Journal of Instruction-Level Parallelism 7 (2005) 1-29 Submitted 10/2004; published 04/2005

Precise Instruction Scheduling

Gokhan Memik MEMIK@ECE.NORTHWESTERN.EDU

Department of Electrical and Computer Engineering
Northwestern University
2145 Sheridan Road
Evanston, IL 60201 USA

Glenn Reinman REINMAN@CS.UCLA.EDU

Department of Computer Science
University of California, Los Angeles
Boelter Hall
Los Angeles, CA 90095 USA

William H. Mangione-Smith BILLMS@EE.UCLA.EDU

Department of Electrical Engineering
University of California, Los Angeles
Engineering IV Building
Los Angeles, CA 90095 USA

Abstract

Pipeline depths in high performance dynamically scheduled microprocessors are increasing steadily. In addition, level 1 caches are shrinking to meet latency constraints - but more levels of cache are being added to mitigate this performance impact. Moreover, the growing schedule-to-execute-window of deeply pipelined processors has required the use of speculative scheduling techniques. When these effects are combined, we are faced with performance degradation and increased power consumption due to load misscheduling, particularly when considering instructions dependent on in-flight loads.

In this paper, we propose a scheduler for such processors. Instead of non-selectively speculating, the scheduler predicts the execution delay of an instruction and issues them accordingly. This, in return, can eliminate the issuing of some operations that will otherwise be squashed. Clearly, load operations constitute an important obstacle in predicting the latency of instructions, because their latencies are not known until the cache access stage, which happens later in the pipeline. Our proposed techniques can estimate the arrival of cache blocks in various locations of the cache hierarchy, thereby enabling more precise scheduling of instructions dependent on these loads. Our scheduler makes use of two structures: A Previously-Accessed Table that stores the source addresses of in-flight load operations and a Cache Miss Detection Engine that detects the location of the block to be accessed in the memory hierarchy.

Using the SPEC 2000 CPU suite, we show that the number of instructions issued can be reduced by as much as 52.5% (16.9% on average) while increasing the performance by as much as 42.1% (14.3% on average) over the performance of an aggressive processor.

1. Introduction

In the last decade, the most important factor in increasing the performance of high-end microprocessors has been arguably the increase in the clock rates. Increased clock rates are achieved mostly by technology scaling and increasing the pipeline depth. Using technology scaling, designers are able to place more transistors into the chips and achieve lower circuit delays. In addition they use deeper pipelines to increase the clock frequencies of the processors. Although building deeper pipelines has several positive effects on the microprocessor, it also has drawbacks [6]. As the number of pipeline stages is increased, the penalties associated with misscheduled instructions also increase: it takes longer to detect instructions depending on a mispredicted branch or on a memory reference that misses in the cache(s).

In this paper, we propose a precise instruction scheduler that predicts when an instruction can start its execution and schedules the instructions intelligently such that they will arrive at the corresponding position in the pipeline at the exact cycle when the data will be available to them. Usually, this information is readily available if the data to the instruction is provided by a constant-delay operation. For example, if an add operation takes its only input value from a sub instruction and the latency of subtractions are 2 cycles, at the issuing stage we can precisely determine the time when input values will be available. However, operations with varying latency (e.g. load operations) constitute a challenge. The latency of such operations is not known until later in the pipeline. However, we will show that it is possible to predict the latency of such operations effectively. First, we will show that the effects of speculative scheduling are likely to be severe in the future microprocessors and argue that intelligent scheduling mechanisms such as our proposed scheme will be an integral part of high-performance microprocessors in the future. Specifically, our contributions in this paper are:

- we measure the performance implications of instruction misscheduling under different recovery mechanisms,
- we show that the access latencies of load operations are likely to be affected by previously issued load operations indicating a significant load-to-load correlation,
- we present a novel scheduling algorithm that determines cache access latencies by considering in-flight data and for improved delay of load-dependent operations,
- we present two novel cache miss detection techniques that are embedded into our scheduling scheme, and
- we show by simulation that significant performance improvement and power reduction can be achieved with the proposed scheduling scheme.

In Section 6.1, we will show that the load misses have an important impact on the performance of a representative microprocessor architecture. These effects amplify if the cache miss rates increase. Often, the limitation on the on-chip cache sizes has been the die size constraints. Technology scaling allowed designers to utilize larger level 1 data cache sizes. However, as the clock frequencies increase, the cache latencies are becoming the dominating design factor [6]. For example, the level 1 data cache in Pentium 4 is half the size of the level 1 data cache in Pentium III. As the clock frequencies continue to increase further, we believe that the designers will be forced to use even smaller caches and/or caches with lower associativity, compensating for the high miss ratios with more on-chip cache levels or larger high level caches. In addition, it is likely that the importance of cache misses will further increase with the increase in the data footprint of applications.

In current microprocessors the scheduler assumes that the access will hit in the cache and then schedules the dependent operations according to this latency. If the load misses, then the dependent instructions are re-executed. Section 2 will discuss existing re-execution (or replay) mechanisms that are used in current microprocessors. However, as we will show, regardless of the replay mechanism, such instructions will cause bottlenecks. First, in their initial execution they will waste issue bandwidth and other resources. Second, as we will show in Section 2, their execution time is likely to increase. Third, since these instructions are executed twice, they are guaranteed to increase the energy consumption. Our scheduler halts the execution of these instructions until their data is ready and therefore addresses all three bottlenecks.

Power consumption is an important issue in the modern processors. As the semiconductor feature size is reduced, more logic units can be packed into a given area. This increased density reduces the delay of logic units, but increases the overall power consumption of the processor. Therefore, with each new generation of processors, the total power consumption as well as the power per unit area tends to increase, putting the overall value of the processor in jeopardy. In our precise scheduling scheme, the instructions, which otherwise would be squashed, are not issued. Therefore, the power consumption of several key processing elements on the critical execution path, e.g. register file and function units, is reduced. This power reduction will not cause any performance degradation since the execution of these dependent operations would be cancelled anyway once the cache miss was detected. On the contrary, performance can be improved by using precise scheduling. Section 2 will explain this phenomenon in detail. In Section 2, we also explain the details of scheduling and the methods for re-executing the dependent instructions. Section 3 overviews precise scheduling and discusses the address prediction techniques used in our experiments. Section 4 and 5 present different components of our scheduler. Section 6 presents the experimental results. In Section 7, we discuss the related work and Section 8 concludes the paper with a summary of our work.

2. Problem Statement

Modern general-purpose processors are aggressive and optimistic. They aim to execute multiple instructions per cycle, use aggressive branch prediction techniques, execute instructions speculatively, etc. Similar aggressiveness is also observed in execution of load operations and instructions that depend on these load operations. In most processors, these instructions are issued with the assumption that the load will hit in the cache. This assumption usually increases the performance as most of the loads actually hit in the cache. However, as we will show in the following discussion, the performance can be adversely affected by this aggressiveness as the number of pipeline stages is increased. In Section 6, we will present simulation numbers showing that an intelligent scheduling algorithm that is aware of load access latencies can improve the performance of this aggressive scheduling technique by as much as 42.1% and on average 14.3% for SPEC 2000 [19] applications. We will also show that up to 52.5% of the instructions issued should be re-executed indicating the power implications of these scheduling decisions.

Figure 1 illustrates a portion of the Pentium 4 pipeline and shows the schedule-to-execute pipeline latency (*STE pipeline latency*). STE pipeline latency corresponds to the number of pipeline stages between the stage where a scheduling decision is made for the instruction and the stage where the instruction executes. The size of this window has a significant effect on the performance related to load misses. Assume that a load operation is scheduled to execute. If the STE pipeline latency is n, and the processor is an m-way machine, at most

```
m x (n + time_to_identify_miss))
```

instructions may be scheduled before the scheduler knows whether the load will hit or miss in the cache. It is very likely that some instructions that are dependent on the load have to be scheduled to fully utilize the available issue bandwidth. As the STE pipeline latency increases, more dependent instructions will be scheduled before a miss is identified. If the load instruction misses in the cache, these dependent instructions will be re-executed. This re-execution is referred to as *replay*. There are two popular replay mechanisms in the literature: flush replay (used in the integer pipeline of Alpha [9]) and selective replay (used by Pentium 4 [7]). In flush replay, all instructions in the STE window are flushed and re-executed whether they are related to the load operation or not. In selective replay, the processor only re-executes the instructions that depend on the missed load [7]. In this paper we focus on only selective replay. As the STE pipeline latency is increased, the number of recoveries performed increases. Therefore, flush replay is less desirable for deeply-pipelined processors and may reduce the benefit seen from out-of-order speculation.

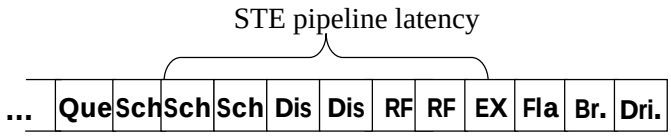

Figure 1: A portion of Pentium 4 pipeline.

There are several possible selective replay mechanisms. We refer to our selective replay mechanism as instruction-based selective replay. In instruction-based replay, an instruction starts its own re-execution. In this mechanism, each issued instruction carries a re-execute bit. When an instruction cannot receive a data value that it is supposed to receive from an earlier instruction, the re-execute bit is set. Before starting the execution of the instruction, the processor checks whether any instruction has this bit set. If so, the scheduler is informed so that the instruction can be re-executed. If the instruction receives its input variables and starts execution, the scheduler is informed such that the instruction can be removed from the issue queue. The advantage of the instruction-based selective replay is its simplicity. The scheduler does not have to keep track of load dependency chain. In addition, a single buffer can be used for all the instructions that are scheduled to execute. The only information required to re-execute the instruction is the index of the entry in which the instruction resides.

The replay mechanisms may result in sub-optimal performance. For the flush replay mechanism, a major portion of the pipeline is flushed after any cache miss. Naturally, if the access latency of the load instruction is known a priori, the processor can halt the execution of the dependent instructions and hence the independent instructions can complete their execution under cache misses. The selective replay can also perform sub-optimally. First, instructions that will be discarded after the cache miss fill extra issue slots. With the access delay information, these slots can be used for independent instructions and hence execution time can be reduced. Second, consider the dependency chain in Figure 2. Assume that the level 1 data cache has two cycles of delay and add/sub operations complete in a single cycle. A traditional scheduler will issue the load instruction, wait for two cycles, issue the add operation, wait for another cycle and issue the sub operation such that in case of a cache hit the latency is minimized. Consider a scenario where

the access misses in the cache. However, the data was requested before and the access completes in four cycles[1]. Since the add operation is scheduled two cycles after the load operation, it will not be able to receive its data and will have to be re-executed. Similarly, the sub operation will be re-executed as well. Regardless of the replay mechanism, the add operation will need to be issued again and go through STE pipeline latency stages, and in this case increase its total execution time. If, on the other hand, the scheduler can predict that access will be completed in four cycles instead of two, it can delay the issuing of the add operation for two more cycles (a total of four cycles) and the sub instruction for five cycles. Assuming that the load operation is issued at cycle 0, with this scheduling, the add operation will be completed at cycle

```
STE_window + 4 cycles
```

instead of at cycle

```
2 * STE_window + 2 cycles
```

that would have been achieved with the traditional scheduling.

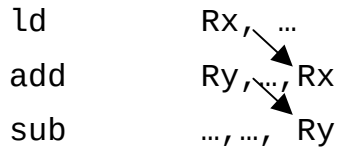

Figure 2: Sample code showing dependency chain

Another important property of the scheduler regarding the execution of load instructions is determining the priorities for the operations to execute. One can imagine a scheduler that gives low priority to those instructions that depend on a load and tries to issue independent operations. However, even with such a mechanism it is unlikely to fill the issue slots. In addition, such a mechanism can slow down the critical instructions. The scheduler used in our experiments gives priority to the oldest instruction in the issue window.

In this section, we have argued that knowledge of load access latency can increase the efficiency of the scheduler significantly. In the next sections we explain how the scheduler can gather this latency information before issuing the corresponding instructions.

3. Overview of Precise Scheduling

The goal of the scheduler is to be able to determine the access latency for load operations and schedule the dependent instructions accordingly. This is achieved with the help of two structures: Previously-Accessed Table (PAT) and Cache Miss Detection Engine (CMDE). Once the exact

[1] This in-flight data might be generated by a previous load instruction that accessed the same address, or can be caused by prefetching. In addition, if the processor uses victim or stream caches [8], the access times may vary between level 1 hit latency and level 2 hit latency (assuming that the access will hit in level 2 cache) for level 1 misses even without in-flight data.

latency of a load operation is found, the arrival time of the source data is stored in the RUU and passed along to the instructions that depend on the load. Since microprocessors have to determine the data dependency between instructions, the only additional requirement incurred is the time field in the RUU.

The PAT determines whether the latency of a load operation will be affected by an "in-flight" data. For example, if the source block of a load is requested by a previous load operation, even if the second load operation misses, it will not see the full L2 latency (or memory latency if L2 also misses). Such load operations, whose latency is affected by other load operations are called to be aliased. Note that, our scheduler needs exact timing information about the latencies. So, information about such aliased loads is crucial. In addition, we will show in Section 5 that in fact a significant fraction of loads is aliased. The PAT is a small cache–like structure addressed by the predicted addresses of the load instructions. When an access latency is predicted for a load source address, the latency is stored in the PAT.

If a load is not aliased (i.e. it accesses a block that is not in-flight), the latency of it is determined using miss detection at each cache level. For example, if the load operation is detected to miss at level 1, and hit in level 2, its latency will be equal to level 2 cache hit latency. This information could be achieved by probing the cache tags. However, the delay for cache tags would be intolerable at the scheduling stage. Instead, we developed two novel cache miss detection techniques. In addition, we use two structures from literature: an MNM machine [11] and a history-based miss prediction scheme [9]. The motivation behind the CMDE techniques is to use small hardware structures to quickly determine whether an access will miss in the cache.

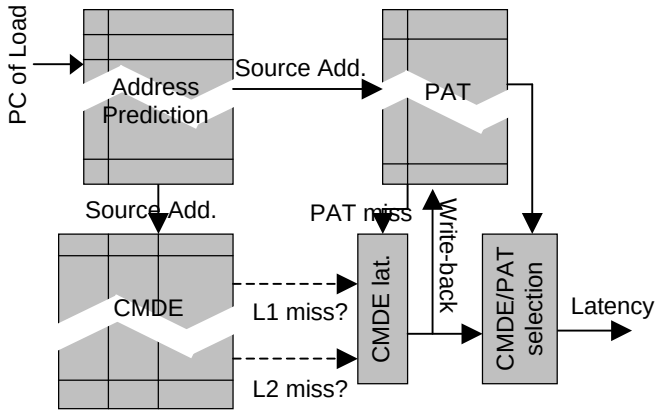

Figure 3: Scheduling scheme.

The load addresses, which are required by the CMDE structures, are not available during the scheduling stage. Therefore, we use address prediction to generate the addresses. In our simulations, we experiment with a 2-delta stride-based predictor [4]. A natural extension of our work is to use the generated information about the cache hits and misses to perform prefetching. Once the scheduler determines that the access will miss in the cache, it can start fetching the data for the access, instead of waiting until the load instruction reaches the execute stage. In Section 6, we investigate the effects of such a prefetching mechanism on our scheduling algorithm.

The overall algorithm used by the scheduler to estimate the access latency is as follows:

1. Source address for the load is predicted,

2. The PAT is probed for the address,
    a. If the address exists in the PAT, the value at the corresponding entry is used for the estimated access latency.
3. If the address does not exist in the PAT, the CMDE structures are accessed to estimate the access latency,
4. If the CMDE structures determine a level 1 cache miss, the estimated latency is placed to PAT (so if a later load is aliased it can use this value).

This overall scheme is depicted in Figure 3 for a processor with 2 data cache levels. The output of the CMDE is hit/miss predictions for level 1 and 2 caches, the output of the PAT is the latency if the address is found in PAT, PAT miss indication, otherwise. PAT miss indication activates the CMDE latency calculation, which sets the latency to one of level 1, level 2 hit latency or memory access latency according to the CMDE output.

4. CMDE Techniques

The CMDE structures work by storing information about each on-chip data cache. If an address is not found in the PAT, the scheduler assumes that this address is not in-flight and accesses the CMDE structures for the level 1 data cache. If the CMDE structures do not indicate a miss, the delay is estimated to be the level 1 cache hit latency. If the CMDE structures indicate a miss, the CMDE structures for the level 2 data cache are accessed. Similarly, if the processor has 3rd (or 4th, etc.) level data caches, the CMDE structures for each cache level is probed until one indicates a possible hit (the estimated delay will be equal to the hit latency of this cache) or the last on-chip cache level is reached.

All CMDE techniques (except History-based Prediction) have reliable outputs. In other words, if they produce a miss output, the access is guaranteed to miss. Otherwise, they produce a maybe output. In this case the access might miss or hit in the cache.

4.1. History-Based Prediction (HP)

The first technique is called the History-Based CMDE, which is similar to the load hit/miss prediction technique used in Alpha 21264 [9]. In this technique, the PC of the load addresses is used to access a history table. The entries in the table indicate whether the load has missed in previous executions. Particularly, each table entry is a 4-bit saturating counter that is incremented by 2 for each time the load misses and decremented by 1 for each time the load hits. The most significant bit is used to predict hit or miss. If this bit is 1, the load is predicted to miss. If it is 0, the prediction is maybe. In Alpha, this history table is used to detect ill load operations that miss frequently and thereby reduce the number of recoveries caused by them. Since Alpha uses a flush-based recovery scheme, this prevention has an important positive impact on the performance. Similarly, we should determine the misses due to these ill load operations and use this information to determine the exact latency of the load operations. The main advantage of this technique is that it does not require the source address of the load instruction; instead the PC address is used.

4.2. Table MNM (TMNM)

The second prediction technique is the Table MNM by Memik et al. [11]. The TMNM stores the least significant N bits of the tag values in the cache. If the least significant N bits of the tag of the access do not match any one of the stored values, then the miss is captured.

The values are stored in an array of size 2N bits. During an access, the least significant N bits of the tag are used to address this table. The locations corresponding to the cache tag values are set to 0 and the remaining locations are set to 1. The value stored at the corresponding location is used as the prediction (0 triggers a maybe, 1 triggers a miss output).

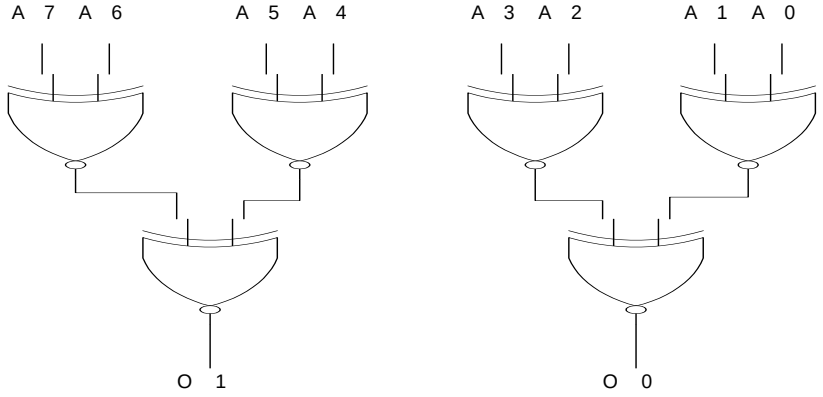

Figure 4: CCMDE example.

4.3. Cross CMDE (CCMDE)

In a tag lookup, all the bits of a tag play a role, i.e. a difference in any one of the bits causes a miss. Therefore, we wanted to design a scheme that considers all tag address bits. Clearly, we cannot use all the bits in a way similar to the table lookup of TMNM, because the complexity of hardware structures would be intolerable. Therefore, in CCMDE we hash the tag address. Particularly, each neighboring bit is XNORed. Then, at the second level the output of neighboring first level XNOR units are XNORed. We continue this until we reduce the tag to a desirable length. An example flow is depicted in Figure 4, which produces a 2-bit output for each 8-bit input. This structure is called an 8-to-2-bit converter. Then, these output bits are used to access a table similar to that of TMNM. In this table, each entry is a 3-bit saturating counter. Whenever a block is placed into the cache, the corresponding position is incremented. During eviction, the counter value is reduced. A counter value of zero means that there is no cache tag that is mapped to the position. If an access address is mapped to such a position, a miss is detected.

To increase the miss detection rate of CCMDE, we use multiple CCMDEs in parallel. For example, if there are 2 parallel 16-to-4-bit converters (generating addresses for two separate tables), the least significant 16 bits of the tag address is directed to the first one and the bits indexed 6 to 21 are directed to the second one. In practice, we have seen that overlapping the input bits increases the CCMDE miss detection rate significantly. In our experiments, we use multiple 16-to-8-bit converters.

4.4. Matrix CMDE (MCMDE)

Because of temporal and spatial locality, when a cache block is accessed, it is very likely that the following load instructions will access blocks that differ in at most one or two bits of the previous access. It is important to capture differences of small number of bits between a cache block and the accessed address. So, we developed a matrix-based scheme. In this scheme, the input bits are aligned in a matrix and then the row XNOR and column XNOR values are found as shown in Figure 5, which presents the basics of the MCMDE for a 16-bit tag address. The main idea behind MCMDE is to match each bit position to two values (one on the column and one on the row), such that if one, two or three bits differ between the accessed address and any cache tag address, the output generated in the second step will be different. This change is captured by looking at the XNOR of the row and column values. Specifically, each Rx value is the XNOR of the bits in the corresponding row. For example

R0 = B0 XNOR B1 XNOR B2 XNOR B3

Similarly, each Cx value is the XNOR of the bits in the corresponding column (C1 = B0 XNOR B4 XNOR B8 XNOR B12). Then, we find the output address bits as shown in Figure 5 (b):

O0 = R0 XNOR C0 XNOR B0

The idea is if there is a single-bit difference that changes both Rx and Cx, it is guaranteed that the change happened in the corresponding diagonal bit index. By including those bits in the output address, we capture any small number of changes. Although not shown in Figure 5, we pad the least significant 5-bits of the tag address to the address generated in Figure 5(b). In the final stage, this address is used to access the MCMDE table, which works similar to a CCMDE table. Therefore, a 5x5 matrix requires a table of size 210 bits.

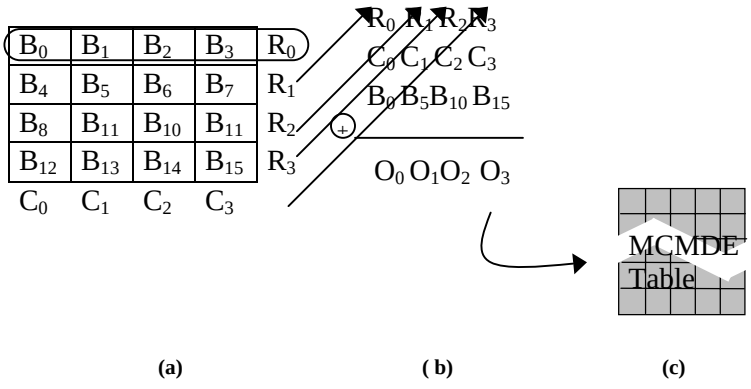

Figure 5: MCMDE example. The steps in capturing a miss with MCMDE: a) The row and column bits are calculated, b) The address is calculated, c) the MCMDE table is accessed that stores the outputs for cache tags.

4.5. Hybrid CMDE (HCMDE)

The previous CMDE techniques explained in this section perform different transformations on the address and use small structures to identify some of the misses. A natural alternative is to combine these techniques to increase the overall accuracy of recognition of the misses. Such a combined CMDE is defined as Hybrid CMDE (HCMDE). Specifically, HCMDE combines the TMNM, MCMDE, and CCMDE schemes. Hence, it also has a reliable output.

Table 1: Operating scenarios that may occur with the scheduler.

	Address Predicted Correctly		Address Predicted Incorrectly	
	Load Misses	Load Hits	Load Misses	Load Hits
TMNM or any CMDE indicate a miss	*Improvement*: Dependent instructions are delayed until data arrives	Is not possible	*Improvement*: Dependent instructions are delayed until data arrives	*Degradation*: Dependent instructions will be delayed unnecessarily
TMNM or any CMDE indicate a maybe	The instructions are executed aggressively as usual (replay performed)	The instructions are executed aggressively as usual	The instructions are executed aggressively as usual (replay performed)	The instructions are executed aggressively as usual

4.6. Discussion

The CMDE techniques (except for HP) discussed in this section never incorrectly indicate a miss. However, they do not detect all cache misses. In other words, if the prediction is a miss, then the block certainly does not exist in the cache. However if the prediction is a maybe then the access might still miss in the cache. The techniques are developed intentionally with this conservative style. The miss predictions should be reliable because the cost of predicting that an access will miss when the data is actually in the cache is high (the dependent instructions will be delayed), whereas the cost of a hit misprediction is relatively less (although the execution is lengthened, the penalties are less severe). Hence, assuming that the address is predicted correctly, CMDE techniques are guaranteed not to degrade performance because of this reliability property. Nevertheless, since the address of the load operations are predicted, there is a chance that the proposed techniques will delay some operations that otherwise would have executed faster. Table 1 presents different scenarios that might occur during the execution. It indicates when the proposed algorithm improves the performance of a traditional scheduler (improvement) and when it might degrade the performance (degradation).

We have measured the delay and energy requirements of the CMDE techniques using either Synopsys Design Compiler or HSPICE. Compared to a 4KB, direct-mapped cache, the MCMDE structures can be accessed within 24% to 46% of the access time of the 4KB cache and use only 1% to 5% of the energy.

The store instructions in the store queue will have priority over PAT-prediction. Therefore, the latency for store aliased loads is known. Once a store leaves the store queue, the data will be

in the level 1 data cache because we assume a write-allocate policy. Therefore, the CMDE will be accurately used for prediction after the stores are retired from the store queue.

5. PAT and Load Aliasing

One of the important problems of finding the exact delay for load accesses is the dependency between different load operations. If a load operation has the same cache block address as another load operation that is scheduled before, the latency of the second load might be affected by the earlier one. Similarly, the access latency of a load may be affected by prefetching. If the latency of a load is affected by another load, we say that there is a load aliasing for the later access. PAT is devised to capture these aliases. PAT is a cache-like structure that is addressed by the source block addresses of the load operation. Note that the earlier load operation will affect the latency of the later load if they have the same cache block address, although they might be accessing different bytes or words. The entries in the PAT are counters. If during the time that the counter value is non-zero another load is predicted to access the same block, the value at the PAT is used as the latency prediction. When the counter reaches zero and the accessed block is placed to the cache, the entry is freed. We can implement these counters by either having each entry as a counter or alternatively we can have a single global counter that is incremented and the entries holding the global time of expected latency.

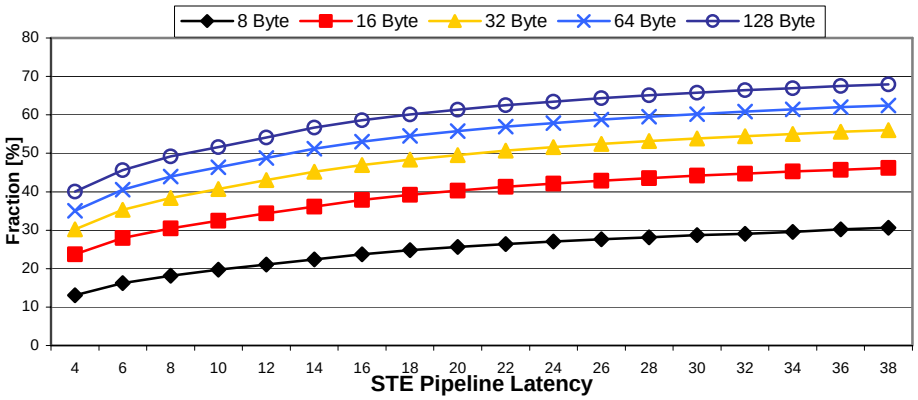

Figure 6: Fraction of dependent loads.

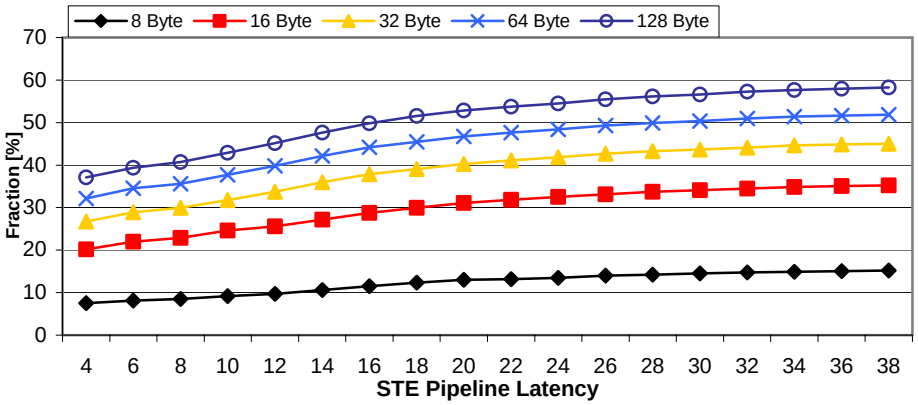

Figure 7: Fraction of aliased loads.

We first investigate the severity of load aliasing. The experimental setup is identical to the simulation parameters as explained in Section 6. We first measure the fraction of loads that have the same source address as another unresolved load operation as the STE pipeline latency is increased. The results are summarized in Figure 6, which plots the fraction of loads that "depend" on a previous load for different cache block sizes. We see that as the STE pipeline latency is increased, the fraction of loads that have the same source address as a previously scheduled load increases.

Next, we studied the fraction of aliased loads among load operations that miss in the first level cache. Note that, all aliased loads have a non-discrete latency (i.e. the latency is not equal to any cache hit latency). Hence, their exact latency can only be found with the help of the PAT. Figure 7 summarizes the results. We see that the access latency for a significant fraction of the loads is affected by load aliasing. For example, for the processor configuration we use in our experiments in the next section (STE pipeline latency of 10), more than 1/3 of the loads that cause recover are aliased.

6. Experimental Results

The SimpleScalar 3.0 [18] simulator is used to measure the effects of the proposed techniques. The necessary modifications to the simulator have been implemented to perform selective replay, address prediction, the CMDE techniques, the PAT, the scheduler, the busses between caches, and port contention on caches. We have also made changes to SimpleScalar to simulate a realistically sized issue queue (as opposed to the whole Reorder Buffer), and to model the events in the issue queue in detail (instructions are released from the issue queue only after we know there is no misscheduling). We simulate 10 floating-point and 10 integer benchmarks from the SPEC2000 benchmarking suite [19]. The remaining SPEC 2000 applications are not included due to simulation problems. The applications are compiled using DEC C V5.9-008 and Compaq C++ V6.2-024 on Digital UNIX V4.0. We simulate 300 Million instructions after fast-forwarding application-specific number of instructions as proposed by Sherwood et al. [17]. Important characteristics of the applications are explained in Table 2.

The base processor is an 8-way processor with an issue queue of 128 entries, an ROB of 256 entries, a 16 KB, 2-way associative level 1 instruction cache and an 8 KB, 2-way associative level 1 data cache. We also present simulation results for a 4 KB, direct-mapped level 1 data cache. Level 1 caches have 4 load/store ports and 2 cycle latencies. The level 2 cache is a 256 KB, 4-way associative cache with 15 cycle latency and has 2 load/store ports. All caches are block-free. The memory latency is set to 320 cycles. In all the simulations, we use a 10-cycle STE pipeline latency. The branch predictor is bimodal with 8 KB table. The branch misprediction penalty is 20 cycles.

In the next section, we present the simulations measuring effects of replay mechanisms. Section 6.2 describes the simulations for measuring effectiveness of the CMDE techniques. Section 6.3 discusses the performance of different PAT structures. Section 6.3 presents the simulations for the proposed precise scheduling. Section 6.5 investigates the power optimizations.

Table 2. SPEC 2000 application characteristics. Execution cycles (cycle), number of load operations executed (Load), number of level 1 data cache accesses (DL1 acc), level 1 data cache miss ratio (DL1 mis), number of level 2 cache accesses (L2 acc), number of level 2 cache misses (L2 mis), number of dependent instructions executed per level 1 cache miss (Dep) until the hit/miss information is received from the cache, and the fraction of level 1 cache hits mispredicted by the HP scheme (HP mis-pred.).

Applications	# cycle [M]	Load [M]	DL1 acc [M]	DL1 mis [%]	L2 acc [M]	L2 mis [M]	Dep	HP mis-pred.
168.wupwise	381.1	68.6	93.0	2.9	4.9	0.5	8.8	19.4
171.swim	864.5	71.8	97.5	17.5	23.2	3.4	0.0	17.3
172.mgrid	801.1	94.7	110.0	16.5	21.2	1.9	5.2	16.7
173.applu	856.6	82.4	114.7	15.5	30.8	3.0	3.7	20.2
177.mesa	369.9	81.2	112.6	2.0	13.9	0.2	6.4	22.8
179.art	3399.5	81.0	103.2	39.7	50.8	22.8	1.9	13.2
183.equake	1535.5	118.9	129.3	18.1	25.6	3.7	8.5	21.8
188.ammp	1158.4	86.2	116.9	8.5	12.8	3.4	4.9	22.5
189.lucas	795.2	51.1	72.0	15.8	17.1	3.3	6.1	12.9
301.apsi	575.7	72.1	111.9	6.6	21.4	1.5	2.3	21.3
164.gzip	405.1	56.2	69.0	14.6	11.4	0.0	15.5	12.8
175.vpr	1195.9	97.1	116.1	6.6	14.2	2.6	8.6	22.4
176.gcc	621.8	81.9	121.2	5.0	25.8	0.4	7.4	24.6
181.mcf	10922.4	83.8	84.6	74.5	63.0	41.2	0.2	4.8
186.crafty	608.5	90.9	115.2	6.1	31.0	0.1	10.8	22.1
197.parser	905.0	63.3	86.2	9.9	9.9	1.8	10.2	16.4
253.perlbmk	430.5	80.6	111.3	2.4	9.5	0.4	13.3	23.0
254.gap	427.2	83.1	110.3	1.2	16.3	0.1	7.5	22.9
255.vortex	553.4	89.6	128.1	1.4	24.9	0.3	7.1	26.7
300.twolf	1173.2	79.9	98.7	8.3	21.2	2.7	10.0	19.5
Average	1399.0	80.7	105.1	13.6	22.4	4.7	6.9	19.1

6.1. Selective Replay

In this section, we present simulation numbers measuring the fraction of the instruction stream that has to be re-executed for selective replay due to cache misses over all the instructions issued. Figure 8 presents the results for increasing the STE pipeline latency in an 8-way processor with 4 KB, direct-mapped cache and Figure 9 presents the results for 8 KB 2-way associative cache. A fraction of 33% means that half of the instructions were re-executed. Note that this also includes

recoveries due to level 2 cache misses. As the STE pipeline latency is increased, we see for both cache types that the fractions of re-executed instructions increase. As the detection time of a miss is increased, the number of instructions dependent on the load also increases. Overall, we see that a major fraction of the instructions have to be re-executed. For example, for the 4 KB level 1 cache, more than half of the instructions are re-executed due to cache misses for an STE pipeline latency of 18 cycles (34.2% re-executed fraction).

Table 3. Coverage of different CMDE techniques for various cache configurations.

Cache Type	Cache Conf. [size, assoc.]	HP 16-entry, 2-way	HP 64 entry, 4-way	TMNM 9x1	TMNM 10x2	CCMDE 16x2	CCMDE 16x3	MCMDE 4x2	MCMDE 5x3	HCMDE 1	HCMDE 2
L1	4K, DM	47.0	57.4	36.3	41.2	48.0	89.4	59.9	83.9	89.8	98.6
	8K, DM	44.0	52.9	36.3	43.0	48.0	88.3	58.9	83.5	88.0	98.3
	32K, 2-way	38.7	43.7	33.8	46.4	41.9	84.0	57.0	82.2	81.1	97.3
L2	256K, 4-way	12.2	14.8	70.1	72.3	59.4	66.9	64.1	90.8	92.7	97.3
	2M, 8-way	4.2	4.4	68.4	73.8	49.4	66.4	54.6	88.6	89.6	97.2

Table 4. The fraction of aliased loads captured by PAT.

STE Pipeline Latency	PAT configuration					
	8 entries, DM	8 entries, 2-way	16 entries, DM	16 entries, 2-way	32 entries, DM	32 entries, 2-way
6	91.1	98.1	94.2	99.5	97.1	99.8
10	87.9	95.8	92.5	99.0	96.0	99.5
14	86.0	94.5	91.3	98.4	95.2	99.2
18	84.2	92.9	90.3	97.9	94.5	98.9
22	82.4	91.3	89.0	97.1	93.7	98.5
24	81.1	89.8	88.0	96.3	92.9	97.9
28	80.1	88.7	87.5	95.8	92.5	97.8

6.2. CMDE Techniques

The success for a CMDE technique is measured in coverage. Coverage is the fraction of the misses that are detected by a CMDE technique. We have performed several experiments to measure the effectiveness of the CMDE techniques for various sizes of level 1 and level 2 caches. The results are summarized in Table 3. Table 3 presents the average coverage of the techniques for the 20 SPEC 2000 applications simulated. In addition, it presents the coverage for various level 1 and level 2 cache configurations. HCMDE 1 combines TMNM 9x1 (a single TMNM table of size 29), CCMDE 16x2 (CCMDE with two 16-to-8-bit converters), and MCMDE 4x2

(MCMDE with two 4x4 matrices). HCMDE 2 combines TMNM10x2 (TMNM with 2 tables of size 210), CCMDE16x3 (CCMDE with three 16-to-8-bit converters), and MCMDE5x3 (MCMDE with three 5x5 matrices). The most notable trend in the results is that for most of the techniques, as the cache size is increased, the coverage reduces.

6.3. PAT Performance

In this section, we experiment with different PAT sizes and investigate how the fraction of the aliased loads captured changes with the PAT size. The results are summarized in Table 4, which presents the fraction of the aliased load operations, for which the accurate access latency was found through different PAT structures. The entries in Table 4 are the averages for all 20 SPEC applications simulated. In Section 5, we have shown that load aliasing was frequent for most processor configurations. Fortunately, these dependencies can be very easily resolved using the PAT. For example, a PAT with 16 direct-mapped entries can recognize 92.5% of the aliased loads.

6.4. Scheduler Performance

In this section, we present simulations that investigate the performance of a processor equipped with the proposed scheduler. The first set of experiments assumes perfect information about load source address (oracle). The second set of experiments uses different address prediction techniques, and the third set of experiments investigate the effects of prefetching on precise scheduling.

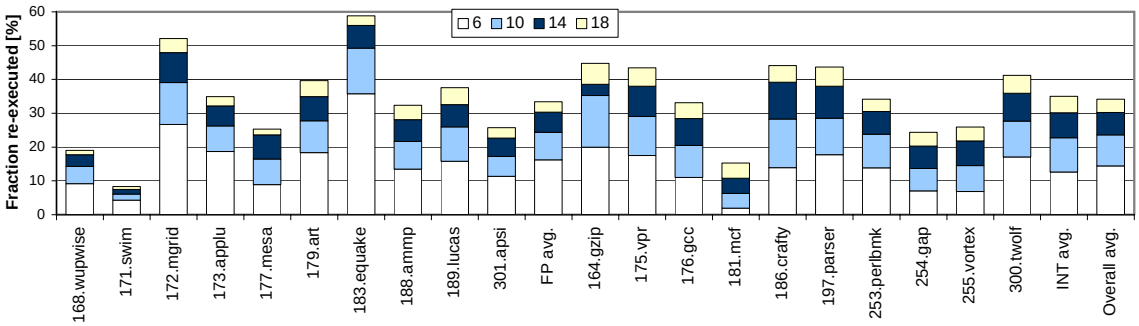

Figure 8: Fraction of re-executions performed over all instructions executed for the processor with 4 KB, direct-mapped cache.

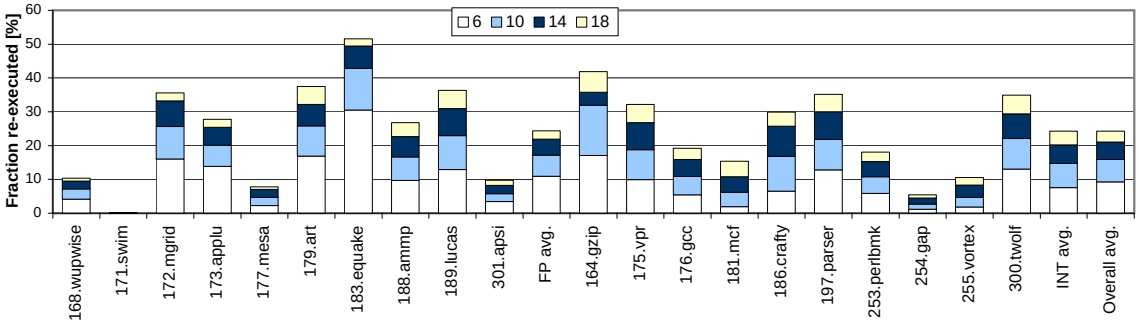

Figure 9: Fraction of re-executions performed over all instructions executed for the processor with 8 KB, 2-way associative cache.

In all the figures, we present simulation numbers for precise schedulers using 6 different CMDE techniques: HP64x4 (History-based Prediction scheme with a 64 entry, 4-way associative table), TMNM10x2, CCMDE16x3, MCMDE5x3, HCMDE 2 (combination of TMNM10x2, CCMDE16x3, and MCMDE5x3). We also present numbers for a scheduler that uses perfect CMDE technique (PERF) with which the scheduler has exact hit/miss information. We always use a 2-way associative PAT with 16 entries. Note that, even a very small PAT is able to capture a very large fraction of the load aliases. Therefore, we do not experiment with different PAT sizes in this section.

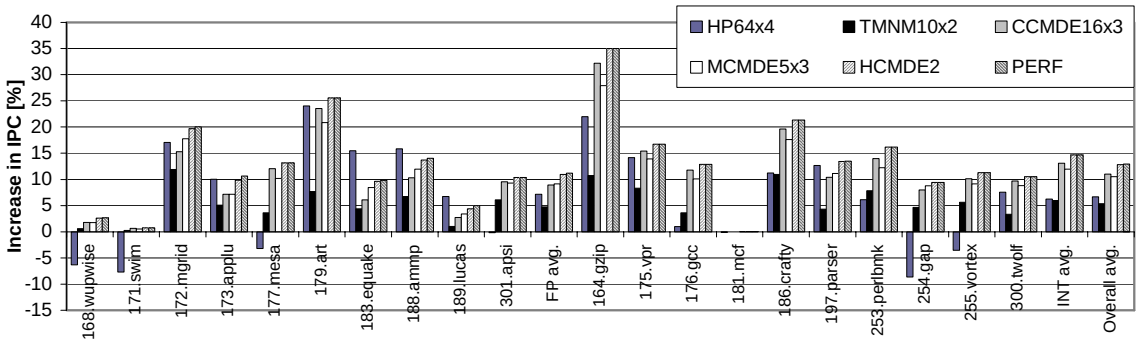

Figure 10: Performance improvement by the scheduler for perfect address prediction. The level 1 data cache is 8 KB, 2-way set associative.

Figure 10 presents the results where an oracle scheduler exactly knows the address of load operations. Although this is not a realistic assumption for many processors, there has been related work that aim to generate load addresses in advance for optimizations [2, 16]. In this scenario, the scheduler that uses HCMDE 2 increases the performance of the processor with 8 KB 2-way associative cache by 12.8%. The scheduler with the perfect cache miss detection improves the performance by 13.0%, showing the effectiveness of the CMDE techniques.

The next set of experiments use a 2-delta stride-based predictor to generate the load addresses used to probe the PAT and the CMDE structures. We use a predictor with 16 K entries. The results are summarized in Figure 11. On average the scheduler reduces the execution cycles by 10.8% using the perfect CMDE and by 10.1% using HCMDE 2. Note that the HP scheme is not affected by the address prediction, because it uses PC of the load instructions. If we compare the precise scheduling with realistic address prediction (Figure 11) and perfect address prediction (Figure 10), we see that for many applications there is little difference. There are three reasons for the similarities. First, the stride predictor is quite successful in predicting addresses: on average 75.3% of the addresses are predicted correctly. For the addresses that are predicted incorrectly, in many cases CMDE returns a maybe, so the instruction is not delayed. Finally, there is a relation between accesses that are predicted incorrectly and the hit/miss outcome of the access. If the predictor cannot successfully predict the address, this means that the access is either not seen before or the load accesses an unusual location. In either case, the CMDE is more likely to return a miss and the access is more likely to miss in the cache than an average access. Therefore, the negative effects of address prediction remain small. Although not presented due to lack of space, the HCMDE 2 technique reduces the number of instruction issues by as much as 52.5% and 16.9% on average.

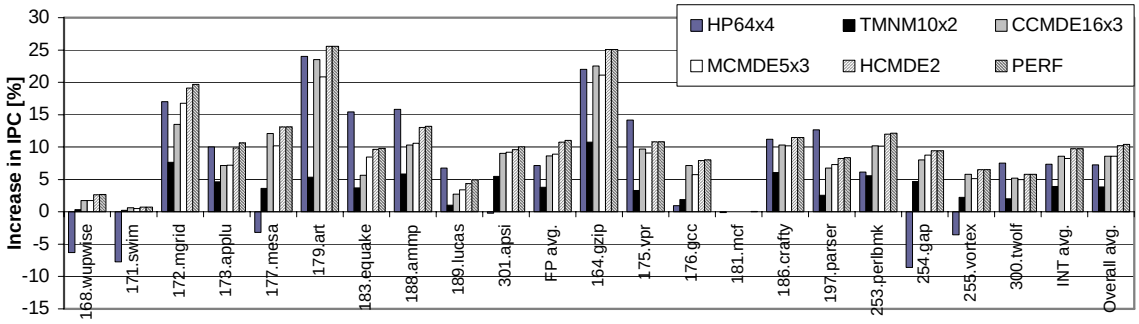

Figure 11: Performance improvement by the scheduler for stride-based address prediction. The level 1 data cache is 8 KB.

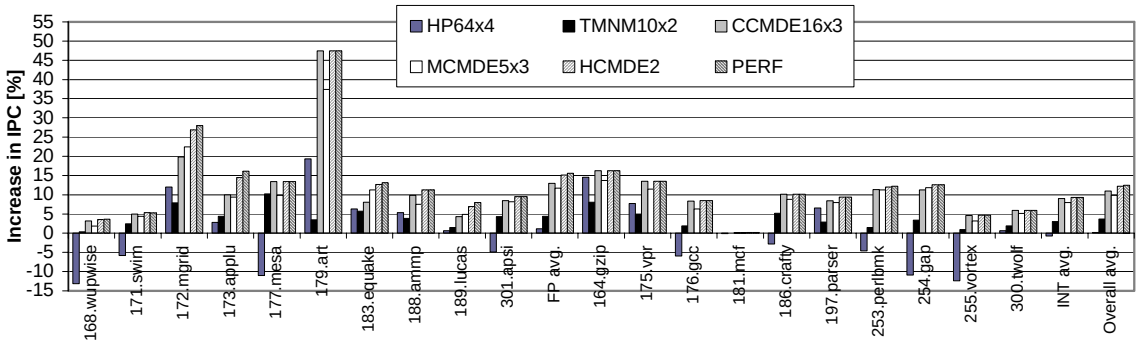

Figure 12: Performance improvement by the scheduler for stride-based address prediction. The level 1 data cache is 4 KB.

Figure 12 presents the simulations for the processor with 4 KB direct-mapped level 1 data cache. Except for the HP technique, the performance of the scheduler generally improves when the cache size is reduced. Specifically, the scheduler with TMNM10x2, CCMDE16x3, MCMDE5x3, HCMDE2, and perfect CMDE increases the performance of the base processor (with 4 KB level 1 data cache) by 5.3%, 11.4%, 11.4%, 13.9%, and 14.2%, respectively. One interesting application is the 181.mcf. Although, this application causes a lot of cache misses (miss rates of 88.1% on level 1 and 65.4% on level 2), the performance is hardly improved with the scheduler. The reason for this is that there are too many level 2 cache misses. Therefore, most load operations see the full memory access latency. Hence, precise scheduling cannot optimize the execution of the dependent instructions. However, as we will present in the next section, power consumption is significantly improved due to the reduced number of replays.

The simulations measuring the effects of load aliasing alone are not presented in detail. In summary, the results were analogous to the fraction of load aliasing: on average 25% to 40% of the IPC improvement is due to PAT structures.

In the next set of experiments, we combine precise scheduling with prefetching. Once the scheduler detects a cache miss indicated by the CMDE structures, it immediately starts prefetching the data from level 2 cache. To prevent cache pollution, the prefetched blocks are placed into a prefetch buffer that is accessed in parallel with the level 1 cache. We use a 32-entry fully associative cache with 256 byte block size to store the prefetch data. On average, the

prefetching mechanism improves the performance of the simulated applications by 9.2% for the processor with the 8 KB, 2-way associative cache. Figure 13 presents the results for a processor with 8 KB, 2-way associative cache. Our goal in these experiments is to measure the effectiveness of precise scheduling rather than evaluating prefetching. Therefore, base processors for each bar in Figure 13 correspond to a processor that determines a miss using the PAT and the CMDEs, prefetches the data but does not perform precise scheduling. We see that, when prefetching is used the performance improvement of the proposed techniques increases. Prefetching increases the number of in-flight data accesses and hence signifies the importance of access delay calculation. On average, the performance of the processor with 8 KB 2-way associative cache is improved by 14.3% and 14.6% using the precise scheduling with HCMDE 2 and perfect CMDE structures, respectively.

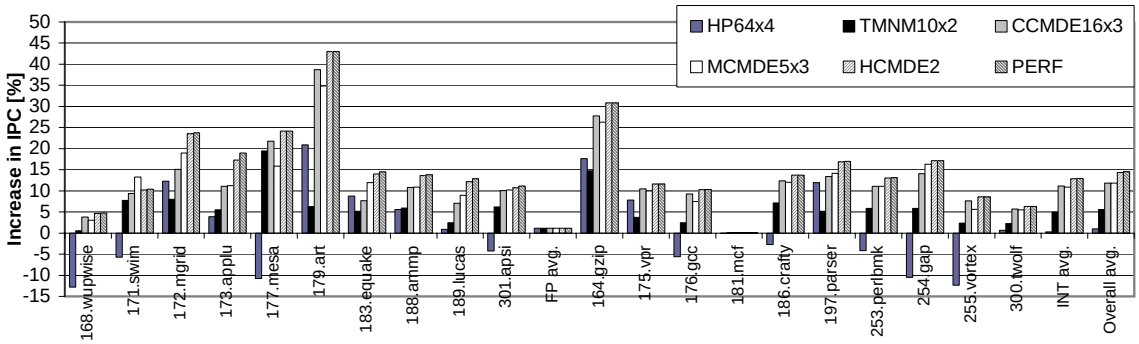

Figure 13: Performance improvement by the scheduler when processor pre-fetches captured misses. The level 1 data cache is 8 KB.

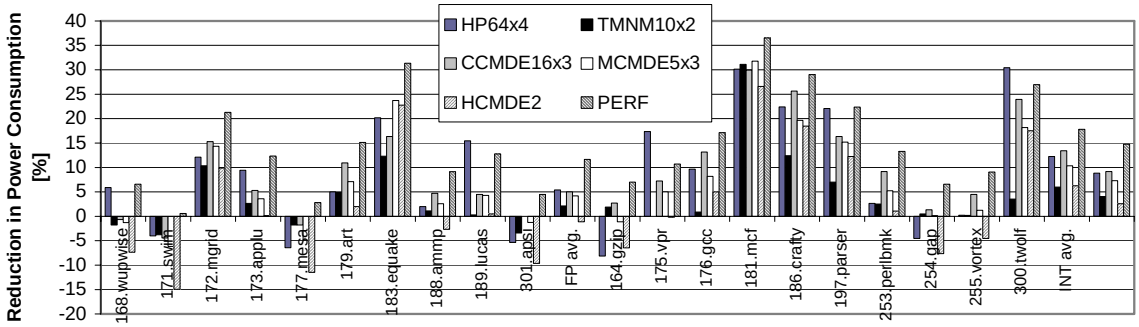

Figure 14: Power reduction by the scheduler. The level 1 data cache is 8 KB.

6.5. Power Reduction

Finally, we develop a model to calculate the power savings achieved using the proposed techniques. To achieve this we have modified SimpleScalar to collect data about the activity in the schedule-to-execute window, dispatch unit, register file, and execution units. We first measure the reduction in bit value change of these units using the proposed techniques. To calculate the power consumption of the overall processor we use the model by Gowan et al. [5]. On average, the abovementioned units are responsible for 48% of the overall power consumed by the processor. The reduction in bit value change is directly proportional to the reduction of dynamic power consumption of the examined units. Therefore, using the model and the values obtained

from the simulator, we can accurately estimate the overall power reduction in the processor. We also add the power consumed by the proposed structures to find the total power consumption if applicable.

We present the results for the processor with 4 KB direct-mapped cache that uses stride-based address prediction. The results are summarized in Figure 14. The figure presents the fraction of power reduced for each simulated application. On average, the scheduler using HCMDE 2 and CCMDE16x2 reduce the power consumption by 1.3% and 9.2%, respectively.

7. Related Work

Various researchers have realized the impact of load scheduling and the instructions dependant on them on the processor efficiency. Both hardware and software techniques have been proposed to generate the load addresses early in the pipeline [1, 2, 16]. Such techniques can be useful to our proposed mechanism by providing the load addresses without the need of address prediction. In this context, load hit/miss prediction has also been studied. Mowry and Luk [13] examine software profiling to predict hit/miss outcome of accesses to dispatch these instructions earlier. Alpha 21264 utilizes a hardware-based hit-miss predictor [9]. Yoaz et al. [20], propose hit/miss prediction using a history table like those used for branch prediction. Raasch et al. [15] use a hit/miss predictor to be used for forming desired dependency chains in the efficient instruction queue they have designed. The CMDE techniques are substantially different from these prediction techniques, which rely on instruction PC for the predictions. Peir et al. [14] study bloom filtering, which is similar to our CMDE techniques. However, they assume the availability of addresses (hence no prediction) and only investigate the effects for a processor with flush replay. We believe that the selective replay is an integral part of deeply-pipelined processors. None of these techniques consider in-flight data, which is an integral part of our scheme.

Moreshet and Bahar [12] did a comprehensive study on the effects of load hit prediction on performance as pipeline depth is varied. Their main focus is to devise an issue queue structure that can efficiently serve execution for deep pipelines under load hit speculation. Their work does not involve predicting exact access times of load instructions and manipulating the scheduler accordingly. Similarly, Lebeck et al. [10] and Ernst et al. [3] propose efficient issue queue designs under load speculation. Our study is orthogonal to these techniques. In fact, precise scheduling can be effectively utilized by these techniques to increase their performance.

8. Conclusion

In this paper, we have proposed a novel precise scheduling technique that determines the accurate access latencies of load operations. This delay is subsequently used to make decisions about issuing of the dependent instructions. These instructions are issued such that they arrive at the corresponding stages in the pipeline at the exact cycle when the data is available to them. We first showed that to find accurate access latency, in-flight data has to be monitored. We proposed a small previously accessed table (PAT) that can efficiently determine aliased loads. We have also presented two novel Cache Miss Detection Engine (CMDE) techniques and also studied two existing schemes. The scheduler that uses the PAT and CMDE combined with a stride-based address predictor increases the performance of a processor with 8 KB 2-way associative cache by 10.1% on average. The performance of a more aggressive processor that uses prefetching is improved by as much as 42.1% (14.3% on average). We have also presented simulation results that show the effects of replay mechanisms for various pipeline and cache configurations. These

simulations showed that the problem of cache misses and re-executions related to these misses are increasing with the increase in the number of pipeline stages. Therefore, it is likely to become an even more serious handicap for future generation processors, which motivates further investigation of techniques that perform extensive planning of instruction scheduling.

Acknowledgements

The authors would like to thank Seda O. Memik for her assistance during the progress of this work. We also thank our anonymous reviewers for their helpful comments.

References

1. Cheng, B.-C., D. A. Connors, and W. W. Hwu. Compiler-Directed Early Load-Address Generation. In International Symposium on Microarchitecture, 1998.

2. Chung, B.-K., J. Zhang, J.-K. Peir, S.-C. Lai, and K. Lai. Direct load: dependence-linked dataflow resolution of load address and cache coordinate. In International Symposium on Microarchitecture, Dec. 2001. Austin / TX.

3. Ernst, D., A. Hamel, and T. Austin. Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay. In International Symposium on Computer Architecture, June 2003. San Diego / CA.

4. Gonzalez, J. and A. Gonzalez. Speculative Execution via Address Prediction and Data Prefetching. In 11th International Conference on Supercomputing, Jul. 1997.

5. Gowan, M. K., L. L. Biro, and D. B. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In 35th Design Automation Conference, 1998. San Francisco / CA.

6. Hartstein, A. and T. R. Puzak. Optimum Pipeline Depth for a Microprocessor. In International Symposium on Computer Architecture, May 2002. Anchorage / AK.

7. Hinton, G., D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, The microarchitecture of the Pentium 4 processor. 2001.

8. Jouppi, N. P. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache Prefetch Buffers. In 25 Years {ISCA}: Retrospectives and Reprints, 1998.

9. Kessler, R., The Alpha 12264 Microprocessor. IEEE Micro, Mar/Apr 1999, 19(2).

10. Lebeck, A. R., T. Li, E. Rotenberg, J. Koppanalil, and J. Patwardhan. A Large, Fast Instruction Window for Tolerating Cache Misses. In International Symposium on Computer Architecture, May 2002. Anchorage, AL.

11. Memik, G., G. Reinman, and W. H. Mangione-Smith. Just Say No: Benefits of Early Cache Miss Determination. In International Symposium on High Performance Computer Architecture, Feb. 2003. Anaheim / CA.

12. Moreshet, T. and R. I. Bahar. Complexity-Effective Issue Queue Design Under Load-Hit Speculation. In Workshop on Complexity-Effective Design, May 2002. Anchorage / AK.

13. Mowry, T. C. and C. K. Luk. Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling. In International Symposium on Microarchitecture, Dec. 1997.

14. Peir, J.-K., S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In International Conference on Supercomputing, June 2002. New York / NY.

15. Raasch, S. E., N. L. Binkert, and S. K. Reinhardt. A Scalable Instruction Queue Design Using Dependence Chain. In International Conference on Computer Architecture, May 2002.

16. Roth, A. and G. Sohi. Speculative Data-Driven Multithreading. In International Conference on High Performance Computer Architecture, Jan. 2001.

17. Sherwood, T., E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In International Conference on Parallel Architectures and Compilation Techniques (PACT 2001), Sep. 2001. Barcelona, Spain.

18. SimpleScalar Llc. SimpleScalar Home Page, http://www.simplescalar.com

19. Standard, P. E. C., Spec CPU2000: Performance Evaluation in the New Millennium, Version 1.1. December 27, 2000.

20. Yoaz, A., M. Erez, R. Ronen, and S. Joourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In International Conference on Computer Architecture, 1999. Atlanta / GA.