Substituting Associative Load Queue with Simple Hash Tables in Out-of-Order Microprocessors*

Alok Garg†, Fernando Castro, Michael Huang†, Daniel Chaver,
Luis Piñuel, and Manuel Prieto

†University of Rochester and Universidad Complutense Madrid
†{garg, huang}@ece.rochester.edu, fcastror@fis.ucm.es, {dani02, lpinuel, mpmatias}@dacya.ucm.es

ABSTRACT

Buffering more in-flight instructions in an out-of-order microprocessor is a straightforward and effective method to help tolerate the long latencies generally associated with off-chip memory accesses. One of the main challenges of buffering a large number of instructions, however, is the implementation of a scalable and efficient mechanism to detect memory access order violations as a result of out-of-order scheduling of load and store instructions. Traditional CAM-based associative queues can be very slow and energy consuming. In this paper, instead of using the traditional age-based load queue to record load addresses, we explicitly record age information in address-indexed hash tables to achieve the same functionality of detecting premature loads. This alternative design eliminates associative searches and significantly reduces the energy consumption of the load queue. With simple techniques to reduce the number of false positives, performance degradation is kept at a minimum.

Categories and Subject Descriptors

C.1.0 [**Processor Architectures**]: General

General Terms

Design, Experimentation, Measurement

Keywords

LSQ, Memory disambiguation, Hash table, Scalability

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. Increasing the number of in-flight instructions, however, requires scaling up different microarchitectural structures. This can cause significant increase in energy consumption, especially if the structure is accessed associatively. One such example is the logic that enforces correct memory-based dependences, generally referred to as the load-store queue (LSQ) and implemented as two separate queues: the load queue (LQ) and the store queue (SQ). Conventional implementations of these queues are age-based, containing complete addresses. Memory instructions need to update one queue and associatively check the other. The associative search operation is a major concern for the scalability of these queues as not only energy consumption increases, the latency of accesses also worsen with the increase of queue size and may present complications in the logic design. As such, unconventional implementations that avoids associative searches should be explored.

In this paper, we focus on the LQ and propose an alternative implementation: instead of explicitly expressing the full address in the queue and implicitly encoding the age information in the allocation order of the entries, we explicitly maintain the age information and implicitly track the address via a hash table. The resulting structure requires only index-based access and thus can be much more energy-efficient and easy-to-scale. We do, however, incur false positive memory order violation detections and thus extra replays. We show that with a few very simple mitigation techniques, the number of replays can be drastically reduced. Furthermore, because our implementation does not place a limit on the number of in-flight loads, we can reduce the chance of stalling processor because the LQ is full. This can offset the performance degradation caused by extra replays. We show that our design can drastically reduce the energy consumption of the LQ at a slight performance cost: on average, about 85% of LQ energy is saved. Performance degradation is around 1% compared to conventional design with optimally sized LQ ignoring any latency issues of a large LQ. Overall, processor-wide energy sayings range from 1-4%.

The rest of the paper is organized as follows: Section 2 discusses the issues of the conventional LQ design and our alternative implementation; Section 3 describes our experimental methodology; Section 4 presents quantitative evaluation of our design; Section 5 summaries related work; and Section 6 concludes.

2. TRACKING LOADS WITH HASH TABLES

2.1 Conventional design

In a typical out-of-order core, the result of a store instruction is not released to the memory hierarchy until the store commits so as to easily support precise exception. A

*This work is supported in part by National Science Foundation under the grant 0509270, the Spanish government through research contract TIN 2005-05619, and by the Hipeac European Network of Excellence.

ISLPED'06, October 4–6, 2006, Tegernsee, Germany.

load instruction, on the other hand, is allowed to access the memory hierarchy as soon as the address is available. This speculative access is premature if a store older in program order executes later and modifies (part of) the data being loaded. In conventional implementations, when a store executes, its age and address information is compared to that of the loads to find out whether there is any premature load. The LQ implicitly stores the age information by using an age-ordered queue. If the portion of the queue belonging to loads younger than the store registers a match, a store-load replay is triggered [7]. Although the high-level concept of the LQ is not overly complex, practical designs have to deal with various issues such as handling of partial overlap between loads and stores. The (physical) address is explicitly stored in the LQ and compared. Unfortunately, address is a very wide operand (more than 32 bits in current generation products) and the width continues to grow. Searching the queue associatively is not only slow but also power-consuming.

2.2 Hash table-based tracking

We propose an alternative design that eliminates associative search and the capacity issue of the LQ. The first key difference between our design and the conventional design is that we explicitly assign and track the age of loads. Recall that to determine whether a store-load replay is needed requires two pieces of information: address and age. Conventional design allocates an LQ entry for each load at dispatch in program order thereby implicitly encoding the age information within the position of the entry. By explicitly encoding and tracking the age, we are no longer bound to perform entry allocation at the early stage of dispatch and therefore can choose from a much wider variety of implementation options. For example, the LQ can be split into multiple smaller ones that are word interleaved.

The second difference is that we use the simple, oft-used indexing table or a hash table to avoid associative comparison of address: each load, upon execution, will use the address to hash into the table and record its age and optionally some address information. (We refer to this table as the load table.) Because the hashing already carries some address information, we only need to record partial information about address – just like the way cache tag keeps partial information. However, one important difference between maintaining a cache and keeping track of loads for order violation detection is that we do not necessarily need to keep the full address. If there is a partial address match between a store and a younger load, we can conservatively assume an address match has occurred. This only affects the efficiency of the system, not correctness. This observation leads to the third difference: in our design, we choose not to record any additional address information (beyond hashing). When two addresses map to the same entry in the load table, we simply assume that they are accessing the same address.

Clearly, multiple loads can hash into the same entry of the load table. We simply keep the age of the *youngest* load. This is because the *existence* of an order violation is all-important, whereas the identity of the load(s) involved is dispensable. Clearly, keeping the age of the youngest load is sufficient to detect the existence of order violation. When a replay is needed, we can simply replay from the instruction following the store in program order – rather than starting from the oldest load among all triggering loads. In fact, trying to identify the oldest load that needs to replay requires additional circuit complexity existing processors such as the IBM POWER 4 chose to avoid [16]. It, too, replays from the store onward.

To summarize, when a load executes, it accesses the load table based on the address and if the age currently stored in the entry is older, it updates the entry with its own age, otherwise, the entry remains unchanged. When a store executes, it similarly accesses the table to read the current age. If the age is younger its own age, a replay (from the instruction following the store) is triggered.

Representing age: To represent the age of memory instructions, we simply take their ROB ID and augment it with a 2-bit prefix to handle wrap-arounds. This prefix increments every time the write (dispatch) pointer of the ROB wraps around. Because at any moment two in-flight instructions can only differ by one in the prefix, we cycle the prefix from 1 to 3 and when comparing two age IDs, the prefix part follows the fixed rule $1 < 2$, $2 < 3$, and $3 < 1$.

When the read (commit) pointer of ROB wraps around, all entries in the load table with the old prefix represent instructions that are committed. We clean up those entries with the old prefix by flash-resetting their prefix to 0 to indicate invalid age. If we do not perform the clean-up, these age IDs will be misinterpreted as future ages. Note that depending on the actual implementation, the flash reset can take multiple cycles. During this clean-up action, we can not start assigning new prefixes, *i.e.*, the ROB write pointer can not wrap around. This guarantees that at any moment, there are only two neighboring prefixes in the load table.

Handling coherence and consistency: The LQ also serves the purpose of maintaining load-load ordering for coherence and consistency. First off, a cache-coherent design requires write serialization: all writes to the same location have to appear in the same order to all processors. As a result, if a younger load obtained some data, subsequently, the data is updated by a store from another processor, then an older load can not obtain the new data. Note that write-serialization is quite a standard requirement even in a uniprocessor environment. This is because even in uniprocessor systems, DMA can also assume the role of a bus master and write to the memory. To simplify software, (almost) all current commercial processors maintain cache coherence with DMA, which implies write serialization.

In a conventional design, an invalidation searches through the entire LQ to find matching addresses. When matches are found, they are marked in the LQ. Every load, upon execution, also searches in the LQ associatively (just like a store). If there is a match of the address with a younger load whose entry is marked (by an invalidation), this suggests that the younger load has consumed the old data and therefore a load-load replay is triggered [7].

The guarantee of write serialization can be supported by the hash table implementation as well. An invalidation will also hash into the load table and mark the entries corresponding to the invalidated addresses. When a load executes, we inspect the invalidation mark in the entry. If the mark is set and the age recorded in the table is younger than that of the current load, we trigger a load-load replay. Note that for load instructions, we already need to perform a read of the hash table to determine whether the age needs to update. Thus, checking of load-load replay is essentially for free in our design. Similar to the store-load replay, we can not pin-point the identity of the younger loads and have to

replay from the instruction following the older load. Again, such a more conservative range of replay is already adopted in existing processors for circuit simplicity [16].

LQ can also be used to ensure that load speculation does not violate the memory consistency model. For example, in MIPS R10000, which implements sequential consistency, loads actually execute, speculatively, before they are allowed to by the consistency model. To guard against mis-speculation, an external invalidation searches the LQ and marks matching entries. When the marked loads reach the commit stage, a "soft" exception – essentially a replay – occurs. This type of replay can also be implemented by the load table. However, because the load table does not provide the identity of loads, only the existence of a match, we can only conservatively replay from the oldest load instruction still in-flight if there is a match. While this would be functionally correct, it is perhaps an inefficient implementation. In this paper, we focus on uni-processors. We leave the study of our design in multiprocessor domain and potential optimizations in that environment to future work.

2.3 Handling multiple data sizes

Unfortunately for memory order tracking, accesses come at different sizes: from byte to 8 bytes or even larger. This creates a challenge to accurately and efficiently track accesses: If tracking happens at a fine granularity (*e.g.*, byte), a wider access needs to simultaneously mark multiple entries, clearly energy-inefficient and complicated circuit-wise. If, on the other hand, tracking is done at a coarse granularity (*e.g.*, 8-byte word), we lose the ability to distinguish two finer-grain accesses to two different portions within the same data chunk. This can potentially result in pathological scenario and create numerous spurious replays: If a loop performs a read-modify-write cycle on an array of bytes, and the issue logic favors loads over stores, then there can be many re-orderings of stores and loads from consecutive iterations to neighboring bytes. These would be incorrectly construed by the coarse-grain tracking logic as ordering violation to the same memory location.

To handle multiple access widths efficiently and without complicated circuitry, we use a main table to track the predominant access width (64 bits, or quad-word in our experimental system) and use a "side" table to handle other widths. Each entry in the main table contains two bits to encode the current tracking width. When a load-quad-word instruction executes, it only accesses the main table. When a load instruction with narrower width executes, it first accesses the main table, if the entry is invalid (the prefix part of the age is 0), it will set the entry's width field to its own width and proceed to the side table to update it. If the main table's entry already has a valid age, then that age will be updated if the load's age is younger. (This way, the main table *always* contain the age of the youngest load that accessed *any* part of the quad-word.) Comparing the entry's existing width and that of the load, there are two possible cases: (a) the entry's width is the same or wider than that of the load, or (b) the entry's width is narrower.

In case (a), because we are already tracking at a coarser granularity, finer-grain information is useless. So we simply "upgrade" the width of the load to the same as the entry's and proceed to the side table if necessary. For example, if the entry's width is double word (4 bytes), then we take the load's address, ignore the last two bit (to get double-word address) and use that to hash into the side table and update the age if necessary.

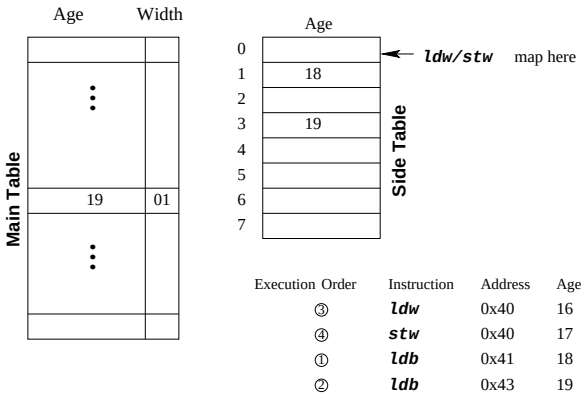


Figure 1. Example of table update with width and age upgrades. *ldw*, *stw*, and *ldb* stand for load-word, store-word, and store-byte, respectively. Example shows an 8-entry side table with a hashing function that takes the 3 least significant bits of byte, word, or double-word addresses, depending on the access width.

In case (b), because we do not want the complexity of having to check multiple entries, we forgo the fine-grain information we have accumulated so far and upgrade the entry's width to that of the load and proceed to update the side table. If the upgraded width is quad-word, the side table is not accessed. Before we access the side table, we upgrade the load's age to the elder of its own age or that of the entry in the main table. The reason to upgrade the age is best explained by an example. In Figure 1, we show a sequence of 4 instructions in which the two younger *ldb* instructions execute first. The figure shows the state of the two tables after their execution. When *ldw* executes, the width of the main table entry will upgrade, indicating that the side table will start to track this quad-word at word granularity instead. *ldw* will map into entry 0 of the side table and the new age needs to reflect all accesses to the entire word starting at address 0x40. Rather than traversing multiple entries of the side table to figure out the new age, we adopt the simple, if conservative, solution: to take the age in the main table which reflects the age of the youngest load accessing anywhere in the quad-word. In fact, all the (consecutive) entries in the side table that correspond to the quad-word need to be updated to at least the age of the load. Finally, we note that, an alternative design is that whenever a width upgrade takes place, we always directly upgrade to quad-word. This simplifies the hardware at the expense of (negligible) increases in the replay rate.

When a store executes, it compares its age to that in the corresponding entry in the main table. If the store's age is younger, then no replay is needed. Otherwise, we may need to replay. However, if the store has a narrower width, we may have more fine-grained tracking information from the side table that can rule out violation. Depending on the width of the store and that of the entry, there are also two cases: (a) the store's width is the same or narrower than the corresponding entry's in the main table, or (b) the store's width is wider. In case (a), we simply treat the store as a wider access (with the same width as the entry) and consult the side table to determine whether we replay. In case (b),

though the side table contains the age information, it does so in a "fragmented" way, and we need to access the side table multiple times to find out the age of the youngest load that overlaps with the store. To avoid complexity, we do not do so. We simply ignore the side table in this case.

In summary, the side table essentially provides some extra space to allow us to "zoom in" and track select quad words at a finer granularity. At any time, a single quad-word is tracked with only one data width. However, that width differs from quad-word to quad-word. The entries in the side table, therefore, are tracking different widths. Hashing conflicts may incur spurious replays, but does not affect correctness.

2.4 Mitigating the effect of pollution

Because of hashing conflicts, we may have spurious replays that an age-ordered LQ recording full address does not have. We also have another and more serious source of spurious replays – table pollution. Recall that updates to the load tables (main table and side table) occur at execution time when the instructions are speculative and may be on the wrong path. Later on, when a recovery of misprediction or replay takes place, the processor rolls back and starts to re-assign older ages to memory instructions on the right path. However, the tables already have many (incorrect) future ages making apparent order violation very likely. Unfortunately, we can not easily undo the updates from the load tables. It is certainly possible to maintain a log of updates and walk through the log to undo updates after a recovery or a replay. However, it is clearly inefficient and undesirable.

We propose a technique to mitigate the effect using a simple, global register. This register can be thought of as a degenerate main table that has only one entry. When a load executes, if its age is younger than that recorded by the register, the register is updated. Because this degenerate table has only one entry, it is very easy to roll back the age to that of the branch upon branch misprediction recovery, or to that of the first instruction to be replayed. If this single register can rule out the possibility of order violation, we do not check the larger tables. This actually has a filtering effect that reduces the energy consumption of accessing the bigger tables.

In practical implementations, we are not limited to a single register. We can use a small number of registers (essentially making a table of a few entries). Intuitively, we get diminishing returns as we increase the number registers. In our limited exploration, adding a second registers can further reduce the number of spurious replays and seems to be a good choice. In this paper, we use a pair of such registers.

Alternatively, we can use a monotonically increasing counter (with wrap-arounds) to replace the ROB ID as the age. With such an age mechanism, when a misprediction recovery or replay happens, we continue to increment age counter rather than reuse already-assigned age IDs and this can significantly cut down spurious replays due to table pollution. The disadvantage is the extra bits needed to store the age. In contrast, ROB ID is already used by the issue logic to update instruction execution status and thus an ROB ID-based age representation is nearly for free. However, as we will show later, the improvement in replay reduction is significant and could easily justify the extra cost in the design.

3. EXPERIMENTAL SETUP

We evaluate our scheme using a heavily-modified SimpleScalar [3] 3.0d tool set with Wattch extension [2]. The Wattch model is extended to include the energy consumption of the hash tables. Besides implementing separate ROB, issue queue, and register files, we have added support to more faithfully model modern microprocessors. Specifically, we allow speculative issue of load instructions when there are prior unresolved stores; we faithfully model store-load replays [7] (instructions are re-fetched and re-issued); we model load rejections [16]; we issue dependent instructions of the load speculatively (assuming a cache hit) and perform scheduler replays [7] if the load misses in the cache.

Our simulator models only uniprocessor and we do not have a source to faithfully model invalidation messages in a uniprocessor. Thus we do not model load-load ordering search in the LQ. This favors the conventional design by reducing the energy expenditure – for our hash table-based design, this checking is for free.

To evaluate our design we use three processor configurations (config1, config2, and config3) as shown in Table 1. *Config1* closely mimics POWER 4 processor whereas *Config2* and *Config3* are scaled-up versions of config1. To focus on energy consumption of the baseline load queue we size it optimally relative to ROB size for all three configurations.

We use highly optimized alpha binaries of all 26 SPEC CPU2000 benchmarks. We simulate 500 million instructions after fast-forwarding 1 billion instructions.

Processor core			
Issue/Decode/Commit width	8 / 8 / 8		
Functional units	INT 8+2 mul/div, FP 8+2 mul/div		
Branch predictor	Bimodal and Gshare combined		
- Gshare	8K entries, 13 bit history		
- Bimodal/Meta table/BTB	4K/8K/4K (4 way) entries		
Branch misprediction penalty	at least 7 cycles		
Memory hierarchy			
L1 instruction cache	64KB, 1-way, 128B line, 2 cycles		
L1 data cache (2 ports)	32KB, 2-way, 128B line, 2 cycles		
L2 unified cache	1MB, 8-way, 128B line 15 cycles		
Memory access latency	120 cycles		
	Config1	Config2	Config3
Issue queue (INT, FP)	(32, 32)	(48, 48)	(64, 64)
ROB	128	256	512
Register (INT, FP)	(100, 100)	(200, 200)	(400, 400)
LSQ(LQ,SQ) 2 search ports	80 (48,32)	144 (96,48)	256 (192,64)
Table size (main, side)	(512, 128)	(1024, 256)	(2048, 512)

Table 1. System configuration.

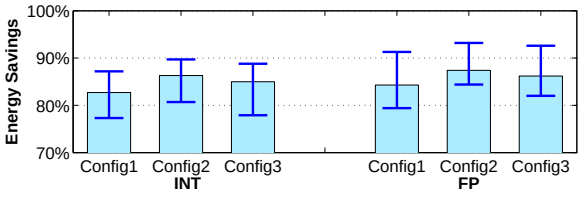
4. EVALUATION

In this section, we perform some quantitative analysis on the proposed design and the optimizations. For brevity, we report data in a compact format. We group applications into integer (INT) and floating-point (FP) groups. All metrics are first normalized to the result of the conventional configuration (baseline) and then averaged within the groups. We show the average in solid bars and the minimum and maximum for the group in superimposed I-beams.

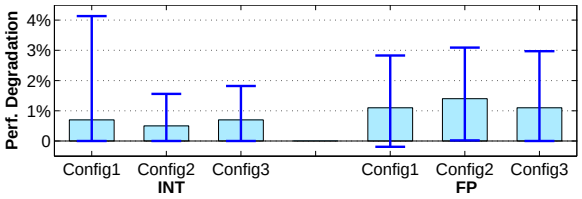
4.1 Energy impact of hash table-based tracking

We first inspect energy savings on the LQ. Intuitively, energy reduction will be significant because we no longer have a wide operand CAM structure and under most circumstances, each access consists of just one main table access. We see in Figure 2-(a) that this is indeed the case. Across configurations and applications, the energy savings consistently fall in the range of 77-94%. The averages are about 83-88% depending on the configuration. Recall that if load-load order

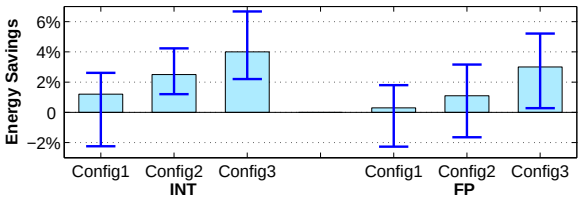
checking is performed, the baseline could spend even more energy.



(a) LQ Energy Savings.



(b) Performance Degradation.



(c) Total Energy Savings.

Figure 2. Average energy savings from the LQ (a), performance degradation (b), and processor-wide energy savings (c) over the conventional systems.

Next, we look at performance degradation. The performance is less straightforward to compare because the difference is sensitive to the LQ size of the baseline. Here we size the LQ in the baseline system to keep stalling due to LQ-fillup very low. Reducing the LQ size even moderately would result in noticeable slowdown for the baseline configuration, especially for floating-point applications. In the scaled-up configurations, we use ever larger LQs, ignoring any practicality issues. With these in mind, we can see that slowdown induced by our design is very small, about 1%. Even the worst-case slowdown is insignificant. The minimum degradation is almost always 0%. The overall processor-wide energy savings are shown in Figure 2-(c). We see that for the smallest configuration (Config1), we already make a net energy gain. As we scale to larger configurations, our gain increases. This is because larger LQs are increasingly inefficient – if they can be built to start with. Note that energy consumption is spread out over different components. Thus it is expected that the drastic energy reduction in the LQ would translate into much smaller global energy savings. Furthermore, the benefit of a table-based tracking mechanism goes beyond energy savings. Large CAM structures are best avoided. Our results have clearly showed that the more sensible table-based implementation can competently and energy-efficiently track memory access order in large scale.

In these statistics, we use monotonically increasing age counter to mitigate pollution effect. If we choose ROB ID-based age and use the alternative of global register for pollution mitigation, the performance will degrade 1.1% and 0.5% on average for the integer and floating-point applications, respectively.

4.2 Understanding the optimization techniques

To understand the effect of the optimization techniques, we show the breakdown of the number of replays and the effect of the optimization techniques in Figure 3. For this analysis, we only show the averages for one configuration (Config2) while the results from other configurations are very similar. We break down the number of replays according to their type, from bottom up: true replays, false replays due to the mismatch between the access width and the tracking width, those due to hashing conflict, and finally, those due to table pollution caused by squashed instructions. We show the results of using four different configurations: using just the main table (M), using the main and the side table (MS), using both tables plus the pair of global registers (MSG), and using the age counter with main and side table (MSC).

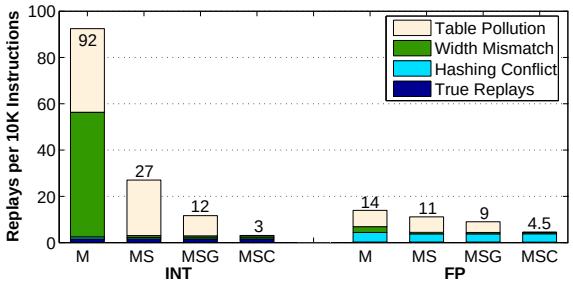


Figure 3. Breakdown of replay rates.

We see that the results are rather intuitive. (1) Integer applications tend to have many narrow-width data accesses, which cause significant number of false replays when we track access order only using the main table at quadword granularity. Floating-point applications, on the other hand, have far fewer replays due to width mismatch when we only use the main table. With the use of the side table, these false replays are almost completely eliminated. (2) Integer applications tend to suffer more from table pollution due to the higher branch misprediction rate. Floating-point applications suffer far less from table pollution, but the effect is still visible. With the global registers, we are able to remove a significant portion of false replays: about 63% and 32% for integer and floating-point applications, respectively. With age counter, this source of replay is all but completely eliminated. (3) Hashing conflict-induced replays are almost negligible in integer applications, whereas they are more noticeable in floating-point applications because the working set is generally larger. The techniques aimed at reducing other false replays have little impact on these conflict-induced replays. A better hashing function may further improve our design for floating-point applications.

5. RELATED WORK

Recognizing the scalability issue of the LSQ, many different proposals have emerged recently. A large body of work adopts a two-level approach to disambiguation and forwarding. The guiding principle is largely the same. That is to make the first-level (L1) structure small (thus fast and energy efficient) and still able to perform a large majority of the work. This L1 structure is backed up by a much larger

second-level (L2) structure to correct/complement the work of the L1 structure [1, 8, 17]. A variation of a two-level approach is the dual-queue approach where memory instructions predicted to have in-flight dependence are kept in an associative queue with conventional searching capabilities and others are kept in less power-hungry FIFOs [5, 6].

Another body of work only uses a one-level structure (for stores) but reduces check frequency through clever filtering or prediction mechanisms such as bloom filter [11, 13].

Exact memory dependence prediction is an alternative to address-based dependence-checking mechanism. Ambitious and complex dependence prediction is used in [14, 15]. Unfortunately, achieving highly accurate prediction of the actual communicating pairs of memory instructions requires a large number of tables, some of which highly-ported. Furthermore, enforcing the predicted dependence requires non-trivial support from the issue logic. Value-based re-execution presents a new paradigm for memory disambiguation. In [4], the LQ is eliminated altogether and loads re-execute to validate the prior execution. To address the energy increases due to re-execution, and the performance impact due to increased memory pressure, various filters are developed to reduce the re-execution frequency [4, 12].

Cooperative disambiguation uses compiler-based analysis to rule out the possibility for certain loads to cause dependence violation and therefore allow them to completely bypass the LQ [10].

Finally, slackened memory dependence enforcement adopts a different philosophy of enforcing memory dependence. Two decoupled executions of all memory instructions are used, the leading front-end execution focuses on speed and efficiency of common-case communication and the trailing execution follows program order to ensure correctness and resets the front-end execution when it failed to correctly enforce dependence [9].

In contrast to this body of prior work, our approach still maintains the conventional address-based memory disambiguation strategy, *i.e.*, we do not rely on dependence prediction or re-execution for validation. Compared to the solutions that still use the LQ, our approach is different in that we explicitly represent age and use hashing functions to implicit represent the address. This allows us to eliminate expensive associative comparisons of wide address operands (32 bits or more) and replace them with a single comparison of the much narrower age ID (about 10 bits) for most memory accesses.

6. CONCLUSIONS

In this paper, we have proposed a novel approach of tracking memory access order violation. The design uses a hash table-based tracking methodology maintaining explicit age information in the table. This data structure provides a number of benefits: First, detecting order violation involves only table indexing and a single age comparison per table. Compared to the fully-associative matching of the much wider address operand, this drastically reduces energy consumption; Second, thanks to the absence of large CAM structures, the access latency is low and does not increase much when scaling up the tables, making them a much better choice for scalable microarchitectures; Third, the design does not place a limit on the number of in-flight load instructions that can be tracked, potentially removing a bottleneck on the instruction-buffering capability. A straightforward implementation of the design, however, does introduce spurious replays due to different data access widths and table pollution as a result of branch misprediction and replays. We have presented two effective mitigation techniques. First, we use a side table to provide finer-grain tracking of non-predominant access widths. Second, we provide two alternatives – a pair of global registers or a monotonically increasing age assignment logic – to filter out a large fraction of pollution-induced false replays. We have shown that the overall implementation is very effective: with very little slowdown, depending on the configuration, we cut the energy consumption of the LQ by an average of about 83-88%. Taking into account the energy cost of extra replays, the processor still makes a net gain in energy on average.

7. REFERENCES

[1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *International Symposium on Microarchitecture*. Dec. 2003.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*. June 2000.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[4] H. Cain and M. Lipasti. Memory Ordering: A Value-based Approach. In *International Symposium on Computer Architecture*. June 2004.

[5] F. Castro, D. Chaver, L. Pinuel, M. Prieto, M. Huang, and F. Tirado. A Power-Efficient and Scalable Load-Store Queue Design. In *International Workshop on Power And Timing Modeling, Optimization and Simulation*. Sep. 2005. Lecture Notes in Computer Science Vol. 2236(8):1-9.

[6] F. Castro, D. Chaver, L. Pinuel, M. Prieto, M. Huang, and F. Tirado. Load-Store Queue Management: an Energy Efficient Design based on a State Filtering Mechanism. In *International Conference on Computer Design*. Oct. 2005.

[7] Compaq Computer Corporation. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, Sep. 2000. Order number: DS-0027B-TE.

[8] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *International Symposium on Computer Architecture*. June 2005.

[9] A. Garg, M. Rashid, and M. Huang. Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification. In *International Symposium on Computer Architecture*. June 2006.

[10] R. Huang, A. Garg, and M. Huang. Software-Hardware Cooperative Memory Disambiguation. In *International Symposium on High-Performance Computer Architecture*. Feb. 2006.

[11] I. Park, C. Ooi, and T. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *International Symposium on Microarchitecture*. Dec. 2003.

[12] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *International Symposium on Computer Architecture*. June 2005.

[13] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *International Symposium on Microarchitecture*. Dec. 2003.

[14] T. Sha, M. Martin, and A. Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *International Symposium on Microarchitecture*. Dec. 2005.

[15] S. Stone, K. Woley, and M. Frank. Address-Indexed Memory Disambiguation and Store-to-Load Forwarding. In *International Symposium on Microarchitecture*. Dec. 2005.

[16] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, **Vol. 46**(1):5–25, Jan. 2002.

[17] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia. Store Buffer Design in First-Level Multibanked Data Caches. In *International Symposium on Computer Architecture*. June 2005.