Reducing Register File Size through Instruction Pre-Execution Enhanced by Value Prediction

Yusuke Tanaka
Department of Computational Science and Engineering
Nagoya University

Hideki Ando
Department of Electrical Engineering and Computer Science
Nagoya University
ando@nuee.nagoya-u.ac.jp

***Abstract*— *Two-step physical register deallocation* (TSD) is an architectural scheme, which enhances memory-level parallelism (MLP) by pre-executing instructions. Ideally, the TSD allows MLP under the unlimited number of physical registers to be exploited, and consequently only a small register file is necessary for MLP. In practice, however, the amount of MLP exploitable is limited, because there are cases where pre-execution is not performed or timing of pre-execution is delayed. This is caused by data dependencies among the pre-executed instructions. This paper proposes the use of value prediction to solve these problems. Our way of the value prediction usage has the advantage over the conventional way of the usage for enhancing ILP, that there is no need to recover from misspeculation. Our evaluation results using SPECfp2000 benchmark show that our scheme can achieve equivalent performance to that of the previous TSD scheme without value prediction, with 75% of the register file size.**

I. Introduction

Supporting many in-flight instructions allows aggressive exploitation of instruction-level parallelism (ILP) and memory-level parallelism (MLP), leading to performance increases. The exploitation of MLP is especially effective in memory-intensive programs. To support many in-flight instructions, a large register file is required. However, a large register file affects the clock cycle time adversely because it takes a long time to access. Although this adverse effect can be alleviated by pipelining, this complicates the bypass logic instead. In addition, having a deep pipeline increases the branch misprediction penalty, lowering IPC. Therefore, it is difficult to remove the adverse effect of a large register file completely. It is important to reduce the register file size without performance degradation.

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

The TSD deallocates physical registers in two phases: 1) the temporal deallocation, which allows the physical register to be allocated to another instruction; and 2) the final deallocation, which allows the result write to be granted. The TSD completely removes the pipeline stall in the rename stage, which is due to 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 be actually available, these instructions are executed (*pre-execution*) if their source operands are available, although the result is not written to the physical register. Instead, the result of a pre-executed instruction is passed to its dependent instructions by the bypass logic. Therefore pre-execution can be performed continuously. This enables 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. In the main execution, a load obtains data from the cache, which would have resided in main memory without pre-execution.

Although the TSD is effective, there are cases where instructions are not pre-executed or not pre-executed early enough, because 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, because the pre-execution of later loads is delayed.

Both problems stem from data dependences. We introduce value prediction to solve these problems. We apply value prediction to the pre-execution candidate instructions whose physical register is only temporarily allocated. This allows instructions to be pre-executed independently of any precedent instructions. Loads, which would have not been pre-executed or not pre-executed early enough in the conventional TSD, are pre-executed early enough, therefore improving MLP. As a result, we can further reduce the register file size.

Study of value prediction was originally aimed at enhancing ILP. Our aim is different in that our target is to enhance MLP. Since pre-execution does not update the architectural state, our scheme does not require the complicated recovery hardware associated with misprediction, unlike the ILP-oriented schemes.

The remainder of this paper is organized as follows. Section II describes related work. Section III explains the TSD scheme proposed previously. Section IV explains the effect of value prediction when it is used for instruction pre-execution in the TSD. Section V discusses power consumption saving. Section VI addresses the applicability of our proposal to other pre-execution schemes. Evaluation results are presented in Section VII, and our conclusions are stated in Section VIII.

II. Related Work

A. Instruction Pre-Execution

A number of studies regarding pre-execution have been carried out [3]–[9]. These schemes extract, either statically or dynamically, the instructions necessary for the generation of memory accesses as a thread, and then 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 multithread environment such as simultaneous multithreading [10] or chip multiprocessors.

The only pre-execution scheme, to our knowledge, that does not need a multithreaded environment is *runahead execution* [11]. 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 succeeding to 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.

B. Enhancing ILP with Value Prediction

Originally, value prediction was studied to enhance ILP by breaking data dependences and executing predicted and dependent instructions speculatively [12]–[14]. Reinman *et al.* examined in detail the effectiveness of value prediction (load-result and load-address predictions) with squash and reissue recovery schemes [15]. Their evaluation results indicate that, because the prediction accuracy is not very high, simple squash recovery exhibits only limited speedup; reissue of only instructions that depend on the mispredicted instruction is required for solid speedup by reducing the misprediction penalty. Unfortunately, instruction reissue complicates the instruction window. In addition, it adds pressure to the instruction window, because an instruction cannot be removed immediately after issue, and it must stay until the prediction is validated at the execution stage.

C. Enhancing MLP with Value Prediction

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

Zho *et al.* advocated the use of value prediction to prefetch data [16]. They remove data dependence (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 enhances MLP to a great extent.

Mutlu *et al.* introduced value prediction in the runahead execution [17]. To our knowledge, this is the only study that has introduced value prediction to 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 for removing data dependence on only the triggered L2 missed load. Unlike this study, we use value prediction generally and aggressively for any instructions, not just a special instruction. Furthermore, the AVD predictor is not useful when attempting to use value prediction for any instructions, because the AVD predictor requires the reference address, which is unknown until the data dependence is resolved.

D. Reducing Register File

Several studies, aimed at reducing register files, have been done in an attempt to reduce the occupation time of physical registers by late allocation [18], [19] or early deallocation [20]–[22].

Early deallocation schemes deallocate registers speculatively by predicting the last consumer, and allocate them at the rename stage. The 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.

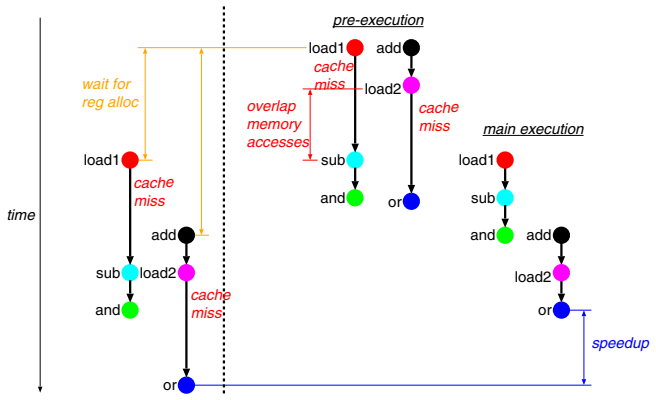
Late allocation schemes do not allocate registers at the rename stage; instead, they are allocated later in the pipeline. The *virtual-physical register scheme* [18], [19] allocates registers at the write-back stage. This scheme is similar to the TSD in that instructions are executed even if physical registers have failed to be allocated, thus realizing pre-execution. Unfortunately, this scheme has the complication of avoiding deadlock due to out-of-order physical register allocation.

III. Instruction Pre-Execution through TSD

The TSD is based on a register renaming scheme, in which a register file contains committed values and temporary values for instructions that have been completed but not yet committed, and 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 and the Digital Equipment Alpha 21264. 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 the extension for pre-execution.

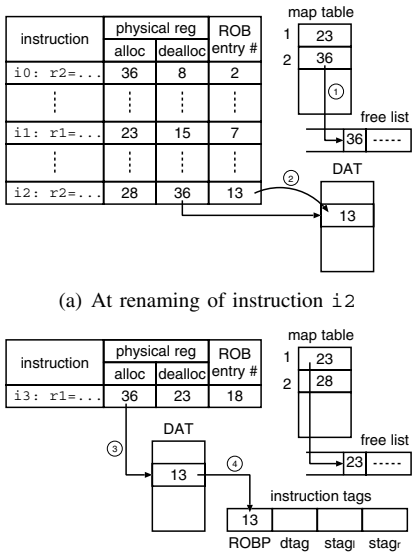
A. Effect of TSD

Fig. 1 illustrates the effect of TSD. An example of the execution timing of two dependent instruction sequences in



(a) Conventional execution (b) pre-execution with TSD

Fig. 1. Effect of TSD.



(a) At renaming of instruction `i2`

(b) At renaming of instruction `i3`

Fig. 2. First-step deallocation.

a conventional processor, where `load1` and `load2` incur a cache miss, is shown in Fig. 1a. Fig. 1b shows the execution timing of the same instruction sequences with TSD. As illustrated by the 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.

B. 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, and 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.

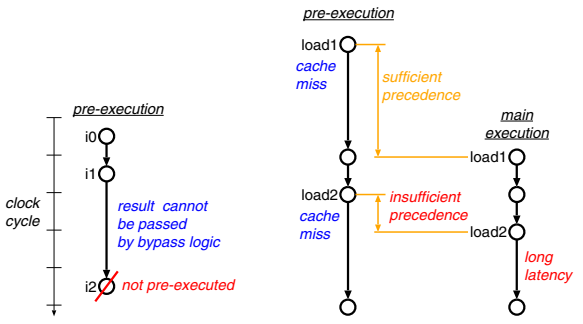
The operations are as follows. First, when an instruction reaches the rename stage, the physical register that is currently allocated to the same logical destination register of the instruction is *temporarily* deallocated, and is appended to the free list. At the same time, the number of the ROB entry, 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 is 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.

Fig. 2 illustrates an example of 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. Fig. 2a 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. 2a). At the same time, the allocated ROB entry number `13` is written into the 36th entry of the DAT (mark (2)).

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

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 write of the result is granted. Then, instructions are issued as per normal.

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



(a) Bypass problem (b) Successive cache miss problem

Fig. 3. Problems in TSD.

C. 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 to dependent instructions via the bypass logic. These instructions form a pre-execution stream, which proceeds earlier than the main execution stream because it exploits the ILP and MLP, where there are no resource constraints on physical registers.

Note that the ready flags of a pre-executed instruction are reset after its issue, and the instruction is not removed from the instruction window. Later, after an ROBP tag is matched, and both ready flags are set again, the instruction is re-executed and the result is written into the destination physical register, as described in Section III-B.

IV. Use of Value Prediction

A. Problems

The TSD has two problems that fails (early) pre-execution, as described in Section I. Fig. 3 illustrates the two problems. The nodes represent instructions, and the edges represent dependences. Fig. 3a shows the first problem, which we call the *bypass problem*, where instruction `i2` is not selected to be issued at the next cycle after instruction `i1` has been issued. In this case, instruction `i1` cannot pass the result to instruction `i2`, because result passing relies on the bypass logic in the pre-execution [1].

The second problem arises from the difference of the throughput between the pre-execution and main execution. The throughput of the pre-execution is lower than that of the main execution, because cache misses occur in the pre-execution while 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 *successive cache miss problem*. In Fig. 3b illustrates this problem. 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.

[1] Actually, the TSD resets the ready flag that was set in the previous cycle, so that an instruction like `i2` is not issued [1], [2].

B. Using Value Prediction

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

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

C. Result Prediction vs. Address Prediction

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

In the result prediction case, a load that potentially causes a cache miss is pre-executed by execution of 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 the address prediction. However, the 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 the address prediction case, the predicted load can access memory immediately without the necessity of other instructions' execution, unlike the result prediction. Thus, the bypass problem can be solved. However, unlike the result prediction, there is no chance of pre-execution if the reference address is unpredictable.

Because of the space limit, in this paper, we will only present the evaluation results from address prediction, though we evaluated both methods. According to our evaluation, the result prediction is inferior to the address prediction because of the bypass problem. Also, the coverage of the result prediction is not so high.

The implementation that incorporates the address prediction into the pre-execution mechanism is identical to that in the conventional method (see Fig. 4). That is, a load is first split into an address calculation instruction and a memory access instruction at the front-end (we assume the split

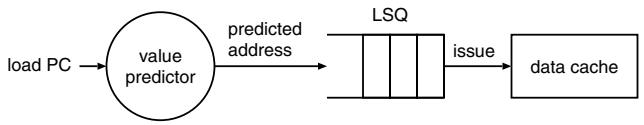


Fig. 4. Issue flow of memory access instruction using address prediction.

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 in the LSQ normally, and the data cache is accessed if issued.

V. Saving Power Consumption

Because every instruction or load consults the value predictor, power consumption is increased. We previously proposed a scheme which suppresses the increase of power consumed by the TSD [23]. The scheme pre-executes only those instructions that have great benefit. We showed that the pre-execution of only a selected small number of instructions is enough to achieve the speedup where instructions are attempted to be pre-executed as much as possible. For saving power consumption of the TSD with value prediction, we can simply extend this scheme. That is, we let only instructions selected for pre-execution consult the value predictor. We do not apply this scheme in this paper, because power consideration is beyond the scope of this paper.

VI. Applicability to Other Pre-Execution Schemes

Although we advocate the use of value prediction in connection to TSD, it is applicable to other pre-execution schemes as well. The bypass problem is unique to the TSD, but the successive cache misses cause problems in any pre-execution scheme, because the delay of pre-execution of a load fails to obtain enough benefit from pre-execution. Pre-execution should run far enough ahead of the main execution to hide memory latency. A data dependence chain with multiple cache misses disrupts this requirement. Breaking dependences using value prediction is effective for ensuring that pre-execution precedes the main execution. This utilization also has the strength of non-necessity of recovery from misprediction.

VII. Evaluation Results

A. 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 SPECfp2000. The programs were compiled using gcc ver.2.7.2.3 with options -O6 -funroll-loops. Table I lists the benchmark programs and their memory statistics during execution on the base processor, whose configuration is described further below.

TABLE I
Statistics of cache and memory accesses.

program	cache miss rate		memory
	L1 data	L2	access rate
ammp	11.1%	31.5%	3.5%
applu	4.3%	48.9%	2.1%
apsi	0.3%	23.3%	0.1%
art	60.5%	45.6%	27.6%
equake	2.9%	24.5%	0.7%
mesa	0.2%	11.4%	0.0%
mgrid	2.4%	25.0%	0.6%
swim	12.0%	31.8%	3.8%

TABLE II
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	64KB, 2-way, 32B line
L1 D-cache	64KB, 2-way, 32B line, 4 ports, 2-cycle hit latency, non-blocking
L2 cache	2MB, 4-way, 64B line, 12-cycle hit latency
Main memory	300-cycle min. latency, 8B/cycle bandwidth
Branch prediction	6-bit history gshare, 8K-entry PHT, 10-cycle misprediction penalty
Mem. disambiguation	perfect

We evaluated the following two models. The first is the *TSD model*, which pre-executes instructions with the conventional TSD. The second is the *TSD-addrpred model*, which uses the address prediction in the TSD, as proposed in this paper.

The configuration of the base processor for the two models is summarized in Table II. We assume perfect memory disambiguation. We have not yet mentioned this, but the TSD has a positive effect on memory disambiguation by pre-executing load address calculation instructions [1], [2]. For fair evaluation, instead of implementing memory dependence predictors in our base simulator, we assume perfect memory disambiguation to exclude the effects thereof.

In the TSD-addrpred model, we use a stride value predictor [24] with a 1024-entry direct-mapped value history table (VHT). The VHT is indexed by the load's PC, and each entry has a tag (same as a cache tag), an immediately previous value, a stride value, and a confidence flag. The value predictor predicts the reference address 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.

B. Reduction of Physical Registers

Fig. 5 shows the geometric mean of IPC for each model, with the number of physical registers varying from 34 to 112 (equal numbers of integer and floating-point registers). The

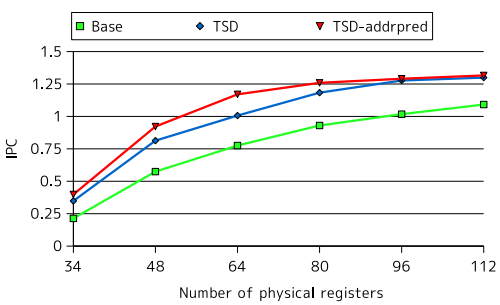


Fig. 5. IPC when varying number of physical registers.

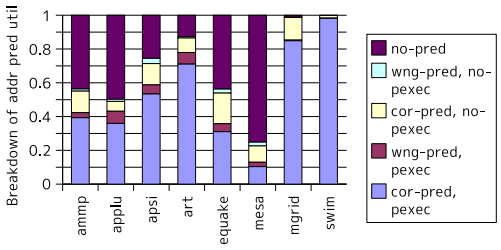


Fig. 6. Breakdown of address prediction utilization.

TSD model can achieve the same performance as the base processor, with a fewer number of physical registers. This effect is augmented by using address prediction.

Here, we attempt to present the reduction rate of the physical registers by obtaining a representative value. As intuitively found, 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:

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

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 in that 1) the ROB size gives the number of supported in-flight instructions, where each in-flight instruction of *ROBsize* requires a physical register, and 2) each committed logical destination register requires a physical register. Because the ROB size is 128 by default, 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, because dynamic instructions with destination registers being either integer or floating point is roughly equal in the benchmark programs. Therefore, there are 96 physical registers each of integer and floating-point. We call this number the *baseline number of physical registers*.

From Fig. 5, we find the TSD and TSD-addrpred models can, with 64 and 48 physical registers, respectively, achieve equivalent performance to the base processor with the baseline number of physical registers. The reduction rate of the physical registers is 33% for the conventional TSD, and 50% using address prediction. Compared with the conventional TSD without the address prediction, the address prediction reduces the number of physical registers by 25%.

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

Unfortunately, the effect of address prediction is deteriorated as the number of physical registers increases. This is because the chance of instruction pre-execution decreases as the number of physical registers increases, and consequently, the chance of using the address prediction becomes less.

C. Breakdown of Address Prediction Utilization

This section shows the breakdown of the address prediction utilization with regard to the correctness of prediction and whether or not the predicted value is used for pre-execution. Fig. 6 shows the result. The data are collected in the TSD-addrpred model with 48 physical registers. This configuration gives the equivalent performance to the base processor as described before. Each bar is partitioned into the five portions. Dynamic loads are categorized into the following classes:

- *cor-pred, pexec*: prediction is correct, and the predicted load is pre-executed.
- *wng-pred, pexec*: prediction is wrong, but the predicted load is pre-executed.
- *cor-pred, no-pexec*: prediction is correct, but predicted load is not pre-executed.
- *wng-pred, no-pexec*: prediction is wrong, and the predicted load is not pre-executed.
- *no-pred*: prediction is not made due to low confidence.

Here, the reason that the predicted load is not pre-executed is either 1) its destination register is deallocated with the second-step deallocation and thus it becomes a non-candidate of pre-execution before it is pre-executed, or 2) the associated address calculation instruction is performed and the reference address becomes available. Therefore, the predicted address becomes unnecessary.

The coverage rate, i.e., the rate of predicted loads relative to the total number of dynamic loads, is fairly high in most programs, as shown in Fig. 6. Also, not surprisingly, many loads whose addresses are predicted are pre-executed in most programs, because such loads are issued with little wait in the LSQ.

D. Pre-Execution Rate

As inferred from Fig. 6, the *pre-execution rate of loads*, the rate of loads that are pre-executed once or more relative to the total number of dynamic loads, is expected to be increased. Fig. 7 shows the pre-execution rate of loads for the TSD and TSD-addrpred models. As in Section VII-C, the data are collected in the case of 48 physical registers.

As shown in the graph, the TSD-addrpred model exhibits the higher rate than the TSD model for all program. In particular, the rate is significantly higher in *applu*, *apsi*, *equake*, and *swim*.

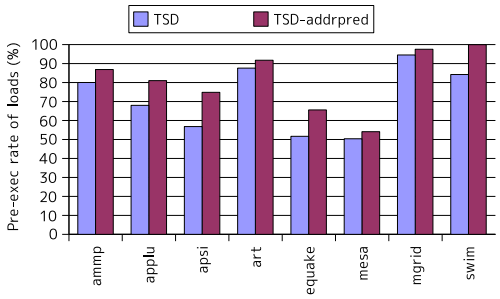


Fig. 7. Pre-execution rate of loads.

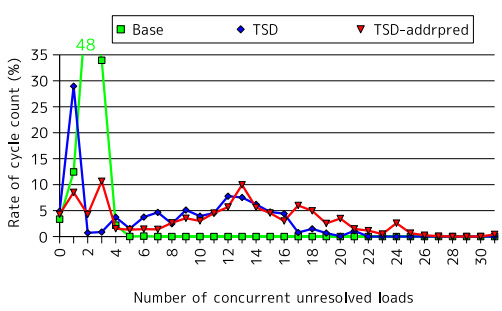


Fig. 8. Distribution of number of unresolved loads whose resolution takes more than or equal to 300 cycles exist simultaneously in art.

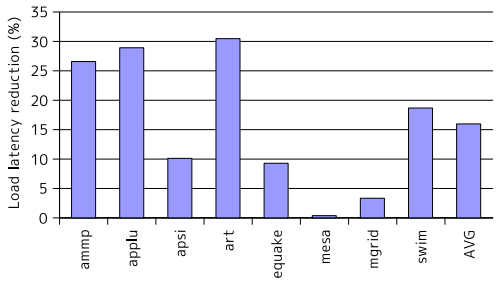


Fig. 9. Load latency reduction rate of TSD-addrpred model relative to TSD model.

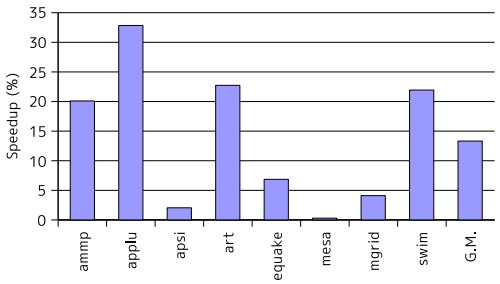


Fig. 10. Speedup of TSD-addrpred model over TSD model.

E. Increase of MLP Exploitation

The increase of MLP exploitation is expected because the pre-execution rate of loads is increased. Fig. 8 shows the distribution of the number of concurrent unresolved loads whose cache miss resolution takes 300 (minimum memory latency) or more cycles in *art*, the program where the memory access rate is the highest among our benchmark programs, as an example. The horizontal-axis represents the number of unresolved loads that exist simultaneously, and the vertical-axis represents the rate of the cycle count for each number of concurrent unresolved loads relative to the total cycle count. We assume 48 physical registers.

As shown in the figure, the distribution peaks at a very few 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 to the right significantly, and the rate for less than 3 unresolved loads is lowered. This indicates that the TSD-addrpred model greatly exploits MLP.

F. Performance

This section discusses how value prediction affects performance. Fig. 9 and 10 show, respectively, the load latency reduction rate and the speedup of the TSD-addrpred model relative to those of the TSD model. We assume 48 physical registers. The load latency is significantly reduced (16% on average), and consequently speedup is considerable (13%). Not surprisingly, the speedup is strongly correlated to the latency reduction rate, but the amount of speedup is very different in each program. The speedup is related to the following three items listed in Table III.

- Memory access rate (*mem*): the rate of the number of loads that access memory in the total number of dynamic loads in the base processor.
- Value prediction accuracy (*pred*): the rate 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 III, we represent how much each item contributes to the effectiveness by "–," "+,"and "++" from weak to strong. Table IV summarizes the result of our analysis. From the table, we can discuss the effectiveness as follows.

In *ammp*, our scheme achieves a great speedup, because *mem*, *pred*, and *pexec* is evaluated to more than "+." In *art* and *swim*, there appears to be little room to improvement because *pexec* is evaluated to "–," but they are considerably memory-intensive, as indicated by *mem* being evaluated to "++." Also, *pred* is evaluated to "++." For these reasons, our scheme achieves a significant speedup despite a *pexec* of "–." In *apsi*, *equake*, *mesa*, and *mgrid*, it is unlikely that pre-execution is effective, because they are not memory-intensive, as indicated by *mem* being evaluated to "–." In *applu*, although *pred* is evaluated to "–," the prediction accuracy is not very low (approximately 50%). As both *mem* and *pexec* are evaluated to "+," the possibility and scope for

TABLE III

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%

TABLE IV

CAUSE ANALYSIS OF EFFECTIVENESS.

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

performance improvement is large. Therefore, our scheme is effective.

VIII. 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 scheme does not need recovery from misprediction, which is usually a considerably negative factor. Our evaluation using SPECfp2000 benchmark programs shows that our scheme exploits more MLP, and significantly reduces the register file size with little performance loss. Although we advocate this new way of using value prediction within the TSD framework, it can also be applied to any other pre-execution schemes.

ACKNOWLEDGMENT

The authors thank anonymous reviewers for their useful comments. This work was partially supported by the Ministry of Education, Culture, Sports, Science and Technology Grant-in-Aid for Scientific Research (C)(No. 19500041).

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," in *MEDEA-8*, September 2007, pp. 41–48.

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

[3] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous subordinate microthreading (SSMT)," in *ISCA-26*, May 1999, pp. 186–195.

[4] C. Zilles and G. S. Sohi, "Master/slave speculative parallelization," in *MICRO-35*, November 2002, pp. 85–96.

[5] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A study of slipstream processors," in *MICRO-33*, December 2000, pp. 269–280.

[6] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *HPCA-7*, January 2001, pp. 37–48.

[7] 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," in *ISCA-28*, July 2001, pp. 14–25.

[8] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *MICRO-34*, December 2001, pp. 306–317.

[9] C. B. Zilles and G. S. Sohi, "Execution-based prediction using speculative slices," in *ISCA-28*, July 2001, pp. 2–13.

[10] D. M. Tullsen, S. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ISCA-22*, June 1995, pp. 392–403.

[11] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," in *HPCA-9*, February 2003, pp. 129–140.

[12] M. H. Lipasti and P. J. Shen, "Exceeding the dataflow limit via value prediction," in *MICRO-29*, December 1996, pp. 226–237.

[13] F. Gabbay and A. Mendelson, "The effect of instruction fetch bandwidth on value prediction," in *ISCA-25*, July 1998, pp. 272–281.

[14] Y. Sazeides and J. E. Smith, "The predictability of data values," in *MICRO-30*, December 1997, pp. 248–258.

[15] G. Reinman and B. Calder, "Predictive techniques for aggressive load speculation," in *MICRO-31*, December 1998, pp. 127–137.

[16] H. Zhou and T. M. Conte, "Enhancing memory level parallelism via recovery-free value prediction," in *ICS-17*, June 2003, pp. 326–335.

[17] 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," in *MICRO-38*, November 2005, pp. 223–244.

[18] A. González, J. González, and M. Valero, "Virtual-physical registers," in *HPCA-4*, February 1998, pp. 175–184.

[19] T. Monreal, A. González, M. Valero, J. González, and V. Viñals, "Delaying physical register allocation through virtual-physical registers," in *MICRO-32*, November 1999, pp. 186–192.

[20] M. Moudgill, K. Pinagli, and S. Vassiliadis, "Register renaming and dynamic speculation: An alternative approach," in *MICRO-26*, December 1993, pp. 202–213.

[21] D. Balkan, J. Sharkey, F. Ponomarev, and A. Aggarwal, "Address-value decoupling for early register deallocation," in *ICPP-2006*, August 2006, pp. 337–346.

[22] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *ASPLOS-11*, October 2004, pp. 107–119.

[23] K. Hyodo, K. Iwamoto, and H. Ando, "Energy-efficient pre-execution techniques in two-step physical register deallocation," *IEICE Transactions on Information and Systems*, vol. E92-D, no. 11, November 2009 (to appear).

[24] T. F. Chen and J. L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. on Comp.*, vol. 44, no. 5, pp. 609–623, May 1995.