Load-Store Queue Management: an Energy-Efficient Design Based on a State-Filtering Mechanism*

Fernando Castro, Daniel Chaver, Luis Pinuel, Manuel Prieto, Francisco Tirado
University Complutense of Madrid
fcastror@fis.ucm.es, {dani02, lpinuel, mpmatias, ptirado}@dacya.ucm.es

Michael Huang
University of Rochester
michael.huang@ece.rochester.edu

Abstract

Modern microprocessors incorporate sophisticated techniques to allow early execution of loads without compromising program correctness. To do so, the structures that hold the memory instructions (Load and Store Queues) implement several complex mechanisms to dynamically resolve the memory-based dependences. Our main objective in this paper is to design an efficient LQ-SQ structure, which saves energy without sacrificing much performance. We propose a new design that divides the Load Queue into two structures, a conventional associative queue and a simpler FIFO queue that does not allow associative searching. A dependence predictor predicts whether a load instruction has a memory dependence on any in-flight store instruction. If so, the load is sent to the conventional associative queue. Otherwise, it is sent to the non-associative queue which can only detect dependence in an inexact and conservative way. In addition, the load will not check the store queue at execution time. These measures combined save energy consumption. We explore different predictor designs and runtime policies. Our experiments indicate that such a design can reduce the energy consumption in the Load-Store Queue by 35-50% with an insignificant performance penalty of about 1%. When the energy cost of the increased execution time is factored in, the processor still makes net energy savings of about 3-4%.

1 Introduction

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. More aggressive implementations go a step further and allow speculative execution of loads when the effective address of a preceding store is not yet resolved. Such speculative execution can be premature if an earlier store in program order writes to (part of) the memory space loaded and executes afterwards. Clearly, speculative techniques have to be applied such that program correctness is not compromised. Thus, the processor needs to detect, squash, and re-execute premature loads. All subsequent instructions or at least, the dependent instructions of the load need to be re-executed as well.

To ensure safe out-of-order execution of memory instructions, conventional implementations employ extensive buffering and cross-checking through what is referred to as the load-store queue, often implemented as two separate queues, the load queue (LQ) and the store queue (SQ). A memory instruction of one type needs to check the queue of the opposite kind in an associative fashion: 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. The logic used is complex as it involves associative comparison of wide operands (memory addresses), requires priority encoding, and has to detect and handle non-trivial situations such as operand size mismatch or misaligned data. These complex implementations come at the expense of high energy consumption and it is expected that this consumption will grow in future designs. This is because the capacity of the queues needs to be scaled up to accommodate more instructions to effectively tolerate the long, and perhaps still growing, memory latencies. Furthermore, scaling of these structures also increases their access latency, which makes it hard to incorporate them in a high-frequency design. Beyond certain limits, further increasing the capacity will require multi-cycle accesses, which can offset the benefit of increased capacity of in-flight instructions.

To address the design challenges of orchestrating out-of-order memory instruction execution, we explore a management strategy that matches the characteristics of load instructions with the circuitry that handles them. We divide loads into those that tend to communicate with in-flight stores and those that do not. This division is driven by the empirical observation that most load instructions are strongly biased toward one type or another. We use the conventional circuitry to handle the former, and a simpler alternative design for the latter. In this paper, we study both static and dynamic mechanisms to classify load instructions into the two categories mentioned above. We also explore hardware support and runtime policies in this design. We show that our design is effective, saving about 35-50% of the LSQ energy at a small performance penalty of a few percent. After factoring in the energy lost due to the slowdown, the processor as a whole still makes net energy savings. Thanks to the improved scalability of our design, we can also increase the capacity of the LSQ much more easily and at a lower energy overhead. This can mitigate the performance degradation and the energy waste due to the slowdown.

The rest of the paper is organized as follows: Section 2 describes our alternative LQ-SQ implementation; Section 3 outlines our experimental framework; Section 4 presents the quantitative analysis of the design; Section 5 discusses the related work; and finally, Section 6 concludes.

2 Architectural Design

2.1 Highlight of Conventional Design

High-performance microprocessors typically employ very aggressive strategies for out-of-order memory instruction execution. Loads access memory before prior stores have committed their data

*This work has been supported by the Spanish goverment through the research contract TIC 2002-750, the Hipeac European Network of Excellence, and by the National Science Foundation through the grant CNS-0509270.

(buffered in the SQ) to the memory subsystem. To maintain program semantics, if an earlier in-flight store writes to the same location the load is reading from, the data needs to be *forwarded* from the SQ (see Figure 1). This involves associative searching of the SQ to match the address and finding out the closest producer store through priority encoding. In most implementations, a load can execute despite the presence of prior store instructions with an unresolved address. Such *speculative load execution* may be incorrect if the unresolved stores turn out to access the same location. To detect and recover from mis-speculation, every store searches the LQ in an associative manner to find younger loads with the same address and when one is found initiates a squash and re-execution. This is referred to as a load-store replay in [8].

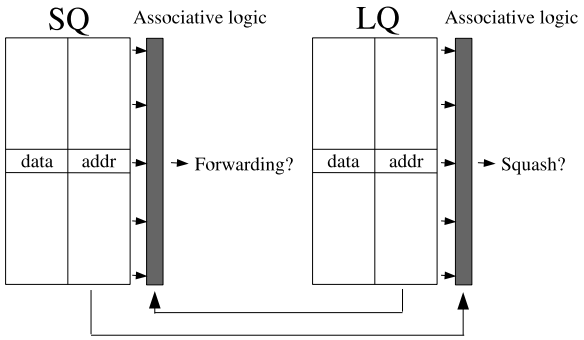

Figure 1. Conventional load-store queue design. A load checks the SQ associatively to forward data from the nearest older store to the same location. A store checks the LQ associatively to find younger loads to the same location that have executed prematurely.

In typical implementations [11, 18], a replay squashes and re-executes all instructions younger than the offending load. To reduce the frequency of replays, some processors use a simple PC-indexed table to predict whether a load will be dependent upon a previous store [8]. Loads predicted to be dependent will wait until all prior stores are resolved, whereas other loads execute as soon as their effective address is available.

2.2 Rationale

While load forwarding, bypassing, and speculative execution improve performance, they also require a large amount of hardware that increases energy consumption. In this paper we explore an alternative load-store queue design with a different management policy that allows for a significant energy reduction. This new approach is based on the following observations:

1. Memory-based dependences are relatively infrequent. Our experiments indicate that only around 12% of dynamic load instances need forwarding. This suggests that the complex disambiguation hardware in modern microprocessors is underutilized.
2. On average, around 74% of the memory instructions that appear in a program are loads. Therefore, their contribution to the dynamic energy spent by the disambiguation hardware is greater than that of the stores. This suggests that more attention must be paid to loads.
3. The behavior of a large majority of load instructions is usually strongly biased. They either frequently communicate with an in-flight store or almost never do so. This suggests that they can be identified either through profiling or through a PC-based predictor at runtime, and treated differently according to their bias.

2.3 Overall Structure

Based on the above observations, we propose a design where we identify those loads that rarely communicate with in-flight stores and handle these loads using a different queue specially optimized for them. For convenience of discussion, we loosely refer to these loads as *independent loads* and the remainder *dependent loads*.

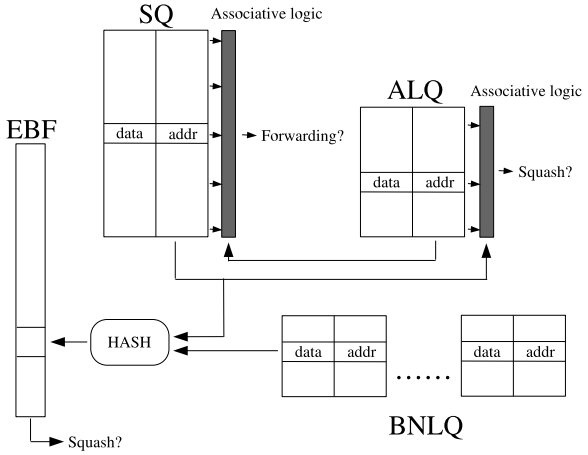

Figure 2. Design of modified LQ-SQ with filtering. The LQ is split into two queues, an Associative Load Queue (ALQ) and a Banked Non-associative Load Queue (BNLQ).

The overall structure of our design is shown in Figure 2. The conventional LQ is split into two different structures: the Associative Load Queue (ALQ) and the Banked Non-associative Load Queue (BNLQ). The ALQ is similar to a conventional LQ, but smaller. It provides fields for the address and the load data, as well as control logic to perform associative searches. The BNLQ is a non-associative buffer that holds the load address and the load data. Its logic is much simpler than that of the ALQ, mostly due to absence of hardware for performing associative searches. The banking of the BNLQ is purely for energy savings and not based on address. Therefore, architecturally, the BNLQ is just a FIFO buffer similar to the ROB (re-order buffer). Though independent loads rarely communicate with in-flight stores, we still need to detect any communication and ensure that a load gets the correct data. To do so, we use a mechanism denoted as Exclusive Bloom Filter (EBF) to allow quick but conservative detection of potential communication between an independent load and an in-flight store and perform a squash when necessary.

2.4 Using BNLQ and EBF to Handle Independent Loads

According to its dependency prediction (Section 2.5), a load is sent to the ALQ or the BNLQ. Since an independent load is unlikely to communicate with an in-flight store, we allocate an entry for it in the BNLQ that does not provide the associative search capability. Additionally, the load does not access the SQ for forwarding at the time of issue. As such, if an in-flight store does write to the same

location as the load, the data returned by the load could be incorrect and we need to take corrective measures.

To detect this situation, we use a bloom filter-based mechanism similar to that proposed in [16]. The filter we use, denoted as EBF, is a table of counters. When a load in the BNLQ is issued, it accesses the EBF based on its address and increments the corresponding counter. (In our setup, counter overflow is very rare and thus can be handled by any convenient mechanism as long as forward progress is guaranteed. For example, the load can be rejected and retried later. It can proceed when the counter is decremented or until it becomes the oldest memory instruction in the processor, at which time it can proceed without accessing the EBF.) When a store commits, it checks the EBF using its address. If the corresponding counter is greater than zero, then potentially one or more loads in the BNLQ have accessed the memory prematurely. In this case, the system conservatively squashes all instructions after the store. Note that an EBF hit can be a false hit when the conflicting load and store actually access different locations. We study a design that mitigates the impact of such false hits in Section 2.6.

To correctly maintain the counter in the EBF, when the load in the BNLQ commits, its corresponding EBF counter is decremented. Additionally, when wrong-path instructions or replayed instructions are flushed from the system, their modifications to the EBF should be undone, otherwise the "residue" will quickly "clog" the filter. This can be achieved by walking through the section of the BNLQ corresponding to the flushed instructions and decrement the EBF counter for any load that has issued. This is the solution we assume in this paper. We have also experimented with "self-cleaning" EBFs that do not actively clean out the residue. Instead, these filters tolerate the accumulation of residue and rely on periodic cleaning of the whole filter to remove any residue. For example, we can use alternating EBFs by dividing the dynamic instructions into fixed-sized blocks and use a different EBF for a different block. When all the instructions in one block have been committed, all entries in the corresponding EBF can be reset to zero, cleaning all residue. However, in our limited exploration, we found that tolerating the residue increases false hit rate of EBF and causes unnecessary flushes. The increase can be quite significant in certain applications.

Load instructions allocated into the ALQ are still handled conventionally. They check the store queue upon execution and the stores check the ALQ too. Thus, when an independent load is dispatched and the BNLQ is full, it is accommodated in the ALQ. Although, this "upgrade" increases the energy expenditure for that load instruction, it avoids stalling dispatch which not only slows down the program but also increases energy consumption. On the other hand, a load predicted to go to the ALQ is not allocated into the BNLQ when the ALQ is full. The reasoning is that there is a high likelihood that doing so would trigger a squash which is very costly both performance-wise and energy-wise. When the ALQ is full during the dispatch of a dependent load, we simply stall the dispatch.

Finally, the hashing function we use is simple: $address \ \% \ EBF\_size$. We use a prime number as the size of the EBF (4001). Through experiments, we confirmed the intuition that false hit rate is lower on average than if the size is similar but is a two's power (4096).

2.5 Dependency Prediction of Loads

In our design of the LQ-SQ mechanism, when a load instruction is decoded, it is suggested by either a dynamic or a profile-based predictor in which queue the load should be accommodated. We study two different predictors.

2.5.1 Profiling-Based Predictor

In a profiling-based system, every static load is tagged as dependent or independent based on profiling information. This way, load dependence prediction is tied to the static instructions. This is a reasonable approach because a large majority of loads have a *strongly* biased behavior: they either frequently or almost never communicate with an in-flight store. Table 1 shows the breakdown. On average, 92% of the static load instructions are strongly biased. The dynamic instances of these instructions constitute a slightly lower 88% of all dynamic load instructions.

	Static instructions		
	INT Avg	FP Avg	TOT Avg
Biased Independent Loads	75.5%	87.9%	82.4%
Biased Dependent Loads	11.8%	8.2%	9.8%
Unbiased Loads	12.7%	3.9%	7.8%
	Dynamic instances		
	INT Avg	FP Avg	TOT Avg
Biased Independent Loads	63.8%	85.4%	75.8%
Biased Dependent Loads	16.1%	8.9%	12.1%
Unbiased Loads	20.1%	5.7%	12.1%

Table 1. Breakdown of static instructions into strongly-biased dependent and independent loads and unbiased loads. The same breakdown when each load instruction is weighted by its total number of dynamic instances. Statistics are shown as the average of the integer applications, that of the floating-point applications, and the overall average. A load is considered a biased independent load when more than 99% of its dynamic instances do not depend on an in-flight store. It is considered a biased dependent load when more than 50% of its dynamic instances are dependent on an in-flight store.

In the static profiling-based prediction, we mark any load instruction whose dynamic instances have a probability of more than a threshold Th to conflict with an in-flight store in a profiling run. More precisely, if t denotes the total number of instances of a certain static load, and d denotes the number of instances where this load hashes to the same EBF entry as an in-flight store, this load is predicted as dependent if $d/t > Th$. This notion of dependency is broader than true data dependence as a load instance is considered to be dependent even when there is a false dependence: load and store to different locations map to the same EBF entry.

Note that in a realistic implementation, the profiling is likely to require simulation support as we need to obtain the microarchitectural information of how many instances of a load communicate with an in-flight store. To explore the optimal threshold Th for each application, in this paper, we simply perform brute-force search in a region and find the one that best balances energy savings with performance degradation. This exploration is discussed further in Section 4. Once an optimal threshold is selected for an application, static loads are marked in the program binary as dependent or independent based on the threshold. This, of course, implies ISA (instruction set architecture) support.

2.5.2 Dynamic Predictor

An alternative to a profile-based system is to generate dependence prediction during execution. In our approach, the prediction infor-

mation is stored in a PC-indexed table similar to the one used in Alpha 21264 [8] to delay certain loads to reduce replay frequency. This information determines which queue a load is allocated to. Initially, all loads are considered independent. This initial prediction is changed if a dependence is detected. As to what updates can be made to an entry already in "dependent" state, we consider two different policies. One possible choice, which we explored in [6], is to hold this prediction for the rest of the execution. This decision, based on the stable behavior observed in Table 1, is simple and works reasonably well. A second policy includes periodic refreshing of the table, restoring all predictions to "independent" [7]. The idea behind this policy is that even loads that rarely communicate with an in-flight store will be predicted as dependent after the first instance of an EBF match, forcing all subsequent instances of the loads to the ALQ. Using periodic refreshing, this effect is limited to within a time interval. We use this policy as it improves performance at a very small hardware cost [6]. Unlike the profiling-based approach, the dynamic alternative needs neither change in the ISA nor profiling. However, it requires extra storage and a prediction training phase after each action of refreshing of the table.

When a store finds an EBF hit with its address, there is a potential dependence between the store and a load in the BNLQ. Without the associative search capability of the BNLQ, however, we can not find the offending load the way we find it in a conventional LQ. Therefore, as mentioned earlier, we simply squash all instructions after the store. However, we need to identify the load(s) that conflict with the store so that we can train the predictor to direct them to the ALQ in the future. For this purpose, we introduce a special DPU mode (dependence predictor update mode). In this mode, after the squash, load instructions are inspected at commit time in order to find out those that triggered the squash. We save the index of the squash-triggering EBF entry and the counter value at the time of squash in two dedicated registers. When an independent load commits during the DPU mode, its EBF entry index is compared to the saved index. Upon a match, its dependency prediction is changed to "dependent" in the PC-indexed predictor. The DPU mode terminates when the number of matching loads reaches the saved count. Note that in the re-execution after the squash, the addresses of loads in the BNLQ are not guaranteed to repeat those prior to the squash – the processor may follow a different predicted path, for example. Thus, the DPU mode should also terminate by time-out.

2.6 Handling of EBF False Hits

The price for the simple and fast membership test using the bloom filter is the existence of false positives: an EBF match does not necessarily suggest a true data dependence. In our base design described above, a false dependence is treated just like a true dependence. In addition to an unnecessary squash, the cost of a false dependence also includes increased pressure on the ALQ, which needs to accommodate all loads considered dependent, truly or falsely. To avoid the cost associated with false hits, we consider architecture support to handle them differently.

When an EBF hit happens at the commit time of a store, instead of immediately squashing all subsequent instructions, we walk through the BNLQ (from the oldest load backward) to determine whether the hit is because of true dependences. This is done by reading out the address of each load and compare it with that of the store. If an address match happens, we squash the offending load and all subsequent instructions. (When using the dynamic predictor, we also update the predictor at this time. Hence ,there is no need for the special DPU mode.) If we finish the searching without finding an address match, nothing needs to be done and we avoided an unnecessary squash.

This sequential searching of the BNLQ takes non-trivial amount of time. Assuming we can only read out one load address from one bank of the BNLQ, the bandwidth of this searching is only a few loads per cycle. Thus, the process can take tens of cycles. However, this search happens in the background and during this period, normal processing continues. The only constraints are that any unchecked load in the BNLQ can not be allowed to commit and that before a search finishes, we can not start another one. In practice, these are unlikely to cause any slowdown: with a reasonable bandwidth, the checking of the BNLQ can easily outpace the retirement of instructions. If no address match is found, the latency of the search should have little if any impact on the performance. Even if a match is found, the delay in commencing the squash is not a cost as it appears: every cycle we push back the squash, we are saving a few independent loads (and other instructions in between) from being squashed, unnecessarily. The occupancy of the BNLQ is not a concern either as an EBF hit (which triggers a BNLQ search) is very rare.

3 Experimental Framework

We evaluate our proposed load-store queue design on a simulated, generic out-of-order processor. The main parameters used in the simulations as well as the applications used from SPEC CPU2000 suite are summarized in Table 2. As the evaluation tool, we use a heavily modified version of SimpleScalar [4] that incorporates our LQ-SQ model and a Wattch framework [3] that models the energy consumption throughout the processor. Profiling is performed using the *train* inputs from the SPEC CPU2000 distribution, whereas the production runs are performed using *ref* input and single sim-point regions [17] of one hundred million instructions.

Processor
8-issue out-of-order processor
Register File: 256 INT physical registers, 256 FP physical registers
Func. Units: 4 INT ALUs, 2 INT Mult-Dividers, 3 FP Adders, 1 FP Mult-Divider
Branch Predictor: Combined; Bimodal Predictor: 8K entries; 2-level: 8K entries, 13 bits history size; Meta-Table: 8K entries; BTB: 4K entries; RAS: 32 entries
Queues: INT-Queue: 128 entries, FP-Queue: 128 entries
Caches and Memory
L1 data cache: 32KB, 4 way, LRU, latency= 3 cycles
L2 data cache: 2MB, 8 way, LRU, latency= 12 cycles
L1 instruction cache: 64KB, 2 way, LRU, latency= 2 cycles
Memory access: 100 cycles
LSQ simulated configurations
Baseline LQ-SQ: LQ: 80 entries; SQ: 48 entries
Proposed LQ-SQ: BNLQ-ALQ: 32-48, 40-40, 48-32, 56-24; SQ: 48 entries; EBF: 4K entries (4 bits per entry)
Benchmarks SPEC CPU2000
Integer applications: bzip2, crafty, eon, gap, gzip, parser, twolf, vpr
FP applications: applu, apsi, art, facerec, fma3d, galgel, mesa, mgrid, sixtrack, wupwise

Table 2. Simulation parameters.

4 Evaluation

4.1 Main Results

We first present some broad-brushed comparison of our LQ-SQ design versus the conventional design. For easier analysis, we keep the overall capacity of the LQ the same in all configurations. Note that this is favoring the conventional design to a large degree as an im-

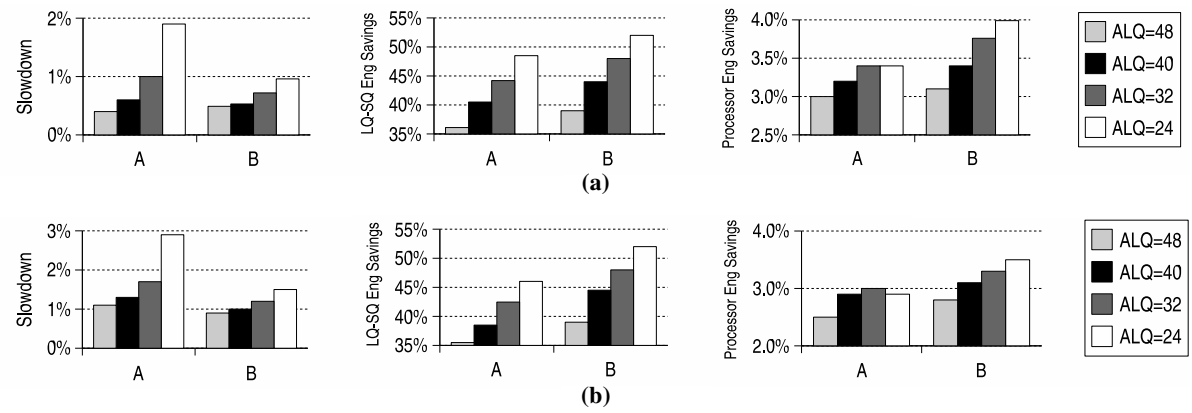

Figure 3. Performance and energy impact of proposed LQ-SQ design using a profiling-based predictor (a) or using a dynamic predictor (b). In all configurations, the combined capacity of the BNLQ and the ALQ is 80, the same as that of the LQ in the baseline conventional system. In option A, a false EBF hit is handled just like a true dependence, whereas in option B, a false EBF hit is detected through sequential searching of BNLQ and ignored.

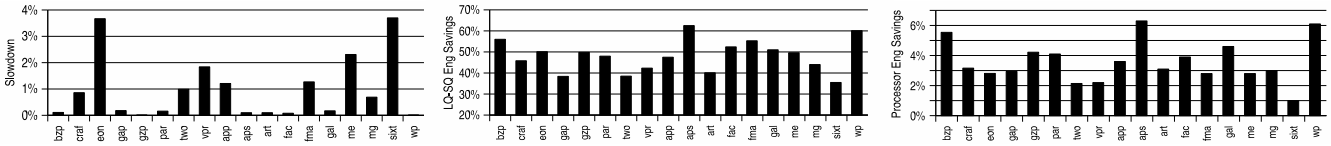

Figure 4. Performance and energy impact of the proposed LQ-SQ design on individual applications. The configuration shown uses a dynamic predictor and does not squash due to a false EBF hit (the "B" option). The size of the BNLQ and ALQ is 48 and 32, respectively.

portant benefit of our design is the scalability of the BNLQ which allows the processor to buffer more in-flight instructions to better exploit long-range ILP. Due to the tremendous amount of data, in many figures, we only show suite-wide averages. Recall that in our LQ-SQ mechanism, a baseline design treats an EBF false hit just the same as a true dependence, whereas a more advanced design sequentially searches the BNLQ upon an EBF hit and ignores a false hit. For notational convenience, we refer to these two options as A and B respectively.

We use the conventional design as the baseline of the comparison and show the slowdown, energy savings in the LQ-SQ, and processor-wide energy savings of our designs. In Figure 3-(a), we show the results of using a profiling-based dependency predictor and in Figure 3-(b), we show those of using a dynamic dependency predictor. In all cases, we explore different distributions of the total LQ capacity into the ALQ and the BNLQ.

From this figure, we can make a few observations. First, our design does achieve the goal of reducing energy consumption of the dynamic memory disambiguation mechanism without inducing much performance penalty. We see that the average performance degradation is about 1% in most configurations and that the energy savings range between 35% and 52% of total LQ-SQ energy consumption. Since energy is dissipated diversely throughout the processor, the total energy savings for the entire processor (factoring in the energy waste due to the increased execution time) are much less significant. Nevertheless, the design makes net energy savings in the processor.

Second, as we reduce the ALQ size, the energy consumption in the LQ-SQ continues to reduce. However, the slowdown also increases due to the more frequent stalls when the ALQ is full. This increased performance degradation incurs global energy waste which can negate the energy saved in the LQ-SQ. Indeed, although infrequent, memory-based dependences still exist and require efficient handling. When the resource is not sufficiently provided, the machine becomes unbalanced and thus inefficient. We have performed some experiments using a degenerated configuration without an ALQ. In this case, the average slowdown is about 19% and the energy consumption of the whole processor increases by about 9% (over the baseline conventional system) despite that the LQ-SQ energy is reduced.

Third, on balance, the profile-based predictor is slightly better. This is largely because that with meticulous tuning, the predictor is able to mark loads that occasionally communicate with an in-flight store as independent. These loads do not increase the pressure on the ALQ. In contrast, with the dynamic predictor, these loads are marked as dependent *reactively* after the squash, not avoiding the penalty, and increasing the pressure on the ALQ until a refresh. Nevertheless, the difference in the results is insignificant and probably does not justify the higher implementation cost of the profiling and the ISA support.

Fourth, special handling of EBF false hits (option B) is effective. As fewer loads are moved to the ALQ, the energy savings in the LQ-SQ increase by a few percent. More importantly, the reduction in ALQ pressure reduces the frequency of stalls due to ALQ full and thus the performance penalty. This is especially apparent when the size of the ALQ is small. For example, in the configuration with the dynamic predictor and a 24-entry ALQ, the performance penalty of using option B is about 1.5%, half as much as the 3%

penalty using option A (Figure 3-(b)). Our results also show that in option A with a dynamic predictor, even when all falsely-dependent loads are moved into the ALQ, the average false EBF hit rate is still about 28%. (This rate changes little when the size of BNLQ and ALQ changes.) When option B is used, this rate increases to about 53% because loads causing EBF false hits are still kept in the BNLQ thus increasing the chance of more false hits. Note that even with the elevated false hit rate, an EBF hit is still a very rare event. On average, one hit occurs in every 1500 committed instructions.

Per-application results In Figure 4, we zoom into one configuration and show the detailed results of individual applications. The behavior seen in this figure is quite representative across all the configurations. We see that there is variation across different applications as expected, but the variation is not significant. In the following discussion, we return to showing just the average results for brevity.

Scaling BNLQ capacity As mentioned earlier, we artificially keep the combined capacity of the BNLQ and the ALQ to be the same as that of the LQ in the baseline conventional design in order to have a cleaner contrast between the two designs. This introduces some performance degradation since a centralized resource is always more fully-utilized than a distributed one. In practical designs, however, a larger BNLQ can be used to offset the performance degradation of using distributed resources. In Table 3, we show the result of a very limited experiment where we use a slightly larger BNLQ of 80 entries together with a 32-entry ALQ. In contrast, we also show the result where the BNLQ is kept at 48 entries as in the earlier experiments.

	BNLQ=48, ALQ=32		
	INT Avg	FP Avg	TOT Avg
Slowdown	1.43%	2.26%	1.78%
Energy Savings in LQ-SQ	40.6%	43.9%	42.4%
Energy Savings in Processor	3.11%	2.85%	2.99%
	BNLQ=80, ALQ=32		
	INT Avg	FP Avg	TOT Avg
Slowdown	1.1%	-2.8%	-1.07%
Energy Savings in LQ-SQ	41.6%	46.9%	44.5%
Energy Savings in Processor	3.41%	5.17%	4.36%

Table 3. Energy and performance impact of increasing the size of the BNLQ. The experiments use the dynamic predictor and option B.

We can see that indeed even this moderate increase in the size of BNLQ can reduce the performance penalty and thus further improve processor-wide energy savings. In fact, for floating-point applications, by slightly increasing the BNLQ size, the performance is higher than that of the baseline. This is because floating-point applications tend to have a higher demand of the LQ resource and we are able to partially meet that demand by increasing the capacity of the BNLQ. As expected, with the increase in performance, the processor-wide energy savings increase even though the LQ-SQ energy savings remain largely unchanged.

4.2 Dependence Prediction

In order to compare the profile-based (static) and the dynamic dependence predictors, and to study the effect of thresholds used in the static predictor, we follow Grunwald et al. and employ the following metrics used in confidence estimation [10]:

- Predictive Value of a Positive test (*PVP*). It identifies the probability that a load dependence prediction is correct. It is computed as the ratio between the number of correctly predicted dependent loads and the total number of loads predicted as dependent.
- Predictive Value of a Negative test (*PVN*). It identifies the probability that a load independence prediction is incorrect. It is computed as the ratio between the number of mispredicted independent loads and the total number of loads predicted as independent.

In our case, using predictors with a high PVP reduces the pressure on the ALQ. This reduction translates into higher energy savings. On the other hand, if a load is incorrectly sent to the BNLQ, a squash is performed, resulting in a significant performance penalty. Therefore, in our design, only very low PVN values are acceptable.

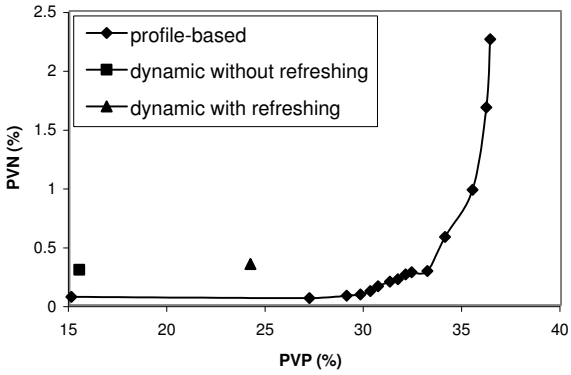

Figure 5. PVN and PVP values for profile-based and dynamic predictor. The results shown are the average values for all applications in a configuration using option A with a 48-entry BNLQ and a 32-entry ALQ. For profile-based predictor, the data points reflect different thresholds Th: 0, 0.01, 0.02, ..., 0.1, 0.2, 0.3, 0.4, 0.5.

In Figure 5, we visually present the measurements of PVP and PVN for different thresholds in the static predictor and for the dynamic predictor with and without refreshing. Intuitively, without refreshing, a dynamic predictor predicts any load that has a dependent instance as a dependent load. It is thus very similar to a static predictor with a very low threshold. We can see that this indeed is the case. Like a static predictor with 0 threshold, it is not optimal. Compared with using a higher threshold, say 0.1, a 0 threshold renders the PVP value much smaller (reduced energy saving opportunities) without reducing PVN (performance degradation) much. When refreshing is applied, the PVP for the dynamic predictor is improved significantly, while the PVN does not degrade noticeably.

4.3 Threshold Exploration in Profiling

From Figure 5 we can also see that when the threshold is above 0.1, the result of a static predictor quickly deteriorates: PVP remains much the same (or even reduces a little bit) and PVN sharply increases. Hence, in the profiling stage, we only need to explore the range between 0 and 0.1 to find the best threshold. We find that a simple algorithm is sufficient in our experiments. We start from the smallest value (0.01 in our case) and gradually increase the value. With each new threshold, we measure the performance degradation

and energy savings. We stop when the ratio between processor energy savings and performance degradation starts to reduce.

In general, as we increase the threshold, we are putting more loads into the BNLQ. This reduces the pressure on the ALQ, thus speeds up the program and saves energy. When the threshold is raised to such a level that the loads are well balanced between the two queues, further increasing the threshold will not improve performance as the gain of reducing dispatch stall due to ALQ full is canceled out by the loss due to squash of dependent loads in the BNLQ. From this point on, the more loads we put into the BNLQ, the more squashes we get. This not only slows down the program but also reduces processor-wide energy savings. Figure 6 shows a typical example that visualizes the above discussion.

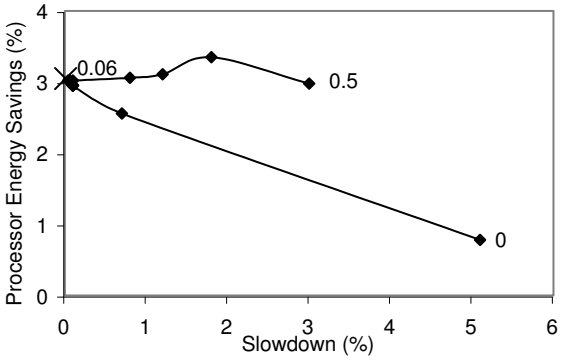

Figure 6. Threshold exploration. The example shown is the results of application *gzip* on a configuration with 40 entries in BNLQ and ALQ each and using option A. The cross in the figure shows the threshold we choose.

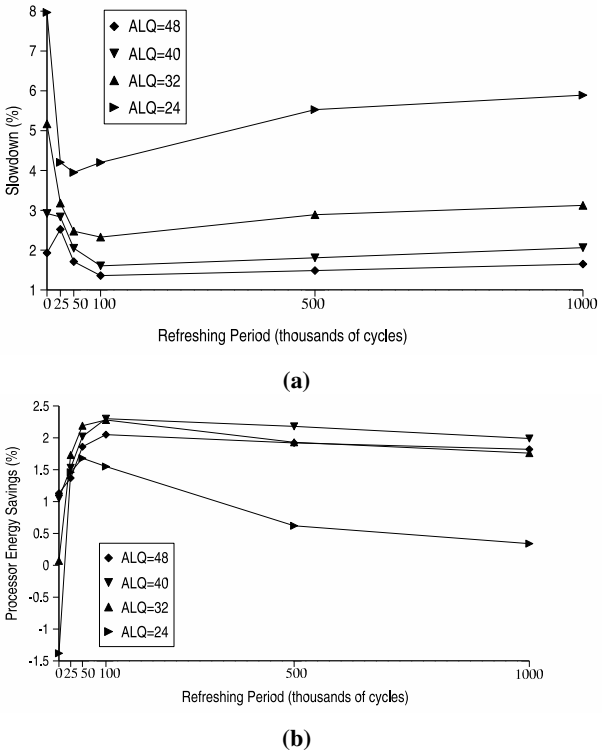

Figure 7. The impact on performance degradation (a) and processor energy savings (b) of the refreshing period in the dynamic predictor. The data shown reflect a system using option A.

4.4 Refreshing Period Exploration

As explained before, in the design using a dynamic predictor without refreshing [6], once a prediction changes to "dependent" for a load instruction, it remains that way for the duration of the program. This increases the pressure on the ALQ and makes some applications increase their execution time dramatically. With refreshing of the prediction table, loads that only occasionally communicate with an in-flight store are given a second chance to be considered independent and thus reducing the ALQ pressure. However, refreshing also forces the predictor to be retrained to recognize biased dependent loads, at a cost of one squash per load. Thus, the period of refreshing needs to be balanced between the two demands. On the one hand, the sooner (more frequent) refreshing happens, the fewer instances of those "occasionally-dependent" loads are sent to the ALQ. On the other hand, the more frequent refreshing is, the more squashes there are due to retraining. Figure 7 shows the performance and energy impact of different refreshing periods. From this figure, we can see that a period of 100,000 cycles tends to be a good choice for all configurations. This is the setting we use in the results shown earlier in this section.

5 Related Work

Recently, there has been a large body of work optimizing address-based memory disambiguation logic, namely the load-store queue. Many designs adopt 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. The L1 structure can be allocated according to program order or execution order (within a bank, if banked) for every store [1, 9, 19] or only allocated to those stores predicted to be involved in forwarding [2, 14]. The L2 structure is also used in varying ways due to different focuses. It can be banked to save energy per access [2, 14]; it can be filtered to reduce access frequency (and thus energy) [1, 16]; or it can be simplified in functionality such as removing the forwarding capability [19]. In contrast to these hierarchical designs, the two queues (BNLQ and ALQ) in our design are in parallel and the contents are mutually exclusive.

Another body of work only uses a one-level structure (for stores) but reduces check frequency through clever filtering or prediction mechanisms [13, 16]. In [16], a conservative membership test using bloom filter can quickly filter out accesses that will not find a match. In [13], only loads predicted to need forwarding checks the SQ. The safety net is for the stores to check the LQ to find mis-handled loads at the commit stage.

Memory dependence prediction is an important alternative to address-based mechanisms to allow aggressive speculation and yet avoid penalties associated with squashing [12]. The key insight is that memory-based dependences can be predicted without depending on the actual address of each instance of memory instructions and this prediction allows for stream-lined communication between likely dependent pairs.

Finally, value-based re-execution presents a new paradigm for memory disambiguation. In [5], the LQ is eliminated altogether and loads re-execute to validate the prior execution. The SQ and associated disambiguation/forwarding logic still remain. Filters are developed to reduce the re-execution frequency [5, 15]. Otherwise, the performance impact due to increased memory pressure can be significant [15].

6 Conclusions

In this paper, we have proposed a split-LQ design where the conventional associative load queue (LQ) is replaced with a smaller associative LQ (ALQ) and a banked non-associative LQ (BNLQ). Loads are processed differently and accommodated in different queues based on the prediction whether they are dependent on an in-flight store. Dependence enforcement for the ALQ is the same as in the conventional design, whereas that for the BNLQ is done through a bloom filter that is inexact and conservative but energy-efficient for the common case where there is no dependence.

For dependence prediction, we have studied the effectiveness of dynamic and profile-based predictors. A profile-based predictor is able to fine-tune the prediction threshold based on each application's characteristics and this leads to better results than a basic dynamic predictor where a load is always predicted as dependent if a past instance has been dependent on an in-flight store. However, with periodic refreshing of the predictor table done at an optimal period, the difference between a dynamic and a static predictor is small.

We have also explored reducing the impact of false hits in the bloom filter by sequentially searching the BNLQ and initiating a squash only when a true dependence is found. The main benefit of this is that loads triggering false hits do not need to be treated as a truly-dependent load and therefore contend for space in the ALQ. This optimization is especially helpful when the size of the ALQ is small.

Overall, the several design options all achieve significant energy savings in the LQ-SQ (about 35-50%) with a negligible average performance penalty of about 1%. Taking into account the energy waste due to the increased execution time, the energy savings in the whole processor are about 2.5%-4% depending on the configuration. Part of the performance penalty is because we artificially keep the combined capacity of ALQ and BNLQ to be the same as that of a conventional LQ in the comparison. When we moderately increase the BNLQ size, the performance penalty is even less and this further improves processor-wide energy savings.

References

[1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *International Symposium on Microarchitecture*, pages 423–434, Dec. 2003.

[2] L. Baugh and C. Zilles. Decomposing the Load-Store Queue by Function for Power Reduction and Scalability. In *Watson Conference on Interaction between Architecture, Circuits, and Compilers*, Oct. 2004.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, June 2000.

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

[5] H. Cain and M. Lipasti. Memory Ordering: A Value-based Approach. In *International Symposium on Computer Architecture*, pages 90–101, June 2004.

[6] 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*, Sept. 2005.

[7] F. Castro, D. Chaver, L. Pinuel, M. Prieto, and F. Tirado. Energy-Aware Load-Store Queue State Filtering. In *Conference on Design of Circuits and Integrated Systems*, Nov. 2005.

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

[9] 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.

[10] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence Estimation for Speculation Control. In *International Symposium on Computer Architecture*, pages 122–131, June–July 1998.

[11] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 9(2):24–36, Mar. 1999.

[12] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *International Symposium on Computer Architecture*, pages 181–193, June 1997.

[13] I. Park, C. Ooi, and T. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *International Symposium on Microarchitecture*, pages 411–422, Dec. 2003.

[14] 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, Sept. 2004.

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

[16] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *International Symposium on Microarchitecture*, pages 399–410, Dec. 2003.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

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

[19] 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.