The University of Jordan

Faculty of Engineering

Computer Engineering Department

Report

Towards Precise Instruction Scheduling

Dr. Gheith Abandah

Prepared By

Shereen Ismael

December 2009

Abstract

Given the relatively high frequency of memory reads (load instruction account for roughly 20%-30% of all instructions executed), memory latency, that is the time it takes for memory to respond to requests, can have a significant impact on performance. One solution is to use multiple faster but smaller cache memory levels.

Many techniques are proposed aim to estimate the cache access time of load instruction in a multi-level caches environment, thereby enabling more precise scheduling of instructions dependent on these loads.

On the other hand sending loads to memory as early as possible requires moving loads up in the execution order, placing them in a position that might be different than the one implied by the program (out of order execution).

So as a consequence, Load may execute before a preceding store on which it may be data dependent (i.e., the store may be writing the data needed by the load). Memory dependence speculation explains that care must be taken to balance the benefits of correct speculation against the net penalty incurred by erroneous speculation.

Introduction

Instructions with a varying latency (e.g load operation) constitute a challenge. This report focus study of load instruction first by resolving memory dependence violation by using logic that enforces correct memory dependences, commonly referred to as the load-store queue (LSQ), and typically implemented as two separated queues: the load queue (LQ) and the store queue (SQ)[9]. These queues contain complete addresses and their entries are allocated in program order. To enable early execution of loads without compromising program correctness, memory instructions are tracked by the two queues and associative searches are used to find the correct producer or to detect dependence violations.

The second part of this report is in orthogonal direction aiming to predict load memory access latency. The goal here is to send loads to memory earlier, as far from the instructions that need the data that will be read, and predicting load access latency in a load use scenario(when preceding store is not available). 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. Newly 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.

1. Memory Dependence Speculation

Memory dependence speculation implies to not delay executing a load until all its ambiguous dependences (i.e., a load consumes a value that may be produced by a store preceding it in the total order) are resolved. Instead, we guess whether the load has any true dependences. As a result, a load may be allowed to obtain memory data speculatively before a store on which it is ambiguously dependent executes. Eventually, when the ambiguous dependences of the load get resolved, a decision is made on whether the resulting execution order was valid or not. If no true dependence has been violated, speculation was successful. In this case, performance may have improved as the load executed earlier than it would had to wait for its ambiguous dependences to be resolved. However, if a true dependence was violated, the speculation was erroneous (i.e., a mispeculation). In the latter case, the effects of the speculation must be undone. Consequently, some means are required for detecting erroneous speculation and for ensuring correct behavior. One of the mechanisms that provide this functionality is age ordered queue as we will explain later.

Though memory dependence speculation may improve performance when it is successful, it may as well lead to performance degradation when it is wrong. We demonstrate either possibility with the example of Figure 1. The reason is that a penalty is typically incurred on mispeculation. The penalty includes the following three components: (1) the work thrown away to recover from the mispeculation, which in the case of squash invalidation, may include unrelated computations, (2) the time, if any, required to perform the invalidation(hardware techniques used today by invalidating and re-executing all instructions following the mispeculated load), and finally (3) the opportunity cost associated with not executing some other instructions instead of the mispeculated load and the instructions that used erroneous data. Consequently, in using memory dependence speculation care must be taken to balance the performance benefits obtained when speculation is correct against the net penalty incurred by erroneous speculation. To gain the most out of memory dependence speculation we would like to use it as aggressively as possible while keeping the net cost of misspeculation as low as possible.

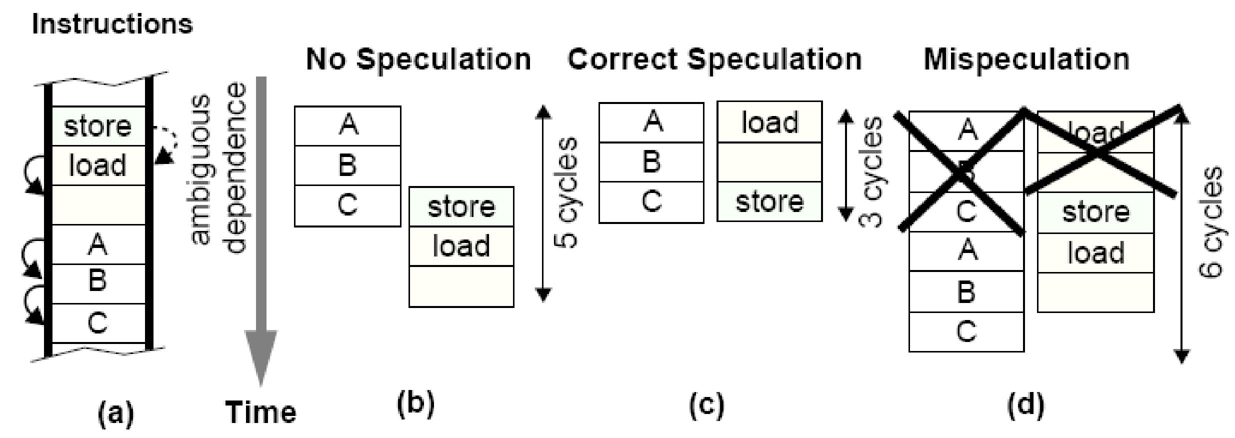


Figure 1: Using memory dependence speculation may affect performance either way. (a) Code with an ambiguous memory dependence. Continuous arrows indicate register dependences. Parts (b) through (d) show how this code may execute in a dynamically-scheduled ILP processor capable of executing two instructions per cycle. We assume that due to other dependences, the store may execute only after two cycles have passed. (b) Execution order when no memory dependence speculation is used. (c) Memory dependence speculation is used and the ambiguous dependence gets resolved to no dependence. (d) Memory dependence speculation is used, and the ambiguous dependence gets resolved to a true dependence.

1.1 Age-ordered queue

Out-of-order microprocessors typically allow early execution of loads for high performance, even if some older store instructions have not yet been resolved. Thus, when an earlier store does access the same memory location, the data returned by the earlier speculative load becomes incorrect and the processor must take a corrective action to ensure the sequential semantics. This dependence enforcement is achieved using age-ordered load queue (LQ) and store queue (SQ).

When a load executes, in addition to the cache access, its address is checked with all older stores in the SQ. If a match is detected, the youngest store forwards the data to the load (load forwarding). Conversely, when a store executes, it must check the LQ looking for younger loads to the same address that have executed prematurely. When matches are found, the processor needs to re-execute (or replay) premature loads and their dependents.

2. Speculative execution of load instruction

Unfortunately, a modern processor schedules an instruction well before it executes, and the latency of some instructions can only be determined by their execution. For example, the latency of a load instruction depends on where in the cache/memory hierarchy its data exists (when preceding store is not available), and can only be determined by executing the load and querying the caches.

At the time the load is scheduled, its latency is unknown. At the time its dependents should be scheduled, its latency may still be unknown. Hence, the timely scheduling of the instructions that are dependent on a load is a problem in modern processors.

One possible solution to this problem is to schedule the dependents of a load only after the latency of the load is known. The processor delays the scheduling of the dependents until it knows the load hit the cache. This effectively increases the load's latency to the amount of time between when the load is scheduled and when its cache hit/miss status is known. This solution introduces bubbles into the pipeline, and can devastate processor performance. Simulations show that a processor using this solution drops 17% of its performance [4].

A better solution is to use data speculation. The processor speculates that a load will hit the cache (a good assumption given cache hits rates are generally over 90%), and schedules its dependents accordingly. If the load hits then schedules the dependent operations according to this latency. If the load misses, any dependents that have been scheduled will not receive the load's result before they begin execution. All these instructions have been erroneously scheduled, and the dependent instructions need to be re-executed. There are two popular re-execution (or replay) mechanisms that are used in current microprocessors:

- Flush replay (used in the integer pipeline of Alpha [6]) all instructions that scheduled to execute are flushed and re-executed whether they are related to the load operation or not. Flush replay is less desirable in deeply pipelined processors where many instructions are scheduled to execute and miss prediction load instructions are frequent.

- Selective replay (used by Pentium 4 [9]). In selective replay, the processor only re-executes the instructions that depend on the missed load [9]. The replay logic keeps track of the dependent uops of each speculative load. When a load misses, all its dependent uops are re-executed with the correct data when that data becomes available. In this mechanism; each issued instruction carries a re-execute bit (as an index). 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.

To reduce the penalty due to data mis-speculations, the processor can predict whether the load will hit the cache, instead of just speculating that the load will always hit. Existing cache hit/miss predictors, however, can only correctly predict about 50% of cache misses. This report focus study on hit/miss predictor that used to identify cache misses early in the pipeline. This early identification of cache misses allows the processor to more accurately schedule instructions that are dependent on loads and to more precisely prefetch data into the cache.

3. **Problem Statement**

To show the problem in scheduling the instructions that are dependent on a load, consider a baseline pipeline model that is similar to Alpha 21264 [6]. In the baseline model, the front-end pipeline stages are: instruction fetch and decode/rename. After decode/ rename, the ALU instructions go through the back-end stages: schedule, register read, execute, writeback, and commit.

Additional stages are required for executing a load. After decode/rename, loads go through schedule, register read, address generation, two cache access cycles, an additional cycle for hit/miss determination ,writeback, and commit.

Thus, there are a total of 7 and 10 cycles for ALU and load instructions, respectively.

Figure 2 shows an example. For simplicity, the front-end stages are omitted. In this example, the add instruction consumes the data produced by the load instruction. After the load is scheduled, it takes 5 cycles to resolve the hit/miss. However, the dependent add must be scheduled the third cycle after the load is scheduled to achieve the minimum 3-cycle load-use latency and allow back-to-back execution of these two dependent instructions. If the processor speculatively schedules the add assuming the load will hit the cache, the add will get incorrect data if load actually misses the cache. In this case, the add along with any other dependent instructions scheduled within the illustrated 3-cycle speculative window must be canceled and rescheduled.

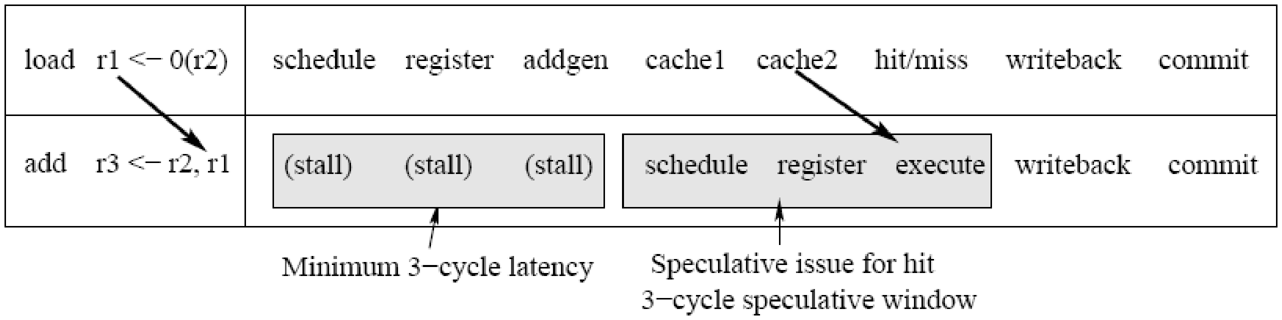


Figure 2: Example of data speculation for a load instruction

Another suggested property of the scheduler is by regarding the execution of load instructions will determine the priorities for the operations that is going to execute. The scheduler will give low priority to the instructions that depend on a load and try to issue independent operations.

However, even with such a mechanism it is unlikely to fill the issue slots and such a mechanism can slow down the critical instructions. The scheduler used in our example gives priority to the oldest instruction in the issue window.

4. 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. Here we will overviews precise scheduling and discuses the address prediction techniques used to identify misses.

Figure 3 show the scheduling scheme which consists of two structures: Previously-Accessed Table (PAT) and Cache Miss Detection Engine (CMDE). Once the exact latency of a load operation is found, the arrival time of the source data is passed along to the instructions that depend on the load.

The PAT is a small cache addressed by the predicted addresses of the load instructions. When access latency is predicted for a load source address, the latency is stored in the PAT.

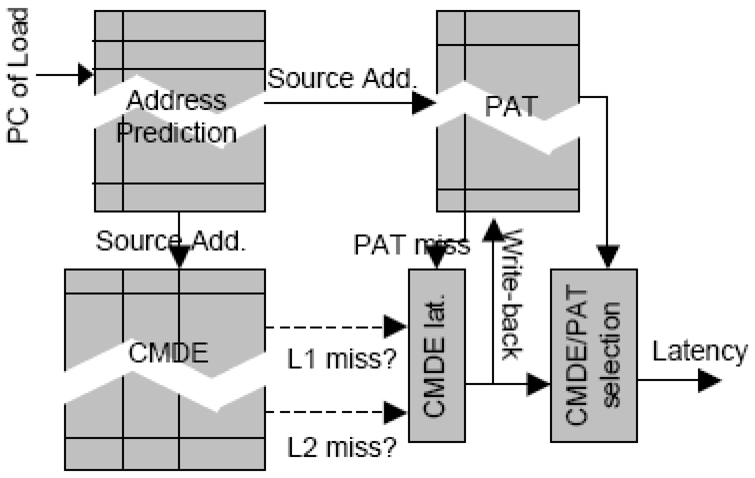


Figure 3: Scheduling scheme

If a load is not aliased (not found in PAT), 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. The predictor use two types of structures from literature: an MNM machine [3] and a history-based miss prediction scheme [6]. The motivation behind the CMDE techniques is to use small hardware structures to quickly determine whether an access will miss in the cache. The load addresses, which are required by the CMDE structures, are not available during the scheduling stage. Therefore, these technique uses address prediction to generate the addresses [7]. The generated information about the cache hits and misses is used 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.

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

1. Source address for the load is predicted (to generate the addresses [7]),

2. The PAT is probed for the address, 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.

4.1 History-Based Prediction (HP)

History-based predictors rely on the assumption that past behavior is a strong indicator of future behavior. This is reason of naming this technique History-Based prediction, which is similar to the load hit/miss prediction technique used in Alpha 21264 [6]. In this technique, the PC of the load address is used to access a history table. The entries in the table indicate whether the load has missed in previous executions. 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 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 miss predicted 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 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 have reliable outputs. Hence, if they produce a miss output, the access is guaranteed to miss. Otherwise, they produce a maybe output then the access might miss or hit in the cache.

Cache miss detection structures (MNM machine)

Returning to the cache miss detection engine built in some structure called MNM machine where cache miss detection techniques store partial information to make the decision whether an access may hit or will definitely miss in the cache. These techniques are built into mostly no machine (MNM machine) structure as figure 4 shows.

MNM structures are much smaller than the corresponding cache structures so MNM reduces the delay and power consumption of the cache system. There are two possible locations of an MNM in a processor categorized as follows:

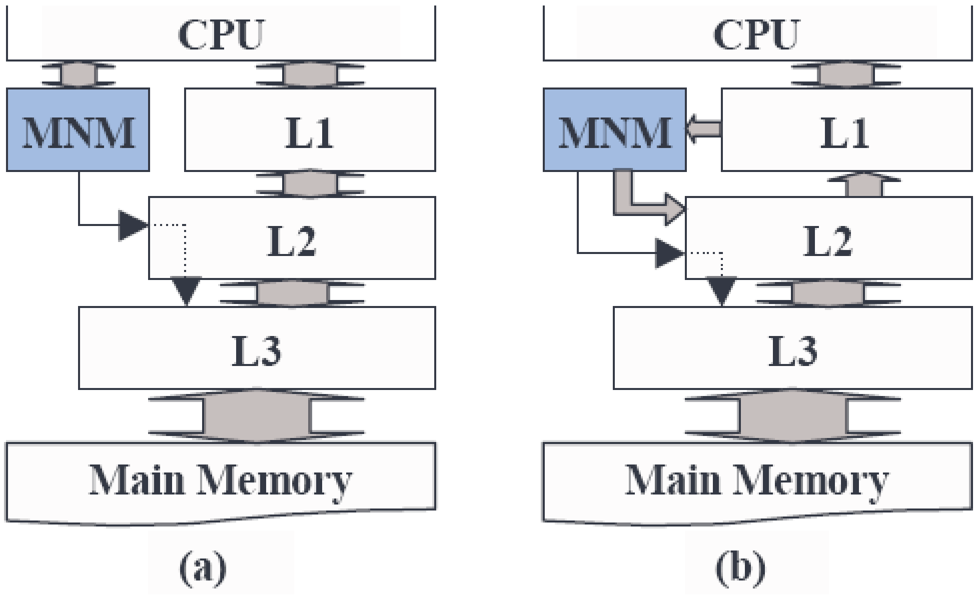


Figure 4: Different Mostly No Machine (MNM) positions in a processor with 3 cache levels. In (a) the MNM and L1 cache are accessed in parallel, in (b) MNM is accessed only after the L1 miss.

- **Parallel MNM**: the L1 cache and the MNM are accessed in parallel. When the address is given to the MNM, it identifies which cache has a block that contains the specific address. When the L1 cache miss is detected, we know which caches to access, because in all techniques, the MNM delay is smaller than the L1 delay. Hence, the miss signals are generated prior to the detection of L1 cache miss, and it is tagged to the control signals after the L1 miss detection.

- **Serial MNM**: the MNM is accessed only after level 1 cache misses. The advantage of this configuration is the reduced power consumption by the MNM. If the level 1 hit ratios are high, the parallel MNM will not generate any useful information for most of the accesses. The serial MNM, on the other hand, has the disadvantage of higher cache access times. For the serial MNM, the access times to any cache level except the level 1 cache are increased by the delay of the MNM.

All the techniques, has separate structures storing information about the different caches. When a decision has to be made, the collective information is processed by accessing all the structures in parallel and interpreting the data stored in these structures.

When a block is placed into a cache, all the bytes in that block are placed into the cache. Therefore, to recognize misses, there is no need to store information about the exact bytes that are stored in a cache. Instead in all the techniques, the block addresses are stored. The block address is composed of the tag and the index sections of an address. Figure 5 shows the block address portion of an address. For example, if the cache has 128-byte block size, each address is shifted 7 bits right before it is entered to the MNM. Therefore, an address entered to the MNM corresponds to a block address placed into or replaced from the cache.

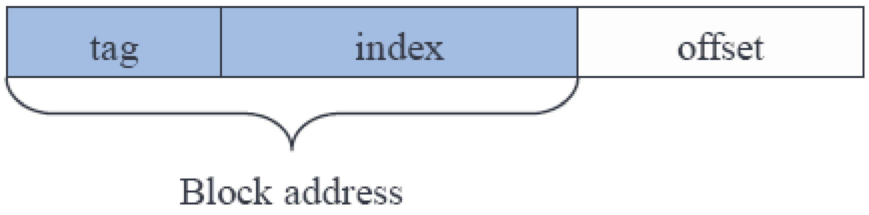


Figure 5: Block address used by MNM

The techniques do not assume the inclusion property of caches. In other words, we assume that if cache level I contains a block b, block b is not necessarily contained in cache level i+1. In addition, the MNM checks for the misses on cache level i+1, even if it cannot identify a miss in cache level i. Since the miss signals are propagated with the access, level i+1 can still bypass the access even if the access is performed in cache level i. For example, if the MNM identifies the miss in L3 cache but could not identify at L2 cache, first the L2 cache will be accessed. If it misses, the L3 cache will be bypassed because a miss was identified for the cache. In all the techniques, a miss for a cache level is indicated by a high output of the MNM structures.

Returning to CMDE techniques, we investigate two variants of this approach: the first is based on partitioned-address matching, and the second is based on partial-address matching. Experimental results show that, for modest-sized predictors, Filters outperform predictors that used a table of saturating counters indexed by load PC. These table-based predictors operate just like the predictor for the Compaq Alpha 21264, except they have multiple counters instead of just one. As an example, for an 8K-bit predictor, the Filter mispredicts 0.4% of all loads, whereas the table-based predictor mispredicts 8% of all loads. This translates to a 7% improvement in IPC over the table-based predictor. Compared to a machine with a perfect predictor, a machine with Filters has 99.7% of its IPC [4].

4.2.1 Partitioned-Address Filter

In this technique, consider a cache block address with n bits (ignoring the offset bits). A large, direct-mapped array of 2n bits is required to precisely record whether each cache line address is in the cache. To reduce the space and allow a quick access, a partitioned-address filter can be constructed. Instead of using the entire cache line address, the address can be split into m partitions, with each partition using its own array of bits. The result is m sub-arrays with 2n/m bits, each of which records the membership of the respective address partitions of lines stored in the cache. A cache miss is identified when one or more of the address partitions for the address of a requested line does not belong to the respective address partition of any line in the cache.

Figure 6 shows how the partitioned-address filter works. A load address is partitioned, in this example, into 4 equally divided groups, A1, A2, A3, and A4. Each of the four address partitions is used to index separate BF arrays, BF1, BF2, BF3, and BF4, respectively. Each entry in the BF arrays contains the information of whether the address partition belongs to the corresponding address partition of any line in the cache. If any of the 4 arrays indicates one of the address partitions is absent from the cache, the requested line is not in the cache. Otherwise, the requested line is probably in the cache, but it's not guaranteed to be. Given the fact that a single address partition can exist for multiple lines in the cache, the primary difficulty of the

Partitioned-address filter is to maintain the correct membership information. When a line is removed from the cache, an exhaustive search is necessary to check if the address partitions for the address of the removed line still exist for any of the remaining lines. To avoid such a search, each entry in the array contains a reference counter that keeps track of the number of cache lines with the entry's corresponding address partition. When a cache miss occurs, each counter for the address partitions for the address of the newly-requested line is incremented, while the counters for the address partitions for the address of the replaced line are decremented.

A zero count indicates the corresponding address partition does not belong to any line in the cache.

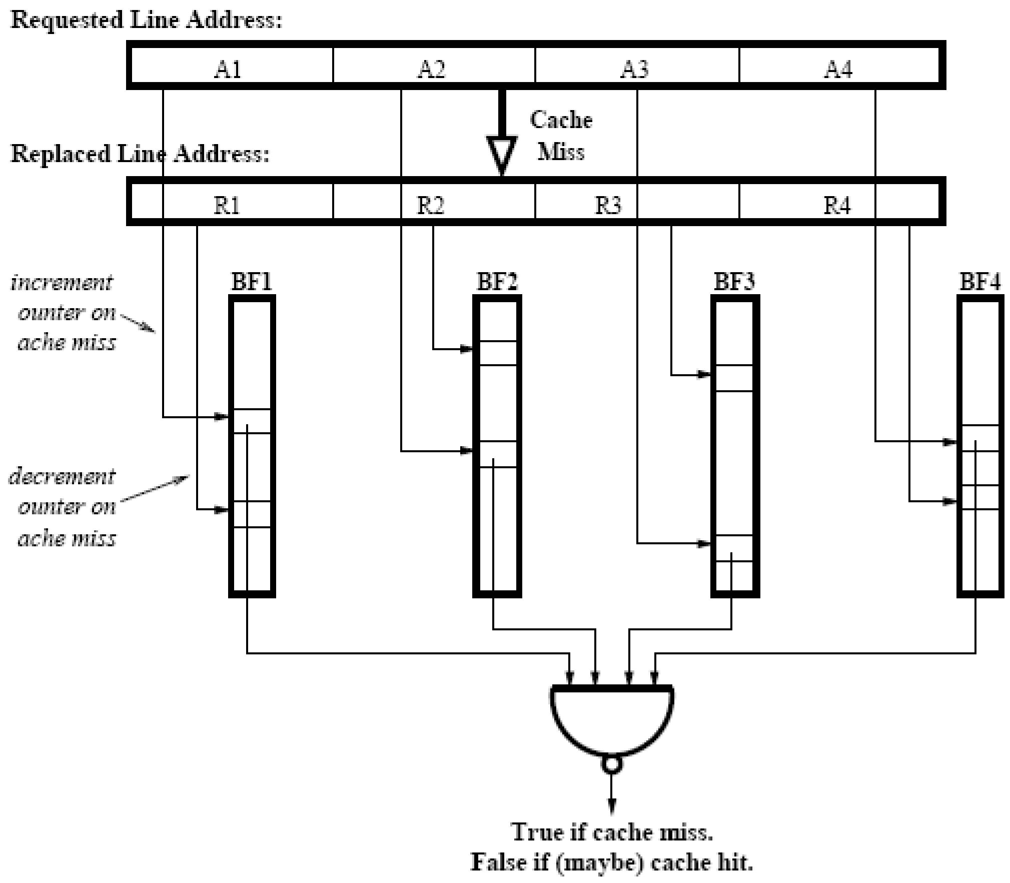


Figure 6: Partitioned-Address Filter for Cache Miss Detection

4.2.2 Partial-Address Filter

The partial-address filter uses the least-significant bits of the line address to index a small array of bits. Each bit indicates whether the partial address matches any corresponding partial address of a line in the cache. The array size is reduced to 2p bits, where p is the number of partial address bits. A filter error occurs when the partial address of the requested line matches the partial address of an existing cache line, but the other portion of the line address does not match. We call such cases collisions. The least-significant bits are selected rather than most significant bits to reduce the chance of collisions. Due to memory reference locality, the most significant line address bits tend to change less frequently. So low order partial address bits that used to represent cache line addresses make collisions to be rare. The design of a partial-address filter is illustrated in Figure 7. A filter array with 2p bits indicates whether the corresponding partial address matches that of any cache line. The filter array is updated to reflect any cache content change. When a cache miss occurs, the entry in the BF array for the replaced line is reset to indicate that the line with that partial address is no longer in the cache. Then, the entry for the requested line is set to indicate that a line with that partial address now exists in the cache.

If the partial address is wider than the cache index, when two cache lines share the same partial address, they must be in the same set in a set-associative cache. The filter array indicates which partial addresses exist in the cache, so if one of these lines is replaced, the filter entry for the replaced line should not be reset, since the partial address still exists for the line that was not replaced. When a cache line is replaced, the collision detector checks the remaining cache lines in the same set as the replaced line to see if any of them have the same partial address as the replaced line. If any do have the same partial address, the filter entry is not reset. Otherwise, the entry is reset. The collision detection is done in parallel with the cache hit/miss detection. The filter array is updated on the detection of a cache miss.

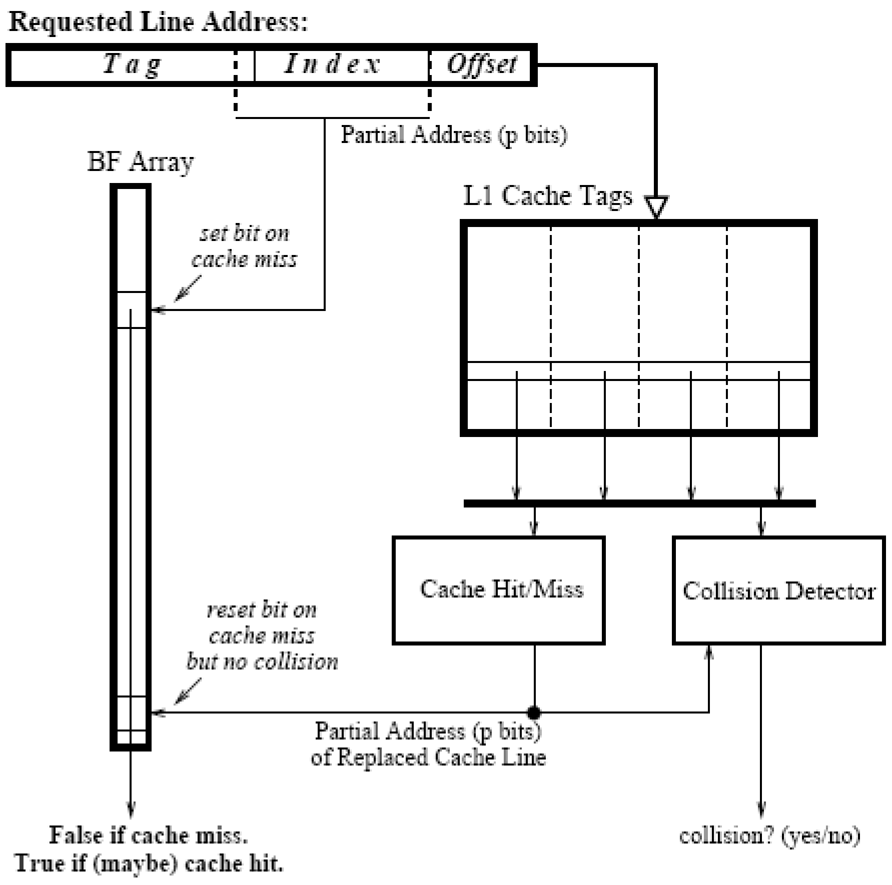
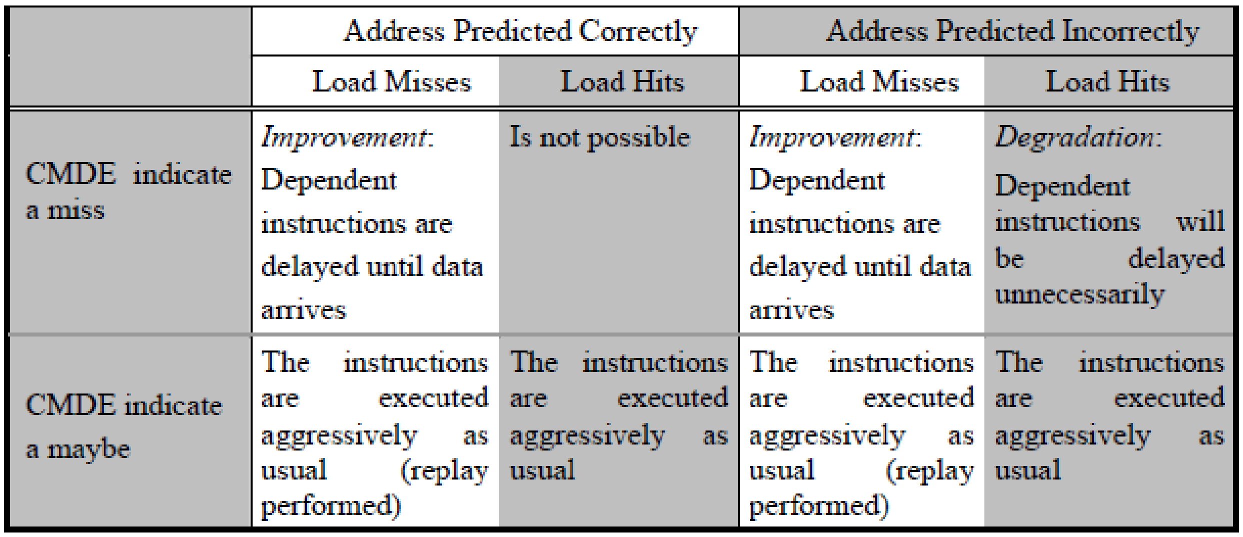


Figure 7: Partial-Address Filter for Cache Miss Detection

4.2.3 Hybrid CMDE

The previous CMDE techniques 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 previous 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
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
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

5. 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 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).

6. 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.

7. Conclusion

Memory latency, the time it takes for memory to respond to requests, can have a significant impact on performance. Memory dependence speculation aims at exposing the parallelism that is hindered by ambiguous memory dependences. Under memory dependence speculation, we do not delay executing a load until all its ambiguous dependences are resolved. Instead, we guess whether the load has any true dependences.

On the other hand CMDE is one of many intelligent speculative scheduling mechanisms aim to predict the time it takes memory to respond to load requests. With a reasonably sized CDME filter, we can correctly predict 99% of all misses [4].

References

[1] Gokhan Memik, Glenn Reinman, and William H. Mangione-Smith. Precise instruction Scheduling. Journal of Instruction-Level Parallelism 7 (2005) 1-29April.2005.

[2] 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] 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.

[4] 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.

[5] 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.

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

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

[8] Fernando Castro, IEEE Transactions on computers. Replacing Associative load Queues, April 2009

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