Invited Paper:

Memory Disambiguation Hardware: a Review

Fernando Castro, Daniel Chaver, Luis Piñuel, Manuel Prieto and Francisco Tirado
ArTeCS Group, Department of Computer Arquitecture, Complutense University
Madrid, Spain
fcastror@fis.ucm.es, {dani02, lpinuel, mpmatias, ptirado}@dacya.ucm.es

ABSTRACT

One of the main challenges of modern processor designs is the implementation of scalable and efficient mechanisms to detect memory access order violations as a result of out-of-order execution. Conventional structures performing this task are complex, inefficient and power-hungry. This fact has generated a large body of work on optimizing address-based memory disambiguation logic, namely the load-store queue. In this paper we review the most significant proposals in this research field, focusing on our own contributions.

Keywords: LSQ, Memory Disambiguation, Energy-Efficiency, Filtering, Hardware Simplification.

1. INTRODUCTION

With high operation frequency, modern out-of-order processors often need to buffer a very large amount of instructions to be able to overlap useful processing with relatively long latencies associated with accesses to lower levels of the memory hierarchy. Processor features such as multithreading further increase the demand on the instruction buffering capability. However, increasing the number of in-flight instructions requires scaling up different microarchitectural structures, which has a significant impact on energy consumption, especially if the structure is accessed associatively.

One such example is the logic that enforces correct memory-based dependences, commonly referred to as the load-store queue (LSQ), and typically implemented as two separated queues: the load queue (LQ) and the store queue (SQ). Conventional implementations of these queues contain complete addresses and their entries are allocated in program order. To enable early execution of loads without compromising program correctness, memory instructions are tracked by the two queues and associative searches are used to find the correct producer or to detect dependence violations. These associative search operations are a major concern for the scalability of these queues. Not only energy consumption increases with the size of the queue, the latency of accesses also worsens and may present complications in the logic design. As such, a range of implementations that avoid associative searches have been explored recently. The main observation behind these designs is that memory-based dependencies are very infrequent and hence, through clever filtering or prediction, it is posible to reduce the number of associative accesses.

Sections II and III recap the conventional design of the LSQ and the main alternatives. Section IV explores our proposals. Finally, Section V concludes.

2. CONVENTIONAL DESIGN

Modern out-of-order processors usually employ an array of sophisticated techniques to allow early execution of loads to improve performance. Almost all designs include techniques such as *load bypassing* and *load forwarding*. Both schemes allow early execution of loads when all preceding stores have calculated their addresses. More aggressive implementations go a step further and allow execution of loads when the address of a preceding store is not yet resolved. Such *speculative execution* can be premature if an earlier store in program order writes to the memory space loaded and executes afterwards. Clearly, this speculation has to be applied such that program correctness is not compromised. Thus, the processor needs to detect, squash and re-execute (or replay) premature loads and their dependents. To simplify implementation, processors typically replay many more instructions (such as all instructions following the store [1]), as these premature loads are rare in general and sometimes extra logic is employed to further reduce their occurrence [2].

The dependence enforcement is achieved using age-ordered load queue and store queue. A memory instruction of one type needs to check the queue of the opposite kind in an associative fashion (see Figure 1): a load searches the SQ to forward data from an earlier, in-flight store and a store searches the LQ to identify loads that have executed prematurely (wrongly speculated).

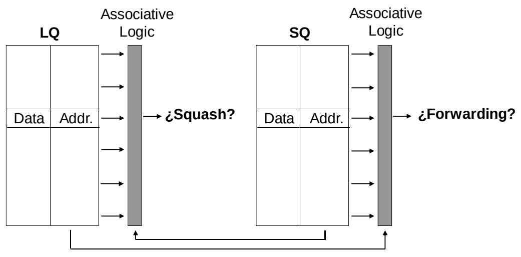

Figure 1 - Conventional LSQ design

3. LSQ: STATE OF THE ART

The LSQ is a hardware structure that exhibits two main problems: 1) its logic is complex as it involves associative comparison of wide operands, which implies a high energy consumption, and 2) the scaling of the LSQ increases its access latency, which makes it hard to integrate it in high-frequency designs. We can identify three different approaches to overcome these problems. Based on the observed behavior of memory instructions (dependences and forwardings are infrequent), many researches have proposed *filtering techniques* to reduce the number of associative searches. Other designs adopt a *two-level approach* for disambiguation and forwarding. The guiding principle is largely the same: use a first level structure small but still able to perform a large majority of the work. This first level is backed up by a much larger second level structure to correct/complement its work. Finally, other designs try to *simplify/remove* the associative hardware of the LSQ looking for a simpler and cheaper management of load store queue operations.

In the following sub-sections we summarize the main contributions. Before reviewing them, we'll start with the most important memory dependence prediction techniques. Memory dependence predicition is an important alternative to address-based mechanisms to allow aggressive speculation and yet avoid penalties associated with squashing. It allows to establish guidelines in the out-of-order memory accessing of load instructions The key insight is that memory-based dependences can be predicted independently on the actual address of each instance of memory instructions and this prediction allows for stream-lined communication between likely predicted store-load pairs.

Memory Dependence Prediction

Moshovos *et al*. [3] propose techniques that attempt to predict those instructions whose immediate execution is going to violate a true dependence, and delay the execution of those instructions as long as necessary to avoid the mis-speculation. They observe that the static store-load instruction pairs that cause most of the dynamic data mis-speculations are relatively few and exhibit temporal locality. This observation suggests using past history to dinamically identify such pairs and a storage structure to cache the information. Thus, when a load speculatively issues and violates a true dependency with an older store, the program counters (PC) of the load and the store are recorded in a table. Subsequent executions of the load instruction will check the table to see if they have conflicted with a store in the past. If so, the load will also check the SQ to see if that store is present. If the store is present in the store queue, then the load must wait until the store has been issued.

Chrysos *et al.* in [4] propose another technique to predict dependences between memory instructions. The proposed memory dependence predictor is based upon the concept of *store sets*. A store set of a specific load is the set of all stores (identified by their PC) upon which the load has ever depended. When a program begins executing, all of the loads have empty store sets, and the processor allows naive speculation of loads around the stores. When a load and a store execute in the wrong order, causing a violation, the store PC is added to the load's store set. Thus, when a load is fetched, the processor will determine which stores in the load's store set are recently fetched but not yet issued, and create a dependence upon those stores. Loads that never cause memory order violations will have no imposed memory dependencies, and will execute as soon as possible.

Subramaniam *et al.* [5] use the idea of dependency vectors from matrix schedulers applied to non-memory instructions and adapt them to implement a new dependence prediction algorithm. The dependency vector is an alternative scheduler topology designed to be significantly more scalable. Goshima *et al.* [6] proposed to replace CAM structures of a conventional scheduler with two dependency matrices to illustrate memory dependences. For a N entry scheduler, each matrix has N rows and N columns; one for each instruction. If instruction *i* is data-dependent on instruction *j*, then the matrix entry at row *i* and column *j* is set to one. So long as instruction *i* has a bit set in its row, then the corresponding input dependency has not been resolved. When instruction *j* issues, it clears all bits in column *j*, thus notifying any dependents in the window that the parent instruction has been scheduled. Store vectors are different than the load-store pairs and store sets approaches in that store vectors do not explicitly track the PCs of stores that collide with loads. Instead, the authors implicitly track load-store dependencies based on the relative age of a store. Thus, a load's store vector records the relative positions or ages of all stores that were involved in previous memory ordering violations. The store vector algorithm has three main steps: update, prediction and scheduling. Initially, all loads are allowed to execute as soon as their addresses are known, so all store vectors are initialized to zero. When load-store ordering violations occur, the store vectors are updated. In the prediction phase each load obtains its own vector; in the scheduling step loads wait until all bits in its vector are zero to proceed to execution.

Fang *et al.* [7] apply a form of memory distance, called store distance defined as the number of store instructions between a load and the previous store accessing the same memory location. Through profiling, the instruction-based store distance distribution is analized and a representative store distance is generated for each static load instruction. Then, a cost effective microachitecture mechanism is developed for the processor to determine accurately on which specific store instruction a load depends according to its distance annotation.

Filtering techniques

Sethumadhavan *et al.* [8] propose a type of filtering scheme named *search filtering*, which uses hashing to reduce both the number of lookups to the LSQ and the number of entries that must be searched. Two Bloom Filters [9] are employed, one for loads and another for stores. Those filters are made up of a table of counters. Store and load instructions, when issued to execution, index (address-based) an entry of its corresponding filter, incrementing the counter. At commit time, the instruction decrements the counter. Besides, at issue time, the instruction accesses the opposite filter, in order to check if a potential memory dependence violation exists. If the stored value into the corresponding counter is bigger than 0, there is a possible memory dependence, and the store/load performs an associative search to the LQ/SQ. If it is 0, then we know for sure that there is no match between memory addresses, and the LQ/SQ search is avoided, filtering this way a large number of associative accesses.

Park *et al.* [10] propose two techniques to reduce the search bandwidth requeriment on the LQ/SQ. The first one extends the store set predictor [4] to predict the matches between loads and stores. They called their scheme *store-load pair predictor*. A load will search the store queue only when the store-load pair predictor predicts that there is a potentially-dependent store in the queue and tells the load to obtain its value from the SQ. While the store-set predictor detects only those store-load pairs that cause dependencies violations, their *store-load pair predictor* detects all matching pairs of loads and stores regardless of whether they cause violations. To detect potential mispredictions stores search the load queue for matching load at commit time. The second one reduces the search bandwidth demand on the load queue. For this propose, they introduce the *load buffer*, which is a small buffer to hold loads that are issued out of order. The key idea is that for detecting load-load order violations (multiprocessors), each load should only check those adresses corresponding to loads issued out of order. The authors highlight that this kind of loads are rare. This way, just adding a small buffer to the LQ, when a load executes, instead of checking the entire load queue, it has to check only the *load buffer*, reducing search time and energy consumption.

Sha *et al.* [11] propose the improvement of the SQ scalability by implementing store-load forwarding using speculative indexed access -*store queue index prediction* (SQIP) -, rather than associative search. This technique uses prediction to identify the single SQ entry from which each dynamic load is most likely to forward. A forwarding mis-prediction, detected by pre-commit filtered load re-execution, results in a pipeline flush. To predict forwarding SQ entries, they use a two-table predictor that is an adaptation of store sets [4]. The first table maps each dynamic load to a small set of static stores from which it has forwarded in the past; the second table maps each of these static stores to the SQ index of its youngest in-flight instance. The predictor selects the youngest of these indices.

Split/two-level LSQ structure

Baugh *et al.* [12] propose an alternative LSQ organization that separates the time-critical forwarding functionality from the process of checking that loads received their correct values. Two main techniques are exploited: first, the store-forwarding logic is accessed only by those loads and stores that are likely to be involved in forwarding, and second, the checking structure is banked by address. The result of these techniques is that the LSQ can be implemented by a collection of small, low-bandwidth structures yielding to a significant reduction in LSQ dynamic power. They propose the usage of a new structure, the *store-forwarding buffer* (SFB), that is much like a traditional store queue but with fewer entries and fewer ports, yielding a reduction in access time and a significant reduction in power consumption. The size of the structure is reduced by allocating entries for only those stores predicted to require forwarding. Similarly, required bandwidth is reduced by snooping only for those loads that are predicted to require forwarding. Because these predictions can be wrong, a mechanism is required to detect faulty predictions. This is performed by a second structure, called MVQ (*memory validation queue*), that is the responsible for detecting load-store ordering violations, consistency violations and forwarding mispredictions. This structure must observe all in-flight loads and stores to identify violations. To efficiently implement this structure, it is banked by address. As a large fraction of static instructions are never involved in forwarding, a single bit per static instruction is sufficient to effectively predict the forwarding behavior of an instruction. Only marked stores are allocated in the SFB. The MVQ, in addition, must detect situations in which load-store forwarding should have been performed on unmarked loads or stores.

Roth [13] proposes a new load-store unit design in which the demand for SQ search bandwidth is reduced. A single bit per load is used to represent the past behavior about forwardings. Marked loads (forwarding in the past) access the SQ, unmarked loads do not. Detecting wrong SQ non-accesses is done by re-executing filtered loads prior to retirement. A load whose re-executed value differs from the one stored in its LQ entry elicits a squash similar to one triggered by a misprediction branch and it is also marked.

Stone *et al.* [14] propose that the functions of store-to-load forwarding, memory disambiguation and in-order retirement of stores were divided among three structures: an address-indexed store forwarding cache (SFC), an address-indexed memory disambiguation table (MDT) and a store FIFO. Because these structures do not include CAMs or priority encoders, they scale readily as the number of in-flight loads and stores increases. The SFC is a small cache to which stores write their values as they complete, and from which loads may obtain their values as they execute. Both loads and stores access the SFC speculatively and out-of-order, but the SFC does not rename stores to the same address. Therefore, violations of true, anti, or output memory dependences can cause loads to obtain incorrect values from this cache. For this reason, the authors employ another structure, named MDT, to detect memory dependences violations. This table buffers the sequence numbers of the latest load and store to each in-flight memory. The processor assigns these numbers that impose a total ordering on all in-flight loads and stores. Thus, just with simple comparisons between sequence numbers the MDT can detect memory dependence violations; if so, it initiates recovery by flushing all instructions subsequent to the load or store whose late execution caused the dependence violation.

Akkary *et al.* [15] propose a hierarchical store queue organization to mitigate the bottleneck that supposes the implementation of store-to-load forwarding, since the store queue must provide the dependent load with data within the data cache access time. The hierarchical proposed store queue splits SQ into two levels: a fast and small first level backed by a much larger and slower second level. The level one buffers the last *n* stores. Since stores typically forward to nearby loads, most forwardings occurs from the first level. When the first level is full, the oldest store is removed and moved into the second level. The level two has membership test buffer (MTB) structures (similar to Bloom filters) to quickly determining wether a given load address matches a store entry in this second level. Thus, when a store is removed form the first level and allocated in the second one, the corresponding MTB entry is incremented. When a store retires and is removed form level two, the corresponding MTB entry is decremented. When a load is issued, the first SQ level and the MTB are accessed in parallel. If the load hits level one, the store data is forwarded to the load. If the load misses the first level and the MTB entry is zero, the data is forwarded to the load from the data cache. If the load misses level one and the MTB is not zero (potential match), the load is penalized a data cache miss penalty to allow sufficient time to acces SQ level 2 and resolve the dependency (if the load hits second level, data is supplied from this level 2; otherwise the data is forwarded from the data cache).

Torres *et al.* [16] propose a two-level SQ well suited to first-level multibanked data caches. The goal is to forward data from in-flight stores to dependent loads with the latency of a cache bank. For that they introduce a particular two-level SQ design in which forwarding is done speculatively from a distributed first level made of extremely small banks, while a centralized, second level checks its correctness and enforces correct store-load ordering a few cycles later.

Sethumadhavan *et al.* in [17] propose a new LSQ implementation to improve area and efficiency by allocationg entries when instructions are issued rather than when they are dispatched. This requires the entries in the LSQ to be unordered with respect to age. To compensate the lack of ordering, the design determines the age by explicitly storing the it in a separate age CAM. To support commits, the age CAM is associatively searched with the age supplied by the ROB. The address and data from the exact matching entry are read out from the CAM and RAM respectively, and sent to the caches. To support violation detection, when a store arrives it searches the address CAM to identify matching loads, and searches the age CAM to identify younger loads. The LSQ then performs a logical *or* of the results of the two searches to flag a violation.

Removing LQ/SQ

Cain and Lipasti [18] propose to solve the associative load queue scalability problem by completely eliminating the associative load queue. Instead, data dependences and memory consistency constraints are enforced by simply re-executing load instructions in program order prior to retirement. Authors propose a mechanism named *value-based replay*, that performs re-execution of load operation in program order prior to commit. If both loads read the same value, then the load correctly resolved its memory dependences; otherwise a violation occurred either due to an incorrect reordering with respect to a prior store or a potential violation of the memory consistency model. Instructions that have already consumed the load's incorrect value are squashed.

Roth [19] proposes a new mechanism to significantly reduces the re-execution requirements in those schemes similar to [18] that allow a first speculative load access and perform another access prior to commit to compare values obtained and verify the correctness of the first execution. The autor introduces the *store vulnerability window* (SVW), a filter that reduces the number of loads that must re-execute to support a given load optimization. SVW assigns each dynamic store a monotonically increasing sequence number. This number is used to associate with each dynamic load a dynamic window of stores to which that load is made vulnerable (potentially dependent). For convenience, this load's SVW is defined as the sequence number of the youngest older store to which the load is not vulnerable. Using a Bloom filter that stores in each entry the sequence number of the last retired store to write to any partially matching address, a violation can be ruled out simply comparing the load's SVW and the corresponding sequence number stored in the Bloom filter. This way the number of loads that need to be re-executed significantly reduces.

Subramaniam *et al.* [20] propose a mechanism similar to the store queue index prediction (SQIP) [11], in which for each load the SQ index of a sourcing store is predicted. The reason why SQIP works is that those loads that receive data directly from stores will usually receive the data from the same store each time. [20] takes a slightly different view on the underlying observation used by SQIP: a store that forwards data to a load ususally forwards to the same load each time. To implement this technique, that results in the complete elimination of the SQ, any store that forwards data to a load will use a predicted LQ index to directly write the value to the LQ entry without any associative logic. Any mispredictions/misforwardings are detected by a low-overhead pre-commit re-execution mechanism.

4. AUTHORS' CONTRIBUTIONS

In this section we summarize our main LSQ techniques recently introduced: Split-LQ, IMDC (*Issue-time Memory Dependence Checking*) and DMDC (*Delayed Memory Dependence Checking*).

Split-LQ

Our first proposal [21] uses a split-LQ design where the conventional associative load queue is replaced with a smaller associative LQ (ALQ) and a banked non-associative LQ (BNLQ). Loads are processed differently and accommodated in different queues (see Figure 2) based on the prediction whether they are dependent on an in-flight store. We hold the CAM structure for the ALQ and the predictor allocates in this queue only the loads predicted to communicate with an in-flight store. Loads predicted to be independent are allocated in a simple FIFO buffer. Dependence enforcement for the ALQ is the same as in a conventional design, whereas that for the BNLQ is done through a

Bloom filter, named EBF, which is inexact and conservative but energy-efficient for the common case where there is no dependence.

Our work exhibits certain similarity with [12] in which both designs splits an associative structure into two parts -one associative and another that does not have this kind of logic- employing a backup mechanism to perform its function. One difference between these works is that we focus on the LQ whereas Baugh *et al.* on the SQ. Furthermore, there are significant differences in the backup mechanism: we just need to check if the value in our EBF is zero, while employing a MVQ a comparison between addresses and sequence numbers is needed.

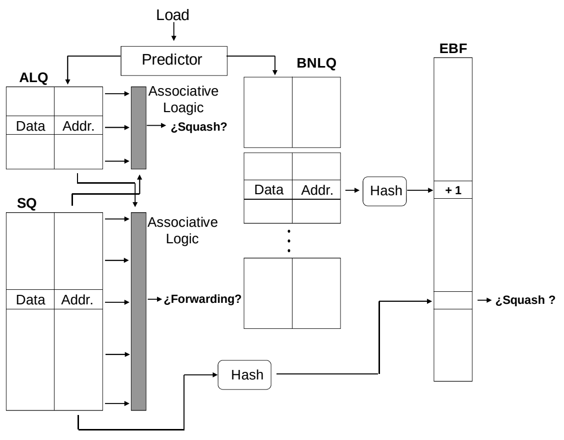

Figure 2 – Split-LQ design

IMDC

The key difference between our IMDC [22][24] design and the conventional design is that we explicitly assign and track the age of loads. Conventional design allocates an LQ entry for each load at dispatch time in program order thereby implicitly encoding the age information within the position of the entry. IMDC, by explicitly encoding and tracking the age, does not need to allocate an entry for every load at the early stage of dispatch. Instead, we use a simple hash table –named *load table*- that replaces the associative LQ. Upon execution, (see Figure 3) each load uses the address to hash into the table and records its age. When a store executes, the address is also used to hash into the table. If the age recorded in the entry is younger than that of the store, then the load and store executed out of order with respect to each other, so a potential memory dependence violation occurred. To enforce sequential semantics, a replay is needed, thus we simply replay from the instruction following the store in program order.

Our design is conceptually close to [14], since our load table is similar to Stone's MDT. Nevertheless, our table is much simpler. Thus, the MDT needs to employ associative hardware to mitigate the effect of collisions.

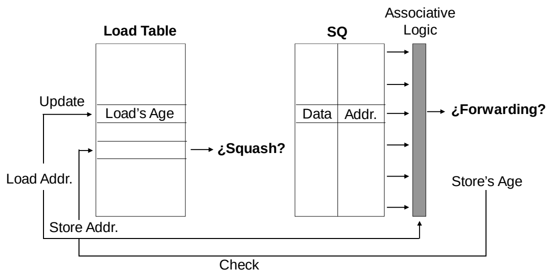

Figure 3 – IMDC design

DMDC

This design [23][24] replaces the LQ with an address-based table in which only write a very small fraction of all stores. The scheme works in a decoupled way, so that in the first step an age-based filtering is performed. To do that we employ just a simple register, named YLA, comparing the age of each store upon execution with the age of the youngest issued load stored in the mentioned YLA. As result of this process, store instructions potentially dependent with some in-flight load are marked. Furthermore, using this YLA register, a instruction's window containing potential offending loads is delimited. In the second phase (previous to commit time), simple addresses comparisons are established between both kind of instructions. For that purpose, a small and very simple hast table is used. In case of conflict in the table, a potential memory dependence violation is detected and the corresponding re-execution of offending instructions in triggered.

Our filtering effect is similar to [8], but our YLA significantly ouptperforms the Setumadavan's results. Other mechanisms propose to delay the dependence checking to commit phase too, but all of them imply a load re-execution prior to retirement, something that does not occur in our design. Only the SVW [19] exhibits certain similarity with our work, but the nature of triggered windows is significantly different.

5. CONCLUSIONS

Many proposals to simplify the management and the hardware of conventional LSQ have been introduced recently. We have reviewed the main techniques of memory dependence prediction, the schemes that reduce the check frequency through clever filtering and the mechanisms that modify substantially the conventional structure of the LSQ. Finally, we have recapped our own contributions.

ACKNOWLEDGMENTS

This work has been supported in part by Spanish government through the research contract CYCIT-TIN 2005/5619, Consolider Ingenio 2010 2007/20811 and the Hipeac2 European Network of Excellence.

6. REFERENCES

[1] J. Tendler, J. Dodson, J. Fields, H. Le and B. Sinharoy, “Power4 System Microarchitecture”, IBM Journal of Research and Development, Vol 46, No. 1, 2002, pp. 5-25.

[2] R. Kessler, “The Alpha 21264 Microprocessor”, IEEE Micro, Vol. 9, No. 2, 1999, pp. 24-36.

[3] A. Moshovos, S. Breach, T. Vijaykumar and G. Sohi. “Dynamic Speculation and Synchronization of Data Dependences”. In Int’l Symp. on Computer Architecture, 1997, pp. 181-193.

[4] G. Chrysos and J. Emer. “Memory Dependence Prediction using Store Sets”. In Int’l Symp. on Computer Architecture, 1998, pp. 142-153.

[5] S. Subramaniam and G. Loh. “Store Vectors for Scalable Memory Dependence Prediction and Scheduling”. In Int’l Symp. on High-Performance Computer Architecture, 2006, pp. 65-76.

[6] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura and S. Tomita. “A High-Speed Dynamic Instruction Scheduling Scheme for Superescalar Processors. In Int’l Symp. on Microarchitecture, 2001, pp. 225-236.

[7] C. Fang, S. Carr, S. Onder and Z. Wang. “Feedback-Directed Memory Disambiguation through Store Distance Analysis”. In Int’l Conference on Supercomputing, 2006, pp. 278-287.

[8] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, S. W. Keckler. “Scalable Hardware Memory Disambiguation for High ILP Processors”. In Int’l Symp. on Microarchitecture, 2003, pp. 399-410.

[9] B. Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors”, Communications of the ACM, Vol. 13, No. 7, 1970, pp. 422-426.

[10] I. Park, C. L. Ooi, T. N. Vijaykumar. “Reducing Design Complexity of the Load-Store Queue”. In Int’l Symp. on Microarchitecture, 2003, pp. 411-422.

[11] T. Sha, M. M. K. Martin, A. Roth. “Scalable Store–Load Forwarding via Store Queue Index Prediction”. In Int’l Symp. on Microarchitecture, 2005, pp. 159-170.

[12] L. Baugh and C. Zilles, “Decomposing the Load-Store Queue by Function for Power Reduction and Scalability”, IBM Journal of Research and Development, Vol. 50, No. 2-3, 2006, pp. 287-298.

[13] A. Roth. “A High-Bandwidth Load-Store Unit For Single- and Multi- Threaded Processors”. Technical report (CIS), Development of Computer and Information Science, University of Pennsylvania, 2004.

[14] S. S. Stone, K. M. Woley and M. I. Frank. “Address-Indexed Memory Disambiguation and Store-to-Load Forwarding”. In Int’l Symp. on Microarchitecture, 2005, pp. 171-182.

[15] H. Akkary, R. Rajwar and S. Srinivasan. “Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors”. In Int’l Symp. on Microarchitecture, 2003, pp. 423-434.

[16] E. Torres, P. Ibañez, V. Viñals and J. Llaberia. “Store Buffer Design in First-Level Multibanked Data Caches”. In Int’l Symp. on Computer Architecture, 2005, pp. 469-480.

[17] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger and S. W. Keckler. “Late-Binding: Enabling Unordered Load-Store Queues. In Int’l Symp. on Computer Architecture, 2007, pp. 347-357.

[18] H. W. Cain and M. H. Lipasti. “Memory Ordering: a Value-Based Approach”. In Int’l Symp. on Computer Architecture, 2004, pp. 90-101.

[19] A. Roth. “Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization”. In Int’l Symp. on Computer Architecture, 2005, pp. 458-468.

[20] S. Subramaniam and G. Loh. “Fire-and-Forget: Load-Store Scheduling with no Store Queue”. In Int’l Symp. on Microarchitecture, 2006, pp. 273-284.

[21] F. Castro, D. Chaver, L. Piñuel, M. Prieto, M. Huang and F. Tirado “Load-Store Queue Management: an Energy-Efficient Design Based on a State-Filtering Mechanism”. In Int’l Conference on Computer Design, 2005, pp. 617-624.

[22] A. Garg, F. Castro, M. Huang, L. Piñuel, D. Chaver and M. Prieto. “Substituting Associative Load Queue with Simple Hash Table in Out-of-Order Microprocessors”. In Int’l Symp. on Low-Power Electronics, 2006, pp. 268-273.

[23] F. Castro, L. Piñuel, D. Chaver, M. Prieto, M. Huang and F. Tirado “DMDC: Delayed Memory Dependence Checking through Age-Based Filtering”. In Int’l Symposium on Microarchitecture, 2006, pp. 297-308.

[24] F. Castro, R. Noor, A. Garg, D. Chaver, M. Huang, L. Piñuel, M. Prieto and F. Tirado. “Replacing Associative Load Queues: a Timing-Centric Approach”. To appear in IEEE Transactions on Computers, 2008.