Onur Mutlu
10/9/2001

Memory Dependence Prediction and Access Ordering for Memory Disambiguation and Renaming

1. Introduction

The objective of modern superscalar processors is to maximize the instruction-level parallelism (ILP) that can be extracted from programs. The most basic method used for extracting more ILP from programs is out-of-order execution [1]. Unfortunately, out-of-order execution by itself does not provide a desired level of ILP. The program's control flow [2] and data flow [3] impose serious limits on the level of parallelism that can be extracted. Therefore, most modern processors employ aggressive branch prediction mechanisms to relax the control flow constraints that limit the ILP. To overcome the data-flow limits, researchers have suggested the use of data speculation [3,4,5], but these schemes have not yet been implemented in superscalar processors.

In this paper, we would like to survey a number of published methods for increasing the ILP by relaxing the constraints imposed by memory dependences. In particular, we will survey methods for dynamic memory disambiguation and memory renaming. In Section 2, we will discuss the importance of relaxing the memory data-flow constraints and discuss the distinctions and similarities between memory disambiguation and memory renaming. Section 3 will present a survey of recent research articles published on both topics and will discuss the advantages and shortcomings of each approach. Section 4 will discuss the differences between those mechanisms and comment on their effectiveness. In Section 5, we will describe how some modern processors tackle the memory dependence problem.

2. The Memory Dependence Problem

Modern processors exploit ILP by executing instructions in an order that is different from the sequential program order, which is called out-of-order execution. In other words, independent instructions whose operands are ready can be scheduled and executed before older instructions that are still waiting for their operands. Hence, to support out-of-order execution, the hardware needs to be able to precisely determine the dependencies among instructions so that sequential program semantics will not be violated. In case of register dependencies, determining which instructions are dependent is easy due to the explicit encoding of architectural register names (numbers) in the instruction format. Memory dependencies are much harder to determine, because memory addresses are not explicitly encoded in the instruction format and need to be dynamically generated. However, this dynamic generation of memory addresses is not done in sequential program order. Hence, when a load instruction is ready to be scheduled, it is likely that there are older store instructions in the instruction window whose addresses have not yet been determined[1]. This problem is known as the "unknown address problem" [6]. A related concept, the process of determining if two memory instructions access the same memory location is called *memory disambiguation*.

[1] Here, we would like to point out that the problem does not exist with the scheduling of ready store instructions, because they are completed in sequential program order to facilitate easier recovery from control-flow mispredictions.

2.1 Non-speculative memory operation scheduling

There are several ways to attack the "unknown address" problem. One possible solution is to execute all stores and loads in the total program order. Considering that load and store instructions comprise a large fraction of instructions in most programs, imposing a total order on memory reference instructions would seriously limit the ILP that can be extracted from programs. A slightly less conservative approach is to delay the scheduling of a load until all previous store addresses become available. This approach limits the amount of ILP extracted from memory disambiguation, because it is unlikely that a load will conflict with many of the previous stores.

The ideal approach to non-speculative memory disambiguation is to delay a load only until a previous conflicting store operation's address and data become known (If there is no conflicting store, the load can be scheduled right away). In this case a load is denied execution if the address or data of the youngest conflicting store that is older than the load is unknown. Hence, the conflicting load needs to have both the store address and store data for memory disambiguation. The dependency matrix of HPS [6] can be used for such an approach. This matrix relates each memory operation to every other memory operation. Memory operations are assigned a unique row in the dependency matrix. When a store with an unknown address is encountered and it corresponds to row i of the dependency matrix, bits in column i are set to 1. When the address becomes known, the bits of column i are cleared to 0. A memory operation corresponding to row k of the dependency matrix is allowed to proceed only when no preceding store operation has unknown address. This means that no bits in parts of row k that correspond to older operations should be 1. This approach opens more opportunities to exploit parallelism compared to the "store queue" approach of IBM 360/91, which stalls a store and all younger memory operations in the issue stage if the store address is not known yet [7]. However, with a large instruction window, the implementation cost of such a dependency matrix could be very high.

The approaches described above were all non-speculative, meaning that using the above approaches, no load will load a wrong value into a register. Hence, no recovery action needs to be taken. The non-speculative nature of these schemes makes them unattractive if we want to maximize the opportunities to exploit more parallelism in programs. Hence, it makes sense to build predictors to predict whether a load will conflict with a previous store and make scheduling decisions based on the outcome of the predictors. The purpose of this survey is to examine several of these prediction mechanisms and determine their effectiveness with respect to two important issues: memory disambiguation and memory renaming.

2.2 Speculative memory operation scheduling

The aggressiveness of memory disambiguation using speculative scheduling also depends on how much more parallelism we want to exploit. A less aggressive approach is to predict whether a load will conflict with *any* older store in the instruction window. In this case, the load cannot be scheduled until *all* older stores execute[2]. Clearly, this is not the most aggressive approach, because the load may be unnecessarily delayed by having it to wait *all* older stores in the instruction window. A more aggressive scheme is to

[2] By "execute" we mean that the store instruction determines its address and data value.

predict that a load conflicts with *a pa rticular* earlier store, if any, and delay the scheduling of the load until *that particular* store executes. This requires the predictor be able to form load-store conflict pairs. However, a load instruction will not be unnecessarily delayed when the prediction is correct.

One extreme form of speculative memory operation scheduling is to *always* assume that the load that is to be scheduled will not conflict with any of the unknown store addresses. Hence, a load will always be scheduled regardless of the number of older stores with unknown addresses. This kind of extreme speculation is not the best performing technique due to the cost of recovery as a result of mispredictions. More intelligent predictors are needed to keep the misprediction rate low.

So far we have assumed that when a store instruction executes, it writes its data into a store buffer. A later load that is dependent on the store will access the store buffer and read the data from the store buffer. This is called *load forwarding* [8]. In order to be able to do load forwarding, a load needs to have its address calculated. A more aggressive form of memory dependence prediction/access mechanism, *memory renaming*, can enable the load instruction to retrieve its data before its effective address is calculated. In order to be able to perform memory renaming, the relationship between the load and the previous store instruction that generates the data the load needs to be identified [9]. The identified store-load pair can be associated with an identifier used to address the store data value, which bypasses the normal memory addressing mechanism. Hence, memory renaming assigns a new name (and location) to a memory address that is produced by a store and consumed by a later load. Through the use of this new name, the memory latency is hidden, even though the memory access for the load still needs to be completed for verification purposes. In a sense, *load forwarding* is a very naive form of memory renaming. However, by using the address of the memory location as the name of the new space allocated for the store data value, load forwarding requires that a load determine its address before it can be scheduled. Memory renaming, as proposed by Tyson and Austin [9], does not impose such a requirement and hence can hide memory latency of loads exposing more parallelism.

2.3 Outcomes of Memory Dependence Prediction

It is important to note that although memory disambiguation and memory renaming can use the same memory dependence predictor, the measure of success of the predictor is different for each case. To explain this better, we will use the notation of Yoaz, et. al. [10]. The memory dependence predictor has two possible outcomes for each load: The load can be predicted as conflicting with a previous store (PC: Predicted colliding) or the load can be predicted as non-conflicting with any previous store (PNC: Predicted not colliding). The actual execution may show that the load was actually conflicting with a previous store (AC: Actually colliding) or it may turn out that the load was actually not conflicting with any older store (ANC: Actually not colliding). Hence, the speculation space of the predictor is divided into four: A load can be PC-AC (Predicted colliding, actually colliding), PC-ANC (Predicted colliding, actually not colliding), PNC-AC (Predicted not colliding, actually colliding), or PNC-ANC (Predicted not colliding, actually not colliding). First and last cases correspond to correct predictions. Second and third cases (PC-ANC and PNC-AC) correspond to the misprediction cases and they deserve more attention.

In case of memory disambiguation, PC-ANC case is undesirable but not very harmful. The cost of the PC-ANC misprediction is a lost opportunity in increasing parallelism. In other words, a load will be delayed unnecessarily, but no recovery action needs to be taken. However, PNC-AC misprediction is extremely undesirable, because in that case load will be supplied with the wrong data value and, in the best case, load and all of its dependent instructions need to be re-executed. As this recovery and re-execution is usually costly, we would like to avoid PNC-AC mispredictions for memory disambiguation.

The argument goes the opposite way for memory renaming. If a load suffers a PC-ANC misprediction, the load will be predicted conflicting with a wrong store and will be supplied a wrong value through renaming. Hence, the load and all its dependent instructions need to be recovered and re-executed. On the other hand, a PNC-AC misprediction is not as costly, because the load will not get the wrong value through renaming. Only an opportunity for renaming will be lost.

3. Review of Some Memory Dependence Prediction Mechanisms

In this section, we will review several mechanisms proposed for memory dependence prediction. For the most part, we will follow a chronological order. We will first start with the Address Resolution Buffer (ARB) proposed by Franklin and Sohi [11].

3.1. Address Resolution Buffer

Franklin and Sohi, in [11], recognize that the dependency matrix of HPS [6] and the store queue of IBM 360/91 [7] have two drawbacks:

1. They do not provide the full *speculative* flexibility to the reference reordering process.

2. They require very wide associative searches in the disambiguation step (These searches are costly in terms of delay and hardware, especially in high frequency systems.)

The basic ARB directs memory references into bins based on their address, and the bins are used to enforce a temporal order amongst references to the same address. The ARB is a banked structure. Each bank of the ARB contains a number of rows, let’s say k. This means that each bank can hold k addresses to which a memory operation is pending in the current instruction window. The banking of the ARB interleaves the addresses among the banks. Hence, multiple disambiguation requests can be dispatched in one cycle, provided that they are all to different banks. Besides, the associativity of the search is reduced because an address needs to be compared only with the addresses in the matching bank. Hence, the ARB does reduce the associative search required by the dependency matrix and the store queue.

A figure of the 4-bank, 6-stage ARB is displayed in Figure 1. Each row in a bank corresponds to a memory address. In Figure 1, the top row in Bank 0 corresponds to memory address 2000. The rest of the entries in the row (other than the address) show the pending operations on that address. Each stage corresponds to a sequence number. The active sequence numbers are delineated by the head and tail pointers of the ARB. In Figure 1, sequence number 1 is the oldest instruction in the machine and corresponds to stage 1. We see that the instruction with sequence number two (stage 2) is a load to address 2000. Also, instruction with sequence number 3 (stage 3) is a non-committed store to address 2001 with a data value of 10. When a *load* with sequence number i is executed, first, the ARB bank is determined using the load address. Then an associative search within the bank is performed to see if an earlier store is executed to the same address in the active ARB window. If so, the store with the closest sequence number is determined and the value of the store is forwarded to the load. If no preceding store to the same address exists in the ARB, the load is speculatively sent to the data cache. If the load address is not present in the ARB, a new row is allocated for the address and the load is entered into the appropriate spot. This is necessary to be able to initiate recovery if an older store later writes to the same address. When a *store with* sequence number j is executed, again the ARB bank is determined first. If no row exists for the address, a new one is allocated. The store bit of stage j of the ARB row entry is set to 1 and the value to be stored is recorded in stage j. If the store address was already in the ARB, the row is searched to see if there is a younger load that has executed without any intervening stores in-between. If that is the case recovery action is initiated. All instructions including and after the incorrect store are squashed. Tail pointer is moved back to point to the sequence number before the incorrect load. Hence, this scheme resembles a reorder buffer for memory instructions. It is worthwhile to note that a store can also be entered to ARB before its data value is available. This decreases the probability of loads getting the incorrect value from the data cache.

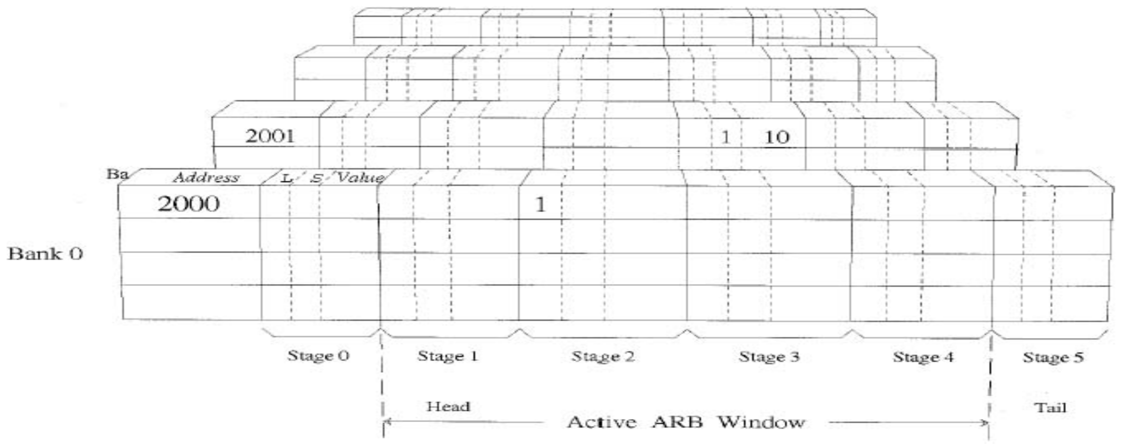

Figure 1. A 4-way interleaved, 6-stage ARB as shown in [11].

This brief description of the ARB shows that it can support speculative loads and load forwarding. However, the speculation mechanism of the ARB is not based on a predictor. Rather, a load is speculatively sent to the cache if an older store entry happens to not have executed yet. Hence, the misprediction rate of the ARB might be high for some benchmarks. Franklin and Sohi show that ARB outperforms the dependency matrix. The most attractive feature of the ARB is perhaps the reduction of the associative search in the memory disambiguation and load forwarding process. Such a reduction may be able to prevent the load forwarding and disambiguation process from becoming the bottleneck in a processor with a very large instruction window.

3.2. Moshovos' Work on Data Dependence Speculation

Andreas Moshovos was one of the first to extensively publish on the dependence behavior of loads and stores and techniques to predict and bypass these dependences. In [5], he proposes a technique that attempts:

1. to predict those instructions whose execution will violate a true data dependence
2. to delay the execution of those instructions as long as it is necessary to avoid the mis-speculation.

One of the most important observations of [5] is the fact that static store-load instruction pairs that cause most of the dynamic data mis-speculations are relatively few and exhibit temporal locality. This observation suggests that store-load dependences can be predicted based on past history. Hence, the store-load pairs can be allocated in a cache-like structure. When one of the instructions in the pair is later encountered, some steps need to be taken to synchronize the two instructions. This synchronization can be performed by the use of a hypothetical condition variable. When a store-load conflict is detected first, the store-load association can be assigned a condition variable, which is set to false. In the next instance of the load, when the load is ready to be scheduled, it first checks the condition variable of the association. Seeing that the condition variable is false, the load *wait*s. Once the next instance of the store is executed it *signals* the condition variable by setting it to true. The load can now be scheduled and after it is scheduled it needs to set the condition variable back to false. Hence, synchronization is achieved between the store-load pair by means of simple synchronization structures.

Another important observation in [5] is the notion that the *path* followed to execute a load instruction might affect whether or not it is dependent on an older store. Hence, a dependence predictor might perform better if it includes path information. The *dependence distance* (the difference in the instance numbers of the instructions which are dependent) can be used to distinguish different paths taken to execute a load. This is especially important in loop-based dependences. Depending on the loop distance only some of the dynamic instances of a static load will be dependent on the dynamic instances of a store. Incorporating the dependence distance information to the prediction mechanism thus could save a lot of false dependences.

3.2.1. Implementation Aspects of [5]

[5] proposes two tables to implement the suggested prediction/synchronization mechanism. The first table is the *memory dependence prediction table* (MDPT), which identifies a static dependence and provides a prediction as to whether the next dynamic occurrence of the store-load pair needs to be synchronized. Each entry of the table contains the load instruction address, store instruction address, and the dependence distance. Also an optional predictor, such as two-bit counters can be incorporated in the entry. One interesting question is whether an entry should always predict that the store-load association requires synchronization. If the association is stable over time, having a sticky predictor (which always predicts that the pair will conflict) would be a good option. However, if this is not the case, then using a 2-bit counter scheme to provide hysteresis would be desirable.

The second table is the *memory dependence status table* (MDST), which tracks the status of current associations that are predicted as conflicting by the MDPT. An entry of the MDPT supplies a condition variable to be used for synchronization. This status entry is used to coordinate the synchronization of the load and the store instructions that are in the instruction window. When the MDPT predicts that a store-load pair will conflict it allocates an MDST entry for the pair and initializes the condition variable. The hardware structures are updated based on the explanation of the synchronization above and the synchronization is satisfied.

It is important to note that for an MDPT entry to be allocated, a static load-store pair should incorrectly execute out-of-order. Hence, the proposed predictor learns from the past behavior of the pairs and bases its prediction on this past behavior. The memory dependence predictors we will examine later will all utilize history-based learning.

3.2.2. Evaluation of the Prediction Scheme of [5]

In [5], different data dependence speculation policies are simulated on a multiscalar [12] processor simulator. The simulated policies are NEVER (Loads are never speculated), ALWAYS (Blind speculation), WAIT (Loads with true dependences wait for all previous stores to generate their addresses), and PSYNC (Perfect memory disambiguation: Loads with no memory dependences execute as soon as possible, loads with true dependences are synchronized with the corresponding stores). It is shown that ALWAYS scheme results in a 30% average speedup over the NEVER scheme on 5 SPEC92 integer benchmarks (compress, espresso, gcc, sc, xlisp). It is also shown that ALWAYS scheme sometimes performs better than WAIT scheme but sometimes performs significantly worse due to the mis-speculation recovery penalties. The instruction window size also significantly affects the effectiveness of any scheme; as the instruction window gets larger, the effectiveness of dependence speculation increases.

The prediction scheme described in the paper is compared to the ALWAYS scheme. The proposed scheme achieves a speedup of around 10% for the five benchmarks.

3.3. Moshovos' Work on Memory Renaming and Bypassing

In [13], Moshovos and Sohi extend their work to bypass the memory on load accesses that are dependent on stores. They view the memory as an "inter-operation communication agent". This means that memory stores the value generated by an operation and that value will be sourced by a later operation. This dependency is implicit in the addressing modes. [13] tries to makes this dependency explicit by handling the memory communication in a separate name space other than memory address space. They use memory dependence prediction to generate this dynamic name space through which the dependent loads and stores can communicate without incurring the overhead of address calculation, memory disambiguation, and data cache access. Hence, they propose an aggressive way of doing *memory renaming*, which they call *speculative memory cloaking*.

3.3.1. Speculative Memory Cloaking

It deserves some attention to explain the high-level mechanism of memory cloaking, because this approach speculatively exposes a high level of parallelism by possibly providing the load with its data value in the very early stages of the pipeline. The first step in cloaking is to build an association between a store-load pair. This is done through the use of a *dependency detection* table (DDT). Once a load instruction generates a conflicting address with a previous store instruction in the DDT, a tag for the association is created and the dependent pair is stored in the *dependency prediction and naming table* (DPNT) along with the tag. When a later instance of the store instruction is brought to the processor, the DPNT is accessed and an association with the load will be found. At this point, a synonym is generated for the linkage between the two instructions and space is allocated in the *synonym file* (SF) and a pointer to this space is recorded in the DPNT. This synonym serves as a new name for the memory location that is accessed by the store-load pair. The space allocated in the SF is used to hold the data value produced by the store instruction. Initially, no valid value exists in this space, but when the store produces its value, the value is written into the allocated space in the SF. Note that the store does not need to have computed its address to be able to write its data value to the synonym file. Hence, the store instruction is essentially broken into two pieces: store address and store data (We will see later that a similar approach is taken by Pentium Pro). The traditional memory access that is required to verify the correctness of the cloaking needs the store address and the store data whereas the mechanism used for cloaking only requires the store data.

When the load associated with the store is brought into the instruction window, it accesses the DPNT and sees that space is allocated in the SF for its association with the store. Thus, the load accesses the synonym file and obtains the data value of the store if it is already computed. The instructions dependent on the load can therefore start executing speculatively using that value. When the load computes its address, the traditional memory system is accessed to obtain the real data of the load. This data is compared with the data value supplied by the SF. If they are the same, cloaking was successful. If not, recovery action needs to be taken to purge and re-execute the load and its dependent instructions.

3.3.2. Prediction of Memory Dependences in [13]

As mentioned in Section 2.3, the predictor for memory cloaking (a form of renaming) needs to minimize the PC-ANC mispredictions to minimize the cost of recovery actions. Also, it is necessary to predict exactly with which store instruction the incoming load collides. In [13], the latter requirement is accomplished by linking the dependent store and load using a newly allocated tag for the association. This scheme assigns a common tag to all dependences that have common producers (stores) or consumers (loads) and use that tag to identify all these dependences collectively. For example, if a load conflicts with multiple different stores based on the control flow path taken[3], say store$_1$ and store$_2$, both store$_1$-load and store$_2$-load association will be assigned the same tag (A similar but more flexible approach is also taken by the store set predictor [14], which we will examine later). The correct association will be enforced based on which store is present in the instruction window. Of course, there is a slight problem if multiple instances of the same dependence (association) are in the instruction window at the same time. But this problem can be easily solved by creating different synonyms for different instances.

[3] In code: *if (condition) then* store$_1$ *A; else* store$_2$ *A; load A* the load depends on both store$_1$ and store$_2$ but only with one of those for a given control flow path.

This prediction scheme yields a PC-ANC misprediction rate of around 2% on SPEC95 Integer benchmarks (compiled for MIPS ISA) with reasonable hardware resources (2K DPNT entries) and for a 256-instruction window. It is important to point out that, as instruction window size increases and pipelines get deeper and deeper maintaining or improving the PC-ANC misprediction rate is crucial, because of the increasing mis-speculation recovery penalty.

3.3.3. More Parallelism: Memory Bypassing

A more aggressive form of memory dependence speculation can be employed by observing that memory is a communication agent between arithmetic instructions in load-store architectures. Hence, DEF-store-load-USE dependency chains that typically exist in these architectures can be sped up by converting them to DEF-USE chains (hence bypassing the memory access). This can only be done when the store-load dependence is predicted and when the DEF and USE instructions simultaneously exist in the instruction window. Although, this is a promising way to extract more parallelism, we will not go into details of this mechanism for the purposes of this paper except for noting that the key to effective bypassing is extremely accurate dependence prediction.

3.4. Tyson and Austin's Memory Renaming Scheme

At the same time with Moshovos [13], Tyson and Austin also published a scheme to implement memory renaming [9]. Our discussion will not be as extensive here, due to the similarities between [13] and [9]. The essential idea is very similar: Assign a common tag to store-load associations and access a *value file* (very similar to synonym file in [13]) in case of the recurrence of the association and forward the store data from the value file to a load that is in the very early stages of the pipeline. One distinction between the two schemes is the fact that store instructions do not write into the value file unless they are committed in [9]. Whereas, in store file [13], store data is written as soon as it becomes available. This probably reduces the amount of parallelism exposed by [9] compared to [13]. However, the implementation of [13] would be more complex because it requires detecting when the store data becomes available (separate from the store address availability).

The initial binding of stores to loads is also done differently in [9]. When a store forwards a value to a later load, an association is formed between the store and the load in what is called the *store/load cache* and a tag is assigned for the association to index the *value file.* Hence, no separate structure (e.g. DDT in [13]) is used to detect dependences. The advantage of this is the reduced hardware cost. The disadvantage is that only store-load dependences *within the instruction window* are detected and hence can be renamed. This disadvantage might be critical for obtaining higher performance, since, as shown in [13], most store-load dependences are distant, which means that the dependence is impossible to detect in a single instruction window because the associated store and load are never coexistent in the same instruction window.

Tyson and Austin's paper also discusses two different recovery mechanisms in case of mis-speculation and hence deserves a little more attention. One recovery scheme is what they call *squash recovery* in which all instructions including and succeeding the mis-speculated load are squashed and re-fetched. A higher-performance recovery scheme is to only squash the dependent instructions and not squash the independent instructions, called *re-execution recovery*. The implementation of this scheme is not presented, but it might be quite complex. However, it certainly reduces the cost of mis-speculation. Using this recovery scheme they report that their scheme achieves a speedup between 1% to 42% for 10 benchmarks from SPEC95 suite.

3.5. Two Other Memory Dependence Predictors

This section concludes our survey of dependence prediction mechanisms with the discussion of two relatively simple yet powerful predictors proposed in [14] and [10].

3.5.1. The Store-set Predictor

This predictor [14] is proposed for memory disambiguation. The aim of the authors is to be able to schedule load instructions as soon as possible without causing any memory order violations. The predictor proposed is based on *store-sets*. A store set for a specific load is the set of all stores upon which the load has ever depended. The processor adds a store to the store set of the load if a memory order violation is caused when the load executes before that store. In the next instance of the load instruction, the store set is accessed to determine which stores the load will need to wait for before executing.

One important observation is that multiple loads can also depend on the same store. Hence, the same store can exist in the store-sets of different loads. [14] shows that the predictor needs to have this flexibility of a store existing in multiple different store sets in order to achieve high correct prediction rates. Hence, they describe a store-set *merging* predictor implementation.

The store set predictor consists of two tables. The first one is the *store set ID table* (SSIT), which connects the store-load associations. The second table is the *last fetched store table* (LFST), which keeps a track of the store currently in the instruction window for a particular ID. When a memory violation occurs, the SSIT might already have an entry for the load or the store. If only one of the instructions already have an ID, the other instruction is assigned that same ID. If neither the load nor the store already has an ID, a new ID is allocated and written into the SSIT in the following way: The SSIT is indexed using both load and store instruction's program counters and the newly-allocated tag is written into those locations in the SSIT. Hence, the linkage between the load and store is formed through the SSIT. If both the load and the store have ID's in the SSIT, one of the ID's (smaller one) is declared the winner and both the load and store are assigned the same ID. This operation effectively merges the store sets of two different loads.

A diagram of the predictor is given in Figure 2. When a load is fetched, it accesses the SSIT and gets its store set ID. Using this store set ID, it accesses the LFST and gets the sequence number of the most recently fetched store in its store set. The load should not be ordered to execute before that store.

When a store is fetched it accesses the SSIT. If it finds a valid store-set ID, it first accesses the LFST and gets the most recently fetched store instruction in the store-set. The new store becomes dependent on the store in the LFST. This ensures that the stores are executed in the correct order. The fetched store also updates the LFST by inserting its sequence number to the appropriate entry. After the store instruction issues, it accesses the LFST and invalidates the entry if the entry still refers to itself.

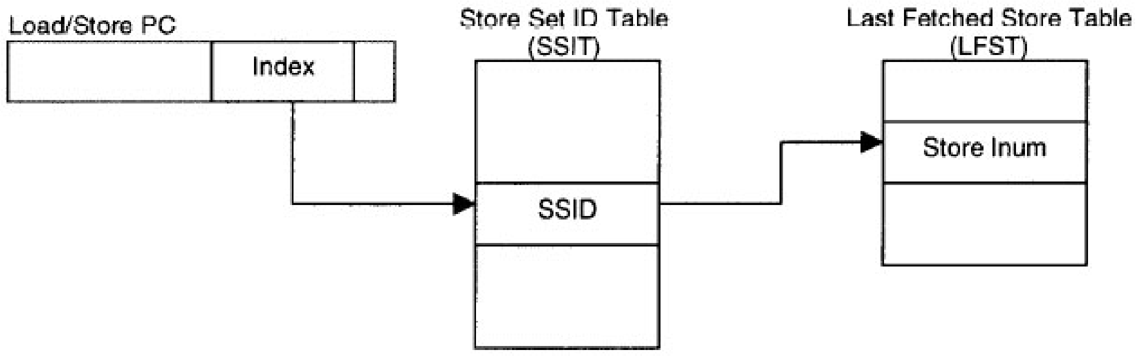

Figure 2. Implementation of the store-set predictor as shown in [14].

The elegance of the store-set predictor comes from its simplicity and effectiveness. Chrysos and Emer report that this mechanism, with reasonable hardware resources, achieves very close to the performance provided by a perfect memory disambiguation mechanism, where perfect means loads are scheduled as soon as the stores they are dependent on have finished execution and no memory order violations or false dependencies exist. They also report that blind speculation performs worse than no speculation (loads wait until all previous store addresses are known) for some benchmarks and the IPC difference for perfect memory disambiguation and no speculation is drastic (up to 300% performance improvement for some benchmarks).

3.5.2. Collision History Table (CHT) Predictor

The last predictor we will describe is the CHT predictor proposed by Yoaz et. al. [10]. As this predictor has many variants, we will only describe the basic idea without going into a lot of detail. The proposed *Full CHT* predictor only provides a prediction as to whether a load instruction will conflict with *any* store within the instruction window. It does not predict *which* store instruction the load will conflict with. Hence, it is easier to design but it does not provide the best possible information for disambiguation purposes.

However, predicting only whether a load will conflict simplifies the predictor drastically. No associations of stores and loads need be kept. Besides, the predictor can store some "dependence distance" information about the dependences of a load instruction and make a worst case estimate as to how far in up in the scheduling window the load instruction can be moved so that it does not collide with an older store.

An entry in the CHT consists of a tag (part of the load program counter), an n-bit counter that generates the prediction, and an optional distance field, which tells how many instructions in the scheduling window the load can be moved up. All loads are inserted into this table when they are decoded. When a mis-speculation occurs for a load, the counter associated with the load is incremented. When a false dependence prediction is made for the load, the counter for that load is decremented. Hence, the CHT is very much like the current branch predictors.

[10] reports that a variation of this CHT mechanism is able to capture most of the benefit that can be gained from perfect memory disambiguation.

4. Comments on the Examined Prediction Mechanisms

We have surveyed six different mechanisms that can be used to attack the memory disambiguation problem and increase parallelism using memory renaming[4]. Each scheme has its advantages and disadvantages. Based on the surveyed papers, it is impossible to determine which scheme will work best on a given processor configuration. As suggested in Section 2.3, some predictors would be good for memory disambiguation and some would be good for memory renaming. Hence, the choice of the predictor depends on how aggressive we would like a superscalar processor handle the *memory dataflow*. Here, we would like to classify the discussed mechanisms and briefly discuss their limitations.

ARB is a mechanism for handling memory disambiguation. It does not provide enough support for memory renaming except for load forwarding. The biggest advantage of ARB is its ability to reduce the very-wide associative search in order to determine whether a load conflicts with a store. None of the other disambiguation schemes address this issue. The store-set predictor and the CHT predictor need to be supplemented with an ARB-like structure to be effective. As mentioned before, the store-set predictor and the CHT predictor are solely memory dependence predictors. The entry of loads and stores in these predictors require the detection of store-load conflicts. This detection mechanism can either be a store queue, a dependency matrix, or the ARB as discussed in sections 2 and 3.1. If we think about the issue in another way, supplementing the ARB with a better prediction mechanism such as the store-set predictor or the CHT predictor also would improve the speculativeness and the accuracy of memory disambiguation.

To choose between the store-set predictor and the CHT predictor, we need to simulate different configurations of each predictor and choose the one that best fits our needs. This choice cannot be made based on the published papers.

For implementing memory renaming, Moshovos' cloaking mechanism and Tyson and Austin's memory renaming structures are attractive options. The choice of which one to implement, again depends on the available hardware resources and how aggressive we want the superscalar processor to get. Moshovos' memory bypassing mechanism is by far the most aggressive suggestion in that it tries to convert DEF-store-load-USE chains into DEF-USE chains. However, the proposed bypassing scheme is only limited to instances when both DEF and USE instructions are in the instruction window. A more aggressive mechanism would try to apply this transformation to instances where DEF and USE do not co-exist in the instruction window. Memory cloaking does not try to reduce the DEF-store-load-USE chains. Rather, it tries to service store-load dependences in a separate name space different from the memory name space. It has the advantage that the store and the load need not co-exist in the instruction window. Hence, it may be possible that a load gets its value from the synonym file, which was written into by a store 8K instructions ago. This flexibility of memory cloaking is its most important advantage over Tyson and Austin's memory renaming. One drawback to implementing memory cloaking is its complexity and possible impact on cycle time.

[4] We especially refrained from spending time to report the performance results presented in the papers, because the simulation environments, instruction set architectures, benchmarks, and microarchitecture designs on which the mechanisms are evaluated are extremely different. Hence, a comparison of the mechanisms based on the results presented in the surveyed papers would not be accurate.

We would like to conclude this section by noting that the success of an aggressive *memory dataflow engine* depends heavily on the correctness of the predictors. As pipelines get deeper and instruction windows get larger, the mis-speculation penalty will bear more impact on the overall performance of the processors. Hence, the discussed predictors may need to be improved to accommodate the needs of very deep pipelines.

5. Real Life: Memory Disambiguation in Some Commercial Processors

Some of current processors (HP-PA 8000 [5], Alpha 21264 [15]) allow speculative loads to be issued to the memory system to alleviate the unknown-address problem. However, the memory dependence prediction mechanisms are not sophisticated. None of the processors (to our knowledge) employ aggressive memory renaming.

5.1. Alpha 21264

The Alpha 21264 [15] performs blind speculation, meaning that if any of the earlier stores' address is not available when a load is ready to get scheduled, the scheduler will always predict that the load will not conflict with the store. Hence, the load is always sent out to the memory system in the presence of unknown store addresses. In Alpha there are two queues for memory operations: LDQ for loads and STQ for stores. These queues both have 32 entries. Both queues position instructions in their fetch order, although they enter the queue out of order when they issue. Loads exit the LDQ in fetch order after loads retire and the load data is returned. Stores exit the STQ in fetch order after they retire and write their data into the data cache. To detect mis-speculations, when a store issues into the STQ, the younger load addresses in the LDQ are associatively compared with the store's address. If a match is found, the LDQ squashes the matching load and all later loads and initiates recovery. The STQ also performs the function of load forwarding in Alpha 21264.

Once a mis-speculation is detected, the Alpha 21264 sets a bit in what is called a *load wait table* to signal that the mis-speculated load should not be executed out-of-order in the next execution. The implementation of this table is not explained in [15]. The load wait table is periodically cleared to avoid potential unnecessary waits. This cyclic clearing of prediction tables is also a concern with all of the previously described mechanisms. Chrysos and Emer [14] suggest periodic clearing of the SSIT. The reason for this is the fact that the program might have entered a different phase and the conflicting store load pair may not be conflicting any more. The CHT predictor [10] does not suffer from the problem as much because it incorporates n-bit counters to adjust the predictions. If the load instruction does not conflict with a store any more, the counters will pick up the new behavior and will not predict the load instruction as conflicting.

5.2. Pentium Pro

The Pentium Pro processor implements a memory order buffer to avoid memory order violations. Each load instruction is decoded into a single micro-operation (uop), whereas each store instruction is decoded into two uops. One is the STA (Store Address Calculation). The other is the STD (Store Data) [10]. Hence, the store is broken into two instructions. This has the advantage of breaking the dependencies of a store. Hence, the store data does not need to wait for the dependencies of the address calculation to be satisfied. Nor does the store address needs to wait for the dependencies for the store data to be satisfied, which is more important for performance. This means that the store address can be made available earlier than the data (at times), which will aid memory disambiguation. Memory disambiguation in P6 processor family follows two basic rules [10]:

1. A load cannot be dispatched if an earlier, unresolved STA uop exists in the instruction window. In Pentium Pro, the existence of such an instruction is easily determined by checking the memory order buffer.

2. A load cannot execute out of order with an older STD coupled with an STA that references the same address of the load.

The memory order buffer is used to prevent loads from being blocked by earlier stores. It is also used to avoid memory order violations. Stores are buffered in the MOB and they exit the MOB when they retire. Incoming load instructions are checked associatively with all previous store addresses in the MOB to determine if they conflict. Only non-conflicting loads can execute out of order with earlier stores.

MOB of the Pentium Pro also implements load forwarding. For this to occur, the following conditions should be met [16]:

1. The store is complete in the MOB.
2. The addresses referenced by the store and load have the same alignment.
3. The data requested by the load is a subset of the data written by the store.

If these conditions are not met, the load forwarding (which is called *store forwarding* by Intel) cannot be completed and the load has to wait until the store is retired. This is a significant performance penalty. Hence, unaligned load-store dependences should be avoided in code compiled for Pentium Pro.

6. Conclusion

In this paper, we have surveyed a number of approaches to memory dependence prediction and commented on their effectiveness in extracting more parallelism from programs. Then we examined the schemes employed by two of the current processors (Alpha 21264 and Pentium Pro) and saw that none of the proposed aggressive speculation techniques are employed by these processors. The importance of the memory dataflow would increase as the instruction windows get larger and memory system becomes more of a bottleneck and hence we would expect newer processors employing aggressive techniques such as memory renaming and bypassing to extract more ILP from programs. Hence, more research is needed in this area to improve the accuracy of the memory dependence predictors and to find out more aggressive ways to do memory operation reordering and renaming.

References:

[1] Tomasulo, R.M., “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” *IBM Journal of Research and Development,* Vol. 11, 1967, 25-33.

[2] Lam, M. S. and Wilson, R. P., “Limits of Control Flow on Parallelism,” *Proceedings of the 19th Annual Symposium on Computer Architecture*, May 1992.

[3] Lipasti, M. H. and Shen, J. P., “Exceeding the Dataflow limit via Value Speculation,” *Proceedings of the 29th Annual ACM/IEEE Symposium on Microarchitecture,* December 1996.

[4] Lipasti, M. H., Wilkerson, C. B., and Shen, J. P., “Value Locality and Load Value Prediction,” *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1996.

[5] Moshovos, A., Breach, S. E., Vijaykumar, T. N., and Sohi, G. S., “Dynamic Speculation and Synchronization of Data Dependences,” *Proceedings of the 24th Annual Symposium on Computer Architecture,* June 1997.

[6] Patt, Y. N., Melvin, S. W., Hwu, W. W., and Shebanow, M., “Critical Issues Regarding HPS, A High Performance Microarchitecture,” *Proceedings of the 18th Annual ACM/IEEE Workshop on Microprogramming,* December 1985.

[7] Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M., “The IBM System/360 Model Machine Philosophy and Instruction-Handling,” *IBM Journal of Research and Development,* Vol. 11, 1967, 8-24.

[8] Johnson, M., *Superscalar Microprocessor Design.* Englewood Cliffs, N.J.: Prentice Hall, 1991.

[9] Tyson, G. and Austin, T.M., “Improving the Accuracy and Performance of Memory Communication Through Renaming,” *Proceedings of the 30th Annual ACM/IEEE Symposium on Microarchitecture,* December 1997.

[10] Yoaz, A., Erez, M., Ronnen, R., and Jourdan, S., “Speculation Techniques for Improving Load Related Instruction Scheduling,” *Proceedings of the 26th Annual Symposium on Computer Architecture,* May 1999.

[11] Franklin, M. and Sohi, G. S., “ARB: A Hardware Mechanism for Dynamic Memory Disambiguation,” *IEEE Transactions on Computers,* 45(5):552-571, May 1996.

[12] Sohi, G. S., Breach, S. E., and Vijaykumar, T. N., “Multiscalar Processors,” *Proceedings of the 22nd Annual Symposium on Computer Architecture,* June 1995.

[13] Moshovos, A. and Sohi G. S., "Streamlining Inter-Operation Memory Communication via Data Dependence Prediction," *Proceedings of the 30th Annual ACM/IEEE Symposium on Microarchitecture*, December 1997.

[14] Chrysos, G. Z. and Emer, J. S., "Memory Dependence Prediction Using Store Sets," *Proceedings of the 25th Annual Symposium on Computer Architecture*, July 1998.

[15] Kessler, R. E., "The Alpha 21264 Processor", *IEEE Micro*, 19(2), pp. 24-36, March-April 1999.

[16] "Optimizations Corner: Cleaning Memory and Partial Register Stalls in Your Code", http://www.gamasutra.com/features/19991221/barad_pfv.htm.