IEICE TRANS. INF. & SYST., VOL.E92–D, NO.11 NOVEMBER 2009

PAPER

Energy-Efficient Pre-Execution Techniques in Two-Step Physical Register Deallocation

Kazunaga HYODO[†*], **Kengo IWAMOTO**[†], ***Nonmembers*, *and* Hideki ANDO**[††a)], ***Member***

SUMMARY Instruction pre-execution is an effective way to prefetch data. We previously proposed an instruction pre-execution scheme, which we call *two-step physical register deallocation* (TSD). The TSD realizes pre-execution by exploiting the difference between the amount of instruction-level parallelism available with an unlimited number of physical registers and that available with an actual number of physical registers. Although previous TSD study has successfully improved performance, it still has an inefficient energy consumption. This is because attempts are made for instructions to be pre-executed as much as possible, independently of whether or not they can significantly contribute to load latency reduction, allowing for maximal performance improvement. This paper presents a scheme that improves the energy efficiency of the TSD by pre-executing only those instructions that have great benefit. Our evaluation results using the SPECfp2000 benchmark show that our scheme reduces the dynamic pre-executed instruction count by 76%, compared with the original scheme. This reduction saves 7% energy consumption of the execution core with 2% overhead. Performance degrades by 2%, compared with that of the original scheme, but is still 15% higher than that of the normal processor without the TSD.

key words: *microarchitecture, microprocessor, instruction pre-execution, low power*

1. Introduction

The load latency in cycles increases as LSI technology advances because the rate of improvement of memory access time is much slower than that of the processor's clock frequency. This gap between processor and memory is often called a memory wall. A general method of reducing latency due to a memory wall is to fill the gap with several cache hierarchy levels, and to satisfy load requests at as high a level as possible. However, this method is both very costly and often insufficient.

Data prefetching is an alternative or additional method of solving this problem. Many hardware schemes have been proposed for prefetching of data with both a regular pattern (e.g., [1]) and an irregular pattern (e.g., [2]). Instruction pre-execution (e.g., [3]) is effective for prefetching data with an irregular pattern. Pre-execution of loads that miss the cache moves data from a lower level to an upper level of the memory hierarchy, before the main execution that builds the architectural state.

We previously proposed an instruction pre-execution scheme, which we call *two-step physical register deallocation* (TSD) [4], [5]. A feature of this scheme that is different to other pre-execution based schemes is that the TSD realizes prefetching within a single thread context with a modest amount of simple hardware. The TSD removes the pipeline stall in the rename stage that 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 or a small buffer called the *forwarding buffer* [6], and thus pre-execution can be performed continuously. If the pre-executed load incurs a cache miss, the requested data is moved from memory to a cache, thus realizing data prefetching. Later, when the temporarily allocated physical register has actually been available, the instruction is notified, and is 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 will hit the cache, which would have been a miss without the prefetch in the earlier pre-execution.

Although TSD can reduce load latency, the previous study did not take into account energy consumption. The TSD attempted to pre-execute as many instructions as possible for maximal performance improvement. The instruction that can, however, potentially contribute to reducing load latency are those that are directly or indirectly related to a load causing a cache miss. Since pre-execution consumes extra energy of the execution core (the instruction window, function units, D-caches, etc.), to improve energy efficiency, only instructions that contribute greatly to performance improvement should be selectively pre-executed.

In this paper, we propose a scheme that achieves selective pre-execution for TSD [7]–[9]. Our scheme comprises the following two steps:

1. The scheme finds a *delinquent load* [10], and then searches for instructions on which the load depends, either directly or indirectly.
2. It optimizes the length of the instruction sequence obtained to achieve high energy efficiency.

Manuscript received June 8, 2009.

[†]The authors are with the Department of Computational Science and Engineering, Nagoya University, Nagoya-shi, 464–8601 Japan.

[††]The author is with the Department of Electrical Engineering and Computer Science, Nagoya University, Nagoya-shi, 464–8601 Japan.

[*]Presently, with Denso Corp.

a) E-mail: ando@nuee.nagoya-u.ac.jp

DOI: 10.1587/transinf.E92.D.2186

In the first step, the scheme finds delinquent loads by measuring the frequency of the L2 cache misses for each load. Here, delinquent loads are those that cause frequent L2 cache misses. It is well known that a small number of loads are responsible for the majority of L2 cache misses. Next, the scheme stores the instructions preceding the delinquent load in a buffer, and then searches for the instructions on which the load depends, either directly or indirectly using dataflow analysis. We call such instructions *transitive producers* or *TP instructions*. The delinquent load and the TP instructions are marked, and become the initial set for the next optimization.

In the second step, the scheme attempts to shorten the TP-instruction sequence as much as possible for high energy efficiency by learning. Specifically, for the delinquent load obtained, the scheme measures *precedence cycles*, which indicate how many cycles earlier the pre-execution of the delinquent load is performed relative to the main execution of the load. The scheme then *prunes* enough TP instructions from the head of the sequence, as long as the number of precedence cycles is not reduced.

In this paper, we propose a scheme based on TSD. However, conventional thread-based pre-execution schemes have the same problem, and our scheme (at least, the basic concept) is applicable to them as well.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 explains the original TSD scheme proposed previously. Sections 4 and 5 describe the first and second steps of our scheme, respectively. Section 6 explains applicability of our scheme to other pre-execution schemes. Evaluation results are presented in Sect. 7, and our conclusions are stated in Sect. 8.

2. Related Work

A number of prefetch studies based on pre-execution have been carried out [3], [10]–[15]. Most of these schemes extract the instructions necessary for prefetching *statically* as a thread, and then spawn this thread at a certain point in the program execution to a different context of the processor. An energy-aware thread selection scheme proposed by Petric *et al.* [16] estimates performance benefit and energy consumption analytically, and then selects threads with good performance/energy tradeoff. The disadvantage of these schemes, including Petric's scheme, is that they require a multithreaded environment such as simultaneous multithreading [17] or chip multiprocessors. Even in such an environment, a disadvantage arises in that the context is consumed; it may be more profitable to allocate other threads to the context for a better throughput. Furthermore, static thread extraction and thread selection requires recompilation.

Among the multithreaded pre-execution schemes, a scheme in which pre-execution instructions are extracted *dynamically* is, to our knowledge, only the scheme proposed by Collins *et al* [18]. Like our scheme, this scheme also focuses on delinquent loads for energy efficiency, but unlike our scheme, it greedily extracts pre-execution instructions for high performance and thus does not optimize the length of the pre-execution instruction sequence for energy saving.

The only pre-execution scheme, to our knowledge, that does not need a multithreaded environment is *runahead execution* [19], which attempts to exploit memory-level parallelism (MLP) when an L2 cache miss occurs. Mutlu *et al.* proposed several schemes to improve energy efficiency, for example, by suppressing ineffective runahead execution based on past history [20]. Yet, they neither focus on delinquent loads nor optimize the length of the runahead (i.e., pre-executed) instruction sequence, unlike in our scheme.

3. Instruction Pre-Execution through TSD

The TSD assumes 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. In this section, we shortly present the TSD scheme that forms the basis of our study. See [4], [5] for more details. First we illustrate the effect of TSD, and then explain the scheme, by describing both the basic TSD and the extension for pre-execution.

3.1 Effect of TSD

Figure 1 illustrates the effect of TSD. An example of the execution timing of four dependent instructions in a conventional processor, where `load` incurs a cache miss, is shown in Fig. 1 (a), while Fig. 1 (b) shows the execution timing of the same instructions 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 cache miss of `load` occurs earlier. Handling this cache miss moves data to the upper level in the memory hierarchy. As a result, in the main execution, `load` hits the L1

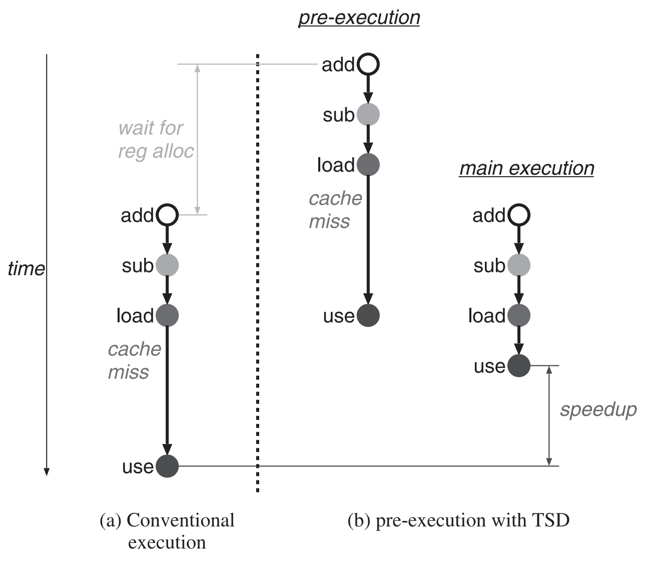


Fig. 1 Effect of TSD.

data cache, resulting in a speedup.

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, and holds the number of the 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.

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

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 to dependent instructions via the bypass logic or *forwarding buffer* [6], which is a small fully-associative buffer that stores recent results. These instructions form a pre-execution stream, that proceeds faster than the main execution stream because it exploits the instruction level parallelism, where there are no resource constraints with regard to 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 Sect. 3.2.

4. Selecting Pre-Executed Instructions

This section explains the first step of our scheme; that is, finding a delinquent load and then searching for the TP instructions of the load.

4.1 Finding Delinquent Loads

Delinquent loads are those that incur L2 cache misses frequently. To find delinquent loads, we count the L2 cache misses for each load in a given interval. We prepare a table called the *miss count table* (MCT), which is indexed by the load PC. Each entry in this table contains the following fields:

- a valid flag (V flag)
- a counter that counts L2 cache misses
- a flag that indicates whether or not the search for TP instructions has been performed (C flag)

When a load causes an L2 cache miss, it reads the corresponding entry in the MCT. If the entry is valid, the counter in the entry is incremented. Otherwise, the entry is initialized by setting the counter to 1, clearing the C flag, and setting the V flag. The MCT is reset at constant intervals by clearing all the V flags. If a load increments the counter value, which then reaches a predetermined threshold with the C flag not set, we determine that the load is delinquent, and start the search for its TP instructions (as described in Sect. 4.2).

Note that the energy consumed by the MCT accesses is small, because the MCT is accessed when an infrequent L2 cache miss occurs and a delinquent load is decoded (as described in Sect. 4.2). We give a quantitative evaluation of the energy consumption in Sect. 7.6.

4.2 Dynamic Search of TP Instructions

In the search for the TP instructions of a delinquent load (referred to as a *TP search*), we prepare a FIFO buffer called the *retired instruction buffer* (RIB). A TP search involves two steps.

The first step stores committed instructions (precisely, the PC and the designation numbers of the destination and source registers of the committed instructions) in the RIB. The RIB normally sleeps (dropping or stopping the power supply), but is activated to append committed instructions. The activation is triggered when an instruction marked as a *trigger* is decoded. In our implementation, the trigger instruction is a delinquent load. When a delinquent load is found by the MCT, it is marked as a trigger. To denote the mark, we add a flag called the *TG flag* to each instruction in the I-cache. If a trigger instruction is decoded, the MCT is consulted. If the C flag in the corresponding entry is not set, this indicates that a TP search has not yet been performed in the current interval. In this case, the trigger instruction activates the RIB, and the committed instructions are sequentially appended to the RIB until the trigger instruction (i.e., delinquent load) is committed.

The second step starts when the trigger instruction is committed. The TP instructions of the delinquent load are searched for using dataflow analysis by reading the instructions of the RIB from tail to head. This analysis conceptually traverses the edges of dataflow graph in a reverse direction from the node of the delinquent load.

Figure 2 gives the algorithm, which is similar to live register analysis. First, the source register *sreg* of the delinquent load is assigned as the initial element in the set *LIVE* (1st line). Next, instructions are read from the RIB from tail to head, and the following procedure performed until *LIVE* is empty. If the destination register *dreg* of the instruction read is a member of *LIVE*, the instruction is marked as a TP instruction (5th line). Then, the *dreg* is removed from *LIVE* (6th line), and the *sreg(s)* added to *LIVE* (7th line).

We represent the *LIVE* set with a bit vector. That is, if register *i* is a member of *LIVE*, the *i*-th bit is set. This implementation allows set operations to be carried out by bit-wise logical operations, thus simplifying the hardware.

Note that a flag called the *TP flag* is prepared for each instruction in the I-cache to denote the TP mark. Instructions with their TP or TG flag set are pre-executed, if possible (this control is modified later as stated in Sect. 5).

After finishing the TP search, the C flag in the MCT entry corresponding to the delinquent load is set. This prevents the TP search for the associated delinquent load from being performed again in the current interval. Also, the RIB switches to sleep mode to minimize energy consumption.

All TP and TG flags are cleared at the end of each interval. This keeps track of the transitions in the program execution phases.

Although we define a delinquent load as a trigger instruction, which instruction should be the trigger is a performance issue. However, the ideal case, in which a sufficiently large RIB (1K entries in this evaluation) is always active, improves performance by only 0.5% (we assume that the TP search consumes no clock cycles independently of the RIB size in this evaluation, unlike the evaluation in Sect. 7). Therefore, our trigger timing is early enough. On the other hand, an early trigger may extract more TP instructions than is necessary, but unnecessary TP instructions are pruned in the later optimization, which is described in the next section.

```
1: $LIVE := sreg$ of $delinquent\ load$;
2: foreach $inst \in RIB$ (from tail to head) {
3:     if $(LIVE = \phi)$ break;
4:     if ($dreg$ of $inst \in LIVE$) {
5:         mark $inst$ as "TP";
6:         $LIVE := LIVE - dreg$ of $inst$;
7:         $LIVE := LIVE \cup sreg(s)$ of $inst$;
8:     }
9: }
```

Fig. 2 Algorithm of TP search.

5. Optimizing the Length of TP-Instruction Sequence

The effectiveness of the prefetch depends on the *precedence cycles* of a pre-executed delinquent load, where precedence cycles of an instruction are the number of cycles by which the instruction is pre-executed earlier than its main execution. In TSD, precedence cycles are obtained by removing the resource constraint with respect to physical registers. However, there are instructions that does not receive such benefit (e.g., instructions whose data dependence constraint is more severe). Apparently, their pre-execution is energy-inefficient. This motivates our optimization of the length of the TP-instruction sequence[†].

5.1 Optimization Algorithm

The adjustment to the length of the TP-instruction sequence *prunes* the appropriate number of TP instructions from the head of the sequence. Specifically, pruning inhibits pre-execution of the specified number of instructions from the head instruction (*HD instruction*) of the sequence, even if such instructions are TPs. Note that pruned instructions are executed normally, and the following dependent TP instructions are pre-executed using the results. To implement this, we first define an extra flag per instruction in the I-cache, which we call the *HD flag*, to represent an HD instruction. Next, we prepare a table called the *TP pruning table* (TPPT). Each entry in the TPPT is associated with an HD instruction and contains three fields: *nprune*, *max_pcy*, and *ntp*. *Nprune* is the number of pruning instructions, *max_pcy* is the maximum of the precedence cycles of the delinquent load associated with the HD instruction obtained from past executions, and *ntp* is the number of TP instructions included in the TP sequence.

When a TP search finishes, we initialize the entry in the TPPT associated with the HD instruction of the searched TP-instruction sequence. That is, *nprune* and *max_pcy* are reset to 0, and *ntp* is set to the number of searched TP instructions.

The number of pruned instructions is determined by

```
1: $pcy := precedence\ cycles\ of\ delinquent\ load$;
2: if $(pcy \le max\_pcy)$ {
3:     if $(max\_pcy - pcy < PCY_{th})$
4:         $nprune := min(nprune + \Delta_t, ntp)$;
5:     else
6:         $nprune := max(nprune - \Delta_t, 0)$;
7: } else {
8:     $nprune := max(nprune - \Delta_t, 0)$;
9:     $max\_pcy := pcy$;
10: }
```

Fig. 3 Learning algorithm for pruning TP instructions.

[†]We use the term *sequence*, but, of course, TP instructions are not necessarily contiguous instructions. Remember that they are the instructions on which a delinquent load depends, either directly or indirectly.

learning. The algorithm is given in Fig. 3. When the execution of a delinquent load terminates, the precedence cycles, *pcy*, are obtained (1st line). If *pcy* is smaller than or equal to the current *max_pcy*, the algorithm evaluates by how much *pcy* has decreased compared to *max_pcy*. If the decrease is smaller than the predetermined threshold PCY_{th} (3rd line), the algorithm predicts that pruning more instructions will not affect the precedence cycles. Then, *nprune* is increased by the predetermined constant value Δ_t (4th line). Otherwise, the algorithm learns that too many instructions were pruned in the past. Therefore, *nprune* is decreased by Δ_t (6th line). If *pcy* is larger than *max_pcy* (7th line), the algorithm detects a move of the baseline for pruning, and thus it attempts to transfer the algorithm to the initial state slowly. That is, *nprune* is decreased to Δ_t (8th line). Also, it updates *max_pcy* (9th line). Note that the fluctuation in *max_pcy* is mostly caused by a fluctuation in the instruction latency (e.g., cache misses).

5.2 Confirming Validity

The evaluation of the precedence cycles in the optimization is valid, only if the control of execution traverses the same path as that covered by the TP search. If the control of execution reaches a delinquent load by taking a different path, some dependence edges in the TP-instruction sequence are cut. Due to this, a TP instruction will wait for main execution of a non-TP instruction, as a result, the precedence cycles are reduced. To confirm the validity in a simple manner, we use heuristics to monitor the number of TP instructions fetched starting with the HD instruction and ending with the delinquent load. The optimization algorithm given in Fig. 3 is performed, only if this number matches the number of TP instructions found during the TP search, and which is stored in the TPPT as *ntp*.

6. Applicability to Thread-Based Schemes

Although we proposed our scheme as one strongly connected to TSD, it (at least, the basic concept) is applicable to conventional thread-based pre-execution schemes. First, focusing on delinquent loads allows good power/performance threads to be extracted, compared to most previous schemes which do not have an effective policy for extracting power-efficient threads. From our evaluation results, load latency can be significantly reduced with only a small number of threads related to delinquent loads. Delinquent loads can be found in a dynamic scheme using our scheme. In a static scheme, they can be found by profiling at compile time. Focusing on delinquent loads also suppresses code inflation in a static scheme, because the number of static threads is reduced.

Second, our pruning algorithm of the TP-instruction sequence is an effective and simple way to optimize the length of a thread, particularly for dynamic schemes. In these, it is difficult to find the minimum length of a thread while still obtaining maximal performance gain. This is because hardware does not have a global view of data dependence information, and thus it is difficult to calculate how much each instruction in the thread contributes to load latency reduction.

7. Evaluation Results

To evaluate our scheme, we built a simulator based on the SimpleScalar Tool Set version 3.0a [21]. The instruction set is SimpleScalar/PISA, which is an extension of the MIPS R10000 ISA. We use eight programs from SPECfp2000. The programs are 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, whose configuration is described later. Note that the L2 miss rate and the memory access rate do not include the accesses by the stride prefetcher.

We evaluate the energy consumption as follows. The energy consumed by a structure is derived by multiplying the energy consumed by a single operation on the structure by the total number of operations on the structure. The number of operations on each structure is obtained from the SimpleScalar simulation. To obtain the energy consumed by a single operation on an array and CAM structure, we used the CACTI [22] modified by us to use the real wire parameters [23] instead of the scaled parameters used in the original CACTI. This modification is important to estimate the precise energy consumption. Furthermore, we used the models from the Wattch [24] as combinational structures, and scaled their energy for the assumed technology. We assume the 70 nm technology and a supplied voltage of 1.0 V.

We evaluated the following two models. The first is the *full model*, which attempts to pre-execute all instructions through TSD as far as possible. The second is the *energy-efficient model*, which selectively pre-executes instructions based on the scheme described in this paper.

The configuration of the base processor for the two models is summarised in Table 2. Not to overestimate the prefetch effect of the TSD, we introduce a stride prefetcher using a per-load stride predictor [25], and a stream buffer [1] between the L1 data cache and the L2 unified cache to avoid pollution of the L1 data cache. Also, 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 [4], [5]. For fair evaluation, instead of implementing memory

Table 1 Statistics of cache and memory accesses.

program	miss rate			memory
	L1 D-cache	stream buffer	L2	access rate
ammp	9%	59%	29%	1.59%
applu	5%	68%	29%	0.96%
apsi	1%	20%	3%	0.00%
art	48%	31%	85%	12.79%
equake	4%	31%	22%	0.29%
mesa	1%	18%	10%	0.02%
mgrid	3%	3%	9%	0.01%
swim	13%	59%	13%	1.03%

Table 2 Base configuration.

Pipeline width	4-instruction wide for each of fetch, decode, issue, and commit
Branch prediction	6-bit history gshare, 8K-entry PHT, 512-entry, 4-way BTB 10-cycle misprediction penalty
ROB	128 entries
LSQ	64 entries
Instruction window	64 entries
Physical registers	168 (int 84, fp 84)
Function unit	4 iALU, 2 iMULT/DIV, 4 fpALU, 2 fpMULT/DIV/SQRT
L1 I-cache	64 KB, 2-way, 32 B line
L1 D-cache	64 KB, 2-way, 32 B line, 2 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
Data prefetcher	stride prefetcher, incremental prefetching [25]
Stream buffer	8-way, 4 KB each, 32 B line, 1-cycle hit latency
Mem. disambiguation	perfect

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%

Table 4 Configuration of full and energy-efficient models.

common	8-entry, fully-associative forwarding buffer
energy-efficient model	1024-entry tagless MCT. 64-entry RIB. Threshold of miss count to identify delinquent load is 16. Reset interval of MCT and TP, TG, and HD flags is 1 M cycles. 128-entry tagless TPPT. PCY_{th} is 30 cycles. Δ_t is 3 cycles.

dependence predictors in our base simulator, we assume perfect memory disambiguation to exclude the effects thereof.

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 used the following equation:

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

where $Npregs$ and $Nlregs$ are the total number of physical and logical registers, respectively. The coefficient α is the ratio of dynamic instructions that have integer or floating-point destination registers to the total number of dynamic instructions. 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 $\alpha \times ROBsize$ requires a physical register, and 2) each committed logical destination register requires a physical register. Next, we divided the total number of physical registers equally between integer and floating-point registers. This is reasonable as the number of dynamic instructions with destination registers being either integer or floating point is roughly equal in the benchmark programs we used. Table 3 lists the percentage of dynamic instructions categorized by the type of the destination register in the baseline processor. From the table, we obtain 0.83 as α. Since the ROB size is 128 by default, we obtain 170 ($128\times0.83+64$) as the number of physical registers. Finally, we round the number to the nearest multiple of 8, giving us 168 physical registers.

The specific default parameters for the full and energy-efficient models are listed in Table 4. We also conservatively assume that the TP search consumes 10 cycles per entry in the RIB. During the search, the scheme accesses the I-cache to set the TP flags. Therefore, we also conservatively assume that the instruction fetch is stalled during the search

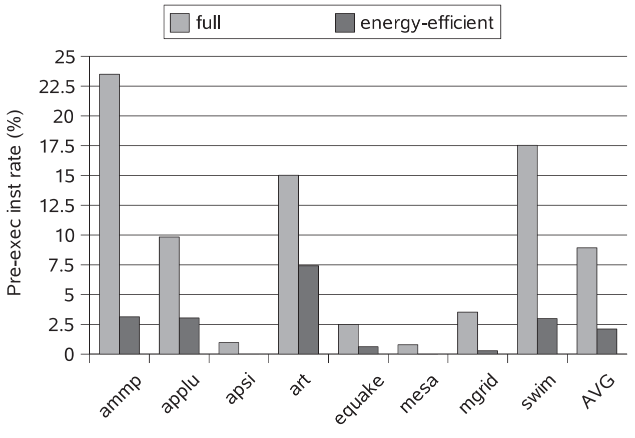


Fig. 4 Pre-executed instruction rate.

for TP instructions, regardless of whether or not a TP flag is written.

7.1 Pre-Executed Instruction Rate

Figure 4 shows the *pre-executed instruction rate*, the ratio of pre-executed instructions to committed instructions, for both the full and energy-efficient models. As seen in the figure, the energy-efficient model significantly reduces the pre-executed instruction rate (76% reduction on average). As expected, the reduction rate of the pre-executed instructions correlated negatively to the memory access rate (see Table 1). For example, the reduction rates of *apsi*, *mesa*, and *mgrid* are more than 90%, while the memory access rates of these programs are very low. In contrast, the reduction rate of *art* is not as high (51%), but the memory access rate is very high.

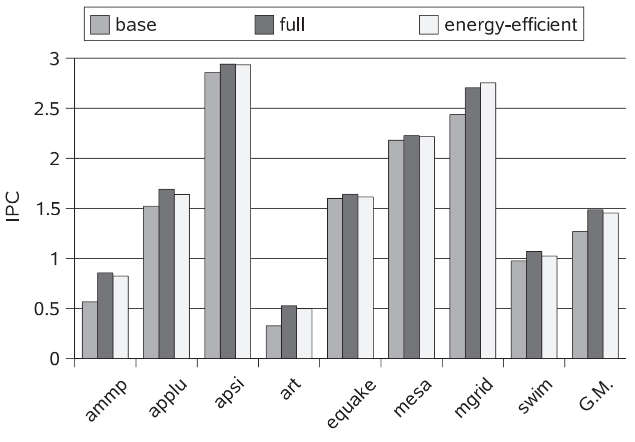


Fig. 5 IPC.

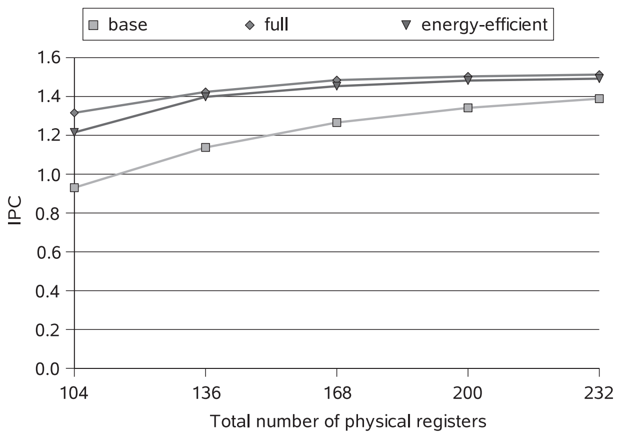


Fig. 6 IPC when only total number of physical registers is varied.

7.2 Performance

Figure 5 depicts IPC for the base, full, and energy-efficient models, and show that the degradation of the energy-efficient model relative to the full model is small (2% on average). There are three possible reasons for this performance degradation. First, our scheme cannot hide the latency of loads that incur L2 cache misses infrequently. The impact of this on the performance is small as seen in Fig. 9 shown below. The difference in performance degradation for a threshold between 1 and 64 that detects delinquent loads is negligibly small (note that loads that rarely incur L2 cache misses are selected for pre-execution when the threshold is 1). Second, our scheme does not hide most L1 cache miss penalties. Third, our scheme hardly receives any benefit from the removal of true dependence between an address calculation and load, which is part of the TSD's ability. Although we have not explained this ability, the TSD pre-executes address calculation instructions, and the results are written into the LSQ (we assume a split load/store, where the load/store operation is split into an address calculation and a memory access). A dependent load then uses the calculated address in the main execution, without having to await its calculation [4], [5]. This removes the true dependence between an address calculation and a load. Our scheme loses most of this benefit. We believe that most of the performance degradation is due to the latter two reasons.

Although the performance of the energy-efficient model is slightly inferior to that of the full model, it is still 15% higher than the base model.

Figure 6 shows the IPC of the three models, while varying only the total number of the physical registers. As the total number of physical registers increases, the improvement rate of the two TSD models over the base model decreases. However, the energy-efficient model still exhibits 7% better performance over the base model even with 232 physical registers (38% more than the default number).

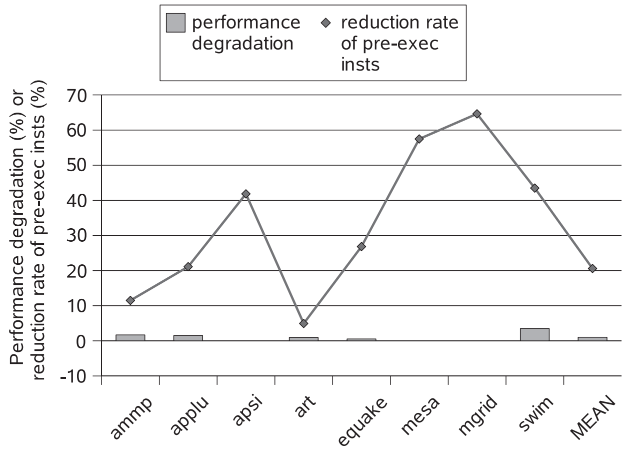


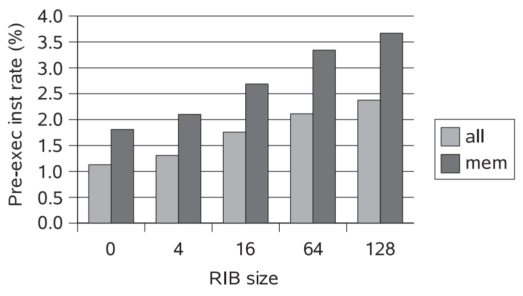
Fig. 7 Effect of TP-instruction sequence length optimization.

7.3 Effect of TP-Instruction Sequence Length Optimization

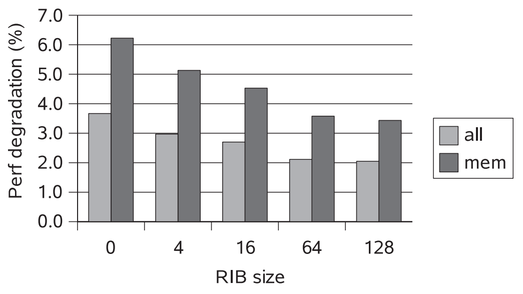
Figure 7 shows the effect of the optimizing the TP-instruction sequence length. The line and bars represent respectively, the pre-executed instruction reduction rate and performance degradation of the optimized model, relative to the model without the optimization. The rightmost values of the graph are the arithmetic mean and geometric mean of the pre-executed instruction reduction rate and performance degradation, respectively. As shown in the figure, the optimization is very effective in reducing the number of pre-executed instructions (21% reduction on average), while the adverse effect on performance is only slight (1% degradation on average).

7.4 Sensitivity to RIB Size

Figures 8 (a) and (b) show, respectively, the pre-executed instruction rate of the energy-efficient model and its performance degradation relative to that of the full model, for varying RIB sizes. Note that the meaningful maximum size

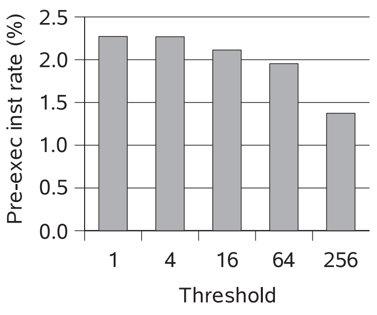


(a) Pre-executed instruction rate

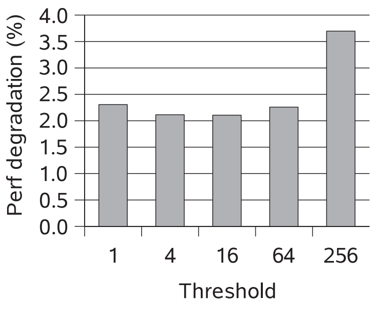


(b) Performance degradation

Fig. 8 Sensitivity to RIB size.



(a) Pre-executed instruction rate



(b) Performance degradation

Fig. 9 Sensitivity to threshold of delinquent load detection.

of the RIB is the ROB size (i.e., 128). The left bar for each RIB size represents the average of all benchmark programs, and the right bar represents the average of only memory-intensive programs (*ammp*, *applu*, *art*, *equake*, and *swim*).

As expected, the pre-execution instruction rate increases as the RIB size increases. As this rate increases, the prefetch timeliness improves, and consequently, the performance degradation decreases. These characteristics are seen more clearly in memory-intensive programs.

7.5 Sensitivity to Threshold of Delinquent Load Detection

Figures 9 (a) and (b) show, respectively, the pre-executed instruction rate of the energy-efficient model and its performance degradation relative to that of the full model, for varying thresholds for detecting delinquent loads. In general, as the threshold increases, the detected delinquent loads decreases and the learning time for the detection increases. This leads to greater performance degradation, but lower pre-executed instruction rates, as illustrated by the two figures. An oddity is observed in that the performance improves for a threshold of between 1 to 16, though the difference is only slight. This is because the cycles of instruction fetch stall during the TP search decreases as the threshold increases, and this benefit outweighs the disadvantage of a decrease of the prefetch.

7.6 Energy Consumption

Our scheme reduces the pre-executed instruction rate, and thus reduces the energy consumed by the execution resources, such as the instruction window, D-cache, and function units. The scheme, however, requires additional hardware, which consumes extra power. Most of the extra power is consumed by the three flags of the I-cache, MCT, RIB, and TPPT. In our CACTI simulation, the three flags increase the energy consumption by only 6% over that of an I-cache access. For each of the MCT, RIB, and TPPT, Table 5 gives the size and operation rate, which specifies how many cycles out of the total number of execution cycles the respective hardware operates. As shown in the table, the size of all hardware is small, and the operation rate is also small. Therefore, we expect the consumed energy to be small.

Table 5 Size and operation rate of MCT, RIB, and TPPT.

	MCT	RIB	TPPT
size	0.8 KB	0.4 KB	0.3 KB
operation rate	6.3%	0.9%	5.4%

Table 6 lists the evaluated average energy consumed by the components related to the TSD scheme, for the base, full, and energy-efficient models. The components are categorized to the execution core, TSD overhead, and energy-efficient scheme overhead. The energy of each component is normalized by the execution core energy of the base.

As seen in Table 6, although the energy consumed by the execution core of the energy-efficient model is still 4% larger than that of the base, it is reduced by 7% compared with that of the full model. Not surprisingly, the overhead energy of our scheme is negligibly small, being only 2% of the execution core energy.

The overhead energy of the TSD is 12% of the execution core energy in the full and energy-efficient models. Reducing this overhead is required to further improve the energy efficiency. This is our work in progress. Specifically,

Table 6 Energy consumption normalized by execution core energy (EE stands for "energy-efficient model").

category	component	model		
		base	full	EE
execution core	instruction window	0.32	0.36	0.34
	LSQ	0.07	0.08	0.07
	register files	0.09	0.10	0.10
	function units	0.28	0.31	0.29
	D-cache	0.23	0.27	0.24
	D-TLB	0.00	0.01	0.01
	total	1.00	1.12	1.04
TSD overhead	DAT		0.05	0.05
	ROBP broadcast/match		0.06	0.06
	forwarding buffer		0.00	0.00
	total		0.12	0.12
EE scheme overhead	I-cache flags			0.01
	MCT			0.01
	RIB			0.00
	TPPT			0.00
	total			0.02

we are investigating a scheme that makes the TSD mechanism work only when it is necessary.

7.7 Access Time Overhead to I-Cache

Our scheme adds a three-bit flag for the TG, TP, and HD to each instruction in the I-cache. This may increase the access time for the I-cache. We evaluated the access time using our CACTI simulator, and the result show that the wordline delay of a data array increases by 2% of the cache access time. The critical path of the cache access, however, is the tag array access path, and this increase does not affect the cache access time.

8. Conclusions

In this paper, we proposed a scheme to improve the energy efficiency of the TSD previously proposed. Our scheme selectively pre-executes only those instructions that contribute significantly to performance improvement. The evaluation results using the SPECfp2000 benchmark programs show a significant reduction in the number of pre-executed instructions, leading to a reduction in energy consumption, with only a modest performance degradation, compared to the original TSD.

Acknowledgments

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] N.P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers," Proc. 17th Annual International Symposium on Computer Architecture, pp.364–373, May 1990.

[2] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," Proc. 24th Annual International Symposium on Computer Architecture, pp.252–263, June 1997.

[3] 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.

[4] 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. Eighth Workshop on Memory Performance: Dealing with Applications, Systems and Architectures, pp.41–48, Sept. 2007.

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

[6] E. Borch, E. Tune, S. Manne, and J.S. Emer, "Loose loops sink chips," Proc. Eighth Annual International Symposium on High Performance Computer Architecture, pp.299–310, Feb. 2002.

[7] K. Hyodo and H. Ando, "A low-power design of instruction pre-execution mechanism with two-step physical register deallocation," IPSJ SIG Technical Reports, pp.169–174, Aug. 2007.

[8] K. Hyodo and H. Ando, "A low-power scheme for two-step physical register deallocation through selective pre-execution," Proc. 2008 Symposium on Advanced Computing Systems and Infrastructures, pp.237–244, June 2008.

[9] K. Iwamoto and H. Ando, "Evaluation of power saving scheme for two-step physical register deallocation," IPSJ SIG Technical Reports, April 2009.

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

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

[13] 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.

[14] J.D. Collins, S. Sair, B. Calder, and D.M. Tullsen, "Pointer cache assisted prefetching," Proc. 35th Annual International Symposium on Microarchitecture, pp.62–73, Nov. 2002.

[15] 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.

[16] V. Petric and A. Roth, "Energy-effectiveness of pre-execution and energy-aware p-thread selection," Proc. 32nd Annual International Symposium on Computer Architecture, pp.322–333, June 2005.

[17] 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, Jan. 1995.

[18] 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.

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

[20] O. Mutlu, H. Kim, and Y.N. Patt, "Techniques for efficient processing in runahead execution engines," Proc. 32nd Annual International Symposium on Computer Architecture, pp.370–381, June 2005.

[21] http://www.simplescalar.com/

[22] P. Shivakumar and N.P. Jouppi, "Cacti 3.0: An integrated cache tim-

ing, power, and area model," WRL Research Report 2001/2, Aug. 2001.

[23] 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.

[24] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," Proc. 27th Annual International Symposium on Computer Architecture, pp.83–94, May 2000.

[25] I. Farkas, P. Chow, N.P. Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," Proc. 24th Annual International Symposium on Computer Architecture, pp.133–143, June 1997.

Kazunaga Hyodo was born in 1983. He received his B.E. and M.E. degrees from Nagoya University in 2006 and 2008, respectively. Since 2008, he has been engaged in development on engine control unit at Denso Corp.

Kengo Iwamoto received his B.E. degree from Nagoya University in 2008. He is currently a graduate student of the department of computational science and engineering of Nagoya University. His research interests include computer architectures.

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 a 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.