Speculation Techniques for Improving Load Related Instruction Scheduling

Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan
Intel Corporation
Intel Israel (74) Ltd., BMD Architecture Dept., MS: IDC-3C
P.O. Box 1659, Haifa 31015, Israel
{adi.yoaz, mattan.erez, ronny.ronen, stephan.jourdan}@intel.com

Abstract

State of the art microprocessors achieve high performance by executing multiple instructions per cycle. In an out-of-order engine, the instruction scheduler is responsible for dispatching instructions to execution units based on dependencies, latencies, and resource availability. Most existing instruction schedulers are doing a less than optimal job of scheduling memory accesses and instructions dependent on them, for the following reasons:

- *Memory dependencies cannot be resolved prior to execution, so loads are not advanced ahead of preceding stores.*
- *The dynamic latencies of load instructions are unknown, so scheduling dependent instructions is based on either optimistic load-use delay (may cause re-scheduling and re-execution) or pessimistic delay (creating unnecessary delays).*
- *Memory pipelines are more expensive than other execution units, and as such, are a scarce resource. Currently, an increase in the memory execution bandwidth is usually achieved through multi-banked caches where bank conflicts limit efficiency.*

In this paper we present three techniques to address these scheduler limitations. One is to improve the scheduling of load instructions by using a simple memory disambiguation mechanism. The second is to improve the scheduling of load dependent instructions by employing a Data Cache Hit-Miss Predictor to predict the dynamic load latencies. And the third is to improve the efficiency of load scheduling in a multi-banked cache through Cache-Bank Prediction.

1. Introduction

One of the key elements in achieving higher performance in microprocessors is executing more instructions per cycle. Existing microprocessors are already capable of executing several instructions concurrently. However, dependencies among instructions, varying latencies of certain instructions, and execution resources constraints, limit this parallelism considerably. In particular, load instructions introduce three scheduling problems.

The first problem arises from the fact that addresses of memory operations are not known at schedule time. This prominent problem, termed *memory ambiguity*, prevents the scheduler from advancing loads ahead of preceding stores, since a load's dependency on a store is defined through their memory addresses. *Speculative memory disambiguation* is a technique that deals with this problem by predicting whether a load is dependent on a specific store prior to execution. In this paper we describe and test several simple, economic, yet effective disambiguation mechanisms and show that there is a large potential performance increase in employing them.

The second issue associated with loads is the scheduling of load dependent instructions. Efficient instruction scheduling algorithms rely on the fixed execution duration of each instruction. This is not the case for load instructions, since the accessed data may reside in any one of several memory hierarchies, having different latencies. We suggest *data cache hit-miss prediction* as a method of predicting a load's latency through a prediction of the data's hierarchical location.

Another major hurdle to increasing performance is the number of memory pipelines available. Currently, at most two pipelines are available, and that at great cost. The multi-banked cache has been suggested as a sub-ideal, yet economic, solution to this problem. We offer *cache-bank prediction* as a way of simplifying the pipeline and of avoiding the bank conflict problem (increasing efficiency) by scheduling only independent loads to execute simultaneously.

The rest of section 1 presents a more elaborate introduction and prior work related to these ideas. Section 2 explains the prediction mechanisms. Our simulation environment is described in section 3 and the results appear in section 4. Section 5 concludes the paper.

1.1 Memory Disambiguation

Current state-of-the-art processors are limited in performance due to true data dependencies. There are two types of data dependencies between instructions, register and memory dependencies. At schedule time true register dependencies are known, but memory dependencies are yet to be resolved (addresses are computed during execution). Memory dependencies occur when the addresses referenced by a load instruction and a preceding, in program order, store instruction are the same. Since the scheduler does not know this at

schedule time, a problem arises when trying to re-order a load in front of a preceding store. Memory reference ordering may follow the conservative (traditional) approach in which loads are dispatched as they appear in sequential program order with respect to stores, introducing false data dependencies. Such restrictions cause a significant loss of performance, since, in practice, most loads can bypass prior stores whereas in this scheme their execution is needlessly delayed. The other extreme is the opportunistic approach in which loads are dispatched as soon as possible assuming they are not dependent on stores. In this approach most of the load-store reordering cases result in improved performance, but the high penalty for erroneously advancing loads that truly depend on preceding stores offsets much of this performance gain. Incorrect memory ordering may incur a large performance penalty since the wrongly advanced load and all its dependent instructions must be re-executed or even re-scheduled.

The problem of not knowing whether a load should be deferred until earlier stores execute motivates us to try to predict memory dependencies. We present a family of simple, yet powerful, memory dependence predictors. The prediction information may be kept either within a dedicated hardware structure, or as a run-time hint along with the load op-code (in the instruction or trace cache).

Prior Work: Memory disambiguation through dependence prediction is an issue of active research. First let us describe how memory operation ordering is conducted in the P6 processor family [Inte96]. A load instruction is decoded into a single uop (micro-operation), whereas a store is decoded into two uops, a STA (Store Address) and a STD (Store Data). Breaking stores into two uops increases parallelism, since the store address is often calculated before the data to be stored is. Traditional memory disambiguation schemes (P6 processor family) follow two basic rules:

- A load can not be dispatched if an earlier, unresolved, STA uop exists in the instruction window.
- A load cannot execute out-of-order with an earlier STD coupled with a STA that references the same address as the load. The problem is that to detect this case the load must be dispatched without knowing whether it is truly dependent on the earlier STD.

The first rule is a pessimistic rule for memory disambiguation, since it assumes that all prior stores are likely to access the same address as subsequent loads. The second rule is required for correctness.

In the P6 processor family there is a way of retaining the instructions in the RSs, and re-dispatching in case of a collision. Newer implementations with no ability to stall would need to re-execute the load and its data dependent instructions until the STD is successfully completed.

Previous work on memory disambiguation was done in the context of compiler and hardware mechanisms for non-speculative disambiguation ensuring program correctness. These can be divided into four categories: static compiler techniques in which disambiguation is statically performed by the compiler; software only dynamic disambiguation, in which the compiler inserts specific disambiguation code into the command stream [Nico89][Huan94]; hardware assisted disambiguation where the compiler uses special op-codes which assist disambiguation and verification [Gall94]; hardware only dynamic disambiguation as proposed by Franklin and Sohi [Fran96].

More recently, work on speculative memory disambiguation has been done in industry and academia.

Academic work done by Moshovos and Sohi [Mosh97] advised a relatively complex memory dependence prediction technique for improving memory ordering and store value forwarding. The idea is to predict the load-store pairs and to use this information to either bypass loads beyond stores or to forward the store's data value to the matching load. To perform this load-store pairing, they use a set of fully-associative tables holding pairs that were wrongly reordered in the past. Similar work was done by Austin and Tyson [Aust97]. Moshovos and Sohi [Mosh97b] also proposed to improve the communication of data between the producer and consumer by eliminating or bypassing store-load pairs. A more thorough examination of such communication improvements is presented in [Jour98].

Chrysos and Emer proposed a simplification of the fully associative tables [Chry98]. Their mechanism uses two tables, one for associating loads and stores into sets and the other to track the use of these store sets. After receiving its set ID the load checks when the last store of that set was dispatched and executes appropriately.

In industry, Hesson *et al.* suggested the *Store Barrier Cache* [Hess95]. In this implementation each store that caused an ordering violation increments a saturating counter in the barrier cache. At fetch time of a store, the barrier cache is queried and if the counter is set all following loads are delayed until after the store is executed. If the store did not cause a violation the counter is decremented.

Our mechanism is in a sense similar to [Hess95] yet more refined, since it deals with specific loads, and to [Chry98] but much more cost effective. In its most basic configuration our proposed predictor stores only one bit per load, which may modify, or be appended to, its entry in the trace/instruction cache. In this way a significant performance increase can be achieved at almost no added storage.

1.2 Hit Miss Prediction

It is the scheduler's job to ensure that instructions are executed as soon as possible. Most instructions have a static latency, so instructions that are dependent on them can be scheduled precisely. Load instruction latencies, on the other hand, vary depending on the location of the data. If the data resides in the highest cache level, latency is short, whereas going to a lower hierarchy results in a longer delay. This complicates and reduces the efficiency of scheduling load dependent instructions, since the scheduler needs to schedule the dependant instructions before the actual delay is known. Therefore, predicting this latency allows for more efficient use of CPU resources, achieving higher performance.

A possible implementation may assume all loads hit the

higher level cache. This is a reasonable assumption since more than 95% of the dynamic loads are cache hits. Loads that actually miss the cache will cause instructions that depend on them to be re-scheduled and re-executed. Although this method is quite effective, re-scheduling and re-execution are still expensive in terms of performance, power, execution bandwidth, and especially in increased hardware complexity.

The new concept presented here is to predict which loads will miss the cache, thus delaying the dependent instructions until the needed data is fetched. This increases performance directly by saving a few clocks through the scheduling of load-dependent instruction to execute at the exact time the data is retrieved. More importantly, resource utilization is improved and execution bandwidth requirements are reduced. The reduction in execution bandwidth is particularly beneficial for multi-threaded architectures, where the temporarily available execution units can be used by other threads of execution.

Related Work: Mowry and Luk [Mowr97] used software profiling for load hit-miss behavior, in order to hide the latency of cache misses (e.g. dispatching these loads earlier).

Recently, Compaq, when presenting the Alpha-21264 processor (Hot Chips 10 [Kess98]), mentioned the existence of a hardware based hit-miss predictor.

Our hardware approach is based on a per-load binary prediction of a hit or miss in the cache, and employs adapted versions of well-known branch predictors. Branch predictors have received much attention in industry and academia. References to some of the predictors we tested can be found in works by Yeh and Pat [Yeh97], McFarling [Mcfa93], and Michaud and Seznec [Mich97].

1.3 Bank Prediction

In order to achieve high performance through the execution of multiple instructions in a single cycle, an implementation should allow for more than one concurrent memory reference. There are several ways to accomplish this goal. Ideally, the first-level cache should accommodate several accesses in a single cycle. A truly multi-ported cache fulfills this goal, but this solution increases latency and is quite complex and expensive in terms of area and power. Another well-known solution is a multi-banked cache. In this scheme, the cache is split into several independently-addressed banks, where each bank supports one access per cycle. This is a sub-ideal implementation since only accesses to different banks are possible concurrently. However, this reduces the complexity and cost of the cache, while still allowing more than one memory access in a single cycle. We suggest to minimize the effects of this restricted access by avoiding simultaneous execution of bank-conflicting loads. Bank conflicts may be avoided if the bank associated with each load instruction were known, then, the processor may schedule load instructions in such a way, so as to maximize the utilization of the banks. However, in current processors this is not done since the scheduling precedes the bank determination. We suggest a *bank predictor* to predict the bank associated with each load thus allowing more efficient use of the cache. Also, as will be explained later, a good bank predictor will allow the simplification of the multi-banked memory pipeline as well as other memory-related structures, reducing latencies.

Related Work: Several mechanisms have been suggested to approximate the benefits of a true multi-ported cache, without paying the power and area costs associated with it. These can be divided into four basic categories:

- Single ported cache enhancements such as in [Wils95].
- Cache duplication, in which several single ported copies of the cache are kept and accessed simultaneously [Digi97].
- Single ported caches run at twice the core speed, thus emulating a truly multi-ported cache [Weis94].
- Multi-banked cache designs such as [Simo95][Hunt95].

In these multi-banked designs a scheme was provided for minimizing the bank conflict problem. In this scheme load/store instructions undergo dual scheduling - after the load's address is calculated it enters a second level scheduler for accessing the banked cache, in this way the bank utilization is maximized. This scheme has been implemented, but it increases the load execution latency and is exceedingly more complex than our suggested design.

Recently, the Locality Based Interleaved Cache design has been proposed as an enhanced multi-banked cache that allows for scalability [Rive97].

To the best of our knowledge no previous work or proposals have been made for Bank Prediction and its repercussions on the memory subsystem described later.

2. Prediction Mechanisms

In this section we describe the prediction mechanisms proposed for memory disambiguation, hit-miss prediction, and bank prediction, as well as different design considerations.

The predictors vary greatly in the way they are employed, and are therefore affected differently by specific machine parameters.

2.1 Memory Dependence Prediction

The memory dependence predictor presented here is an accurate and implementable mechanism for performing speculative memory disambiguation. The basic idea is not to predict the load-store pairs, but rather just to predict whether a given load *collides* with any preceding, not yet executed, store in the scheduling window (*inclusive collision predictor*). If the load is predicted not to collide, it can be advanced prior to all stores in the scheduling window. This mechanism enables the bypassing of stores by certain loads and delaying colliding loads until the prior stores complete. Since it is essentially a binary predictor that keeps no complex pairing information and does not enable data forwarding, it is much simpler to implement and may accommodate more elaborate history information within a given area. Furthermore, since the predictor needs to predict the collision of a load with a non-specific store and not the exact load-store pair, its prediction is more accurate.

The enhanced version of our memory dependence predictor (*exclusive collision predictor*) is more refined, thus it is capable of creating more opportunities for advancing loads. This is done by annotating the minimal *distance* to the store that matches a certain load. This information enables a colliding load to bypass some (but not all) the stores in the scheduling window. This minimal distance may also provide a simple way of performing load-store pairing, enabling data value forwarding as well as better memory disambiguation. For each load, in addition to its collision history, the information of how far (measured by number of stores or instructions) a load can be safely advanced is kept. The prediction tables used to hold the required information will be described later. Even in this enhanced version, the predictor is much simpler than fully-associative tables holding load-store pairs.

When predicting collisions of loads, the misprediction penalty is not symmetric and depends on the direction of the misprediction. To clarify this we will first classify loads into two groups – *conflicting* and *non-conflicting* loads, and then into several categories (Figure 1). A load is considered to be conflicting if at schedule time (all source operands are ready and there is a free execution unit) there is a prior store with an unknown address. Colliding loads are conflicting loads for which the conflicting store's address matches that of the load. Conflicting loads are further divided into four categories according to their predicted and actual collision status. Our goal is to reduce the size of the two edge cases (wrong predictions). We pay special interest in decreasing the AC-PNC part: predicting an *Actually-Colliding* (AC) load as *Non-Colliding* (PNC) may cost a full re-execution penalty (in case the load was incorrectly advanced). On the other hand, predicting an *Actually-Non-Colliding* (ANC) load as colliding costs a lost opportunity to increase parallelism.

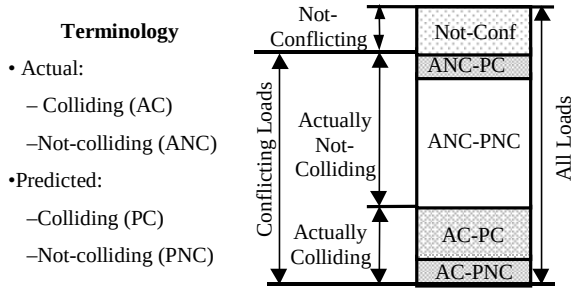

Figure 1 Load Classification Terminology

As explained later, in its simplest form our dependence predictor needs only a single bit to operate. The idea is based on a Collision History Table (CHT), which is organized as a cache, and can use various associativity, indexing, and replacement policies. It collects the history of load-store collisions to be used for subsequent predictions. A CHT may be very similar to a branch history table that predicts the outcome of a branch according to its previously seen behavior. The main idea is to mark loads as "bad," meaning they should be delayed (predicted as colliding). Our observation is that loads exhibit a recurrent collision behavior, therefore future collisions can be predicted based on history. Another option is to include the run-time disambiguation information along with the load instruction op-code (annotated in the instruction or trace cache) saving the area and complexity of separate tables. Storing disambiguation hints within the trace cache may also improve the disambiguation quality by allowing different behaviors for the same load instruction based on execution path. In this paper we explore several varieties of collision history tables. Each CHT entry may contain the following fields (Figure 2):

- Tag (optional): The tag consists of the linear instruction pointer of the load. The tag may be a full tag, partial tag, or omitted altogether. In the latter case, the table is treated as a tagless, direct mapped cache, indexed by a subset of the linear instruction pointer bits.
- Collision predictor: the predictor may be any binary predictor (colliding/non-colliding). In practice, however, a sticky bit (after its first collision, the load is always predicted as colliding) or a 1-bit saturating counter (the load may change its behavior from colliding to non-colliding) are enough.

  Note: a sticky predictor has several advantages here:
  - It is biased to mispredict on the safe side (reducing the number of AC-PNC loads).
  - The bit representing a "bad" load may actually be removed to provide a 0-bit predictor (tag only).
  - It eliminates the need to convert a colliding load into a non-colliding one. Such a conversion is not simple to perform precisely (as explained later), however, as suggested in [Chry98], the table may be cleared occasionally to provide for behavior changes.
- Collision distance (used in conjunction with the exclusive predictor): determines the dynamic distance between the load and the store it collides with. The load can be advanced no more than this distance. This number will eventually converge on the minimal allowable distance. This is useful for push/load parameter pairs (from different call sites) and register save/restore pairs. Note that different stores may be used for the same load.

A combination of some of these structures is also possible. A list of practical CHT structures follows:

- Full CHT: uses tags, a one bit (or more) counter for the predictor and, optionally, a collision distance. This structure contains colliding as well as non-colliding loads and is useful for maintaining additional load related information such as data prefetch or value prediction information. Various allocation and invalidation policies for an entry in the CHT can be implemented. For example, allocating a new entry only when a load collides for the first time and invalidating its entry when its state changes to non-colliding.
- Implicit predictor CHT: uses tags only and implicitly marks each entry as colliding (a collision distance is used for the exclusive predictor). Such a CHT contains only colliding loads. Being sticky, this predictor is good at reducing the number of actually-colliding loads predicted as non-colliding.

- Tagless CHT: uses 1-bit counters (or other binary predictors, e.g. bimodal) with no tags. collision distance information is optional. The CHT is directly mapped by a subset of the linear instruction pointer bits. Its small entry size allows for many entries, but it suffers from interference (aliasing).
- Combined Implicit-predictor and Tagless: uses the Implicit-predictor outcome when the tag matches and the Tagless result otherwise (predict a load as non-colliding only when there is no tag match in the Tag-only CHT and the Tagless state is non-colliding). This configuration tries to maximize the number of AC-PC. In a machine configuration where maximizing the number of ANC-PNC loads is more important, a prediction for colliding is given only when both tables predict a collision. This predictor allows many entries (Tagless) with less interference ("best of both worlds").

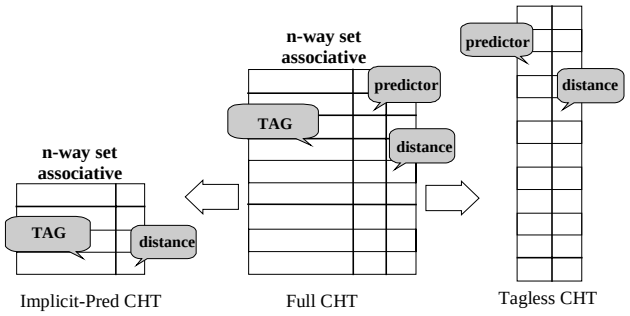

Figure 2 Variations of Collision History Tables

In addition to the CHT, the processor should keep information of actual load-store collisions. This information can be recorded in an existing MOB/ROB like structure (instruction **R**e**O**rder **B**uffer and **M**emory **O**rdering **B**uffer) which ensures correct program behavior and provides a forwarding mechanism for memory operations). The requirements are to identify for each executed store-address whether a future (in sequential order) load refers to the same address, and to determine for every to-be-executed load whether a preceding store refers to the same address. Out-of-order microprocessors today have such capabilities.

A description of the maintenance and use of the collision information follows. For the sake of simplicity we refer to the inclusive version using a sticky predictor.

1. When a load instruction appears in the instruction stream (prior to scheduling) the CHT is searched. If the load is found and its predicted state is colliding the load is marked as predicted-colliding and is treated as if it depends on all preceding store addresses. In all cases where the load is not found in the CHT or when it is predicted as non-colliding, it is treated as independent of all preceding stores in the scheduling window.
2. When a predicted non-colliding load has all its operands ready (and there is an available execution unit - AGU), it can be executed even before older stores. If the load is predicted as colliding it waits for all older stores (STAs and STDs) before executing.
3. When a store address becomes known, a check is made to see if there is any following (in program order) load that has already been executed with the same address. If found, that load is marked as actually-colliding; The load and all its dependent instructions are restarted.
4. When a load retires, the prediction is verified. If the load is predicted-non-colliding but was actually-colliding, an entry is allocated for that load in the CHT (if not there already).

Handling the exclusive version is a straightforward extension of the above description. Extending the scheme to a non-sticky predictor is more complex since it requires additional mechanisms for identifying whether a predicted-colliding load would have really collided or not and, in the latter case, converting predicted-colliding loads into non-colliding ones.

2.2 Hit-Miss Prediction

Once all instruction dependencies have been resolved, the scheduler's remaining task is to dispatch instructions in a way that maximizes the utilization of available resources, while minimizing instruction latencies. In order to do an optimal job on this task the execution latencies of all instructions must be known a-priori. This condition is met for most instructions, but not for loads. In modern processors the memory is constructed from several hierarchies which vary in their access time. Therefore, a load whose data is present in the fastest memory hierarchy will have a shorter latency. Today, most processors employ two levels of caches as well as relatively slow memory.

To facilitate the scheduling operation, we suggest dynamically predicting whether a specific load will hit or miss the cache (for the first level only or for all levels). Since the latencies for accessing each level are known, this will allow scheduling instructions that depend on this load to execute at the exact time the data will be available. Most processors today already use simplistic hit-miss prediction. Since most load operations result in a first level cache hit, this latency is assumed for all loads (always predict a cache hit). However, on a cache miss, significant performance may be lost. Predicting hit-miss for the first cache level is more important than predicting it for the other levels. Misses in lower levels will result in much longer stall periods reducing the relative penalty of re-execution.

To demonstrate this, consider the execution aspects of a deeply pipelined processor that continues current design trends. Such a processor will probably have an execution pipeline similar to the one in Figure 3.

In this machine a load which hits L1 will execute with a latency of 8 cycles after scheduling (2 cycles for register file access and bypass, one cycle for address calculation, and 5 to access the cache), whereas an L1 miss that is a hit in the L2 cache will take 15 cycles. The major problem with a deep pipe in case of a varying-latency instruction (e.g. L1 miss) is that several instructions may have already started execution by the time the actual latency is known. In this single-issue pipe, since hit/miss indication is received 5 cycles after the dependent instruction started scheduling, up to 5 instructions

may have started scheduling/execution. The dependent instruction cannot wait for the hit/miss indication since scheduling is not a zero-cycle operation, but may take several cycles. Therefore, even though a bypass network exists the scheduling must start several cycles earlier. All the dependent instructions, which have already been dispatched, will need to be re-scheduled and re-executed. This consumes extra execution resources and will probably delay program execution since the recovery process is not immediate.

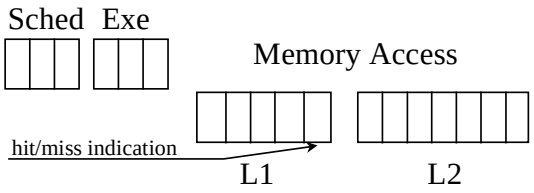

Figure 3 Pipeline Example

This problem is compounded by the fact that most processors are also super-scalar and that the miss indication may be received even later assuming partial address translation [Patt95], so more than 5 instructions may have already started execution.

If a hit-miss predictor (HMP) were made available to this processor, many of these lost cycles could be saved, resulting in faster execution due to an effective increase in execution resources, and a reduction in wasted cycles caused by re-scheduling.

Employing a HMP may also help in the struggle to reduce processor power consumption. This is achieved by reducing the number of re-executions, which may be considerable in applications that have a large number of L1 cache misses.

Another concept in computer architecture that may benefit from hit-miss prediction is *multi threading* [Tull95]. Here, the prediction may be used to govern a thread switch if a load is predicted to miss the L2 cache, and suffer the large latency of accessing main memory.

As explained above, since a load's latency is unknown during scheduling, the load dependent instructions are scheduled speculatively. During the execution stage of the dependent instruction, when the load data should be ready, the dependent instruction must verify that the data is actually available before continuing execution. Other instructions, later in the dependency chain, may have also started execution and will also perform verification. This complex scheme may be replaced with an accurate HMP. Now, the load's latency is "known" during scheduling, and on the rare event of a misprediction a simple flush replaces the complex re-schedule/re-execution recovery mechanisms, without sacrificing much performance.

As with memory dependence prediction the penalties associated with mispredictions are not symmetric. Here loads are also divided into four groups: Actual Hit - Predicted Hit (AH-PH), Actual Miss - Predicted Hit (AM-PH), Actual Hit - Predicted Miss (AH-PM) and Actual Miss - Predicted Miss (AM-PM).

Correct Predictions:

AH-PH loads are handled in the same way they are today and have no effect on performance.

AM-PM loads (misses that were caught by the predictor) have a positive effect on performance as described above.

Mispredictions:

AH-PM loads will result in a slight degradation in performance, since the dependent instruction may be dispatched only when the hit indication arrives (in the example above this instruction's execution will be delayed by 5 cycles).

AM-PH loads carry a high misprediction penalty. These loads are the misses that were not caught and will result in re-executions. However this is precisely the situation in current processors were all loads are predicted to hit the cache.

We evaluated HMPs with respect to L1 cache misses only. Since a hit-miss prediction is a binary prediction nearly all branch prediction techniques may be adapted to this task.

We tested many predictors and configuration but present only two predictors that require reasonable resources. Also, note that the results for these predictors are brought only to demonstrate the validity of the prediction technique and are far from optimal.

The first and simpler predictor is an adaptation of the well-known *local predictor*. Instead of recording the taken/not-taken history of each branch, we record the hit/miss history of each load. This paper does not address the whole spectrum of design consideration such as number of entries and history length. Instead we will present the results for a predictor with a tagless table of 2048 entries and a history length of 8 (~2KBytes in size).

As will be shown in the simulation results, this predictor suffers from a relatively high number of AH-PM mispredictions. Depending on the specific machine parameters, this may cause a large degradation in performance. To alleviate this problem we suggest a hybrid predictor. The components are a local predictor (512 entries) and two global predictors, a *gshare* (history length of 11 loads) and a *gskew* (each table has 1K entries, and the hash functions operate on a history of 20 loads). The chooser mechanism between the three predictor components is a simple majority vote (the total predictor size is less then 2KBytes).

To improve prediction performance, other information and predictor types may be employed. For example, *timing information* – using temporal proximity of loads (distance in cycles) to improve prediction. If a load misses the cache and a later load tries to accesses the same cache line before that line has arrived it will also miss the cache (*dynamic miss*). On the other hand, if the second load is executed after enough time has past for the first load to have been serviced, it will most likely be a hit. Most processors already have a structure that tracks dynamic misses (outstanding miss queue) and a small buffer for tracking serviced loads is a simple addition. To use these (or similar) structures, the *cache-line dependence* between loads should be predicted. This prediction can be done in several ways, for example:

- Dedicated cache-line dependence predictor such as a binary predictor to predict dependence with recent loads.
- An address predictor can be queried and the result used to check cache-line dependence.

Another way of making hit/miss predictions is by using an address predictor to directly check whether the data is in the cache or not. Unfortunately, this requires a tag lookup in the cache, and since the pressure on this resource is high for the L1 cache, this option is costly to implement. However, if we are interested in L2 hit/miss prediction, this may be a viable option. A simple mechanism for alleviating some of the pressure on tag lookup was suggested in [Pinte96], and may be used to enable this type of prediction for L1 also. If the load address is predicted correctly we can of course fetch the data ahead of time and not use it for hit-miss prediction only, but this requires different mechanisms and is beyond the scope of this paper.

2.3 Bank Prediction

Bank Prediction affects the processor in several ways. The most important use of bank prediction is to allow a multi-banked cache to closely approach the memory performance of a truly multi-ported cache. The major weakness of multi-banked designs is that an improperly ordered instruction stream leaves part of the memory bandwidth unused. This is because each bank can only service one access simultaneously. Therefore, bank conflicts reduce performance, and the extent of this effect is dependent on the CPU micro-architecture. With bank prediction, the bank information is available early in the pipeline before instruction scheduling. Thus, scheduling can be performed in a way that efficiently utilizes the multiple bank structure, i.e. memory operations predicted to access the same bank are not dispatched simultaneously.

Another possible use of a bank predictor is the simplification of the memory execution pipeline, as well as other memory related structures. By providing a prediction in an early stage of the processing pipeline, the load instruction can be scheduled to a simplified memory execution pipeline. This *sliced* memory pipeline does not have a cross-bar linking it to all cache banks, but is hard-wired to one bank only (single-bank pipelines). Interconnects are very costly, and their complexity increases super-linearly with the number of ports whereas the single-bank pipeline scheme is highly scalable. Similarly, disambiguation and memory reordering are usually performed with structures featuring a CAM where the number of entries determines the access time. Slicing the disambiguation hardware into several banks makes the implementation simpler and faster. The sliced pipeline is also shorter than the regular multi-banked cache pipeline, since the decision stage, where the proper bank is picked, is no longer required. This reduces the latency for all load operations thus increasing overall performance, provided the bank is predicted with high-accuracy.

In the sliced pipeline, however, a load whose bank is mispredicted, will not be able to execute, and must be flushed and re-executed once the bank is known, even if there is no bank conflict. In order to minimize this degradation in performance, when there is no contention on the memory ports, or if the confidence level of the bank prediction is low, the memory operation may be dispatched to all memory pipelines. Once the actual bank is known, all instructions in the wrong pipelines are canceled, wasting one cycle in each pipe in the case of a low confidence prediction. Note that stores are never on the critical path, thus they may always be duplicated to all pipelines.

A comparison of memory pipelines for a truly multi-ported cache, a dual-scheduled implementation, a conventional multi-banked, and our sliced multi-banked cache appear in Figure 4. The sliced pipe requires a bank predictor for operation, while both the sliced and conventional multi-banked configurations will benefit from a bank predictor due to scheduling improvements. However, the accuracy of the predictor is more important for the sliced pipe. If the predictor is not accurate enough, either too many loads will need to be replicated to both pipes (little advantage over a single-ported cache), or large penalties would be incurred.

The various implementations also differ in their data fetch latencies and bank conflict penalties. The truly multi-ported cache has no latency penalties and no conflicts. The conventional multi-banked configuration has an increased latency due to the crossbar setup and decision stage, and suffers from conflict penalties, since conflicting loads must either stall the pipe or re-execute/re-schedule. The dual-scheduled mechanism eliminates conflict penalties but increases the load latency due to the second scheduler. The sliced multi-banked pipeline has the same latency as the ideal multi-ported pipeline. However, when a misprediction occurs and two loads are dispatched simultaneously to wrong banks (if only a single load is dispatched it is duplicated to both banks) the loads must be re-executed.

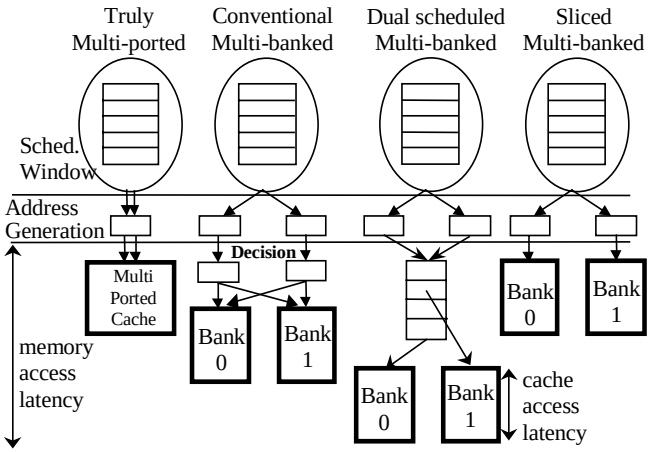

Figure 4 Memory Pipeline Comparison

In evaluating a bank predictor, there are two factors which are of particular interest in determining its performance: *prediction rate* (how many memory operations are predicted?) and *accuracy* (how accurate are the bank predictions?). The importance of the accuracy and misprediction-rate are dependent on the use of the predictor (conventional or sliced multi-banked). If the predictor is used for better scheduling only, the penalty may be less significant since we can stall the pipeline until the bank is available rather than re-executing the instruction from scratch. Confidence can be used in the sliced pipeline scheme to reduce the number of

mispredictions. On the other hand, low confident memory operations waste scheduling slots. The choice of a specific predictor configuration is therefore highly dependent on its intended use. The predictor may either be configured for high prediction rate and lower prediction accuracy, or high prediction accuracy, but with fewer memory operations predicted.

We focus in this study on the general idea of bank prediction and test a configuration with two banks only. With two banks, almost all binary predictors may be adapted to deliver bank predictions. Bank predictors can be based on bank history information, control flow information, and/or load target-address information. We tested many well-known binary predictors based on bank history, as well as combinations of them. An address predictor is obviously extremely well suited to be adapted for bank prediction, since the bank is based solely on the load's effective address (one bit is required to choose between two banks).

The basic configuration tested was a combination of several binary predictors. Each of them supplied a predicted bank along with a confidence level for the prediction. Then several policies were evaluated:

- The prediction was a simple majority vote of the different binary predictions.
- A weight was assigned to each predictor, the different predictions were summed, and a prediction was given only if this sum exceeded a predefined threshold.
- Only those predictions with a high confidence were taken into account.
- A different weight was assigned according to the confidence level supplied.

We also present the results of a bank predictor based on address prediction (address prediction mechanism described in [Beke99]).

Scaling to more than two banks may either be done using a non-binary predictor (such as an address predictor) or by extending binary prediction. Each bit of the bank ID can be independently predicted and assigned a confidence rating. If the confidence level of a particular bit is low, the load will be sent to both banks (as described above) minimizing the disadvantages of independent bit prediction.

3. Simulation Environment

Our simulated architecture consists of a general out-of-order machine with a detailed memory hierarchy model and a branch prediction unit. The simulator is trace driven where each trace is comprised of 30 million consecutive IA-32 instructions translated into micro-operations (uops) on which the simulator works. Each trace is representative of the entire program execution. We conducted our simulations using seven trace groups:

SpecInt95 – 8 traces
SpecFP95 – 10 traces
SysmarkNT – 8 traces
Sysmark95 – 8 traces
Games – 5 traces (of various popular games)
Java – 5 traces (from Java programs)
TPC – 2 traces from the TPC benchmark suite

For each of the three ideas presented in the paper a slightly different approach was taken for simulating it.

3.1 Memory Disambiguation

For memory disambiguation we focused on the scheduling module, and how it should react to wrong memory operation ordering. Whenever a load uop (micro-operation) is wrongly scheduled with respect to a STA or STD uop, a *collision penalty* is added to delay the data retrieved by this load.

The basic machine configuration simulated:

- Memory: 16K Icache, 16K Dcache and 256K unified second level cache (4 way set associative, 64 byte cache lines).
- Front end: 6 uops fetched and renamed per clock.
- Advanced branch prediction mechanism.
- Renamer Register Pool: 128 entries.
- Scheduler: Modeled various scheduling window sizes with different memory ordering schemes. The base line scheduling window had 32 entries.
- Execution: 2 integer, 2 memory, 1 floating point, and 2 complex operation units (as a base-line configuration).
- Retire Rate: up to 6 uops per clock
- Load store collision penalty: 8 cycles.

Six memory ordering schemes were tested:

I. Traditional: each load waits for all STAs, but can advance ahead of STDs. On a wrong load-STD ordering a collision penalty is added.
II. Opportunistic: assumes each load is always non-colliding and advances it as much as possible. If it actually collides then the load waits for both the STA and STD with which it collides, and a collision penalty is added.
III. Postponing Collision Predictor: like in the traditional scheme each load waits for all STAs. But a CHT is used to predict the colliding loads, which are postponed until all older STDs are executed. In the case of AC-PNC, the load waits for the STD with which it collides, and a collision penalty is added.
IV. Inclusive Collision Predictor: A CHT is used to predict which loads are colliding (with any store). When the CHT predicts that a load is going to collide, it is delayed until all STA/STDs complete (hence we may lose some opportunities on "too general" / "not specific enough" predictions). In the case of AC-PNC, the load waits for the STA/STD pair with which it collides to be executed, and a collision penalty is added.
V. Exclusive Collision Predictor: An enhanced CHT with distance information is used for predicting which store is colliding with the candidate load. When the CHT predicts that a load is going to collide, this load waits for the specific STA/STD it collides with.
VI. Perfect memory disambiguation: the scheduler knows exactly which loads collide with which STA or STD and advances or delays loads accordingly.

3.2 Hit-Miss and Bank Prediction

The hit-miss and bank predictors were added to the simu-

lator described above and their statistical performance was evaluated. Since the main objective of the paper is to bring forth these ideas and show their general validity in the context of high performance computing, we made some performance simulations for hit-miss prediction. However, these performance numbers should be regarded only as a flavor of the potential of this idea since the specific machine configuration and design is crucial to the choice and success of our predictors.

4. Simulation Results

We performed two types of simulations. Full performance simulations were carried out for memory disambiguation and hit-miss prediction. Bank prediction was evaluated in terms of statistical predictor performance only.

4.1 Disambiguation Results

To evaluate the effect that different memory ordering techniques have on performance, we ran simulations using 30 traces of the SpecInt95, SysmarkNT, Sysmark95, Game, Java, and TPC groups.

Before we explore the benefits of each policy let us examine the statistical distribution of dynamic loads encountered in the trace into the categories defined in 2.1. This classification gives us an indication of the potential performance gain of the various schemes. This performance gain has two sources: ANC loads can be advanced, saving the needless delay added by traditional schedulers, and AC loads can be delayed eliminating re-execution (see section 2.1 for terminology). The data for the different trace groups (with a 32 entry scheduling window) is shown in Figure 5.

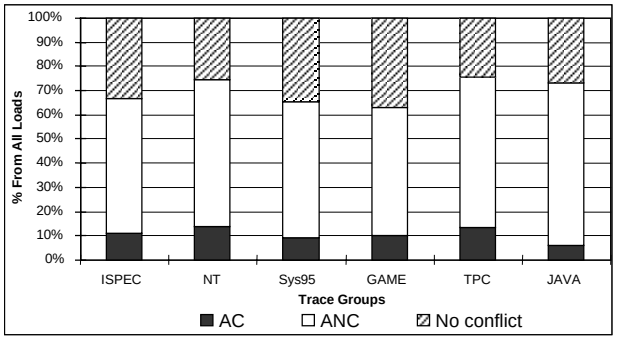

Figure 5 Load Scheduling Classification

By and large, we can see that only 10% of the loads collide with older stores (in the scheduling window), about 60% of the loads do not collide with earlier unexecuted stores and the rest of the loads (~30%) have no ordering conflicts at schedule time. These results imply that between 60% - 70% of the loads can benefit from a collision predictor.

With a larger scheduling window (the scheduling window is the subset of the instruction window from which scheduling is actually performed, e.g. reservation stations) and more execution units, collision rates increase. Looking, for example, at the SysmarkNT traces (Figure 6) we can see that increasing the scheduling window size from 8 to 128 entries results in a steady increase of AC loads, while the percentage of non-conflicting loads decreases. So, as the window size is increased, the potential performance gain of superior memory ordering schemes increases as well.

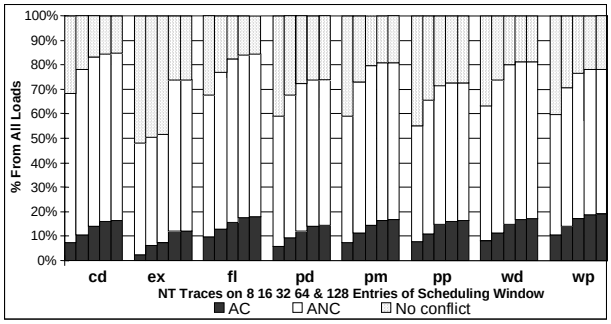

Figure 6 Opportunities vs. Window Size

Figure 7 shows the performance gain expected from the different memory reference ordering methods. The results for the prediction techniques were obtained by using 2K entries of a 2-bit saturating counter Full-CHT, organized as a 4-way set associative table (a new entry is allocated only after a load actually collides). The baseline performance is for the Traditional approach. The maximum performance gain possible from memory disambiguation (Perfect bar) is quite high, about 17% for the SysmarkNT on average. The gain achieved by the different methods follows a consistent curve from the 6% and 9% of the Postponing and Opportunistic schemes to the 14% and 16% of the two predictor based schemes. As can be seen, the Inclusive and Exclusive prediction schemes provide most of the gain possible through memory disambiguation.

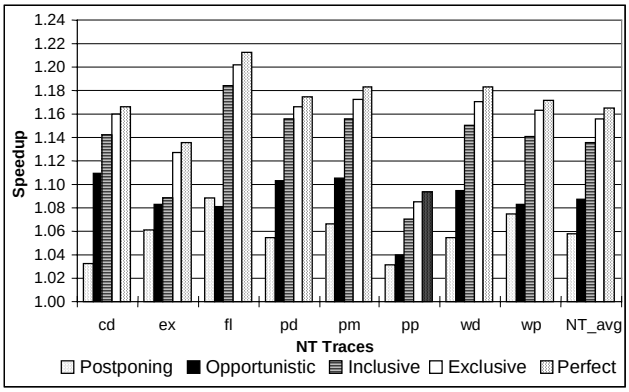

Figure 7 Speedup vs. Memory Ordering Scheme

The effect of varying the number of integer and memory execution units is demonstrated in Figure 8 (marked in the figure as EU# and MEM#). As a general rule, we can see that wider machines gain more performance when using a better memory ordering mechanism. SysmarkNT and SpecInt traces benefit much from advancing loads (8%-17%), while Sysmark95, TPC, Games and Java traces show a less pronounced performance gain (5% to 10%).

We studied the effects of predictor size and other properties on its accuracy. Figure 9 shows the performance for the four configurations presented earlier.

The Full CHT is best for limiting the number of ANC-PC

loads, since it allows for changing load-behavior. This also leads to less stringent predictions of colliding load. For example, the results for the 2K Full-CHT are 3.4% ANC-PC and 0.9% AC-PNC (% of all loads). Since there is a limited number of unique loads in each trace, the predictor's accuracy does not increase much with its size.

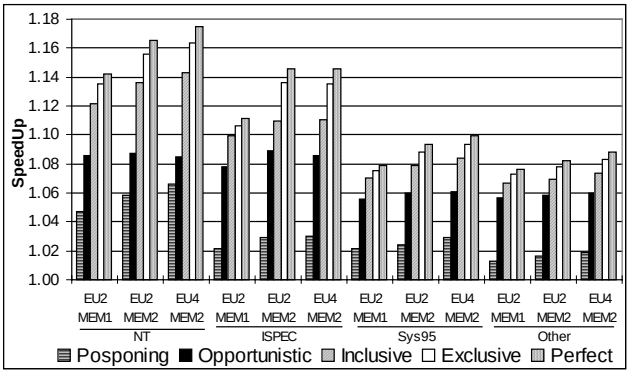

Figure 8 Speedup vs. Machine Configuration

The tagless predictor, which uses a 1-bit counter, shows consistent performance improvement as its size is increased, since it suffers from less aliasing when more entries are available (2K predictor gives 3.8% ANC-PC and 0.8% AC-PNC).

The tagged-only configuration shows an improvement in the accuracy of AC predictions, while giving up several ANC loads (ANC-PC). This behavior is the result of the sticky property: once a load is marked as colliding its behavior is set until it is discarded from the table. After a load is replaced, it is again considered non-colliding (the default prediction). As the table's size is increased there are fewer replacements, thus loads that do change from colliding to non-colliding are now mispredicted (2K predictor gives 11% ANC-PC and 0.2% AC-PNC). As mentioned in [Chry98] cyclic clearing of the tables (once every several million instructions) may alleviate this problem.

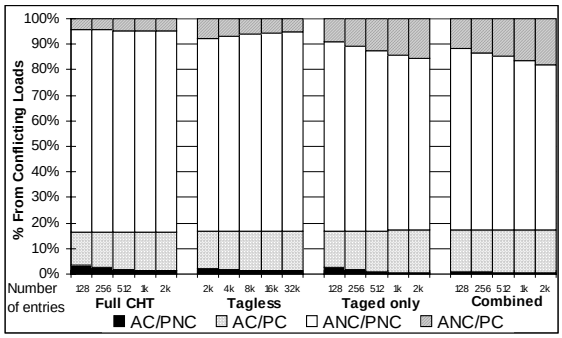

Figure 9 CHT Performance

The combined predictor employs both the tagless (4K entries) and tagged only predictors, it predicts a load as non-colliding only when both predictors agree. Therefore it is best for limiting the percentage of AC-PNC loads, but obviously the ANC-PC loads are more numerous (2K predictor gives 12.6% ANC-PC and 0.16% AC-PNC). Note that Figure 9 focuses on the predictors behavior, and not on performance speedup.

4.2 Cache Hit-Miss Prediction Results

To evaluate the general prediction accuracy of our suggested configurations, statistical simulations (no effect on scheduling) were run on four trace groups: SpecFP95, SpecInt95, SysmarkNT, and "Other" (Games, Java and TPC). To get a feel as to how these results translate into performance gains, we ran performance simulations for SpecINT95 and SysmarkNT.

The following graph (Figure 10) presents the average predictor performance. All results are in percentage of all loads in the trace. The right most bar in each group is the average number of cache misses per trace group (the number of "mispredictions" in the traditional method), the middle bar represents the number of misses which were predicted correctly (AM-PM, higher is better), and the left most bar shows the mispredicted hits (AH-PM - Actual Hit Predicted Miss, lower is better).

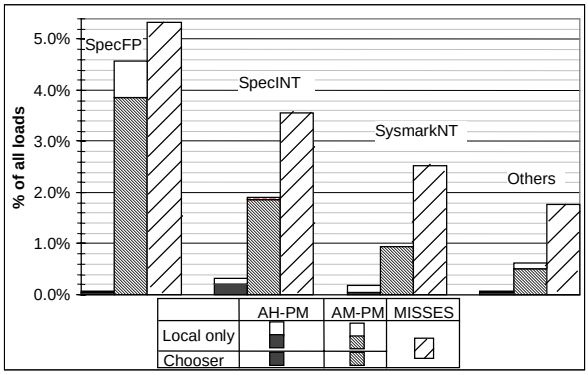

Figure 10 Hit-Miss Predictor Performance

The two left bars of each group show the two predictors' performance and are divided into two parts. The total bar height is the percentage of all loads predicted/mispredicted by the local-only predictor, and the bottom part is for the chooser's results.

The behavior of the two different predictor configurations is consistent between trace groups. The local-only predictor reduces the number of misses from about 34% - 85% (SysmarkNT and SpecFP95 respectively), but mispredicts 0.32% - 0.07% of the hits. Adding a confidence mechanism, via the chooser, reduces the mispredictions significantly (0.2% - 0.04%) while sacrificing little in the AM-PM rate.

On all the traces, the number of misses caught (AM-PM) outweighed the mispredicted hits (AH-PM) by at least 5 to 1 (except in SpecINT xlisp which had a 2/1 ratio). In general, the predictors performed best on FP traces and worst on the "other" trace group.

Performance simulations were carried out on the machine described earlier and on top of our highest performing configuration (4 gen. / 2 mem. EUs and perfect disambiguation). The results show a good correlation between the statistical predictor performance and speedup. The potential speedup of hit-miss prediction (using a perfect predictor) was encouraging, about ~6% speedup on average. With this machine

model the best performing predictor was the local only predictor that also employs timing information as described in section 2.2. It achieved about 45% of the speedup potential, increasing performance by about 2.5%. But again, we emphasize the point that the performance simulations are presented only to demonstrate the idea and to show the correlation between the results of the statistical simulations and the actual portion of the speedup potential that may be achieved. Moreover, execution speedup is only one of the benefits that may be gained by employing hit-miss prediction.

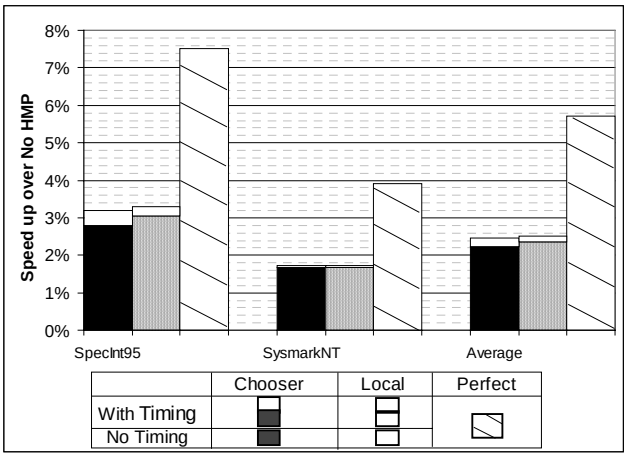

Figure 11 Speedup of Hit-Miss Prediction

4.3 Bank Prediction Results

We concentrated on evaluating the statistical results of the bank predictor. As explained in 2.3, the effectiveness of the bank predictor is greatly influenced by the machine configuration and related penalties. However, we can still evaluate the relative performance aspects, which are the *prediction rate* - percent of loads for which a prediction is made, and the *prediction accuracy* - percentage of predictions that were correct. To better understand the effect of these parameters on performance we have developed a simple metric. This metric represents the relative performance improvement that should be achieved with a specific predictor, compared to that of a perfect predictor. The assumptions for this metric are that each load takes 1 cycle (or 1 unit) to execute, so the maximum gain of a two-ported cache is 0.5 cycles for each load. If the predictor is not ideal a penalty is paid for each load which is mispredicted. The following relations can be derived between the different parameters of the predictor:

$$P = PredictionRate \qquad R = \frac{CorrectPredictions}{WrongPredictions} >> 1$$

$$LoadExecutionTime = (1-P) + P \cdot \frac{0.5R + Penalty}{R+1}$$

$$GainPerLoad = 1 - LoadExecutionTime$$

$$GainPerLoad = P \cdot \frac{0.5R + 1 - Penalty}{R+1} \approx P \cdot (0.5 - \frac{Penalty}{R})$$

$$\boldsymbol{Metric} = \frac{GainPerLoad}{IdealGain} = \frac{GainPerLoad}{0.5} \approx \boldsymbol{P \cdot (1 - \frac{2 \cdot Penalty}{R})}$$

This approach is somewhat simplistic, since it does not take into account the dynamic pipe behavior and, more importantly, the fact that not all loads should be predicted since there may be no contention on the ports. Yet we can still conclude that the misprediction penalty is crucial in choosing a predictor. A small penalty means we must choose a predictor with a high prediction rate, even if it is less accurate. Whereas a higher penalty calls for a more accurate predictor.

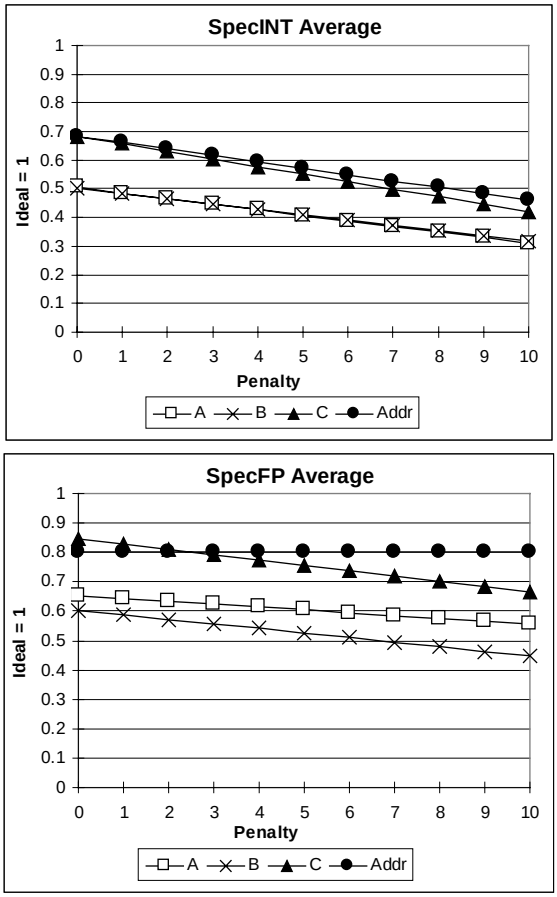

Figure 12 Bank Predictor Comparison (metric)

To make this point clear we present the results for four different predictors. Three predictors are based on the chooser mechanism of Section 2.3 and one is based on an address predictor (2*gshare is weight 2 for gshare):

Predictor A = local+gshare+gskew
Predictor B = local+gshare+bimodal
Predictor C = local+2*gshare+gskew
Local - 512 entries (untagged), 8 bit history (0.5KB)
Gshare - 11 bit history (0.5KB)
GSkew - 17 bit history, 3 tables of 1024 entries (0.75KB)
Addr = the address predictor results as appear in [Beke99]

We present the results using our metric (true multi-porting=1) for SpecINT95 and SpecFP95 in Figure 12. SysmarkNT traces behaved very similarly to SpecINT.

From the graphs in Figure 12 we can learn the prediction rate of the different predictors, which is the *Metric* at the point where the *Penalty* is 0. The predictor accuracy can be deduced from the slope, in general, the steeper the slope the less accurate the predictor is. Typical prediction rates for SpecINT are 50% for predictors A and B, and 70% for the

address predictor and predictor C. Typical accuracy is 97% for predictors A and C, and 98% for the address predictor and predictor B. As can be seen, the address predictor and Predictor C exhibit a high prediction accuracy (large *R*). This makes them suitable for the sliced pipe configuration suggested in 2.3. It should be noted that since the utilization of the memory ports will not be 100%, our simplification method, where low confidence and non-contended loads are duplicated to all pipes should approach ideal multi-porting.

5. Conclusions

In this paper we present three techniques for improving instruction scheduling in high-end processors. As demonstrated through extensive simulations there is significant potential speedup in making load-store reordering possible. To achieve most of the available gain we used a simple, yet accurate, prediction method, which at its simplest form keeps only one bit per load. The predictor finds the loads that can not be reordered ahead of older stores (*colliding loads*) utilizing the fact that colliding loads tend to repeat their behavior, and that most loads do not collide and can be advanced to increase execution speed. The second method improves the scheduling of load dependent instructions by using a cache Hit-Miss Predictor for predicting the dynamic load latencies. This results in execution bandwidth savings, performance speedups and has other potential uses (such as for multi-threading). Bank Prediction is a new technique that tackles a major problem of multi-banked cache architectures (*bank conflicts*) by allowing the scheduler to simultaneously execute only such loads which do not conflict, and allowing simplifications of memory related structures.

Acknowledgments

The authors would like to thank Bob Valentine (Intel) for his creative ideas improving and simplifying our prediction techniques, Jonathan Kliegman (CMU) for his 1997 summer work on Hit-Miss Prediction and Bishara Shomar (Intel) for his major contribution to memory disambiguation.

References

[Aust97] T. Austin and G. Tyson – "Improving the Accuracy and Performance of Memory Communication Through Renaming" – MICRO-30, Dec. 1997.

[Beke99] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz and U. Weiser – "Correlated Load-Address Predictor" – ISCA-26, May 1999.

[Chry98] G. Chrysos and J. Emer – "Memory Dependence Prediction Using Store Sets" – ISCA-25, July 1998.

[Digi97] Digital Equipment Corporation, Maynard MA – "21164 Alpha Microprocessor Hardware Reference Manual" – Digital Equipment Corporation , 1997.

[Fran96] M. Franklin and G.S. Sohi – "ARB: A Hardware Mechanism for Dynamic Memory Disambiguation" – IEEE Transactions on Computers Vol. 45 No. 5, May 1996.

[Gall94] D. Gallagher, W. Chen, S. Mahlke, J. Gyllenhall and W. Hwu – "Dynamic Memory Disambiguation Using the Memory Conflict Buffer" – ASPLOS-VI, Oct. 1994.

[Hess95] J. Hesson, J. LeBlanc and S. Ciavaglia – "Apparatus to Dynamically Control the Out-Of-Order Execution of Load-Store Instructions" – US. Patent 5,615,350 Filed Dec. 1995, Issued Mar. 1997.

[Huan94] A. Huang, G. Slavenburg and P. Shen – "Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation" – ISCA-21, July 1994.

[Hunt95] D. Hunt – "Advanced Performance Features of the 64-bit PA-8000" – COMPCON '95, March 1995

[Inte96] Intel Corporation – "Pentium® Pro Family Developers Manual " – Intel Corporation, 1996

[Jour98] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz – "A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification" – MICRO-31, Dec. 1998.

[Kess98] R.E. Kessler – "The Alpha 21264 Microprocessor: Out-of-Order Execution at 600 MHz" – HOT-CHIPS 10, Aug. 1998.

[Mcfa93] S. McFarling – "Combining Branch Predictors" – WRL Technical Note TN-36, June 1993.

[Mich97] P. Michaud, A Seznec and R Uhlig – "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors" – ISCA-24, June 1997.

[Mosh97] A. Moshovos and G.S. Sohi – "Dynamic Speculation and Synchronization of Data Dependencies" – ISCA-24, June 1997.

[Mosh97b] A. Moshovos and G.S. Sohi – "Streamlining Inter-operation Memory Communication via Data Dependence Prediction" – MICRO-30, Dec. 1997.

[Mowr97] T.C. Mowry and C.-K. Luk – "Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling" – MICRO-30, Dec. 1997.

[Nico89] A. Nicolau – "Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies" – IEEE Transactions on Computers Vol. 38 No. 5, May 1989.

[Patt95] D. Patterson and J. Hennessy – "Computer Architecture A Quantitative Approach, Second Edition (section 5.5)" – Morgan Kaufmann Pub., 1995.

[Pinte96] S.S. Pinter and A. Yoaz – "Tango: a Hardware-based Data Prefetching Technique for Super-scalar Processors" – MICRO-29, Dec. 1996.

[Rive97] J. Rivers, G. Tyson, E. Davidson, and T. Austin – "On High-Bandwidth Data Cache Design for Multi-Issue Processors" – MICRO-30, Dec. 1997.

[Simo95] M. Simone, A. Essen, A. Ike, A. Kishamoorthy, T. Maruyama, N. Patkar, M. Ramaswami, M. Shebanow, V. Thirumalaiswamy, and D, Tovey – "Implementation Trade-offs in Using a Restricted Data Flow Architecture in a High Performance RISC Microprocessor" – ISCA-22, June 1995.

[Tuls95] D. Tulsen, S. Eggers, and H. Levy – "Simultaneous Multithreading: Maximizing On-Chip Parallelism" – ISCA-22, June 1995.

[Weis94] S. Weiss and J. Smith – "POWER and PowerPC" – Morgan Kaufmann Publishers, 1994.

[Wils96] K. Wilson, K. Olukotun, and M. Rosenblum – "Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors" – ISCA-23, May 1996.

[Yeh97] Yeh and Patt – "Two-Level Adaptive Training Branch Prediction" – ISCA-24, June 1997.