Data Prefetching and Address Pre-Calculation through Instruction Pre-Execution with Two-Step Physical Register Deallocation

Akihiro Yamamoto[†], Yusuke Tanaka[‡], Hideki Ando[‡], Toshio Shimada[†]
yamamoto.akihiro@renesas.com, tanaka@ando.cse.nagoya-u.ac.jp,
ando@cse.nagoya-u.ac.jp, shimada@nuee.nagoya-u.ac.jp

[†] Department of Electrical Engineering and Computer Science, Nagoya University
[‡] Department of Computational Science Engineering, Nagoya University

ABSTRACT

This paper proposes an instruction pre-execution scheme that reduces latency and early scheduling of loads for a high performance processor. Our scheme exploits the difference between the available amount of instruction-level parallelism with an unlimited number of physical registers and that with an actual number of physical registers. We introduce a scheme called two-step physical register deallocation. Our scheme deallocates physical registers at the renaming stage as a first step, and eliminates pipeline stalls caused by a physical register shortage. Instructions wait for the final deallocation as a second step in the instruction window. While waiting, the scheme allows pre-execution of instructions. This enables prefetching of load data and early calculation of memory effective addresses. In particular, our execution-based scheme has the strength on prefetch of data with an irregular access pattern. Considering the strength of an automatic prefetcher for a regular access pattern, combining it with our scheme offers the best use of our scheme. The evaluation results show that the combined scheme significantly improve performance over a processor with an automatic prefetcher.

1. INTRODUCTION

Load latency in cycles becomes longer as LSI technology advances because the rate of improvement for memory access time is much slower than that for processor clock frequency. The gap between processor and memory is often called a memory wall [22]. A general method to reduce 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 to solve this problem. Many hardware schemes have been proposed for prefetching [4, 6, 10, 17, 20]. The schemes currently implemented in processors generally predict an access pattern, and prefetch data according to the prediction as a trigger of a cache miss. Although these schemes can be implemented with simple hardware, they are ineffective for irregular patterns that are difficult to predict. Schemes for irregular patterns have been proposed [3, 5, 9, 11, 19] but they have disadvantages in that they require high cost or multi-threaded environments.

Even if memory latency is perfectly hidden by prefetching, loads are generally still on the critical paths. The schedule timing is constrained by true data dependences, which arise from program semantics and so are difficult to remove. Some predictive schemes are proposed which remove true dependences by predicting a load address or a load value (e.g., last value predictor [12]). A common disadvantage of these schemes is that it is difficult to achieve a sufficient improvement in performance without complicated hardware for efficient recovery from misprediction [18].

This paper proposes a scheme that provides instruction pre-execution within a single thread for data prefetching and load address pre-calculation. Our scheme assumes a split load/store, where the load/store operation is split into an address calculation and a memory access. Our scheme creates a pre-execution stream that precedes the main execution stream that builds the architectural state. In general, instruction execution is constrained by dependences and resource constraints. The precedence of the pre-execution stream is ensured by relaxing the imposed resource constraint. Our scheme focuses on physical registers as a resource constraint. Generally, a register file large enough to fully exploit instruction-level parallelism (ILP) contained in a program is not implemented, because of space, time, and power constraints. This causes pipeline stalls at the register rename stage due to the shortage of physical registers. To avoid such stalls, our scheme splits physical register deallocation into two steps at different pipeline stages. As a first step, deallocation is carried out in the rename stage, which eliminates stalls at the rename stage and so the instructions advance and are stored in the instruction window. Our scheme allows these instructions to have a *dry run* (i.e. execution without write), forming the pre-execution stream. Final deallocation is carried out as a second step at the commit stage, as in the conventional manner. This deallocation is notified to the instructions that were allowed to use this physical register in the first step. These instructions form the main execution stream that is allowed to write results.

Our pre-execution provides two benefits. First, it realizes data prefetching. Our prefetching relies on the actual execution, and not on prediction, and thus is effective for memory accesses that are difficult to predict. Second, pre-calculation of the load address removes true dependence of a load during the main execution. Our pre-calculation of the address writes the result to the load/store queue (LSQ). Thus, a load can immediately use the calculated address in the main execution, without waiting for address calculation.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 proposes our pre-execution scheme. Section 4 presents evaluation results. Finally, Section 5 concludes the paper.

2. RELATED WORK

2.1 Data Prefetching

Much effort has been done for reducing data cache misses using compile-time techniques (e.g., [2]). However, cache misses are difficult to remove completely. From the industrial viewpoint, it is important that cache misses are reduced by hardware when we execute an existing binary without recompilation. With these reasons, many studies regarding hardware schemes have been done. There are several approaches, and one of them is data prefetch.

Automatic prefetchers [6, 10], such as stride prefetchers, are effective for regular access patterns. For irregular patterns, another approach is necessary. Several schemes that hold irregular pattern information have been proposed [9, 11], but unfortunately these are very costly. Execution-based schemes have been devised as yet another approach. Our scheme belongs to this category. A number of studies have been carried out using a thread-based approach (e.g., [3, 5, 19]). The disadvantage of these schemes is that they require a multithreaded environment such as simultaneous multithreading (SMT) or chip multiprocessors. The only pre-execution scheme, to our knowledge, that does not need a multithreaded environment is *runahead execution* [16]. The disadvantage of this scheme is the large overhead of mode switching in both checkpointing and restarting. Therefore, its effectiveness is reduced for cache misses with relatively small but still important latencies like L1 cache misses. Furthermore, no computation can be overlapped with handling the triggered L2 miss, because the computed results during the runahead mode are discarded when returned to the normal mode. Later, they proposed a method that reuses the computed results of instructions independent of the L2 cache miss when returned to the normal mode, but they found such result reuse is ineffective [15].

2.2 Elimination of Address Calculation Dependence

Load address prediction is effective in eliminating the true dependence between an address calculation and a load. The effectiveness of this predictive scheme depends on the prediction accuracy and the efficiency of the recovery from mispredictions. Comprehensive performance evaluation was carried out by Reinman et al. [18]. They found that a simple recovery method using pipeline squashing significantly reduces the effectiveness, and a more elaborate method where only dependent instructions are re-executed is necessary for the performance improvement to be sufficient. However, this makes issue logic complicated, thus adversely affecting the clock cycle time.

2.3 Effectively Enlarging Register File

Several studies, aimed at effectively enlarging the register file, have been done in an attempt to reduce the occupied time of physical registers by late allocation [7, 13] or early deallocation of physical registers [1, 14, 21].

Early deallocation schemes deallocate registers speculatively by predicting a last consumer, and allocate them at the rename stage. Their shortcomings of such schemes include the large penalty imposed by mis-speculation recovery, and the requirement of a large checkpointing register file, which includes shadow storage.

Unlike the conventional scheme, late allocation schemes do not allocate registers at the rename stage. Instead, they allocate them later in the pipeline. The *virtual-physical register scheme* [7, 13] allocates registers at the write-back stage. This scheme is similar to ours in that instructions are executed even if physical registers have failed to be allocated, thus realizing pre-execution. Unfortunately, this scheme has a complication in avoiding deadlock due to out-of-order physical register allocation. Feasible, but still complicated implementation imposes a considerable cycle count penalty, and causes a significant increase in the dynamic instruction counts by ad hoc register reallocation and resulting instruction re-execution.

3. PRE-EXECUTION MICRO-ARCHITECTURE

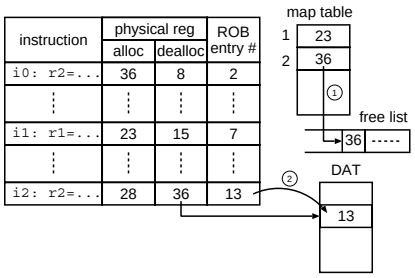
Our scheme assumes a register renaming scheme, where the register file contains committed values and temporary values for instructions that have been completed but not yet committed, and a map table translates a logical register number into a physical one. This type of register renaming is, for example, implemented in the MIPS R10000. In this section, we first propose a two-step physical register deallocation scheme as a basic scheme. We then extend the scheme to enable pre-execution.

3.1 Basic Scheme

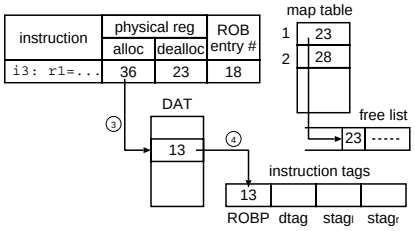
3.1.1 First-Step Deallocation

The first-step deallocation is performed in the rename stage. Besides the map table and free list, we prepare a table called the *deallocation table* (DAT). Each entry in the DAT is associated with a physical register, and holds a valid bit and the number of the reorder buffer (ROB) entry, where the instruction that finally 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. This is carried out regardless of whether or not free physical registers exist. At the same time, the number of the ROB entry, where the instruction is allocated, is written into the DAT entry associated with the deallocated physical register, and the entry is validated. 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 operation. At this time, we obtain the number of the ROB entry (ROBP), where an instruction that will *finally* deallocate the physical register is placed, by looking up the DAT, if such the instruction is still in the ROB (in this case, the DAT entry looked up is valid; otherwise, the entry is invalid, meaning such the instruction was already committed and performed the final deallocation). The ROBP (if obtained) is attached to the renaming instruction as



(a) At renaming of instruction `i2`



(b) At renaming of instruction `i3`

Figure 1: First-step deallocation.

a tag to find the timing of the second-step deallocation later in the instruction window (as described in Section 3.1.2).

Figure 1 illustrates an example of the operation 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 1a illustrates the operations when instructions `i0` and `i1` have already been renamed, and `i2` is being renamed. First, the physical register `36` that is currently allocated to a logical destination register `r2` is deallocated, and is appended to the free list (mark (1) in Figure 1a). At the same time, the allocated ROB entry number `13` is written into the 36th entry of the DAT (mark (2)).

Next, Figure 1b illustrates the operations when instruction `i3` is being renamed. Physical register `36` that was deallocated by instruction `i2` has been allocated to this instruction. After deallocation and allocation of the physical registers as described, we obtain by referring to the DAT (mark (3) in Figure 1b) the ROB entry number `13`, where instruction `i2` that will finally deallocate the allocated physical register `36` (as a second-step deallocation) is placed.

This number is attached to the renaming instruction `i3` as an ROBP tag, and 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)). Note that physical register numbers are not used as operand tags. Instead arbitrary unique numbers are used to identify the dependences of instructions in flight, because physical register numbers cannot identify different dependences due to the temporary deallocation in the rename stage.

3.1.2 Second-Step Deallocation

The renamed instruction is stored in the instruction window, and waits for the second-step deallocation of its destination physical

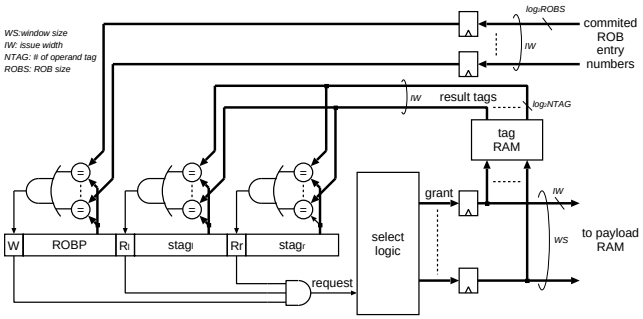


Figure 2: Instruction scheduler in our basic scheme.

register. The second-step deallocation is performed in the commit stage. The deallocated physical register at this time is one that was previously allocated to the logical register as is the case in the conventional scheme. The scheme differs, however, by broadcasting 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. Also, the DAT entry associated with the deallocated physical register is invalidated.

In the example shown in Figure 1, the second-step deallocation of a 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 the instruction `i3`, and its result write of this instruction is granted.

The logic circuit of the instruction scheduler in our basic scheme is shown in Figure 2. The three tags, ROBP, $stag_l$, and $stag_r$, are held in the wakeup logic for each instruction. The three flags, W, R_l, and R_r, associated with each tag are also held. Flags R_l and R_r are the conventional ready flags that indicate that the corresponding source operand is available. Flag W indicates that the result write of the corresponding instruction is granted. When an instruction is issued, its destination tag, which is also called a result tag, is read from the tag RAM, and is broadcast to the wakeup logic. The tag match is checked with the comparator, and the ready flag is set if matched as in the conventional scheme. On the other hand, the ROB entry number sent from the ROB is compared with each ROBP tag, and the flag W is set if matched. If flags W, R_l, and R_r are all set, an issue request is sent to the select logic. If selected, the instruction is executed and write the result as in the normal system. Note that, for instructions that do not obtain a valid ROBP in the rename stage, flag W is initially set when they are written into the instruction window. Since their destination physical register is already finally deallocated, issue control by flag W is not necessary. Also, for instructions that do not have destination registers such as the address calculation instructions of memory instructions, they are controlled in the same manner, because these instructions do not require register writes.

3.2 Extension to Instruction Pre-Execution

In the previous section, we described that instructions waiting in the instruction window are not allowed to be issued until their writes are granted. However, it is possible for such instructions to be executed; though the execution result is not written, it can be passed to dependent instructions via the bypass logic if the instructions are issued continuously. These instructions form a pre-execution stream, which proceeds faster than the main execution stream because it

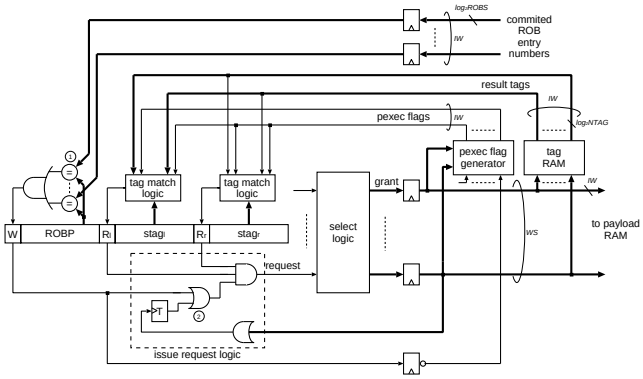


Figure 3: Instruction scheduler for pre-execution.

exploits ILP, where no resource constraint regarding physical registers exists.

Figure 3 shows the logic circuit diagram of the instruction scheduler that implements our pre-execution scheme. The main differences between this extended scheduler logic and the logic of the basic scheme shown in Figure 2 are as follows: (1) A T-FF is placed in the issue request logic to allow pre-execution only once before a write is granted. (2) A flag we call *pexec* is attached to the each result tag of the pre-executed instructions. This flag is used to reset the ready flag that was set in the previous cycle.

Function. When an instruction is written into the instruction window, the T-FF in the corresponding entry is set. When the execution of an instruction finishes, its result tag is broadcast. The ready flag in the entry where the tag is matched is set. When both ready flags, R_l and R_r, are set, the issue request is sent to the select logic through the issue request logic, even if flag W is 0, because the T-FF holds 1. If the request is granted, the selected instruction is issued and pre-executed. The grant signal toggles the T-FF to 0, and thus an issue request will not be produced until flag W is set. Also, *pexec* flags are generated, and will reset the ready flags in the next cycle

Note that pre-executed instructions are not removed from the instruction window. Later, when the flag W is set by ROBP tag match, and the both ready flags are set again, the instruction is re-executed and writes the result into the destination physical register, as described in 3.1.2.

3.3 Potential Problems

3.3.1 Delay

Since the instruction scheduler is one of critical circuits that determine the clock cycle time of a processor, we need to check whether the circuits added to the instruction scheduler affect the delay or not. The critical path of the original scheduler consists of the grant drive, the tag RAM access, the tag drive, the tag match, ORing the match results, setting the ready flag, ANDing the ready flags, and the select. Because of the space limitation, all potential problem paths are not explained. The path that is most concerning is the one related to flag W. This path consists of the drive of the ROB entry numbers, the match with ROBP (mark (1) in Figure 3), ORing the match results, setting flag W, ORing flag W with the T-FF (mark (2)), ANDing the ready flags, and the select. Compared with the original critical path, its delay is shorter with respect to the tag RAM access time, but is longer with respect to the time for ORing with the T-FF. Apparently, the time for ORing with the T-FF is much shorter than the tag RAM access time. Therefore, the delay for the path related to flag W is shorter than that for the original critical path.

3.3.2 Precise Exception

When an exception is committed, the processor state must be rolled back to the committed point. In the normal register renaming scheme, this roll-back is performed by rolling back the rename map table from the latest map to the map when the excepted instruction is renamed. This roll-back is carried out by reading the ROB from the tail (newest instruction) to the top (oldest instruction). Each ROB entry has the number of a physical register that was previously allocated to the same logical destination register of the instruction the entry holds. The roll-back mechanism writes the previously allocated physical register number into the map table entry associated with the logical destination register. In our two-step deallocation scheme, the roll-back of the map table is performed with the exactly same manner. Note that although the map table is speculatively updated, the previously allocated register number each ROB entry holds is not changed, and therefore the roll-back is performed correctly.

The DAT is not necessary to be rolled back with the following reasons. First, it does not hold the architectural state. Second, there are no speculatively deallocated physical registers after rolling back to the committed exception point, and thus an instruction always reads a DAT entry after a preceding instruction writes an ROB entry number into the entry.

Although we described the scheme for handling exception, any type of mis-speculation recovery that is related to register renaming is handled with the same manner.

3.4 Combination with Automatic Prefetcher

Automatic prefetchers, like stride prefetchers, have a big advantage in that they can automatically and promptly prefetch much data in a long stream with a regular pattern. On the other hand, although our pre-execution scheme can prefetch data with both regular and irregular patterns, the prefetching rate is bounded by the fetch rate of the load instructions. This restriction is particularly disadvantageous in a long stream with a regular pattern.

Considering both the strengths and weaknesses, combining the two prefetching schemes compensates for the weaknesses of each scheme; the automatic prefetcher covers the miss stream with a regular pattern, and our pre-execution prefetching covers the miss stream with an irregular pattern. This combination also provides additional benefits. First, the automatic prefetcher helps advance the pre-execution faster, because part of the cache misses during the pre-execution can be avoided with the automatic prefetcher. This improves the timeliness of prefetching with an irregular pattern. Second, our pre-execution triggers the automatic prefetcher earlier, improving the timeliness of prefetching with a regular pattern.

4. EVALUATION

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 use nine memory intensive programs from SPECfp95 and SPECfp2000, which are the highest nine programs of the L1 data cache miss rate in the programs of the two suites we could compile for SimpleScalar.

Table 1 lists the benchmark programs and their L1 data and L2 unified cache miss rates. The programs are compiled by gcc ver.2.7.2.3 with options -O6 -funroll-loops. The configuration of the baseline processor for our evaluation is summarised in Table 2. Note that we assume perfect memory disambiguation for fair evaluation. Our pre-calculation of the load address has the positive effect on memory disambiguation. For fair evaluation, instead of implementing memory dependence predictors in our base simulator, we assume perfect memory disambiguation to exclude the effect on memory disambiguation.

Table 1: Cache miss rate.

suite	program	L1 miss rate	L2 miss rate
SPECfp95	hyrdro2d	8%	31%
	su2cor	6%	0%
	wave5	4%	7%
SPECfp2000	ammp	9%	32%
	applu	5%	49%
	art	48%	44%
	equake	4%	41%
	mgrid	3%	30%
	swim	13%	37%

Table 2: Baseline processor configuration.

Pipeline width	4-instruction wide for each of fetch, decode, issue, and commit
ROB	64 entries
LSQ	64 entries
Instruction window	64 entries
Function unit	4 iALU, 2 iMULT/DIV, 2 Ld/St, 4 fpALU, 2 fpMULT/DIV/SQRT
L1 I-cache	64KB, 2-way, 32B line
L1 D-cache	64KB, 2-way, 32B line, 2 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
Physical register	total 128 (64 for each of int and fp)
Mem. disambiguation	perfect

4.1 Effect of Data Prefetching

We evaluate average load latency of the following three models. The first model is the *stride prefetcher model*, which is the baseline processor with a stride prefetcher using a per-load stride predictor [6]. It has a stream buffer [10] between the L1 data cache and the L2 unified cache to avoid pollution of the L1 data cache. The stream buffer is accessed in parallel with the L1 data cache, and the requested data is assumed to be obtained with single cycle latency, if it is found. The configuration of the stream buffer is determined based on the hardware prefetching scheme of an Intel Pentium 4 [8]. The buffer has eight ways, the capacity is 4KB each, and the line size is 32B. The buffer is allocated on an L1 data cache miss. To suppress prefetching of useless data, the incremental prefetching scheme [6] proposed by Farkas et al. is introduced.

The second model is the *pre-execution model*, which models the baseline processor with our pre-execution scheme. To boost performance, it prefetches a single succeeding line, in addition to the requested line, to the L1 data cache, if not explicitly explained. A stream buffer is not placed.

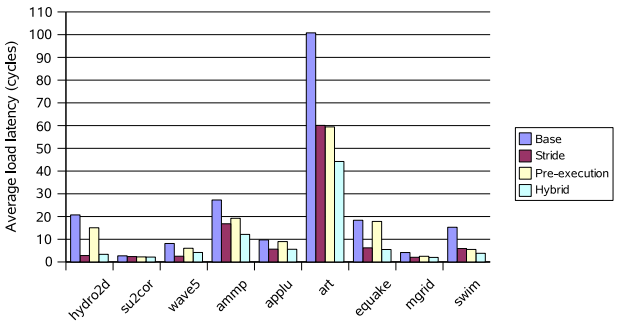


Figure 4: Average load latency.

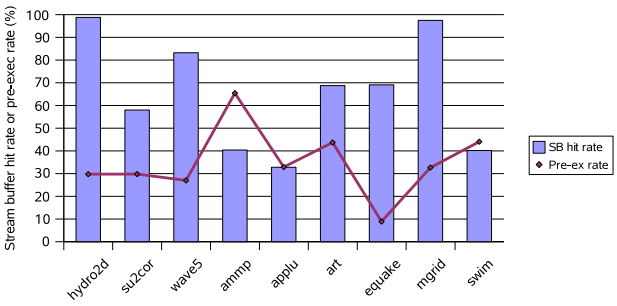


Figure 5: Pre-execution rate and stream buffer hit rate.

The third model is the *hybrid model*, which combines pre-execution and the stride prefetcher model.

Figure 4 shows the average load latency for each model. As shown in the figure, the pre-execution model reduces the load latency significantly below that of the base model in many programs. The effectiveness is comparable in *su2cor*, *ammp*, *art*, *mgrid* and *swim*, compared with the stride prefetcher model. However, in other four programs, the stride prefetcher model is better than the pre-execution model. These results are roughly correlated to how many load instructions are pre-executed in the pre-execution model and the hit rate of the stream buffer in the stride prefetcher model. Figure 5 shows the percentage of pre-executed load instructions to the total number of committed load instructions (*pre-execution rate*) in the pre-execution model, and the hit rate of the stream buffer in the stride prefetcher model. For example, in *art*, the pre-execution rate is high (44%), but the stream buffer hit rate is not very high (69%). Thus, pre-execution is comparable with stride prefetching. However, in *hydro2d*, the pre-execution rate is not high (30%), but the stream buffer hit rate is very high (99%). Thus, stride prefetching is more effective than pre-execution.

As expected, in the hybrid predictor model, the two models compensate for each other, and thus this exhibits the lowest load latency in most programs. In particular, in *ammp* and *art*, the hybrid model significantly reduces the load latency when compared to both the stride prefetcher model and the pre-execution model.

4.2 Effect of Address Pre-Calculations

We evaluate the effect of address pre-calculations. To exclude the prefetch effect, we assume that the L1 data cache is perfect in our evaluation of this section. Figure 6 shows the speedup of the pre-execution model over the baseline model, with the perfect L1 data cache for both models. As shown in Figure 6, our address pre-

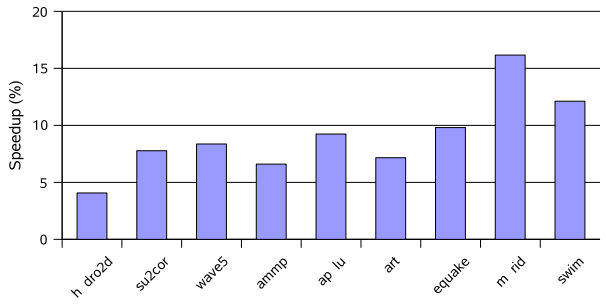


Figure 6: Effect of address pre-calculations.

calculation greatly improves the performance of every program. The speedup is 9% on geometric mean.

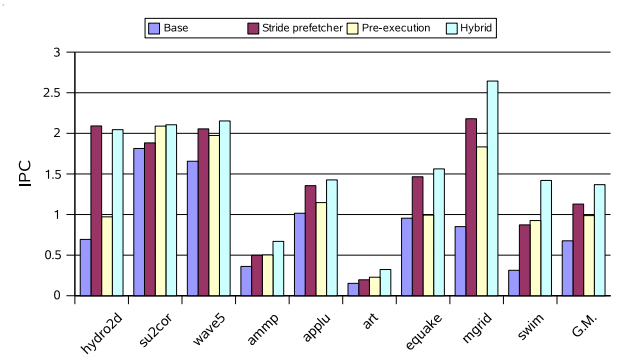
4.3 Overall Performance

We evaluate the overall performance of our scheme. Figure 7a compares the IPC of the four models in the case where the next-line prefetch (NLP) is performed on pre-execution, and Figure 7b without the NLP on pre-execution. As shown in these figures, our pre-execution model achieves a great speedup of 46% or 17% with or without the NLP, respectively, on average over the base model. Note that our scheme achieves this great speedup with a modest hardware cost and with little adverse impact on the clock cycle time. Although the performance of the pre-execution alone is 13% or 30% worse with or without the NLP than that of the stride prefetcher model on average, the hybrid model exhibits considerably better performance. The speedup over the stride prefetcher model is 21% or 15%, with or without the NLP, respectively.

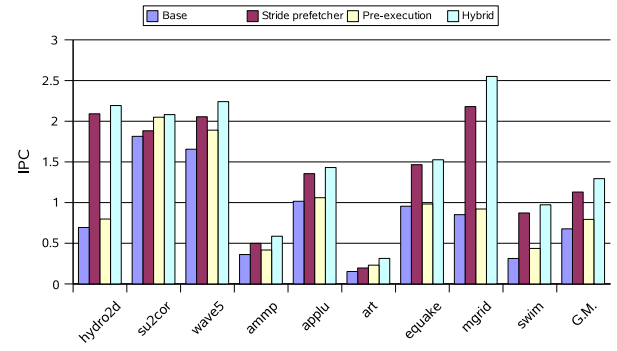
4.4 Performance Sensitivity to the Number of Registers

The effectiveness of our pre-execution is affected by the number of physical registers available. Figure 8 shows the IPCs (geometric mean) of the four models for different numbers of total physical registers for integers and floating points per thread (the number of integer and floating-point physical registers is equal). According to the number of the physical registers, we set the ROB size to be balanced at each evaluated point, so that the processor can handle the proper number of in-flight instructions. We determine the ROB size with the following equation: the ROB size = the number of total physical registers – the number of total logical registers. Other parameters are not changed. We also plot the total number of physical registers per thread for several commercial processors on the X-axis for references (note that the number of the available registers per thread is a half of the total number of physical registers in the Intel Pentium 4 and the IBM Power5, because these processors employ two-way SMT). In the case of processors where the rename register file is separated from the architectural register file, the total number of physical registers is the sum of the architectural and the rename registers.

As anticipated, the effectiveness of our pre-execution diminishes as the available number of physical registers increases (the speedup of the pre-execution model over the base is 42% for 68 physical registers, and 35% for 256 physical registers; the pre-execution rate decreases from 52% to 22%. However, the effectiveness is still found to be substantial even in the extreme case in our evaluation (256 registers). In the range of registers from 96 to 192, which covers all of the commercial processors we plotted, the hybrid model significantly outweighs the stride prefetcher model by between 18% and 30%.



(a) With next-line prefetch



(b) Without next-line prefetch

Figure 7: Overall performance.

4.5 Resource Size Sensitivity

Figure 9 shows the speedups of the pre-execution model for different sizes of instruction window, LSQ, and ROB over the base model. Each speedup is calculated independently for each different sizes. There are three bars for each instruction window size: the left, middle, and right bars represent the speedup when the ROB size is respectively 1.0x, 1.5x, and 2.0x larger than the instruction window size. The LSQ size is equal to the instruction window size. In general, the speedup is increased as the resource constraints are relaxed, simply because more the resources exist, more instructions are pre-executed. A small ROB with a small instruction window (32-entry ROB with 32-entry instruction window) severely deteriorates the effectiveness of our scheme. Instructions are stalled with these resources before stalled with a shortage of physical registers (64 physical registers). Also, at the 64-entry instruction window (our default), we find that more ROB entries over the default value (64 entries) are beneficial. As a summary, with a reasonable size of the resources (e.g., 64-entry instruction window and 64 or 96-entry ROB), the effectiveness is sufficiently large.

4.6 Impact of No Storage in Pre-Execution

As described in Section 3.2, pre-executed instructions pass their results to succeeding instructions only via the bypass logic. This is

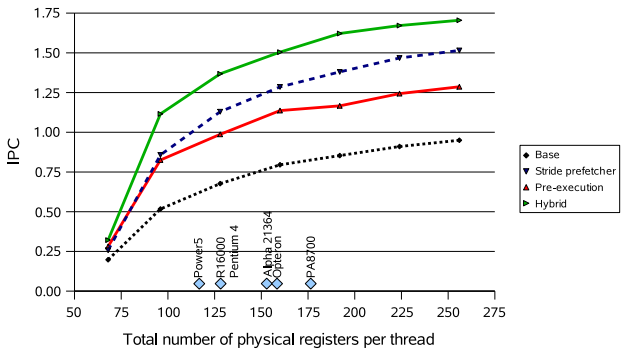


Figure 8: IPC vs. total numbers of physical registers per thread.

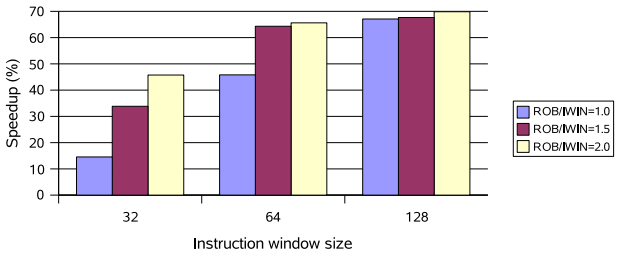


Figure 9: Effect of instruction window and reorder buffer size.

because pre-executed instructions have no storage for results. To quantify this negative impact, we evaluated models where this disadvantage is excluded. Specifically, we introduce an assumption that the results of pre-executed instructions are available for succeeding instructions that are issued in any timing. We call this assumption an *ideal bypass*. Figure 10 compares IPCs with the real bypass to that with the ideal bypass in the pre-execution and hybrid models. As shown, the negative impact of no storage for pre-execution is small.

4.7 Contribution of Next-Line Prefetch

As describe in Section 4.1, in the evaluation so far (except for Figure 7b), our pre-execution scheme used next-line prefetch (NLP). To evaluate this effect, we introduce two options regarding the NLP to the four models. (1) *Full NLP* always performs the NLP on both main execution and pre-execution of loads. (2) *NLP on pre-exec* performs the NLP only on pre-execution of loads. Figure 11 shows the evaluated results. As shown in the figure, the NLP is effective in the models without the stride prefetcher (base and the pre-execution models). In contrast, the NLP is not so effective for the models

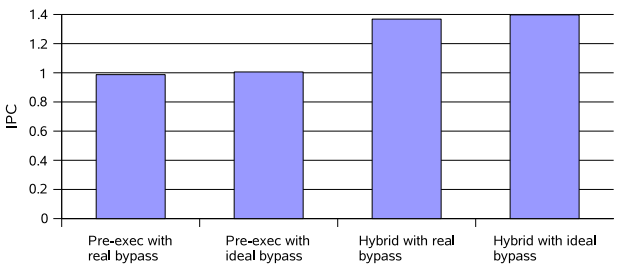


Figure 10: Performance impact of no storage in pre-execution.

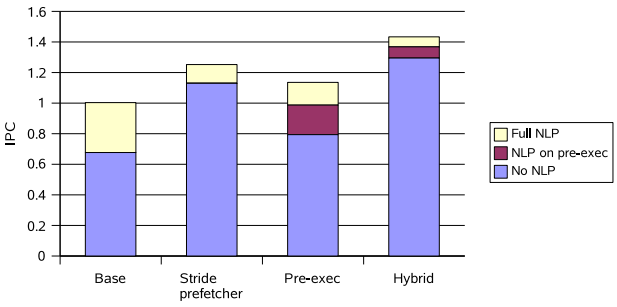


Figure 11: Effect of next-line prefetch.

with the stride prefetcher. This is because the stride prefetcher often detects a constant stride in the case where the NLP is effective. Considering the complexity of the NLP (i.e. twice access to the D-cache) along with small contribution, the exclusion of the NLP may be a better design decision, when using our pre-execution scheme with a stride prefetcher, which we believe that offers the best use of our scheme.

4.8 Contribution of Late Physical Register Allocation

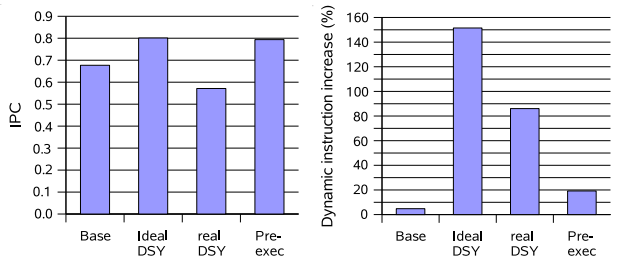
In our scheme, a physical register is temporally allocated at the rename stage, but the write to it is granted when the instruction resides in the instruction window. This is virtually a late allocation of physical registers. The late allocation reduces the register life time, and thus the available number of physical registers increases. How large this effect is depends on how long the scheme can reduce the register life time. In the case of our scheme, the reduced time is very short; only the rename stage. Therefore, the effectiveness is anticipated to be very small. To quantify the effectiveness, we evaluate a processor with our basic scheme described in Section 3.1 (no pre-execution extension). As a result, the speedup of this processor over the base is found to be only 5% on average.

4.9 Performance in SPECint2000 Programs

We evaluate performance using eight programs we select from SPECint2000 benchmark, which are those we could compile for SimpleScalar. The programs are listed in Table 3 with their cache miss rates. Unfortunately, the average speedup of the pre-execution model over the base is only 1%, unlike in the floating-point programs. The reasons of this small speedup are as follows. First, most SPECint2000 programs are not memory intensive. Second, the available amount of ILP contained in a program is small, and thus the number of physical registers we assumed in this evaluation is enough.

Table 3: Cache miss rate of SPECint2000 benchmark.

program	L1 miss rate	L2 miss rate
bzip2	3%	9%
gcc	11%	0%
gzip	3%	3%
mcf	23%	46%
parser	4%	1%
perlbmk	1%	0%
vortex	0%	6%
vpr	2%	24%



(a) IPC

(b) Dynamic instruction count increase

Figure 12: Comparison with virtual-physical register scheme.

4.10 Comparison with Virtual-Physical Register Scheme

As described in Section 2.3, the virtual-physical (VP) scheme has the ability to prefetch data like ours. Here, we compare our scheme with the VP scheme, which employs DSY (on-demand with steeling from younger) [13] to avoid deadlock. We introduce two models for the VP scheme with different cycles consumed for register reallocation in DSY. One is an *ideal DSY model*, which is ideal in that no cycle is consumed for register reallocation. Another is a *real DSY model*, which limits the ROB read bandwidth to the commit width (eight in our assumption) for searching a victim to reallocate a register and stalls the execution and commit stages of the pipeline while searching (the memory system is not stalled). In this evaluation, we disable the NLP in our scheme for a fair comparison.

Figure 12a shows the IPCs. As shown in this figure, the IPC of our scheme is comparable with the ideal DSY. However, taking into account the register reallocation cost, the real DSY model significantly degrades the performance. In addition, the VP scheme considerably increases dynamic instruction, as shown in Figure 12b (the vertical axis indicates the percent increase of the dynamic instruction count over the committed instruction count). This increases power consumption.

5. CONCLUSIONS

In this paper, we have proposed a scheme that prefetches data and address pre-calculation of loads. Our scheme allows instructions to be pre-executed in a single context by exploiting the difference between the available amount of ILP without resource constraints of the physical registers and that of ILP with those constraints. Execution-based data prefetching enables prefetching of data with an irregular access pattern. Our scheme is implemented by a simple table that maintains early register deallocation and a modestly modified instruction scheduler. Our evaluation results show that our scheme significantly improves performance over a processor without a prefetcher. Considering the strength of an automatic prefetcher for a regular access pattern, we believe that combining it with our scheme offers the best use of our scheme. We confirmed that the combined scheme considerably improves the performance over a processor incorporating only a stride prefetcher.

6. ACKNOWLEDGMENTS

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).

7. REFERENCES

[1] D. Balkan, J. Sharkey, F. Ponomarev, and A. Aggarwal. Address-value decoupling for early register deallocation. *ICPP-35*, pages 337–346, August 2006.
[2] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. *ASPLOS-8*, pages 139–149, October 1998.
[3] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading(SSMT). *ISCA-26*, pages 186–195, May 1999.
[4] T. F. Chen and J. L. Baer. Reducing memory latancy via non-blocking and prefetching caches. *ASPLOS-5*, pages 51–61, October 1992.
[5] 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. *ISCA-28*, pages 14–25, July 2001.
[6] I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. *ISCA-24*, pages 133–143, June 1997.
[7] A. González, J. González, and M. Valero. Virtual-physical registers. *HPCA-4*, pages 175–184, February 1998.
[8] Intel Corporation. *Intel Pentium 4 Processor Optimization Reference Manual*, 1999.
[9] D. Joseph and D. Grunwald. Prefetching using Markov predictors. *ISCA-24*, pages 252–263, June 1997.
[10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. *ISCA-17*, pages 364–373, May 1990.
[11] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. *ISCA-28*, pages 144–154, July 2001.
[12] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. *ASPLOS-7*, pages 138–147, October 1996.
[13] T. Monreal, A. González, M. Valero, J. González, and V. Viñals. Delaying physical register allocation through virtual-physical registers. *MICRO-32*, pages 186–192, November 1999.
[14] M. Moudgill, K. Pinagli, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. *MICRO-26*, pages 202–213, December 1993.
[15] 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. *MICRO-38*, pages 223–244, November 2005.
[16] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An effective alternative to large instruction windows. *HPCA-9*, pages 129–140, Feburary 2003.
[17] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. *ISCA-21*, pages 24–33, April 1994.
[18] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. *MICRO-31*, pages 127–137, December 1998.
[19] A. Roth and G. S. Sohi. Speculative data-driven multithreading. *HPCA-7*, pages 37–48, January 2001.
[20] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. *MICRO-33*, pages 42–53, June 2000.
[21] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. *ASPLOS-11*, pages 107–119, October 2004.
[22] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, volume 23(1), pages 20–24, March 1995.