A Review of Memory Disambiguation

Abstract—**Memory disambiguation techniques are used to resolve aliasing of load and stores to the same address. As a result, better scheduling decisions can be made by processor or compiler, resulting in higher single thread performance. Various such strategies have been proposed by researchers till date. This short paper, tries to review some of those strategies in chronological order.**

I. Introduction

Out of order pipelines dynamically reschedule instructions to execute independent instructions while dependent instructions are still waiting to get their source operands. This re-ordering of instructions raises a new problem associated to memory operations. Processor cannot figure out any dependence of a load operation on a store operation unless addresses of those instructions are calulated. As a result load instructions cannot be scheduled if any store instruction yet has to commit. This problem also known as ”unknown address problem” [1] can severely hurt performance. Memory disambiguation refers to determining if loads and stores access the same address. Applying memory disambiguation techniques and letting loads execute before store instructions if possible, can boost up performance. For example, 40% performance improvement was seen on Intel Core microarchitecure [2] by applying memory disambiguation technqiques and allowing loads to execute ahead of previous stores.

Early techniques to resolve this problem were non-speculative. Such techniques used to delay load instructions until younger store instructions did not get executed (i.e their data and address become known). One of these techniques was proposed as part of work done to create a new high performance micro-architecture by Yale et al [1]. They proposed a solution for the unknown address problem using “dependency matrices”. These dependency matrices are used to show relationship that exists between operations. Each store instruction with unknown adress makes all entries in a particular column (assume i) of the matrix equal to 1. Load instructions get a unique entry in a row. No load instruction can proceed if first k (k is equal to row number) elements of row are equal to 1. These entries are made 0 when address of particular store instruction becomes known. This technique can be costly to implement in hardware. It also cannot fully exploit the instruction level parallelsim (ILP) available in the program. Since then, various speculative memory disambiguation techniques have been proposed which can exploit more available ILP as compared to non-speculative techniques. In this short paper we will review five speculative memory disambiguation techniques. First three techniques are purely hardware bases, fourth one is purely software based and fifth discussed technique involves hardware software cooperation. Next sections discuss these techniques one by one.

II. Address Resolution Buffer

Franklin and Sohi [3] proposed one of the first speculative memory disambiguation solutions. They observed that non-speculative techniques like that in [1] and Tomasulo machine [4] require wide associative searches (meaning that they are complex in hardware) and they are not very flexible in providing reordering of references. They proposed address resolution buffer (ARB) to support speculative mmeory operations. ARB supports laod speculation and load forwarding. ARB contains many banks and each of them can contain a fixed number of rows. Each row corresponds to an address on which a memory operation is pending in the current instruction window. Due to banking of this structure more than one disambiguation requests can be dispatched per cycle. The scope of associative research is reduced as well due to banking, because an address needs to be compared with addresses of the matching bank only. Each row of any bank contains a number of stages as shown in Figure 2, where each stage corresponds to a sequence number (temporal sequence number of instruction). When a load instruction executes, its load address is used to determine ARB bank. Then within that ARB bank, an associative search is performed to figure out if an earlier store is executed to the same address in the active window. If an earlier store exists in the bank, associated data value (from closest store) is forwarded to the load. In case no store is found, load request is sent to the data cache speculatively. If no row exists for address of a load or store instruction a new row is created. If the address for store instruction exists in a bank, particular row is searched to see if there is any younger load that has executed and no other store instruction has executed on the same address yet. If this is the case, recovery is performed by squashing all intstructions after this store. Tail pointer which determines the boundary of active window as shown in Figure 2 is also restored.

In this technique it is assumed that load does not depend on an older stores and if an older store entry is not executed yet, load is always speculatively sent to data cache. This assumption, instead of learning from past behaviour can result in lower performance. As a result misprediction rate of the address resolution buffer for some of the benchmarks can be high. In [3] Franklin and Sohi have shown that ARB

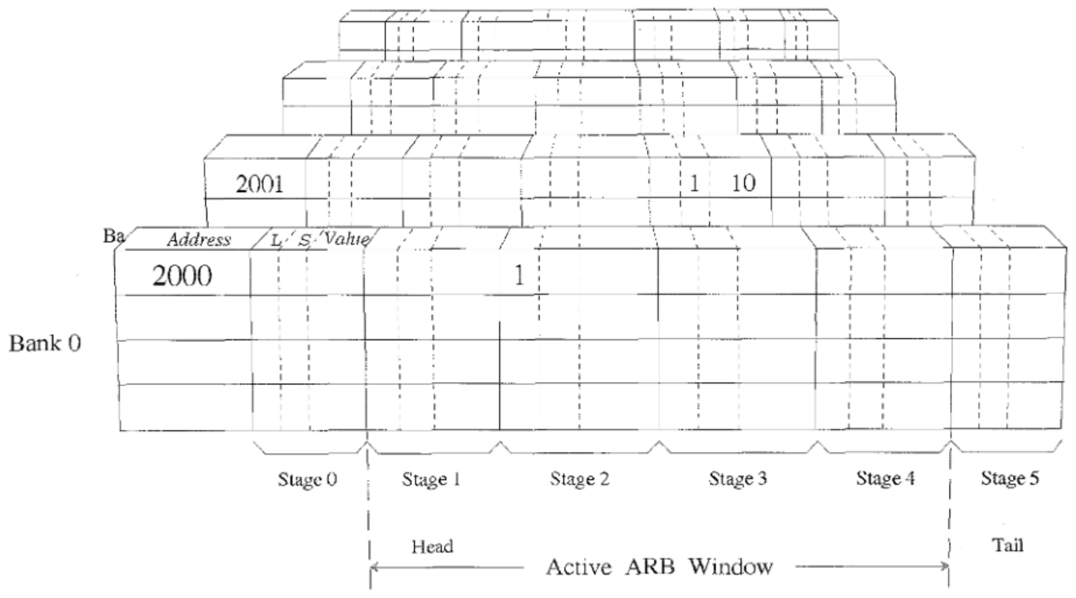

Fig. 1. Address Resolution Buffer [3]

performs better than dependency matrix and is less expensive in hardware cost due to reduced associative search.

III. Memory Renaming

Memory renaming was proposed by Tyson and Austin [5]. They used register communication techniques to improve the performance of memory traffic. They showed an overall improvement of 14% in execution time after over the base case (no speculation). The name memory renaming is derived from the similar technique of register renaming. In memory renaming, instead of identifying the address of store instruction used for transmission of value, particular store instruction is identified thta writes the data value. To support this technique a memory dependence predictor and a register value file (an extension to normal register file) are incorporated into pipeline. Memory dependence predictor (also termed as load store cache) which uses program counter (PC) of load and store instructions is used to indicate dependence between load and store instructions. The pair that is predicted to communicate is bound to same physical register in value file. Thorugh their experiemnts Tyson and Austin found out that for SPEC92 benchmakrs, producer locality (reuse of same source / producer instruction) ranges from 43% to 75% for SPEC integer and floating point benchmarks. That is why load store dependence pairings are used to determine when speculation would be beneficial.

While at decode stage, load or store insturction looks in load store cache (memory dependence predictor) to get value file index which is then used to find entry in value register file. If the entry is found: it is checked that if the value can be used from value file, based on the status of a confidence counter or if their is an in-flight store that can update register file value. If that is the case it returns its entry number from reorder buffer. Otherwise, the last existing value from register file is returned. If entry does not exist in value file, a new entry in cache is created and it is associated with a new entry in value file. Later if the value is found to be correctly predicted i.e. the value used is validated against the correct value, load is commited. Otherwise, recovery can be initiated. Two recovery mechanisms are presented by the authors: squash recovery and re-execution recovery. In squash recovery, all instructions including and after the mis-predicted load are squashed and execution restarts from that point. In case of re-execution recovery only dependent instructions are squashed leaving the independent ones. Obviously this is a high performance recovery mechanism but it is more costly in hardware.

They have used SimpleScalar simulator for simulation studies. An average of 62 % of memory dependencies are predicted correctly by the predictor for all programs. The experiments performed that use confidence mechanism show at least 69.22 % confidence for all of them. Performance improvements of more than 16 % are shown overall. This mechanism can only detect load store dependences within the instruction window. Since most load store instructions as shown in [6] are distant, this can be a limitation for higher performance [7].

IV. Store Set Predictor

Store set predictor [8] is one of the relatively simple proposed approaches for memory disambiguation. A store set is maintained for any specific load. This store set contains all

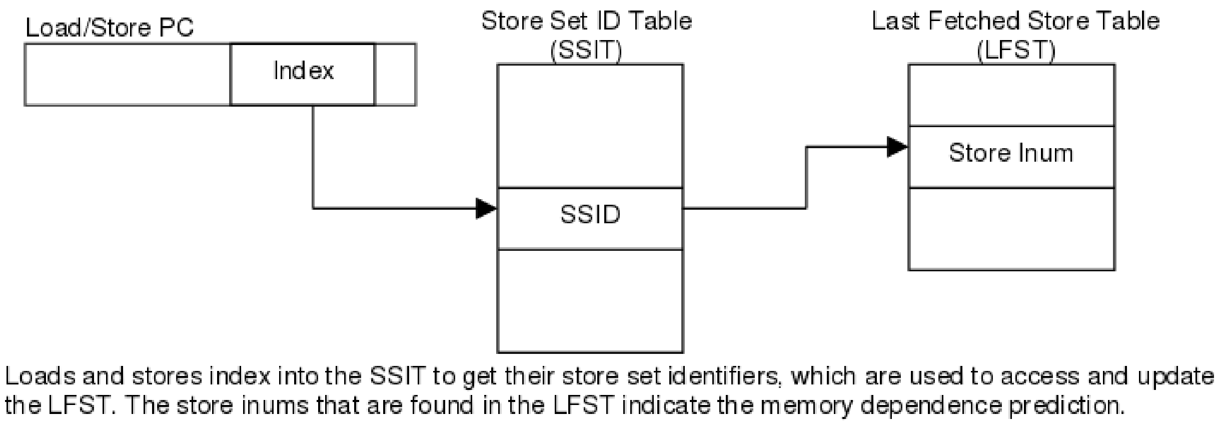
Load/Store PC
Index
Store Set ID Table (SSIT)
SSID
Last Fetched Store Table (LFST)
Store Inum
Loads and stores index into the SSIT to get their store set identifiers, which are used to access and update the LFST. The store inums that are found in the LFST indicate the memory dependence prediction.

Fig. 2. Store Set Memory Dependence Prediction [8]

the stores on which this load has ever depended. Whenever a load executes before a store and a memory order violation is caused, the store instruction is added to the store set. Later when the load instruction executes again, store set is used to predict what store instructions this load depends on. Same store can exist in multiple store sets, since multiple loads can depend on same store. Considering this, authors proposed a store-set merging predictor implementation to redeuce hardware cost. This predictor contains two tables: store set ID table (SSIT) for store-load association connections and last fetched store table (LFST), which keeps track of the store currently in the instruction window for a particular ID. On occurrence of a memory violation, if load or store exists in SSIT, and only one of them has an id , the other is given the same id. If none of them has an id, new id is allocated and written into SSIT. To write this new id, SSIT is indexed using program counters of both load and store isntructions. If both load and store have id's in the SSIT, smaller id is assigned to both of them. As a result, store sets of different loads get merged. When a load is fetched, it accesses the SSIT and gets store set id. Based on this id, LFST is accessed and sequence number of the most recently fetched store in its store set. Load cannot be scheduled bfore this store.

Chrysos and Emer [8] simulated a 8 wide superscalar Alpha machine and used SPEC95 benchmarks for interval of 100 million instructions. To evaluate the perforamnce, infinite configuration (each store set can have as many stores as load depdends on and one store can exist in more than one store set) is used. A 2 bit saturating counter is also incorporated to reduce false dependencies. This predictor is shown to exhibit nearly optimal performance in large instruction window, pipelines when compared to perfect memory dependence predictor.

V. Probabilistic Memory Disambiguation

Ju et al [9] proposed a probabilistic memory disambiguation (PMD) framework to statically find out memory aliasing and make spculation decisions. Authors have focused on memory references from arrays. Giving examples of different code segments authors have made a case that data speculative instructions are profitable only if there is a high probability that original load instruction will not alias with the store instruction being speculated across. An expression of aliasing probability is defined in the paper. Intuitively, the denominator of the expression is the number of times that the load is executed in the iteration space and numerator is the number of times load and store alias to the same address. Numerator is weighted by the probability that the store instruction will execute. Based on the aliasing probabilty following cost model can be used by the compiler to determine if the data specualtion will be profitable or not.

(Lo) should be greater than (Ls + p * (Or + Lc))

Here Lo is the cycle length of schedule in original code. Ls is cycle length of code in data speculatin version, p is the aliasing probability. Or is the overhead to call recovery code and Lc is the cycle length for schedule of recovery code. If the above mentioned expression holds valid, data speculation can result in performance gain. Since, most of these factors are compile time constant, compiler can evaluate above expression and make decisions about speculation statically. A set of heuristics are also developed after studying some common cases to avoid expensive probability computations.

Finally some experiments are performed by authors by modelling HPL PlayDoh VLIW machine. Certain code fragments are chosen from real numeric applications, which are considered to be candidates of data speculation. Results show that for most of the cases aliasing probability is proportional to reciprocal of N (number of loop iterations of code). Cost of recovery code is also high for smaller N, therefore data speculationf for small N results in worse performance than the base version (with no speculation). The performance of speculation version improves with increasing N. Speed up of 1.1 to 1.2 is observed when data speculatin becomes profitable. The threshold value of N where data speculation becomes profitable varies. Finally, significance of guiding data speculation with a cost model like PMD framework to maximize program performance (regardless of value of N)is established.

Thoough this static technique looks promising, but it suffers from the same limitations of any static solution in computer architecture research. If underlying pipeline parameters change, recompilation and establishment of new probabilistic model will be needed.

VI. Feedback-directed Memory Disambiguation

Fang et al [10] proposed a feedback directed memory disambiguation technique. The proposed idea is to generate a representative store distance (store distance is number of stores between a load and a store which access the same address)for each load instruction and then apply compiler/micro-architecture cooperative scheme to perform run-time load speculation. This idea achieves a performance very close to perfect memory disambiguation. If compared to store set technique this idea performs better for small predictor size but gives comparable performance with 16-K entry store set implimentation. Since this technique is a cooperative hardware/software solution it gives the best of both worlds i.e. it does not require a lot of space on chip and it is not limited to array-based codes as well.

Fang et al [10] observed that on average, 82% of the load instructions in SPEC 2000 benchmarks have just one store distance. For other instructions, which can have multiple store distances (result from change in dependences), a summary distance is chosen by compiler based on distance distribution of each instruction. To calulate store distance distribution, for load instruction program is run witha small input for profiling. A global store instruction counter is maintained, to record store cycle. Each store is inserted into a hash table alongwith its current store cycle. Later when a load is fetched it searches the hash table and compares the current store cycle against the previous store cycle with the same address. Store distance can be calculated based on this comparison. For out of order pipelines, if store distance is greater than a threshold termed as ‘speculating distance’ (for example, ROB size in out of order pipelines), store distance is set equal to speculating distance. In this case, load instrucion is not likely to depdend on any store instruction in current pipeline. A store distance is called ‘dominant distance’ if it accounts for more than 95 % of total accesses for this instruction. So, initially speculating distance is set as summary store distance, then if a dominant distance exists for load , summary store distance is set equal to it. Otherwise minimum distance is chosen as summary store distance. This method assumes that small store distances remain constant across different program inputs. On the other hand if store distance is greater than specualting distance, load instructions are likely to be speculated without any mis-specualtion. So, this scheme assumes constant store distances across different inputs.

Supposing that s is the speculating distance, log(s)+1 bits of the offset field of a load instruction are used for encoding of store distance. After encoding of store distance in load instruction, micro-architecture just has to identify store on which this load depends. A store table is kept for this purpose. Every store instruction in the pipeline has an entry in store table with a unique id. When a load instruction is decoded, the encoded store distance is used to index into the store table and get an id of related store instruction. This id is then used in ROB to make speculation decisions.

The results in [10] show that 99.1% of load instructions in CFP2000 benchmakrs and 94.9% of load instructions in CINT2000 benchmarks have constant store distance irrespective of test and reference inputs. Moreover, it is also shown that on floating point benchmarks store distance approach achieves a harmonic mean improvement of 9% over Store set technique with 1K entries and achieves a 4% performance improvement over store set technique with 4K entries. For integer benchmarks a harmonic mean improvement of 8% and 4.5% are achieved over store set technique with 1K and 4K entries respectively. A subset of SPEC 2000 benchmarks is used on FAST simualtor using MIPS ISA, to evaluate perormance of this idea. Though results seem convincing, but the assumption that store distance remains same across workloads seems naive.

VII. Conclusion

Memory disambiguation strategies reviewd in this paper are diverse in their design. Speculative memory disambiguation techniques always provide more leverage to achieve better performance. Depending on the design hardware cost of these techniques can vary as well. For example while address resolution buffer and memory renaming require a lot of hardware, store set technique’s simplicity spares some of hardware cost. Purely software strategies like PMD do not require any hardware cost but they can suffer from serious incaccuracies due to varying workloads of these days. Techniques which uses both hardware and software solutions usually have better opportunity for optimization. For example, feedback directed memory disambiguation discussed in this paper requires less hardware than store set technique and performs better in most of the cases. For good analaysis of different memory disambiguation architectures one should refer to [11], [7]

References

[1] Y. N. Patt, S. W. Melvin, W.-m. Hwu, and M. C. Shebanow, “Critical issues regarding hps, a high performance microarchitecture,” *ACM SIGMICRO Newsletter*, vol. 16, no. 4, pp. 109–116, 1985.

[2]

[3] M. Franklin and G. S. Sohi, “Arb: A hardware mechanism for dynamic reordering of memory references,” *Computers, IEEE Transactions on*, vol. 45, no. 5, pp. 552–571, 1996.

[4] D. Anderson, F. Sparacio, and R. M. Tomasulo, “The ibm system/360 model 91: Machine philosophy and instruction-handling,” *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 8–24, 1967.

[5] G. S. Tyson and T. M. Austin, “Improving the accuracy and performance of memory communication through renaming,” in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pp. 218–227, IEEE, 1997.

[6] A. Moshovos and G. S. Sohi, “Streamlining inter-operation memory communication via data dependence prediction,” in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 235–245, IEEE Computer Society, 1997.

[7] “Literature survey.” https://users.ece.cmu.edu/~omutlu/pub/onur_ss_lit_survey_2001.pdf.

[8] G. Z. Chrysos and J. S. Emer, “Memory dependence prediction using store sets,” in *ACM SIGARCH Computer Architecture News*, vol. 26, pp. 142–153, IEEE Computer Society, 1998.

[9] R. D.-c. Ju, J.-F. Collard, and K. Oukbir, “Probabilistic memory disambiguation and its application to data speculation,” *ACM SIGARCH Computer Architecture News*, vol. 27, no. 1, pp. 27–30, 1999.

[10] C. Fang, S. Carr, S. Önder, and Z. Wang, “Feedback-directed memory disambiguation through store distance analysis,” in *Proceedings of the 20th annual international conference on Supercomputing*, pp. 278–287, ACM, 2006.

[11] B. Calder and G. Reinman, “A comparative survey of load speculation architectures,” *Journal of Instruction-Level Parallelism*, vol. 2, pp. 1–39, 2000.