NoSQ: Store-Load Communication without a Store Queue

THE NoSQ MICROARCHITECTURE PERFORMS STORE-LOAD COMMUNICATION WITHOUT A STORE QUEUE AND WITHOUT EXECUTING STORES IN THE OUT-OF-ORDER ENGINE. IT USES SPECULATIVE MEMORY BYPASSING FOR ALL IN-FLIGHT STORE-LOAD COMMUNICATION, ENABLED BY A 99.8 PERCENT ACCURATE STORE-LOAD COMMUNICATION PREDICTOR. THE RESULT IS A SIMPLE, FAST CORE DATA PATH CONTAINING NO DEDICATED STORE-LOAD FORWARDING STRUCTURES.

Tingting Sha
Milo M. K. Martin
Amir Roth
University of Pennsylvania

Store queues are a painful fixture of modern dynamically scheduled processors. Conventional dynamically scheduled processors associatively search the store queue to perform in-flight store-load forwarding. A drawback of the conventional design is associative search itself, a function whose poor scalability constrains the store queue's scalability and, in turn, the entire instruction window's scalability.

Recent work improves store queue scalability by redesigning store-load forwarding to use mechanisms other than fully associative search. For instance, a previous design by our group uses a traditional store queue but replaces associative search with speculative indexed access to a single entry.[1]

Building on our previous design as well as other techniques (see the "Recent store-load forwarding techniques" sidebar), NoSQ is a new microarchitecture that eliminates the store queue.[2] NoSQ exploits two observations. First, store-load communication patterns are predictable with high accuracy, so search is unnecessary. Second, store input data values already exist in the register file. NoSQ uses speculative memory bypassing (SMB) to convert in-window store-load communication to register communication.[3–6] SMB "short-circuits" the store-load pair in a definition-store-load-use chain by using the register map table to directly connect the definition to the use. NoSQ is not unique in its use of SMB. Prior designs used SMB opportunistically as a lower-latency complement to conventional store-queue-based forwarding, but NoSQ is the only microarchitecture that uses SMB for all in-window store-load communication.

In a conventional microarchitecture, stores execute in the out-of-order core to write their addresses and values into the store queue; without a store queue, there is no reason to execute stores in the out-of-order core. Instead, NoSQ repurposes the register read ports and address generation units formerly used to execute stores in the out-of-order core to now execute stores in

Published by the IEEE Computer Society.

order just prior to commit. This extended commit pipeline also calculates load addresses for verification, allowing NoSQ to eliminate the load queue as well.

The feature that distinguishes NoSQ from other proposed designs is a core data path that contains no dedicated in-flight store-load communication structure. As a result, NoSQ's data path can be simpler, smaller, and faster than a conventional data path. In addition to these implementation advantages, NoSQ has a slightly better instructions-per-cycle (IPC) rate than a conventional design and consumes less data cache bandwidth.

Microarchitecture overview

NoSQ uses SMB to replace the conventional store-load forwarding path in an out-of-order processor. Figure 1 shows pipeline and structural diagrams of a processor with store-queue-based forwarding and of NoSQ. Conventional processors verify load speculation—that is, detect store-load ordering violations—by associatively searching an age-ordered load queue. NoSQ performs speculation of one kind or another on every load and uses a more general verification mechanism. Conceptually, NoSQ reexecutes all loads in order prior to commit, flushing the pipeline when the value read from the data cache doesn't match the value obtained during out-of-order execution.[7] To reduce reexecution overhead, NoSQ uses a mechanism called the store vulnerability window (SVW) to filter 99 percent of would-be reexecutions.[8] To isolate the features specific to NoSQ, we assume that the base design uses SVW-filtered load reexecution to verify load speculation.

Both structural diagrams in Figure 1 show paths for a microarchitecture that executes two loads and one store per cycle. Load 1 does not communicate with—that is, it does not forward from—an older in-flight store and gets its value from the data cache. Load 2 is a communicating load. The conventional design (Figure 1a) dispatches both loads and the store to the out-of-order execution engine. The store writes its address and data to the store queue, and both loads search the store queue at execute. At commit, the

Recent store-load forwarding techniques

Many recent proposals improve the scalability or reduce the complexity of store-load forwarding. These proposals fall into three general classes of techniques. One class maintains an age-ordered store queue structure but uses partitioning, filtering, hierarchy, dependence speculation, and speculative forwarding through the primary data cache or other structures to reduce the frequency of associative store queue search or the number of entries examined per search.[1–5] The second class avoids associative search altogether by replacing the conventional age-ordered structure with a cachelike address-indexed structure.[4,6–8] Techniques in the third class maintain the simplifying age-ordered structure but use dependence speculation to replace associative search with speculative indexed access.[9,10]

NoSQ fundamentally differs from all these techniques: Rather than reducing the complexity of forwarding by optimizing the store queue, NoSQ performs store-load communication without a dedicated intermediary structure. By using speculative memory bypassing (SMB) for all in-flight store-load communication and the store-vulnerability-window (SVW) mechanism for verification, NoSQ eliminates both the store and the load queues.

References

1. L. Baugh and C. Zilles, ''Decomposing the Load-Store Queue by Function for Power Reduction and Scalability,'' *IBM J. Research and Development*, vol. 50, no. 2/3, 2005, pp. 287-298.
2. A. Gandhi et al., ''Scalable Load and Store Processing in Latency Tolerant Processors,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 446-457.
3. I. Park, C. Ooi, and T. Vijaykumar, ''Reducing Design Complexity of the Load/Store Queue,'' *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 03), IEEE CS Press, 2003, pp. 411-422.
4. S. Sethumadhavan et al., ''Scalable Hardware Memory Disambiguation for High ILP Processors,'' *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 03), IEEE CS Press, 2003, pp. 399-410.
5. S.T. Srinivasan et al., ''Continual Flow Pipelines,'' *Proc. 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS XI), ACM Press, 2004, pp. 107-119.
6. A. Garg, M. Rashid, and M. Huang. ''Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification,'' *Proc. 33rd Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 142-153.
7. S. Stone, K. Woley, and M. Frank, ''Address-Indexed Memory Disambiguation and Store-to-Load Forwarding,'' *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 05), IEEE CS Press, 2005, pp. 171-182.
8. E. Torres et al., ''Store Buffer Design in First-Level Multibanked Data Caches,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 469-480.
9. T. Sha, M. Martin, and A. Roth, ''Scalable Store-Load Forwarding via Store Queue Index Prediction,'' *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 05), IEEE CS Press, 2005, pp. 159-170.
10. S. Subramaniam and G. Loh, ''Fire-and-Forget: Load/Store Scheduling with No Store Queue at All,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 06), IEEE CS Press, 2006, pp. 273-284.

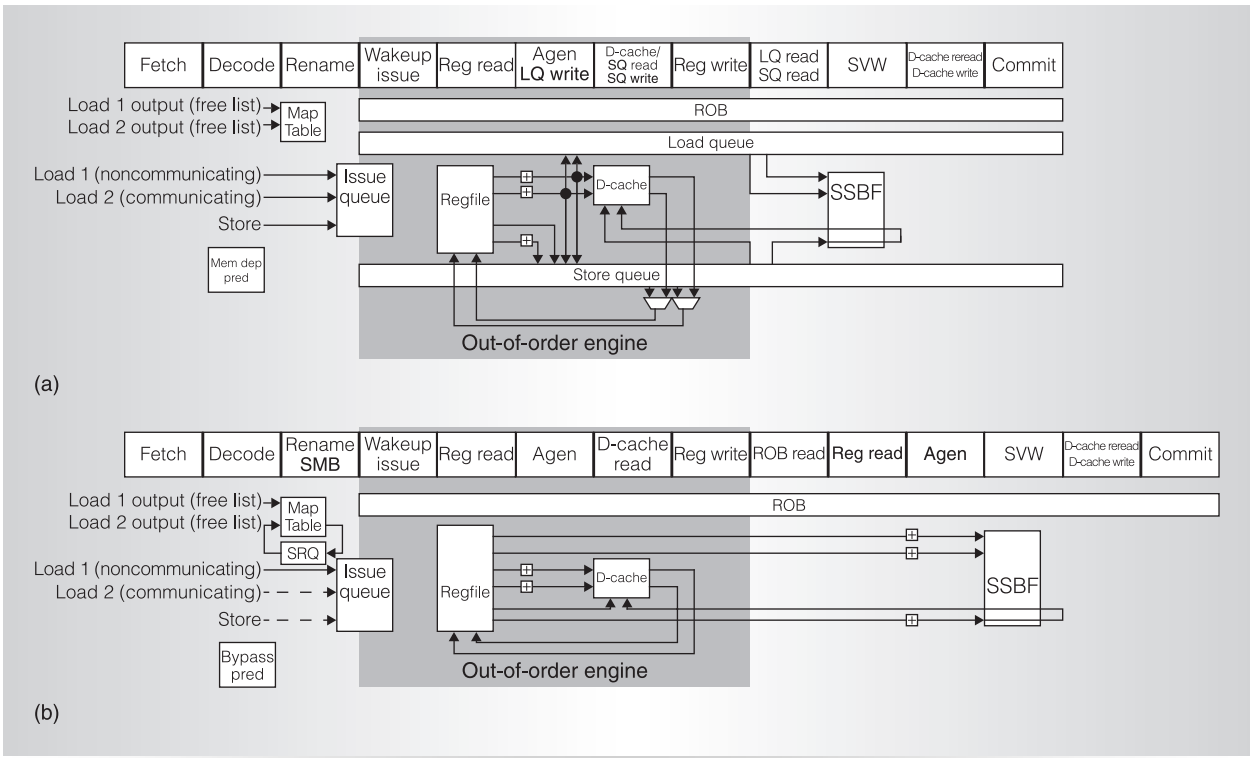

Figure 1. Pipeline and structural diagrams of a conventional design with store-queue-based forwarding (a) and of NoSQ (b). Each diagram shows paths for two loads and one store. Load 1 does not communicate with an older in-flight store, and load 2 does communicate with an older store. Note that NoSQ doesn't dispatch the communicating load to the out-of-order engine. NoSQ also doesn't dispatch any stores to the out-of-order engine. NoSQ has a simpler out-of-order core data path, but also an elongated in-order commit pipeline. Agen: address generation unit; D-cache: data cache; LQ: load queue; mem dep pred: memory dependence predictor; regfile: register file; ROB: reorder buffer; SMB: speculative memory bypassing; SQ: store queue; SRQ: store register queue; SSBF: store sequence Bloom filter; SVW: store vulnerability window;

processor uses the store queue address and data to commit stores to the data cache.

NoSQ uses a bypassing predictor—essentially an enhanced version of store sets[9] or any other memory dependence predictor—to explicitly distinguish bypassing loads from nonbypassing loads and, for bypassing loads, to identify the register that contains the value the load will produce. Bypassing loads, which in a traditional design forward from older in-flight stores, skip out-of-order execution altogether. SMB's renaming extension links their consumers to the register that contains the right value. Nonbypassing loads are injected into the out-of-order core as usual, but they simply read the data cache. Because stores don't actively participate in forwarding, the processor does not dispatch them to the out-of-order engine. As Figure 1 shows, NoSQ has a simpler out-of-order core than the conventional design. Because NoSQ lacks a store or load queue, its in-order commit pipeline accesses the register file to retrieve the base addresses and data values of stores (for commit) and loads (for verification).

NoSQ implementation

Bypassing prediction is more difficult than conventional store-load dependence prediction because bypassing connects a load to a store without first checking that their addresses match. NoSQ bypassing prediction is even more difficult because NoSQ uses bypassing for all in-window store-load communication, not just for high-confidence cases. NoSQ's predictor must generate an accurate prediction for every load; it doesn't have the option of generating no prediction when its confidence is low.

NoSQ uses a distance-based bypassing predictor, which represents a load's dependence on an older store as the distance in dynamic stores between the two instructions.[4,10] A distance-based representation is general; schemes based on store program counters (PCs) have difficulties representing dependencies involving the not-most-recent instance of a given store PC, as in the loop body x[i] = A * x[i–2]. A distance prediction can be converted to a dynamic store instance directly. In contrast, converting a store PC to a dynamic store requires a table that tracks the most recent instance of every store PC. Finally, a distance-based representation dovetails with the SVW reexecution filtering mechanism, allowing the SVW to act as both the verification mechanism and the training mechanism for the bypassing predictor.

SVW background

To determine when load reexecution is unnecessary, SVW assigns monotonically increasing store sequence numbers (SSNs) to the dynamic stores.[8] Two global counters, SSN_{commit} and $SSN_{dispatch}$, track the SSNs of the youngest committed and dispatched stores, respectively. Because SSNs are a fixed number of bits (for example, 20), the processor handles SSN wrap-around by draining its pipeline and clearing all hardware structures that hold SSNs.

The SVW mechanism uses a small, tagged, set-associative table that tracks the SSNs of the youngest stores to write to each address. This table is called the store sequence Bloom filter (SSBF). At commit, stores write their SSNs to the SSBF; that is, $SSBF[store.address] = SSN_{commit}$. When a load executes, it records the SSN of the youngest older store to which it is not vulnerable, $load.SSN_{nvul}$. If the load forwards from an older store, this is that store's SSN; otherwise, this is SSN_{commit}. In the original SVW proposal, a load skips reexecution if $SSBF[load.address] \leq load.SSN_{nvul}$. This outcome indicates that any store that wrote to the load's address either committed before the load executed or is older than the store that forwarded to the load. SVW filtering detects possible writes by other processors by creating an SSBF pseudoentry whenever the cache is forced to evict a block.

Distance-based prediction

SVW serves as a convenient basis for store distance-based dependence prediction because simple arithmetic can convert distances to SSNs—that is, to dynamic store instances—and vice versa.

For each dynamic load, NoSQ's bypassing predictor generates a predicted distance, $load.distance_{bypass}$, to the bypassing store (if any). At rename, the processor converts this distance to a dynamic store instance by subtracting it from the current global SSN_{rename} counter: $load.SSN_{bypass} = SSN_{rename} - load.distance_{bypass}$. Loads that miss in the predictor or whose predicted bypassing store has already committed (as determined by comparing $load.SSN_{bypass}$ to SSN_{commit}) are considered nonbypassing and are dispatched to the out-of-order engine as usual. Loads that hit in the predictor and whose $load.SSN_{bypass} > SSN_{commit}$ are considered bypassing and are not dispatched to the out-of-order engine. Instead, the processor sets their output register mapping to the physical register corresponding to the predicted bypassing store's data input. The processor retrieves this register from the store register queue using the low-order bits of $load.SSN_{bypass}$. The store register queue parallels a traditional store queue in structure, but it is part of the bypassing predictor, not of the out-of-order data path. It contains only physical register numbers (not addresses or values), and it is accessed only at rename, not at execute.

Training and verification with SVW

At commit, the processor uses the SSBF both to filter reexecutions for bypassing loads and to train the predictor. A bypassing load can skip reexecution if $load.SSN_{bypass} == SSBF[load.address]$, indicating that the last store to write to the load's address is the store from which the load speculatively bypassed. Unlike the inequality test used for nonbypassing loads, this equality test requires tagging the SSBF, because equality tests are not safe in the presence of aliasing. If the value obtained from reexecution

doesn't match the original value, the processor squashes the pipeline and trains the predictor with the distance to the store from which the load should have bypassed. This distance is computed as $load.distance_{bypass} = SSN_{commit} - SSBF[load.address]$.

Predictor structure

To capture path-dependent bypassing patterns, NoSQ's bypassing predictor is explicitly path sensitive. Like history-based branch predictors, it uses a hybrid design to reduce both storage requirements and training times, exploiting the fact that many loads have path-independent bypassing patterns.

NoSQ's predictor consists of two set-associative tables. One is indexed by the load's PC and generates path-insensitive predictions; the other is indexed by a combination of the load's PC and path history—one bit for each conditional branch and two PC bits for each procedure call—and generates path-sensitive predictions. Each predictor entry contains a tag and a distance field. To generate a prediction, the processor reads both tables and uses the path-sensitive prediction if one is available. Both tables are updated at commit when a bypassing misprediction is detected. A bypassing misprediction occurs when a nonbypassing load should have bypassed, a bypassing load should have accessed the cache instead, or a bypassing load bypassed from the wrong dynamic store.

Delay

Bypassing cannot handle all store-load communication. Although NoSQ can support some partial-word bypassing—which we described in our conference paper[2]—it cannot perform partial store (that is, narrow-store, wide-load) communication because it cannot combine values from multiple registers. Other communication patterns can pathologically elude the predictor—for example, path-independent but data-dependent patterns or path-dependent patterns whose differentiating signature is longer than the predictor's history. In these cases, to avoid bypassing mispredictions—which cause costly pipeline squashes—NoSQ effectively converts a would-be bypassing load to a nonbypassing load by dispatching it to the out-of-order engine and delaying its execution until the uncertain store commits. At that time, the load retrieves its value from the data cache.

NoSQ implements delay by attaching a short confidence counter to each predictor entry. A prediction with subthreshold confidence causes the load to wait for the corresponding store to commit rather than bypassing from that store. The processor updates the confidence at commit. To allow a low-confidence load to become a high-confidence load, all loads—even loads delayed because of low confidence—update the confidence counters. They increment the counters for correct predictions and decrement them for incorrect predictions.

Extended commit pipeline

In a traditional microarchitecture, the store queue buffers store addresses and data values for in-order commit. In a microarchitecture with in-order load reexecution, the load queue buffers load addresses and data values for in-order verification. NoSQ simplifies the out-of-order core by eliminating the store and load queues. NoSQ must therefore extend the in-order back-end commit pipeline to obtain store and load addresses and data values elsewhere.

NoSQ extends the commit pipeline to

- obtain physical register names, access sizes, and immediate offsets of memory operations from an augmented reorder buffer;
- read base addresses and data values of stores and loads from the register file; and
- use these values to calculate effective addresses for these instructions.

Because stores and bypassing loads skip the out-of-order engine, the commit pipeline now uses the register file ports and address generation units previously used to execute these instructions in the out-of-order core. A small amount of additional register read bandwidth is required for regenerating addresses to verify nonbypassing loads.

NoSQ's commit pipeline needs the addresses of all loads for SVW reexecution

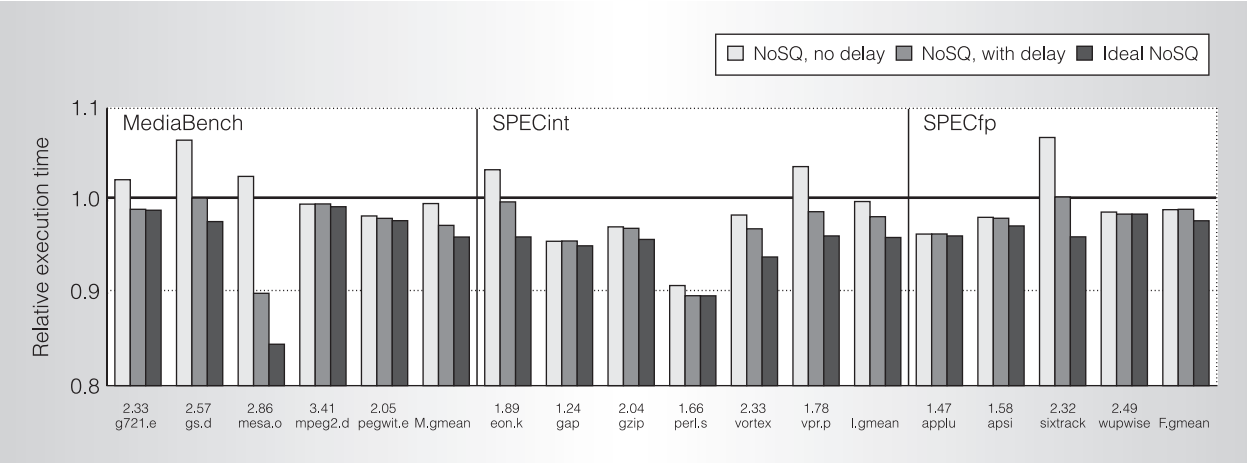

Figure 2. Execution time of three NoSQ configurations relative to a baseline with conventional store-queue-based store-load forwarding. NoSQ without delay has no memory scheduler; with this implementation, several programs suffer from bypassing misspeculation. NoSQ adds delay to avoid misspeculation-induced flushing. Both realistic NoSQ implementations achieve a significant fraction of the performance of an ideal implementation of NoSQ that uses an oracular SMB predictor. M.gmean: geometric mean for the MediaBench benchmarks; I.gmean: geometric mean for SPECint; F.gmean: geometric mean for SPECfp.

filtering. However, the pipeline needs originally executed data values and new data values from the data cache only for loads that actually reexecute—in practice less than 1 percent of all loads. In NoSQ, load reexecution shares the data cache port and the register file data register read port with store commit.

Extending the commit pipeline might increase pressure on core structures such as the reorder buffer, load and store queues, and register file. However, this issue is not a significant concern for NoSQ because the reorder buffer is not latency-critical, and NoSQ has no store and load queues. Moreover, SMB reduces register file pressure by allowing the definition and the load in a definition-store-load-use chain to share a single physical register.

Evaluation

We evaluated NoSQ with a detailed timing simulation of an aggressive current-generation processor (four-way issue, 128-instruction window, 11-stage combined front-end and out-of-order core pipelines) on the SPEC2000 and Mediabench programs. The processor's baseline configuration has a 24-entry store queue, a 48-entry load queue, a 40-entry issue queue, a 2,048-entry store sets load-scheduling predictor, and a six-stage in-order commit pipeline. The NoSQ configuration has no store and load queues, a 2,048-entry bypassing predictor (10 Kbytes total storage), and an eight-stage commit pipeline.

Figure 2 shows execution times for three NoSQ configurations on selected benchmarks relative to a baseline with an associative store queue and perfect load scheduling. The baseline's IPC appears above each benchmark name. The first bar shows NoSQ with no support for delay. In this configuration, NoSQ doesn't use an explicit load scheduler in the out-of-order core. Overall, NoSQ slightly outperforms the conventional design (by 1 percent on average). This improvement is largely due to SMB's latency-reducing effects and to the issue queue capacity amplification caused by not dispatching stores and bypassed loads. Because of bypassing mispredictions, a few programs experience slowdowns relative to the conventional design (as much as 7 percent for gs.decode and sixtrack).

Adding delay to NoSQ (second bar) adds a simplified load scheduler to eliminate excessive flushing for benchmarks that suffer from excess bypassing mispredictions. Delay

improves average performance and reduces the number of benchmarks with more than a 1 percent slowdown to only one (SPECfp's mesa, not shown in the graph). With delay, NoSQ achieves bypassing prediction accuracies of greater than 99.8 percent on every program. To achieve this accuracy, NoSQ delays an average 2 percent of the dynamic loads.

The third bar shows NoSQ with an oracular bypassing predictor; realistic NoSQ with delay achieves half the performance benefit. However, the performance benefits of even an ideal NoSQ are small;[11] our experiments show only a 4 percent speedup over the baseline for ideal NoSQ. Again, NoSQ doesn't use SMB for performance but rather to eliminate the store queue.

In addition to having a slightly better IPC on average than a traditional design, NoSQ performs fewer data cache reads because bypassing loads filtered by SVW avoid accessing the cache even once. As a result, NoSQ reduces data cache reads by 10 percent on average (roughly matching the bypassing rate).

The many recent proposals for simple and more scalable designs reinforce the notion that designing fast, high-bandwidth load and store queues represents a challenge for today's processors and a potential barrier for future large-window processors. NoSQ addresses this challenge by providing a simpler, more scalable microarchitecture.

NoSQ is also a good fit for multithreaded, partitioned, or distributed microarchitectures. Traditionally, these microarchitectures must replicate, partition, or distribute the load and store queues. NoSQ makes these complex actions unnecessary by replacing these queues with the bypassing predictor and the SSBF, structures that are easier to distribute or share. MICRO

Acknowledgments

NSF Career award CCF-0238203, NSF CPA grant CCF-0541292, and donations from Intel supported this work.

References

1. T. Sha, M. Martin, and A. Roth, ''Scalable Store-Load Forwarding via Store Queue Index Prediction,'' *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 05), IEEE CS Press, 2005, pp. 159-170.
2. T. Sha, M. Martin, and A. Roth, ''NoSQ: Store-Load Communication without a Store Queue,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 06), IEEE CS Press, 2006, pp. 285-296.
3. A. Moshovos and G. Sohi, ''Streamlining Inter-Operation Communication via Data Dependence Prediction,'' *Proc. 30th Ann. Int'l Symp. Microarchitecture* (Micro 97), IEEE CS Press, 1997, pp. 235-245.
4. M. Lipasti and J. Shen, ''Superspeculative Microarchitecture for Beyond AD 2000,'' *Computer*, vol. 30, no. 9, Sept 1997, pp. 59-66.
5. V. Petric, T. Sha, and A. Roth, ''RENO: A Rename-Based Instruction Optimizer,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 98-109.
6. G. Tyson and T. Austin, ''Improving the Accuracy and Performance of Memory Communication through Renaming,'' *Proc. 30th Ann. Int'l Symp. Microarchitecture* (Micro 97), 1997, pp. 218-227.
7. H. Cain and M. Lipasti, ''Memory Ordering: A Value Based Definition,'' *Proc. 31st Ann. Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 90-101.
8. A. Roth, ''Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 458-468.
9. G. Chrysos and J. Emer, ''Memory Dependence Prediction Using Store Sets,'' *Proc. 25th Ann. Int'l Symp. Computer Architecture* (ISCA 98), IEEE CS Press, 1998, pp. 142-153.
10. A. Yoaz et al., ''Speculation Techniques for Improving Load-Related Instruction Scheduling,'' *Proc. 26th Ann. Int'l Symp. Computer Architecture* (ISCA 99), IEEE CS Press, 1999, pp. 42-53.
11. G. Loh, R. Sami, and D. Friendly, ''Memory Bypassing: Not Worth the Effort,'' *Proc. 1st Workshop Duplicating, Deconstructing, and Debunking* (WDDD 02), 2002, pp. 71-80; http://www.ece.wisc.edu/~wddd/2002/final/loh.pdf.

Tingting Sha is a PhD candidate in the Department of Computer and Information Science at the University of Pennsylvania.

Her research interests include complexity-effective microarchitectures. Sha has a master's degree in computer science from the University of Pennsylvania. She is a student member of the ACM.

Milo M.K. Martin is an assistant professor in the Department of Computer and Information Science at the University of Pennsylvania. His research interests include scalable microarchitectures, multiprocessor computer architectures, memory systems, and verification. Martin has a PhD in computer science from the University of Wisconsin-Madison. He is a member of the ACM and the IEEE.

Amir Roth is an assistant professor in the Department of Computer and Information Science at the University of Pennsylvania. His research interests include computer architecture and microarchitecture. Roth has a PhD in computer science from the University of Wisconsin-Madison. He is a member of the ACM and the IEEE.

Direct questions and comments about this article to Tingting Sha, Dept. of Computer and Information Science, University of Pennsylvania, 3330 Walnut St., Philadelphia, PA 19104; shatingt@cis.upenn.edu.