction.

2.3 Enhancing MLP with Value Prediction

There are studies similar to ours that use value prediction to enhance MLP rather than ILP.

Zhou *et al.* advocated the use of value prediction to prefetch data [19]. They remove data dependencies (if possible), and prompt instruction execution. Unlike the conventional use of value prediction, the register is not updated, but data is moved from memory to the cache if a speculatively executed load causes a cache miss. Unfortunately, although MLP is improved, the effect is limited. Unlike their study, we advocate value prediction to accelerate instruction pre-execution. In their scheme, instructions are stalled at the rename stage due to a shortage of physical registers, but this does not occur in ours. Therefore, our scheme increases MLP to a greater degree.

Mutlu *et al.* introduced value prediction in the runahead execution [20]. To the best of our knowledge, this is the only study that has incorporated value prediction with instruction pre-execution. The study uses value prediction so that instructions that depend on the triggered L2 missed load can be pre-executed. A new value prediction scheme called *address-value delta* (AVD) was introduced. It predicts the load data from its reference address. Although the study uses value prediction in instruction pre-execution, it is used to remove only a special type of data dependence (that is, the dependence on the triggered L2 missed load). Thus, the AVD predictor is useful only in runahead execution, where the reference address of the load is *known*. On the other hand, where general data dependences must be resolved, as is in the case in our study, the reference address of loads is *unknown*, and thereof the AVD is useless.

2.4 Reducing Register File

Several studies have focused on reducing the size of register files, in an attempt to reduce the occupation time of physical registers by late allocation [21], [22] or early deallocation [23]–[25].

Early deallocation schemes deallocate registers speculatively by predicting the last consumer, and allocate them at the rename stage. Shortcomings of such schemes include the large penalty imposed by misspeculation recovery, and the requirement of a large checkpointing register file that includes shadow storage.

Late allocation schemes do not allocate registers at the rename stage; instead, they are allocated later in the pipeline. The *virtual-physical register scheme* [21] allocates registers at the write-back stage. This scheme is similar to the TSD in that instructions are executed even if physical registers have failed to be allocated, thereby realizing pre-execution. Unfortunately, this scheme has the complication of avoiding deadlock due to out-of-order physical register allocation. To avoid deadlock, a scheme called *DSY* (on-demand with stealing from younger) has been proposed [22]. Although this scheme is feasible, it is still complicated. Furthermore, it imposes a considerable cycle count penalty due to register reallocation, completely eliminating any potential benefit from late register allocation [2].

3. Instruction Pre-Execution through TSD

TSD is based on the following register renaming scheme: 1) a register file contains committed values and temporary values for instructions that have been completed but not yet committed; and 2) a map table translates the logical register number into a physical one. This type of register renaming is, for example, implemented in the MIPS R10000 [26] and the Digital Equipment Alpha 21264 [27]. In this section, we present the TSD scheme [1], [2] that forms the basis of our study. First we illustrate the effect of TSD, and then explain the scheme by describing both the basic TSD and its extension for pre-execution.

3.1 Effect of TSD

Figure 1 illustrates the effect of TSD. An example of the execution timing of two dependent instruction sequences in a conventional processor, where `load1` and `load2` incur a cache miss, is shown in Fig. 1 (a). Figure 1 (b) shows the execution timing of the same instruction sequences with TSD. As illustrated by this figure, pre-execution starts earlier than in conventional execution because it is not stalled by a shortage of physical registers. Thus, the two cache misses occur earlier. Handling the cache misses moves data to the upper level in the memory hierarchy. As a result, in the main execution, two loads hit the L1 data cache, resulting in a speedup. Note that in the pre-execution, MLP is improved by overlapping the memory accesses. This further improves the timing of the data fetch in the main execution.

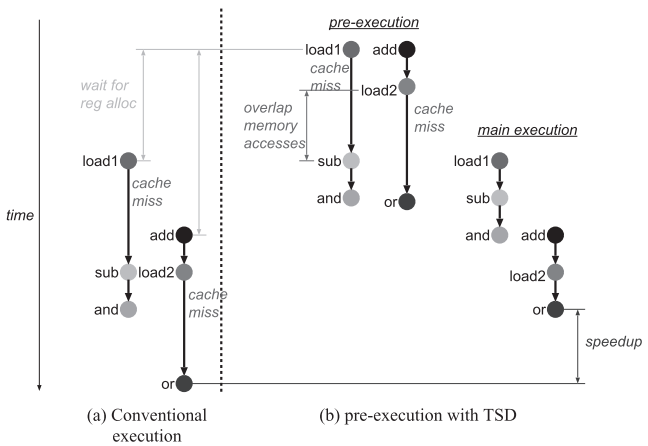


Fig. 1 Effect of TSD.

3.2 Basic TSD Scheme

First-step deallocation. The first-step of deallocation is performed at the rename stage. Besides the map table and free list, a table called the *deallocation table* (DAT) is prepared. Each entry in the DAT is associated with a physical register. It holds the number of the reorder buffer (ROB) entry, where the instruction that finally (in the second step) deallocates the corresponding physical register is placed.

The process is as follows. First, when an instruction reaches the rename stage, the physical register that is currently allocated to the same logical destination register of the instruction is *temporarily* deallocated. Then it is appended to the free list. At the same time, the number of the ROB entry, to which the instruction has been allocated, is written into the DAT entry associated with the deallocated physical register. In addition, an available physical register is obtained from the free list, and newly allocated to the logical destination register as in the conventional method. At this time, by looking up the DAT, we obtain the number of the ROB entry (ROBP), where an instruction that will *finally* deallocate the physical register has been placed. The ROBP is attached to the renaming instruction as a tag to find the timing of the second-step deallocation later in the instruction window.

Using an example, Fig. 2 illustrates the operations described above. The table presents an allocated physical register, a deallocating physical register, and an allocated ROB entry number, for each instruction in the first column. Figure 2 (a) illustrates the operations when instructions `i0` and `i1` have already been renamed, and `i2` is being renamed. First, physical register `36`, currently allocated to logical destination register `r2`, is deallocated, and appended to the free list (mark (1) in Fig. 2 (a)). At the same time, the allocated ROB entry number `13` is written into the 36th entry of the DAT (mark (2)).

Next, Fig. 2 (b) illustrates the operations when instruction `i3` is being renamed. Physical register `36` that was deallocated by instruction `i2` is allocated to this instruction. After the deallocation and allocation of the physical registers as described, we obtain, by referring to the DAT (mark (3) in Fig. 2 (b)), the ROB entry number `13`, containing instruction `i2` that will finally deallocate the allocated physical register `36` (in the second-step deallocation).

This number is attached to the renaming instruction `i3` as an ROBP tag. Then it will be written into the instruction window along with the destination register tag (dtag) and two source register tags ($stag_l$ and $stag_r$) (mark (4)).

Second-step deallocation. The renamed instruction is inserted in the instruction window, and waits for the second-step deallocation of its destination physical register, which

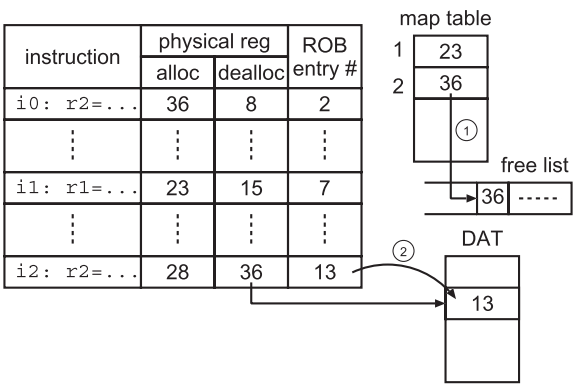
is performed at the commit stage. The deallocated physical register at this time is the one that was previously allocated to the logical register as is the case in the conventional scheme. The scheme differs, however, in that it broadcasts the number of the committed ROB entry, ROBP, to the instruction window. If the broadcasted ROBP matches the ROBP tag of an instruction waiting in the instruction window, the result write is granted. Then, instructions are issued as per normal.

In the example shown in Fig. 2, the second-step deallocation of physical register `36` is performed when instruction `i2` is committed. The ROB entry number `13` is then broadcast to the instruction window. It is matched with the ROBP tag of instruction `i3`, allowing the result write of this instruction to be granted.

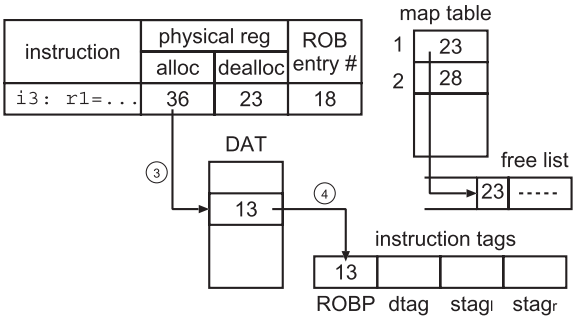
3.3 Extension for Instruction Pre-Execution

In the previous section, we mentioned that instructions waiting in the instruction window are not allowed to be issued until their writes have been granted. However, it is possible for such instructions to be executed if both ready flags are set. Although the execution result is not written, it can be passed on to dependent instructions via the bypass logic. These instructions form a pre-execution stream, which proceeds earlier than the main execution stream since it exploits ILP and MLP, where there are no resource constraints on physical registers.

Note that the ready flags of a pre-executed instruction, which were set by preceding pre-executed instructions, are reset after its issue, and the instruction is not removed from the instruction window. Later, an ROBP tag is matched, and both ready flags are set again. Then, the instruction is re-executed and the result is written into the destination physical register, as described in Sect. 3.2.



(a) At renaming of instruction `i2`



(b) At renaming of instruction `i3`

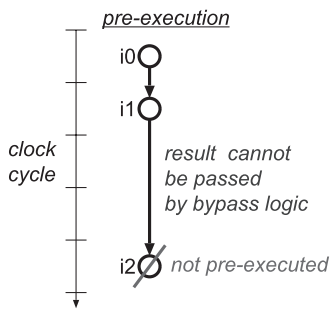
Fig. 2 First-step deallocation.

4. Use of Value Prediction

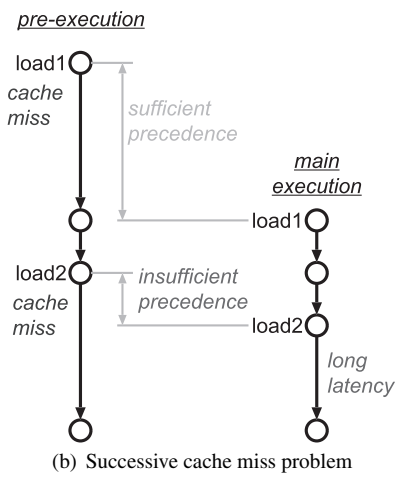
4.1 Problems

The TSD suffers from two problems that prevent (early) pre-execution, as described in Sect. 1. These problems are illustrated in Fig. 3, where the nodes and edges represent instructions and dependences, respectively. Figure 3 (a) highlights the first problem, which we call the *bypass problem*, where instruction `i2` is not selected to be issued in the next cycle after instruction `i1` has been issued. In this case, instruction `i1` cannot pass on the result to instruction `i2`, because result passing relies on the bypass logic in the pre-execution [†].

The second problem arises from the difference in throughput between the pre-execution and main execution. Throughput of the pre-execution is lower than that of the main execution, because cache misses occur in the pre-



(a) Bypass problem



(b) Successive cache miss problem

Fig. 3 Problems in TSD.

[†]Actually, the TSD resets the ready flag that was set by a preceding pre-executed instruction in the previous cycle, so that an instruction like `i2` is not issued [1], [2].

execution whereas they can be avoided in the main execution. If multiple cache misses occur successively in a particular data dependence chain in the pre-execution, the latency of the later misses cannot be avoided sufficiently in the main execution. We call this problem the *successive cache miss problem*, and it is illustrated in Fig. 3 (b). Assume that `load1` and `load2` cause cache misses. The latency of `load1` can be avoided sufficiently, because it is pre-executed early enough compared with the main execution. However, due to the cache miss of `load1`, the pre-execution of `load2` is delayed. As a result, the latency of `load2` cannot be avoided sufficiently in the main execution.

4.2 Using Value Prediction

We propose the use of value prediction to solve the above problems. Value prediction can break up an instruction sequence by cutting data dependences. For the bypass problem illustrated in Fig. 3 (a), 1) we can pre-execute instruction `i2` if its reference address (in the case that `i2` is a load) can be predicted; or 2) we can pre-execute the instructions following `i2` if `i2`'s result can be predicted. In the successive cache miss problem illustrated in Fig. 3 (b), `load2` can be pre-executed early enough, if either its reference address or the result of any preceding instructions can be predicted.

Note that our utilization of value prediction does not require recovery from misprediction, because our scheme does not update the state. Our scheme pollutes the cache slightly with misprediction, but this loss is significantly outweighed by the benefit arising from the increase in pre-executed instructions.

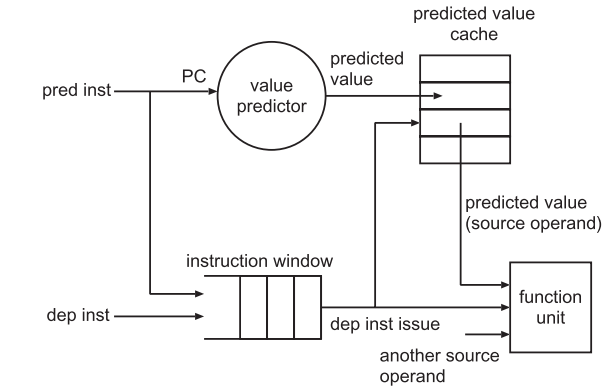
4.3 Result Prediction vs. Address Prediction

In general, there are two ways of using value prediction: predicting an instruction's result and predicting the reference address of a load.

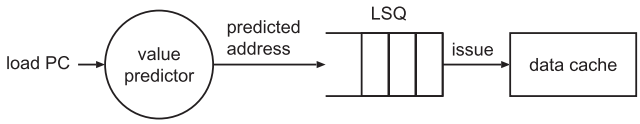
In result prediction, a load that potentially causes a cache miss is pre-executed by executing successive consumers of the predicted instruction. Because prediction can be applied to any instruction with a destination register, in theory, more data dependences are removed and consequently more instructions are expected to be pre-executed, compared with address prediction. However, any successive dependent instructions between the predicted instruction and the load must be pre-executed to obtain the pre-execution benefit. Therefore, the bypass problem remains partially unsolved.

On the other hand, in address prediction, the predicted load can access memory immediately without requiring other instructions to be executed, which is not the case in result prediction. Thus, the bypass problem can be solved. However, unlike result prediction, there is no chance of pre-execution if the reference address is unpredictable.

Although an implementation incorporating result prediction in the pre-execution mechanism differs from that of the conventional method, it is merely a simple extension of



(a) Result prediction case.



(b) Address prediction case

Fig. 4 Incorporation of value prediction.

the conventional implementation (see Fig. 4 (a)). In the conventional method, the predicted value is stored in the destination physical register of the predicted instruction. Since pre-executed instructions do not have a destination physical register writable, we prepare a small buffer, called the *predicted value cache*, to hold predicted values. Depending on the required hit rate, the predicted value cache can be configured with a direct mapped, set-associative, or fully associative organization, indexed or keyed by a physical register number. If the result of an instruction has been predicted in the front-end, it is written into the predicted value cache, and is inserted into the instruction window with marked as *predicted*. Such an instruction outputs an issue request, and issued if granted. Although it is not executed actually, it broadcasts the destination tag to the instruction window. As a result, a dependent instruction is issued subsequently, and then it consults the predicted value cache. If its source register value is found, it is pre-executed using this value. If not found, the issue fails, and the associated instruction is invalidated.

On the other hand, an implementation incorporating address prediction in the pre-execution mechanism is identical to that of the conventional method (see Fig. 4 (b)). That is, a load is first split into an address calculation instruction and a memory access instruction in the front-end (we assume a split load/store architecture). Then, the reference address of the memory access instruction is predicted. The memory access instruction is inserted into the load/store queue (LSQ) with the predicted address. Finally, the memory access instruction is scheduled normally in the LSQ, and the data cache is accessed if issued. Note that this mechanism differs from data prefetch in that instructions that depend on the predicted load are issued, and thus instruction pre-execution continues, which is not the case in data prefetch.

Table 1 Statistics of cache and memory accesses.

program	cache miss rate		main memory
	L1 data	L2	access rate
ammp	11.0%	31.6%	3.5%
applu	4.3%	48.9%	2.1%
apsi	0.3%	23.3%	0.1%
art	59.2%	46.1%	27.3%
equake	2.9%	24.6%	0.7%
mesa	0.2%	11.6%	0.0%
mgrid	2.4%	25.0%	0.6%
swim	12.0%	31.8%	3.8%

Table 2 Base processor configuration.

Pipeline width	8-instruction wide for each of fetch, decode, issue, and commit
ROB	128 entries
LSQ	64 entries
Instruction window	64 entries
Function unit	8 iALU, 4 iMULT/DIV, 4 Ld/St, 6 fpALU, 4 fpMULT/DIV/SQRT
L1 I-cache	64 KB, 2-way, 32 B line
L1 D-cache	64 KB, 2-way, 32 B line, 4 ports, 2-cycle hit latency, non-blocking
L2 cache	2 MB, 4-way, 64 B line, 12-cycle hit latency
Main memory	300-cycle min. latency, 8 B/cycle bandwidth
Branch prediction	6-bit history gshare, 8 K-entry PHT, 512-entry, 4-way BTB 10-cycle misprediction penalty

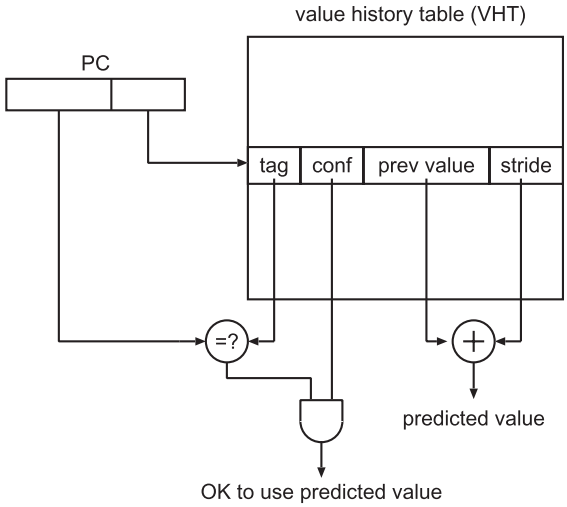


Fig. 5 Stride predictor.

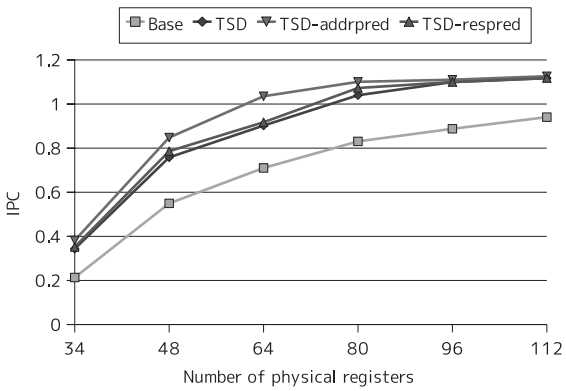


Fig. 6 IPC when varying number of physical registers.

5. Evaluation Results

5.1 Environment

To evaluate our scheme, we built a simulator based on the SimpleScalar Tool Set version 3.0a. The instruction set is SimpleScalar/PISA, which is an extension of the MIPS R10000 ISA. We used eight programs from the SPECfp2000, compiled using gcc ver.2.7.2.3 with options -O6 -funroll-loops. Table 1 lists the benchmark programs and their memory statistics during execution on the base processor, configuration of which is described below.

We evaluated the following three models: the *TSD model*, that pre-executes instructions with the conventional TSD; the *TSD-respred model*, that uses result prediction in the TSD; and the *TSD-addrpred model*, that uses address prediction in the TSD.

The configuration of the base processor for the three models is summarized in Table 2.

In the TSD-respred and TSD-addrpred models, we use a stride value predictor [28] with a 1024-entry direct-mapped value history table (VHT), as shown in Fig. 5. The VHT is indexed by the PC, and each entry has a tag (the same as a cache tag), an immediately previous value, a stride value, and a confidence flag. The value predictor output a predicted value by adding the immediately previous value and the stride value, if the confidence flag is set. The confidence flag is set if the immediately previous prediction was correct.

In the TSD-respred model, we assume a predicted value cache of the same size as the register file. Although it is preferable for the size to be small, we make this assumption to find the maximum benefit of result prediction. In this assumption, the instruction that uses a predicted value always hits the predicted value cache.

5.2 Reduction of Physical Registers

Figure 6 shows the geometric mean of IPC for each model, with the number of physical registers varying from 34 to 112 (with equal numbers of integer and floating-point registers). The TSD model achieves the same IPC as the base processor, with fewer physical registers. The result is enhanced by using value prediction.

Here, we attempt to quantify the reduction rate of the physical registers by obtaining a representative value. Intuitively, the effectiveness of the TSD is sensitive to the balance between the ROB size and the number of physical registers. For balancing, we use the following equation:

Table 3 Percentage of dynamic instructions categorized by type of destination register.

program	type of destination register			
	int	fp	other	none
ammp	24%	55%	1%	20%
applu	60%	31%	1%	8%
apsi	59%	22%	9%	10%
art	37%	39%	2%	22%
equake	58%	16%	0%	26%
mesa	48%	21%	4%	27%
mgrid	43%	55%	1%	2%
swim	48%	48%	0%	4%
AVG	47%	36%	2%	15%

$$Npregs = ROBsize + Nlregs \quad (1)$$

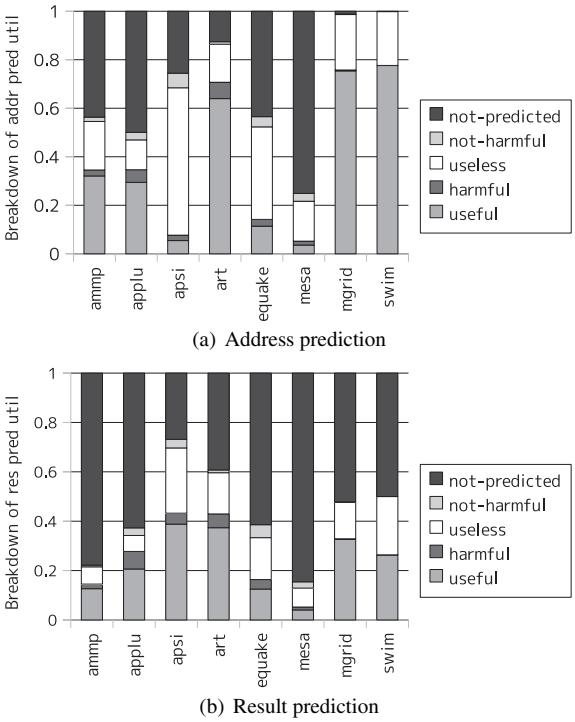
where *Npregs* and *Nlregs* are the total number of physical and logical registers, respectively. The ROB size and the number of physical registers determined by Eq. (1) are balanced with the following reasons. 1) the ROB size gives the number of supported in-flight instructions, where each of most in-flight instructions of *ROBsize* requires a physical register; and 2) each committed logical destination register requires a physical register. Because the default ROB size is 128, we obtained 192 (128 + 64) as the number of physical registers. We then divided the total number of physical registers equally between integer and floating-point registers. This is because there is roughly the same number of dynamic instructions with either integer or floating-point destination registers in the benchmark programs. Table 3 lists the percentage of dynamic instructions categorized by the type of the destination register in the baseline processor. Therefore, there are 96 physical registers each for integer and floating-point values. We call this number the *baseline number of physical registers*.

From Fig. 6, we see that the TSD and TSD-addrpred models can, with 64 and 48 physical registers, respectively, achieve equivalent IPC to the base processor with the baseline number of physical registers. With detailed evaluation for preciseness, the number of physical registers where the TSD and TSD-addrpred model barely outweigh the base with 96 registers in IPC is 62 and 49, respectively. The reduction rate in physical registers is 35% for the conventional TSD, and 49% using address prediction. Compared with the conventional TSD without address prediction, address prediction reduces the number of physical registers by 21%.

Note that reducing the register file size is important despite the increase in hardware due to the VHT. This is because the register file is on the processor's critical path, as described in the Sect. 1.

Unfortunately, the effect of address prediction deteriorates as the number of physical registers increases. This is because the potential for instruction pre-execution decreases as the number of physical registers increases, and consequently, the chance of using address prediction decreases accordingly.

In contrast to the TSD-addrpred model, the TSD-respred model is less effective, despite having a predicted value cache of the maximum size. A qualitative reason for this is discussed in Sect. 4.3. We further analyze the TSD-respred model in the following subsection.



(a) Address prediction

(b) Result prediction

Fig. 7 Breakdown of prediction utilization.

5.3 Breakdown of Prediction Utilization

This section gives the breakdown of prediction utilization with regard to the correctness of prediction and whether or not the predicted value is used for pre-execution. Figure 7 shows the result. The data in figures (a) and (b) were collected with 49 physical registers in the TSD-addrpred and TSD-respred models, respectively. This configuration gives equivalent IPC to the base processor in the TSD-addrpred model as described before. Each bar is partitioned into five portions. Dynamic candidate instructions for prediction (instructions with a destination register in the TSD-respred model, and load instructions in the TSD-addrpred model) are categorized into the following classes:

- *useful*: prediction is correct, and the predicted instruction is pre-executed.
- *harmful*: prediction is incorrect, but the predicted instruction is pre-executed.
- *useless*: prediction is correct, but predicted instruction is not pre-executed.
- *not-harmful*: prediction is incorrect, and the predicted instruction is not pre-executed.
- *not-predicted*: no prediction is made due to low confidence.

Here, the reason that a predicted instruction is not pre-

Table 4 Average of prediction coverage rate and accuracy.

model	coverage	accuracy
TSD-addrpred	69%	93%
TSD-respred	43%	88%

executed is following either case: 1) its destination register is deallocated in the second-step deallocation and it thus becomes a non-candidate for pre-execution; or 2) in result prediction, its source registers become available, or in address prediction, the associated address calculation instruction is performed and the reference address becomes available. Therefore, the predicted result or address is unnecessary.

Table 4 summarizes the average of the prediction coverage rate and accuracy, calculated from Fig. 7. The coverage rate is the ratio of the number of predicted instructions to the number of candidate instructions for prediction. On the other hand, the accuracy is the ratio of the number of correctly predicted instructions to the number of predicted instructions. The equations expressed with reference to the classes defined above are as follows:

$$\begin{aligned} coverage &= useful + harmful + useless + not\text{–}harmful \\ &= 1 - not\text{–}predicted \\ accuracy &= \frac{useful + useless}{coverage} \end{aligned}$$

In the TSD-addrpred model, the coverage rate is fairly high in most programs, as shown in Fig. 7 and Table 4. Also, not surprisingly, many loads with predicted addresses are pre-executed in most programs, because such loads are issued with little wait in the LSQ.

In contrast, the coverage rate is low in the TSD-respred model, preventing the expected benefit, discussed in Sect. 4.3, from being obtained. This, together with the fact that the bypass problem is only partially solved, makes the TSD-respred model ineffective.

In the subsequent sections, we only present evaluation results for the TSD-addrpred model, as the TSD-respred model is far inferior and thus further analysis has no value.

5.4 Pre-Execution Rate

As inferred from Fig. 7 (a), the *pre-execution rate of loads*, that is, the number of loads that are pre-executed relative to the total number of dynamic loads, is expected to increase. Figure 8 shows the pre-execution rate of loads for the TSD and TSD-addrpred models. As in Sect. 5.3, the data were collected using 49 physical registers.

As shown in the graph, the TSD-addrpred model achieves a higher rate than the TSD model for all program.

5.5 Increase of MLP Exploitation

An increase in MLP is expected as the pre-execution rate of loads increases. As an example, Fig. 9 shows the distribution of the number of concurrent unresolved loads whose

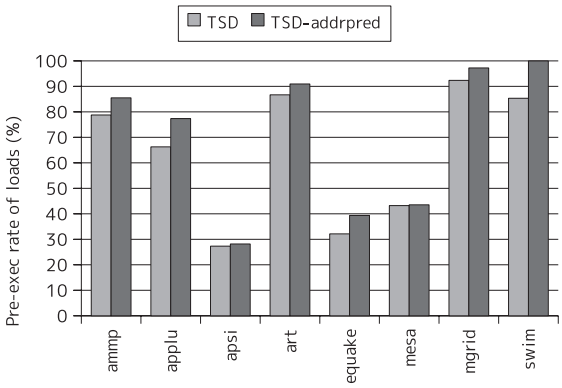


Fig. 8 Pre-execution rate of loads.

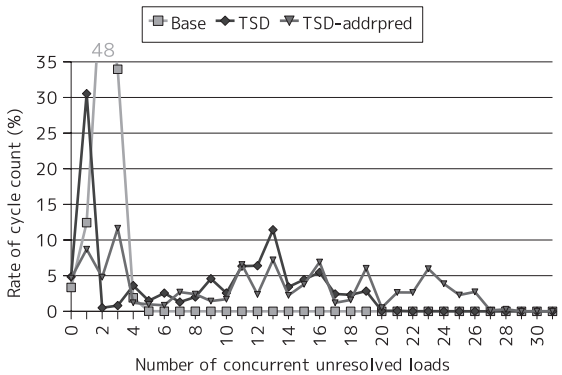


Fig. 9 Distribution of number of unresolved loads whose resolution takes more than or equal to 300 cycles exist simultaneously in art.

cache miss resolution takes 300 (minimum memory latency) or more cycles in *art*. The horizontal-axis represents the number of unresolved loads that exist simultaneously. The vertical-axis represents the ratio of the cycle count for each of the concurrent unresolved loads relative to the total cycle count. We assume 49 physical registers.

As illustrated in this figure, the distribution peaks at a very small number of loads (1–3 loads) for the base processor. This implies that MLP is only slightly exploited. In the TSD model, the distribution is shifted towards the right, indicating that more MLP is exploited. The distribution, however, still peaks at one unresolved load. In contrast, the distribution of the TSD-addrpred model is shifted significantly to the right, and the rate for less than 3 unresolved loads is lower. This indicates that the TSD-addrpred model exploits MLP to a high degree.

5.6 IPC

This section discusses how value prediction affects IPC. Figures 10 and 11 show, respectively, the load latency reduction rate and the speedup of the TSD-addrpred model relative to those of the TSD model. Once again we assume 49 physical registers. The load latency is significantly reduced (22% on average), and consequently speedup is considerable (17%). Not surprisingly, the speedup cor-

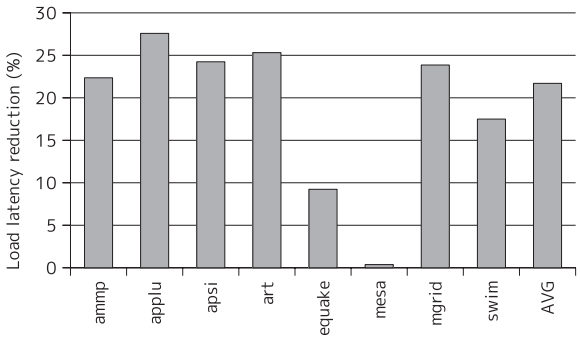


Fig. 10 Load latency reduction rate of TSD-addrpred model relative to TSD model.

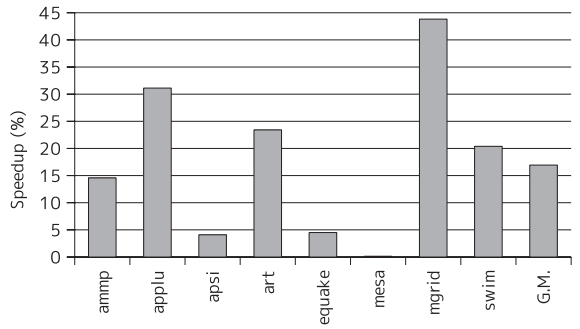


Fig. 11 Speedup of TSD-addrpred model over TSD model.

Table 5 Evaluation standards of items that contribute to speedup.

item	–	+	++
mem	<1%	1–3%	>3%
pred	<50%	50–80%	>80%
pexec	<20%	≥20%	N/A
speedup	<10%	10–20%	>20%

relates strongly with the latency reduction rate, although the amount of speedup differs greatly for each program. Speedup is related to the following three aspects as given in Table 5.

- Memory access rate (*mem*): the ratio of the number of loads that access main memory to the total number of dynamic loads in the base processor.
- Value prediction effective accuracy [†] (*pred*): the ratio of correct predictions relative to the total number of dynamic loads in the TSD-addrpred model.
- Non-attainment rate of pre-execution (*pexec*): the rate of loads that are not pre-executed relative to the total number of dynamic loads in the TSD model.

As listed in Table 5, we represent how much each item contributes to the effectiveness by "–," "+,"and "++" denoting a weak to strong contribution, respectively. Table 6 summarizes the results of our analysis. Based on this table, we can discuss the effectiveness as follows.

In *ammp*, our scheme achieves a significant speedup, because *mem*, *pred*, and *pexec* evaluate to more than "+." In *art* and *swim*, there appears to be little room for improvement because *pexec* evaluates to "–," yet these programs are considerably memory-intensive, as indicated by *mem* evaluating to "++." Also, *pred* evaluates to "++." For these reasons, our scheme achieves a significant speedup despite a *pexec* of "–." In *applu*, although *pred* evaluates to "–," the effective prediction accuracy is not very low (42%). As both *mem* and *pexec* evaluate to "+," the possibility of and scope for IPC improvement is high. Therefore, our scheme is effective.

Table 6 Cause analysis of effectiveness.

program	mem	pred	pexec	speedup
ammp	++	+	+	+
applu	+	–	+	++
apsi	–	+	+	–
art	++	++	–	++
equake	–	–	+	–
mesa	–	–	+	–
mgrid	–	++	–	++
swim	++	++	–	++

In *apsi*, *equake*, *mesa*, and *mgrid*, it is less likely that pre-execution will be effective, because these programs are not memory-intensive, as indicated by *mem* evaluating to "–." However, in *mgrid*, a significant speedup is achieved. This is because, although we have not yet mentioned this, the TSD has a positive effect on memory disambiguation by pre-executing load address calculation instructions [1], [2]. Accelerating pre-execution augments this effect. In fact, under the assumption of perfect memory disambiguation, which excludes this effect, the speedup is decreased to only 4%. This particular result is only seen in *mgrid* by our evaluation with perfect memory disambiguation.

5.7 Discussion of Contribution Breakdown

As described in Sect. 4.1, using value prediction has two effects. One is that the bypass problem is solved, and the other is that the successive cache miss problem is solved. This section discusses the contribution of each effect, by comparing the effect of value prediction in a processor with *perfect bypass logic* to that with real bypass logic. Perfect bypass logic is hypothetical logic that can pass the result of a producer to its consumers at any time (i.e., consecutive clock cycle timing is not necessary to pass the result). Assuming perfect bypass logic eliminates the bypass problem, therefore exposing the effect of solving the successive cache miss problem.

Figure 12 shows the correlation between the speedup of the TSD-addrpred model compared to that of the TSD model under both perfect bypass logic (vertical axis) and real bypass logic (excerpted from Fig. 11) (horizontal axis). The basis of each speedup is the IPC of the TSD model with perfect bypass logic on the vertical axis, and real bypass logic on the horizontal axis. We assume 49 physical registers.

[†]Note that the definition of effective accuracy differs from that of accuracy in Table 4 in that the denominator is different.

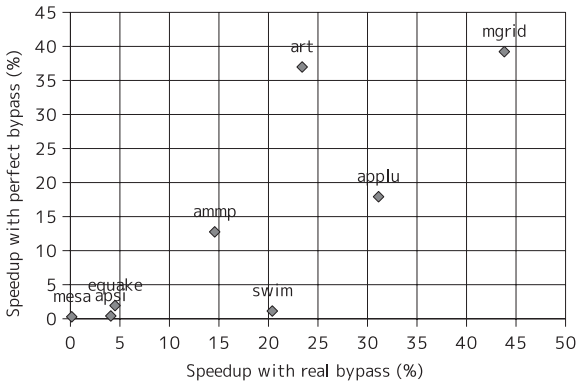


Fig. 12 Correlation between speedup of TSD-addrpred model with perfect bypass logic and that with real bypass logic.

As a program with particular characteristics, *swim* achieved only a small speedup in the case of perfect bypass logic, despite a large speedup in the case of real bypass logic. Therefore, the speedup is mostly due to solving the bypass problem. In contrast, the speedup of *ammp* with perfect bypass logic is equivalent to that with real bypass logic. Also, in *art*, the former is large compared with the latter. These results indicate that the effect of solving the successive cache miss problem contributes much more than that of solving the bypass problem for these programs. In *applu*, the speedup in the case of perfect bypass logic is half of that in the case of real bypass logic. This implies that the effects of the two solutions contribute equally. As described in Sect. 5.6, in *mgrid*, the speedup is obtained by accelerating load address calculation. Since this effect is not eliminated by perfect bypass logic, the speedup in the case of perfect bypass logic is equivalent to that in the case of real bypass logic. Because the speedups of the other programs are small, they are not worth discussing here.

6. Conclusions

This paper proposes the use of value prediction to promote pre-execution in the TSD. Our utilization of value prediction aims to increase MLP rather than ILP. Our evaluation using the SPECfp2000 benchmark programs shows that our proposed scheme with address prediction exploits more MLP, and significantly reduces the register file size with little IPC loss.

Acknowledgment

The authors thank T. Inagaki and K. Ichihara for their help to collect evaluation data. This work was partially supported by the Ministry of Education, Culture, Sports, Science and Technology Grant-in-Aid for Scientific Research (C) (No. 19500041 and 22500045).

References

[1] A. Yamamoto, Y. Tanaka, H. Ando, and T. Shimada, "Data prefetching and address pre-calculation through instruction pre-execution with two-step physical register deallocation," Proc. 8th Workshop on Memory Performance: Dealing with Applications, Systems and Architectures, pp.41–48, Sept. 2007.

[2] A. Yamamoto, Y. Tanaka, H. Ando, and T. Shimada, "Two-step physical register deallocation for data prefetching and address precalculation," IPSJ Trans. Advanced Computing Systems, vol.1, no.2, pp.34–46, Aug. 2008.

[3] Y. Tanaka and H. Ando, "Increasing the effectiveness of instruction pre-execution with physical register deallocation via value prediction," IPSJ SIG Technical Reports, pp.3–8, Oct. 2008.

[4] Y. Tanaka and H. Ando, "Reducing register file size through instruction pre-execution with value prediction," Proc. 2009 Symposium on Advanced Computing Systems and Infrastructures, pp.335–343, May 2009.

[5] Y. Tanaka and H. Ando, "Reducing register file size through instruction pre-execution enhanced by value prediction," Proc. 27th IEEE International Conference on Computer Design, pp.238–245, Oct. 2009.

[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous subordinate microthreading (SSMT)," Proc. 26th Annual International Symposium on Computer Architecture, pp.186–195, May 1999.

[7] C. Zilles and G.S. Sohi, "Master/slave speculative parallelization," Proc. 35th Annual International Symposium on Microarchitecture, pp.85–96, Nov. 2002.

[8] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A study of slipstream processors," Proc. 33rd Annual International Symposium on Microarchitecture, pp.269–280, Dec. 2000.

[9] A. Roth and G.S. Sohi, "Speculative data-driven multithreading," Proc. 7th Annual International Symposium on High Performance Computer Architecture, pp.37–48, Jan. 2001.

[10] J.D. Collins, D.M. Tullsen, H. Wang, Y. Lee, D. Lavery, J.P. Shen, and C. Hughes, "Speculative precomputation: Long-range prefetching of delinquent loads," Proc. 28th Annual International Symposium on Computer Architecture, pp.14–25, July 2001.

[11] J.D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen, "Dynamic speculative precomputation," Proc. 34th Annual International Symposium on Microarchitecture, pp.306–317, Dec. 2001.

[12] C.B. Zilles and G.S. Sohi, "Execution-based prediction using speculative slices," Proc. 28th Annual International Symposium on Computer Architecture, pp.2–13, July 2001.

[13] D.M. Tullsen, S. Eggers, and H.M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," Proc. 22nd Annual International Symposium on Computer Architecture, pp.392–403, June 1995.

[14] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt, "Runahead execution: An effective alternative to large instruction windows," Proc. 9th Annual International Symposium on High-Performance Computer Architecture, pp.129–140, Feb. 2003.

[15] M.H. Lipasti and P.J. Shen, "Exceeding the dataflow limit via value prediction," Proc. 29th Annual International Symposium on Microarchitecture, pp.226–237, Dec. 1996.

[16] F. Gabbay and A. Mendelson, "The effect of instruction fetch bandwidth on value prediction," Proc. 25th Annual International Symposium on Computer Architecture, pp.272–281, July 1998.

[17] Y. Sazeides and J.E. Smith, "The predictability of data values," Proc. 30th Annual International Symposium on Microarchitecture, pp.248–258, Dec. 1997.

[18] G. Reinman and B. Calder, "Predictive techniques for aggressive load speculation," Proc. 31st Annual International Symposium on Microarchitecture, pp.127–137, Dec. 1998.

[19] H. Zhou and T.M. Conte, "Enhancing memory level parallelism via recovery-free value prediction," Proc. 17th Annual International Conference on Supercomputing, pp.326–335, June 2003.

[20] O. Mutlu, H. Kim, and Y.N. Patt, "Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns," Proc. 38th Annual

International Symposium on Microarchitecture, pp.223–244, Nov. 2005.
[21] A. González, J. González, and M. Valero, "Virtual-physical registers," Proc. Fourth Annual International Symposium on High Performance Computing, pp.175–184, Feb. 1998.
[22] T. Monreal, A. González, M. Valero, J. González, and V. Viñals, "Delaying physical register allocation through virtual-physical registers," Proc. 32nd Annual International Symposium on Microarchitecture, pp.186–192, Nov. 1999.
[23] M. Moudgill, K. Pinagli, and S. Vassiliadis, "Register renaming and dynamic speculation: An alternative approach," Proc. 26th Annual International Symposium on Microarchitecture, pp.202–213, Dec. 1993.
[24] D. Balkan, J. Sharkey, F. Ponomarev, and A. Aggarwal, "Address-value decoupling for early register deallocation," Proc. 2006 International Conference on Parallel Processing, pp.337–346, Aug. 2006.
[25] S.T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.107–119, Oct. 2004.
[26] K.C. Yeager, "The MIPS R10000 superscalar microprocessor," IEEE Micro, vol.16, no.2, pp.28–40, April 1996.
[27] R.E. Kessler, "The Alpha 21264 microprocessor," IEEE Micro, vol.19, no.2, pp.24–36, March 1999.
[28] T.F. Chen and J.L. Baer, "Effective hardware-based data prefetching for high-performance processors," IEEE Trans. Comput., vol.44, no.5, pp.609–623, May 1995.
[29] P. Shivakumar and N.P. Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model," WRL Research Report 2001/2, Dec. 2001.
[30] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitecture," Proc. 32nd Annual International Symposium on Computer Architecture, pp.248–259, June 2000.

Appendix: Performance Comparison under the Same Cost

This appendix evaluates the performance of the base and TSD-addrpred models under the same cost. This evaluation is requested by the reviewer and associated editor of this paper. Specifically, they requested us to evaluate the performance comparison between the following two models:

- The TSD-addrpred model with 49 physical registers
- The base processor with a register file enlarged by using the transistors for the VHT and DAT in the TSD-addrpred model.

We refer to the latter model as the *reviewer's model*. From our assumption of the VHT and DAT, the register file can have 1,420 entries for each of integer and floating-point in the reviewer's model.

We first evaluated the access times of the register files with 49 and 1,420 entries under the assumption of 35 nm technology, using CACTI [29] we modified. The major modification is that we introduced the real wire parameters [30], instead of parameters scaled from those in 800 nm the original CACTI uses. The evaluation results show that the access times of the register files with 49 and 1,420 entries are 59 ps and 322 ps, respectively.

We then evaluated IPC for the both models. After that,

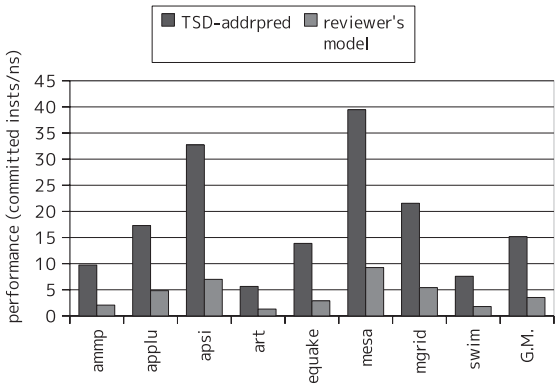


Fig. A· 1 Performance (committed instructions per ns) of TSD-addrpred model and base with a register file enlarged by using the transistors for the VHT and DAT.

we calculated instruction throughput (committed instructions per ns) as performance, which is derived by the IPC divided by the clock cycle time. Here, we simply assume that the access time of the register file determines the clock cycle time, because the register file is on a critical path. The results are shown in Fig. A· 1. As shown in the figure, the performance of the TSD-addrpred model is 4.3 times larger than that of the reviewer's model on average.

Pipelining the register file access may alleviate the degradation of the clock cycle time. In the reviewer's model, the register file must be pipelined to 6 stages to equalize the pipeline stage delay to that in the TSD-addrpred model. However, this in turn complicates the bypass logic, as described in Sect. 1. Specifically, the fanin of the multiplexer at a function unit's input is 24 in the TSD-addrpred model, while it is 104 in the reviewer's model. This simply increases the delay of the multiplexer due to the parasitic capacitance of the transistors. Furthermore, it lengthens the wire of the control signal of the multiplexer and increases the fanout of its driver, according to the number of fanin of the multiplexer. This increases the delay significantly. In addition, the number of comparators of operand tags to control bypassing unrealistically increases. It is 512 in the TSD-addrpred model, while 3,072 in the reviewer's model. This considerably increases the delay of the bypass control; the delay is likely to exceed a single cycle, because the number of the comparators is 3 times larger than that in the issue queue. In summary, the complicated bypass logic degrades the clock cycle time seriously. Finally, having a deep pipeline increases the branch misprediction penalty, and thereby lowering IPC.

Yusuke Tanaka was born in 1984. He received his B.E. and M.E. degrees from Nagoya University, Nagoya, Japan in 2007 and 2009, respectively. Since 2009, he has been with Denso Corp.

Hideki Ando received his B.S. and M.S. degrees in electronic engineering from Osaka University, Suita, Japan in 1981 and 1983, respectively. He received his Ph.D. degree in information science from Kyoto University, Kyoto, Japan in 1996. From 1983 to 1997 he was with Mitsubishi Electric Corporation, Itami, Japan. From 1991 to 1992 he was a visiting scholar at Stanford University. In 1997 he joined the faculty of Nagoya University, Nagoya, Japan, where he is currently a professor in the department of electrical engineering and computer science. In 1998 and 2002, he received the IPSJ best paper awards. His research interests include computer architecture and compilers.