Design and Optimization of the Store Vectors Memory Dependence Predictor

SAMANTIKA SUBRAMANIAM and GABRIEL H. LOH

Georgia Institute of Technology

Allowing loads that do not violate memory ordering to issue out of order with respect to earlier unresolved store addresses is very important for extracting parallelism in large-window superscalar processors. Previous research has proposed memory dependence prediction algorithms to prevent only loads with true memory dependencies from issuing in the presence of unresolved stores. Techniques such as load-store pair identification and store sets have been very successful in achieving performance levels close to that attained by an oracle-dependence predictor, but have relatively complex or power-hungry designs. In this article, we use the idea of dependency vectors from matrix schedulers for nonmemory instructions and adapt them to implement a new dependence prediction algorithm. We show that for conservatively sized processors, a simple PC-indexed table that tracks misordered loads is sufficient to provide most of the performance benefits achieved by more sophisticated predictors. On more aggressive processor configurations, however, our “Store Vector” algorithm provides better performance than the state-of-the-art store sets predictor while maintaining a simpler and more scalable design.

Categories and Subject Descriptors: C.1 [Computer Systems Organization]: Processor Architectures

General Terms: Design, Performance, Scalability

Additional Key Words and Phrases: Memory dependence prediction

ACM Reference Format:

Subramaniam, S. and Loh, G. H. 2009. Design and optimization of the store vectors memory dependence predictor. *ACM Trans. Architect. Code Optim.* 6, 4, Article 16 (October 2009), 33 pages. DOI = 10.1145/1596510.1596514 <http://doi.acm.org/10.1145/1596510.1596514>

1. INTRODUCTION

Each successive technology generation provides more transistors to processor designers. An increase in the device count, however, does not automatically

This article is an extension of the paper entitled “Store Vectors for Scalable Memory Dependence Prediction and Scheduling” which appeared in the *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, in February 2006.

Authors’ address: S. Subramaniam and G. Loh, College of Computing, Georgia Institute of Technology, 265 Ferst Drive, Atlanta, GA 30332-0765; email: {samantik, loh}@cc.gatech.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1544-3566/2009/10-ART16 \$10.00

DOI 10.1145/1596510.1596514 <http://doi.acm.org/10.1145/1596510.1596514>

ACM Transactions on Architecture and Code Optimization, Vol. 6, No. 4, Article 16, Pub. date: October 2009.

result in an increase in performance. Processor microarchitectures evolve with each process generation to convert exponential device scaling into exponential performance scaling.

Microarchitecture contributes to performance scaling through hardware algorithms for extracting instruction-level parallelism (ILP) [Sohi et al. 1995; Patt et al. 1997; Cristal et al. 2004]. A frequently studied technique for exposing more ILP is to increase the number of instructions in-flight in the processor. This may come from implementing very large instruction windows [Michaud and Seznec 2001; Lebeck et al. 2002; Brekelbaum et al. 2002], creating large effective instruction windows [Akkary et al. 2003; Srinivasan et al. 2004], or using nonconventional processor organizations [Sankaralingam et al. 2003; Swanson et al. 2003]. As the instruction window size increases, the processor must consider many load instructions in the presence of earlier unresolved store addresses. Aggressively reordering these loads ahead of unresolved stores may lead to ordering violations and the associated pipeline flushes. Conservatively preventing loads from reordering, however, rapidly reduces the amount of ILP extractable by the larger instruction window.

Out-of-order load scheduling is a nontrivial problem. Load instructions may have data dependencies through memory where an earlier store instruction writes to an address and the load instruction reads from the same memory location. Unfortunately, the effective addresses of all load and store instructions may not be available because the corresponding address computations may not have executed. This leads to a dependency ambiguity: depending on the result of a store's address computation, a later load instruction may or may not actually be data dependent. If the load is actually independent, then it should be scheduled for execution to maximize performance. If the load is data dependent, then it must wait for the store instruction.

Memory dependence prediction is a technique for speculatively disambiguating the relationships between loads and stores. A load instruction that issues too early due to a dependence misspeculation, however, will load a stale value from the data cache and propagate this incorrect value to its dependent instructions. Many cycles may pass between when the load instruction issues and when the processor finally detects the ordering violation. At this point, a large number of instructions from the load's forward slice [Zilles and Sohi 2000] may have already executed. Tracking down and rescheduling all of these instructions is a very difficult task, and so a memory dependence misspeculation is typically handled by flushing the pipeline. Overly aggressive load scheduling combined with the high cost of pipeline flushes can, therefore, result in a net performance decrease.

The most accurate memory dependence predictors rely on identifying relationships between load instructions and one or more earlier stores that are likely data-flow predecessors. Before a load instruction can issue, it must wait until these predicted dependencies have resolved. The identification of the dependencies and the communication of their resolutions typically require additional complex logic. The load-store queue (LSQ), however, is already a very complex circuit with a full set of CAMs for detecting load-store ordering violations as well as supporting store-to-load data forwarding; adding too much

more circuitry to support memory dependence prediction may not be practical because of the negative consequences on the critical-path latency of the LSQ and the overall processor clock frequency.

Conventional implementations of nonmemory instruction schedulers are often CAM-based. The associative logic causes the scheduler to be a timing critical path [Palacharla 1998]. Dependency-vector/matrix scheduler organizations have been proposed for faster and more scalable implementations [Goshima et al. 2001; Brown et al. 2001; Sassone et al. 2007]. In a previous paper, we proposed a new memory-dependence prediction and scheduling algorithm based on this idea of dependency vectors [Subramaniam and Loh 2006]. In this article, we provide extended performance analysis over a wider range of hardware assumptions. We show under which conditions sophisticated algorithms like store vectors are needed, and we also propose a few optimizations to reduce the hardware requirements of store vectors.

The next section discusses prior work on memory dependence prediction, and in particular, it reviews the store sets algorithm in greater detail. Section 3 explains our proposed store vector approach, providing a step-by-step description and example of the technique. Section 4 presents the performance results. Section 5 proposes and evaluates new optimizations of the baseline store vectors predictor to use significantly less hardware. Section 6 provides more detail on why store vectors is able to outperform other proposed memory dependence predictors, especially store sets. Section 7 briefly describes the implementation complexities associated with these memory dependence predictors. Section 8 concludes the article with a discussion summarizing the main results of the article.

2. BACKGROUND

2.1 Memory Dependence Predictors

Like many other properties in microprocessors, memory dependencies exhibit a form of temporal locality. The concept of memory dependence locality was introduced by Moshovos [1998]. In his thesis work, Moshovos characterized two forms of locality: memory dependence status locality, and memory dependence set locality. Memory dependence status locality makes the observation that when a load experiences a store dependency, subsequent instances of the same load will likely experience a dependency as well. Status locality makes no statement about which particular store(s) a load may or may not be dependent on. Memory dependence set locality makes the observation that if a load is dependent on a set of store instructions, then future instances of that load will likely be dependent on the same set of stores.

Previously proposed memory dependence predictors typically predict the dependence status of a load: Whether a dependence currently exists, which determines if the load can issue. The basic dependence predictors do not attempt to exploit the dependence set locality property. The most basic predictor is a naïve or blind predictor that simply predicts all load instructions to not have any store dependencies. A blind predictor will never cause a load to wait

when store dependencies are not present, but it will always cause a costly misprediction if a dependency exists.

Status-based prediction is a technique where the processor uses past behavior to predict future behavior, and based on this speculation, selects some loads to wait for all prior store address to resolve while allowing other loads to speculatively issue. The Alpha 21264 employed such a dependence predictor called the Load Wait Table (LWT), which tracks all loads that experience ordering violations [Kessler 1999]. When a store executes and exposes a load-ordering violation, the load's PC indexes into the LWT and sets a bit. On subsequent instances of the load, the LWT will indicate that the load previously caused a memory ordering violation, and the scheduler will force the load to wait until all earlier store addresses have been resolved. The processor periodically clears the table to avoid permanently preventing loads from speculatively issuing and to allow the predictor to adapt to phasic application behavior in different parts of the program. The Intel Core 2 has a similar mechanism for indicating dependent loads that are forced to wait [Doweck 2006].

Status-based prediction schemes do not take instruction timing into account. A load that is predicted to have a dependency will wait for all previous store instructions before issuing. In practice, the load will only be dependent on one or a few of the previous stores. Even if all of these store dependencies have computed their addresses, the load must conservatively wait until the nondependent stores have resolved as well. A status-based predictor may correctly predict the status of a load and avoid a pipeline flush, but potential ILP may still be lost from preventing loads from issuing in a timely fashion. Memory dependence predictors based on observing the exact sets of stores that a load collides with have shown to provide better prediction accuracy and overall performance. Moshovos proposed a dependence history tracking technique that identifies load-store pairs known to have memory dependencies [Moshovos et al. 1997; Moshovos 1998]. When a load speculatively issues and violates a true dependency with an older store, the processor records the program counters (PC) of both the load and the store in a table. Subsequent executions of the load instruction will check the table to see if it had conflicted with a store in the past. If so, the load will also check the store queue (STQ) (using the store PC recorded in the table) to see if that store is present. If the store is present in the STQ, then the load must wait until the store has issued. This load-store pair approach has the advantage that misspeculations can be avoided, but at the same time, the dependent loads are not delayed longer than necessary.

Different dynamic invocations of a load may have memory dependencies with different static stores. Moshovos also proposed an extension to load-store pair identification that associates a bounded number of stores with each load [Moshovos 1998]. Chrysos and Emer [1998] generalized this approach with their store sets algorithm that allows one or more load instructions to be associated with one or more stores. Previous work showed that a store sets memory dependence predictor provides performance close to that of an oracle predictor.

Fig. 1. Store sets data structures and their interactions.

2.2 Store Sets

The store sets algorithm groups a load's conflicting stores into one logical group or a store set [Chrysos and Emer 1998]. Each store set has a unique identifier, called the store set identifier (SSID). Each load and each store may belong to one and only one store set. Figure 1 shows the hardware organization of the store sets data structures. The Store Sets Identification Table (SSIT) is a PC-indexed, tagless table that tracks the current store set assignment for each load and each store instruction. The example in the figure shows Stores A and C and Load X belonging to Store Set 2, while Store B belongs to Store Set 0. This means that in the past, Load X has had memory ordering violations with Stores A and C. The SSIT combined with proper SSID assignment track the active store sets in the program.

To prevent memory ordering violations, a load instruction must wait on any unresolved store instructions that belong to the load's store set. To determine if any such stores are present in the STQ, each store updates the last-fetched store table (LFST), which indicates the STQ index of the most recent in-flight store. At dispatch, a load instruction consults the LFST and if an active store is present, a dependency is established between the two instructions. In Figure 1(a), the most recently fetched store (from the load's store set) resides in STQ entry 4 as indicated by the LFST and the dependency is illustrated by the bold arrow. To make a load wait on all active stores in its store set, all stores within the same set are also serialized. This is represented by the dependency arc between Stores A and store C in Figure 1. As described, each store can only belong to a single store set determined by the value in the single SSIT entry corresponding to the store. This means that if there are two different loads that are dependent on the same store, then the store will “ping-pong” back and forth between the two loads. To address this problem, Chrysos and Emer proposed a modified SSID assignment rule called store sets merging that allows more than one load to share the same store set.

The store sets algorithm introduces new dependencies from stores to stores and from stores to loads. These dependencies prevent costly memory ordering violations while allowing independent loads to aggressively issue out of order.

Fig. 2. Store Sets implementation using a (a) CAM-based fully associative organization and (b) a RAM-based direct-mapped organization.

However, the load and store queues must incorporate new hardware to track and enforce the store sets dependencies. The load and store queues are already very complex structures, requiring a large amount of content addressable memory (CAM) logic for the detection of memory ordering violations and to support store-to-load data forwarding [Cain and Lipasti 2004; Roth 2005]. Compared to a nonmemory instruction scheduler, the load and store queue CAMs are much larger because they must deal with 64-bit addresses rather than 7- to 8-bit physical register identifiers. Furthermore, the load and store queues must also employ some form of age or order-tracking information because a load must be able to distinguish between an older (in program order) store and a younger store to the same address. In situations where multiple older stores to the same address exist, a load needs the age information to make sure that it receives its data from the more recent store.

The original store sets paper did not clearly describe the exact hardware implementation of the algorithm. Here, we outline two possible approaches. If we implement store sets using associative logic, this requires a complete set of CAMs much like the ones already present in the load and store queue and the related broadcast buses to track the store sets dependencies, as illustrated by the shaded blocks in Figure 2(a). As described earlier, each store updates its LFST entry with its STQ index. For each subsequent load belonging to this store set, the load will read this STQ index from the LFST and store this in the corresponding load queue entry. When the store executes, it broadcasts its STQ index to all of the load queue entries, and any loads waiting on this store will make note of the resolved dependency. Furthermore, the store must also broadcast its STQ index to all of the STQ entries, since all stores within the same store set will be serialized. The net effect of this approach to implementing store sets is the addition of an entire extra set of CAMs to both load and store queues. If multiple loads and/or stores can issue each cycle, then these CAMs will also need to support additional ports. This CAM-based implementation would lead

to load and store queues that do not scale well to larger sizes, grossly increase power consumption, and may possibly limit processor clocks speeds as well.

A second potential implementation of store sets uses a direct-mapped RAM-based organization, as shown in Figure 2(b). Stores update the LFST with their STQ indexes as usual. When a load (or store) in the same store set dispatches/allocates, it first consults the LFST to find the index of the last fetched store (4 in this example). The load then uses this index to directly access (direct mapped) the STQ entry to insert its own load queue index (1 in this example). When the store finally issues, it uses this load queue index to directly access the corresponding load queue entry to notify the load that this dependency has now been resolved (shown by the dashed arrow). This approach does not require CAMs, which have high area/power/latency costs, but there are a few additional complications. Store set merging can result in multiple loads all being dependent on the same store. Like the explicit data forwarding (EDF) technique [Sato and Arita 2001], each STQ entry must be able to support waking up multiple dependent loads, which requires storing more than one load queue index. If each store can have up to k dependent loads, then the store's STQ entry must have the capacity to track k load queue indexes, and the load queue itself needs k write ports if the store is to notify all of these loads without incurring any additional delay. If store set merging and predictor table aliasing effects cause the store to have greater than k dependents, then either dependence prediction will suffer because some loads cannot be added to the store's output dependency list or even more complicated schemes (such as allocation of extra STQ entries) will be required. Another complication with this RAM-based implementation of store sets is that when a younger load inserts its load queue index into a STQ entry, the load may in fact be a speculative instruction in the shadow of a mispredicted branch. When the processor uncovers the branch misprediction, some recovery mechanism is required to remove the load queue index from the parent STQ entry.

The LFST also creates a new critical loop similar to the register renaming logic. It is possible that multiple stores need to update the LFST in the same cycle. If all of these stores belong to the same store set, then some sort of dependency checker is required to correctly set up the intragroup dependencies, and a prioritized write logic is needed to make sure that only the last store in the store set updates the corresponding LFST entry. Similar to the wrong-path pointers in the RAM-based store sets implementation, the LFST entries may also contain STQ indexes that correspond to wrong-path instructions. The LFST would need support for checkpointing or some other recovery/repair mechanism.

2.3 Scheduling Structures

Instruction schedulers typically use CAM organizations. Each issuing instruction broadcasts a unique identifier to notify its dataflow children that the dependency has been resolved. Each instruction must monitor all of the broadcast buses, constantly comparing the identifiers of its inputs with the broadcast traffic. Palacharla analyzed the structure of CAM-based schedulers and found

Fig. 3. (a) two-matrix scheduler and (b) single-matrix unified scheduler.

that the critical path delay increases quadratically with the issue width and the number of entries [Palacharla 1998].

The dependency vector or dependency matrix scheduler organization is an alternative scheduler topology designed to be significantly more scalable. Goshima et al. [2001] proposed to replace the left and right CAM banks of a conventional scheduler with left and right dependency matrices, as illustrated in Figure 3(a). For a W -entry scheduler, each matrix has W rows and W columns; one for each instruction. If Instruction i is data-dependent on Instruction j , then the matrix entry at Row i and Column j is set to one. So long as Instruction i has a bit set in its row, then the corresponding input dependency has not been resolved. When Instruction j issues, it clears *all* bits in Column j , thus notifying any dependents in the window that the parent instruction has been scheduled. The critical-path logic is significantly reduced as compared to a CAM-based scheme. The tag comparison for computing readiness has been replaced by a single wired-NOR and an AND gate to check that both left and right inputs are ready. The multibit tag broadcast has been replaced by a single bit latch-clear signal. The matrix structures only contain one bit per entry, which makes the total area significantly smaller than a CAM-based scheduler that contains registers for the dependency tags, comparators, large broadcast buses, and additional logic. Figure 3(b) illustrates a single-matrix implementation, suggested by Brown et al. [2001]. Each matrix row can contain multiple nonzero entries to denote all of the dependencies, which halves the matrix area and removes the AND gates for detecting both left-and-right input readiness. Our store vector-dependence predictor's hardware implementation is based on this compact, scalable single-matrix scheduler structure.

3. STORE VECTOR-DEPENDENCE PREDICTION

We propose a new algorithm for memory dependence prediction based on store vectors. Store vectors are different than the load-store pair and store set approaches in that store vectors do not explicitly track the PC of stores that collide with loads. Instead, we implicitly track load-store dependencies based on the relative age of a store. Consider the example in Figure 4. The five load and store instructions are listed in program order, and Load W has had ordering violations with Store B in the past, and Load X has had ordering violations with both Stores A and C. Figure 4(a) illustrates the PC-based dependency

Fig. 4. (a) store-address tracking of dependencies and (b) store position or age tracking of dependencies.

Fig. 5. Store vectors data structures and interaction with a conventional load queue.

information used by previous pair or set-based approaches. In contrast, an age-based approach illustrated in Figure 4(b) stores the relative age or positions of the stores. Load X remembers that it had previous conflicts with the most recent store and the third most recent store.

A load's store vector records the relative positions or ages of all stores that were involved in previous memory ordering violations. The store vector for Load X is illustrated in Figure 4(b). In the general case, the length of the store vector will be equal to the number of STQ entries, although we discuss different ways to optimize store vectors to only track a certain number of store dependencies in Section 7.

3.1 Step-by-Step Operation

The store vector algorithm has three main steps: look-up/prediction, scheduling, and update due to ordering violations. These are described in the following text, followed by an example.

Look-up/Prediction. The primary data structure for recording load-store dependencies is the store vector Table (SVT). For each load, the least significant bits of the load's PC provides an index into the SVT, shown in Figure 5. The corresponding store vector is then rotated and copied into the load scheduling matrix (LSM or simply the *matrix*). The process for setting a load's store vector is described later. The LSM consists of one row for each load queue entry, and one column for each STQ entry (the matrix need not be square).

An SVT entry records a load's store vector in a format where the least significant bit corresponds to the most recent store before the load. The rightmost

column of the LSM may not correspond to the most recently fetched store. A barrel shifter must rotate the vector such that the least significant bit is aligned with the column of the most recent store. Any bits in the vector that correspond to already resolved stores must be cleared to prevent the load from waiting on an already resolved dependency (which could result in deadlock). This is accomplished by taking a bitwise AND of the store vector with the bits from each STQ entry, that indicates if the store's address is not ready. Since the STA (store address) and STD (store data) are two separate uops that reside in the STQ entry, the STA bit of the STQ entry can be cleared when the address of the store is known irrespective of whether the data is ready or not. Finally, the vector is written into the matrix.

Note that for both the store sets and store vector techniques, the prediction look-up latency can be overlapped with other frontend activities. Both approaches use PC-based table look-ups, which, in theory, could be initiated as early as the fetch stage, although for power reasons, one would likely defer the look-up to the decode stage to avoid unnecessary predictions for nonload instructions.

Scheduling. After the prediction phase has written a load's store vector into the LSM, there may be some bits in the vector that are set. The position of the bits indicate the stores that this load is predicted to have a dependency with. While any of the bits are still set, the load will not be considered as ready. The hardware implementation of the matrix is identical to the single-matrix scheduler described earlier in Figure 3(b). A wired-NOR determines if any unresolved predicted dependencies remain.

Each STQ entry is uniquely mapped to a column of the matrix. When the store issues, it simply clears all of the bits in its corresponding column. Note that stores in the same vector can issue in any order, whereas with store sets, all of the stores are serialized. Furthermore, a store can be a predicted input dependence for any number of load instructions (more than one entry per column may be set), whereas with store sets, the LFST forces each store to belong to only a single store set.

For those stores in the STQ, whose STA component is ready but whose STD is not ready, some additional control mechanism is required in order to schedule dependent loads. First, such loads can be forced to poll the STQ every cycle to check for the ready STD component and only then be allowed to issue or second, such stores (ready STA and not-ready STD) can be forced to broadcast their ready STD to wake up all loads waiting for them. This problem, however, exists for all memory dependence predictors (store sets, store vectors, and LWT), since all STQs implement separate STA and STD components.

Update. Initially, vectors in the SVT are initialized to all zeros. In this fashion, all load instructions will initially execute as if under a naïve/blind speculation policy. When a load-store ordering violation occurs, the store's relative position/age is determined. The position of the most recent store with respect to the offending load is easily determined because the processor already keeps this information for tracking the ordering of all load and stores. The difference between the STQ indexes of the most recent store and the store involved in the

Fig. 6. Example store vector operation for the ① prediction phase, ② scheduling phase, and ③ update phase.

ordering violation provides the relative age of the offending store. This bit is then set in the load's SVT entry.

Eventually, all bits in the vector may get set, thus making the processor behave as if it was incapable of any memory speculation at all. Similar to the 21,264 LWT and store sets, we periodically reset the contents of the SVT to clear out predicted dependencies that may no longer exist due to changes in program phases, dynamic data values, or other reasons.

3.2 Example

This section provides a detailed example of predicting a load's dependencies, scheduling the load, and updating the load's store vector when an ordering violation occurs. The different steps of the example are illustrated in Figure 6.

① *Prediction:* After decoding the instruction Load X, a hash of the load's PC is used to select a vector from the SVT. The store vector bit that corresponds to the most recent store fetched before this load is indicated with shading. The 1s in the vector indicate that Load X is predicted to be dependent on the most recent store (shaded), the third most recent store, and the sixth most recent store.

In this example, the most recent store is Store D, which has been allocated to STQ entry 2. The STQ-head points to the oldest STQ entry, and the STQ-tail points to the next available STQ entry. If we right barrel shift (least significant bits wrap around to the most significant positions) by an amount equal to the STQ-tail, the store vector bit for the most recent store will now be properly aligned with the most recent STQ entry. In this case, the right barrel shift is by three; note that the position of the shaded bit has moved to reflect the location of the most-recent store.

Corresponding to the nonzero bits in load's store vector, the most recent store is Store D, the third most recent is Store B, and a sixth most recent does not exist. The store vector predicts that Load X is dependent on the most recent store, but at the time of dispatch, Store D had already issued, so Load X should not wait on Store D. Each entry of the STQ has a bit that indicates whether the corresponding store address is unknown. In the case of invalid entries, the bit is cleared to indicate that the address is known. By taking a bitwise AND, we clear the store vector bit that corresponded to the invalid store (the sixth most recent store) as well as a store that had already been resolved (Store D). This final store vector is written into the LSM in the row specified by the load's load queue entry index.

② *Scheduling*: In this example, Load X only has one remaining dependency in its store vector. At some point in the future, Store B will issue. At this point, Store B clears its column in the LSM. Each STQ entry can simply have a single hardwired connection to the clear signal for the corresponding column. Load X's store vector no longer contains any 1s, so the wired-NOR will raise its output to indicate that Load X's predicted store dependencies have all been satisfied. Assuming that Load X's address has already been computed, it will proceed to bid for a memory port and then issue.

③ *Update*: If Load X issued and the load and STQs do not eventually detect any memory ordering violations, then no other actions are required. If after Load X speculatively issues and Store A ends up writing to the same address as Load X, then there will be a memory ordering violation. In this case, Load X and all instructions afterward are flushed from the pipeline and refetched. Store A is the fourth most recent store with respect to Load X, although it may not be the fourth most recent store in the STQ because other store instructions may have been fetched since Load X was dispatched. To prevent future memory ordering violations between Store A and Load X, the store updates Load X's store vector in the SVT, setting the bit for the fourth most recent store.

4. RESULTS

This section presents the performance evaluation of the various memory dependence prediction algorithms.

4.1 Evaluation Methodology

We use cycle-level simulation to evaluate the performance of the memory dependence predictors. In particular, we use the MASE simulator [Larson et al. 2001] from the SimpleScalar toolset [Austin et al. 2002] for the Alpha instruction-set architecture. We made several modifications related to memory dependence prediction and scheduling, including support for separate load and store queues (as opposed to a unified LSQ), store sets and the store vectors algorithms. In addition to comparing the performance of store sets with store vectors, we also simulate the LWT described in Section 2.1, since it has low-implementation complexity. For LWT, store sets and store vectors, we reset the predictor tables every 1 million instructions. The minimum latency from fetch to execution is twenty cycles, and we simulate a full in-order frontend pipeline as opposed to

Table I. Simulated Processor Configurations

Configuration	Medium	Large	Extra-Large
Processor Width	6	8	8
Scheduler Size (RS)	64	96	128
Load Queue Size	32	96	128
Store Queue Size	32	96	128
ROB Size	256	512	2,048
Memory Ports	2	4	4
ALU	4 Int, 4 FP		
Mult	1 Int, 1 FP		
IL1 Cache	16KB, 4-way, 3-cycle		
DL1 Cache	16KB, 4-way, 3-cycle		
L2 Cache	512KB, 8-way, 10-cycle		
L3 Cache	4MB, 16-way, 30-cycle		
Memory	250 cycles		
ITLB	64-entry, FA		
DTLB	64-entry, FA		
L2 TLB	1,024-entry, 8-way, 7-cycle		

the default SimpleScalar behavior of simply stalling fetch for some number of cycles.

We model three main processor configurations, whose parameters are listed in Table I, to evaluate the performance of the different memory dependence predictors. The parameters that differ between the configurations are noted in the specific configuration column while all other parameters are common to all the configurations. The “Medium” configuration represents a moderately aggressive processor similar to current microarchitectures. The “Large” configuration has larger hardware structures and a wider machine width to represent a more aggressive processor. Finally, we also model an “Extra-Large” configuration, which represents a highly aggressive machine that exposes more opportunities for load-store dependencies to exist. While the extra-large configuration is definitely more aggressively positioned than current processors and would be difficult to implement using conventional design methodologies, there have been some innovative research proposals that have put forth implementable designs of similar, extra-large machines [Sankaralingam et al. 2003; Cristal et al. 2004]. Due to the large buffer structures, these designs could encounter several load-store dependencies and may benefit by using advanced memory dependence prediction algorithms. Hence, we include this configuration design point in the evaluation.

We simulated a variety of applications from the following suites: SPEC-cpu2000, MediaBench [Lee et al. 1997], MiBench [Guthaus et al. 2001], Graphics applications including 3D games and ray-tracing, Pointer-intensive benchmarks [Austin et al. 1994], as well as Bioinformatics benchmarks [Albayraktaroglu et al. 2005]. All SPEC applications use the reference inputs, where applications with multiple reference inputs are listed with different numerical suffixes. To reduce simulation time, we used the SimPoint 2.0 toolset to choose representative samples of 100 million instructions [Perelman et al. 2004]. All “average” IPC speed-ups are computed using the geometric mean.

Fig. 7. Performance of the medium-sized configuration across all benchmark suites. Memdep refers to those applications that were found to exhibit memory dependence sensitivity.

Fig. 8. Performance of the large-sized configuration across all benchmark suites. Memdep refers to those applications that were found to exhibit memory dependence sensitivity.

4.2 Performance Results

Our baseline processor configuration uses naïve/blind speculation. We also use a perfect oracle-dependence predictor as an “upper bound.” The oracle predictor is perfect in the sense that it correctly predicts which stores a load has true dependences with while ensuring that the load is only made to wait until these specific stores have computed their addresses. The oracle guarantees perfect dependence prediction, but this does not necessarily result in maximum performance because in rare cases misspeculated loads can still have performance benefits due to prefetching effects or early branch misprediction detection. Figures 7, 8, and 9 show the performance of the simulated memory dependence predictors for the medium, large, and extra-large configurations, respectively. We call an application dependence-sensitive, labeled as Memdep in the plots, if perfect-dependence prediction results in at least a 1% IPC speed-up over blind speculation. Many applications are not sensitive to memory dependences and only experience few (single digits) ordering violations over the course of millions of instructions. The detailed performance results of the various memory dependence predictors for all benchmarks (both dependence sensitive and

Fig. 9. Performance of the extra-large-sized configuration across all benchmark suites. Memdep refers to those applications that were found to exhibit memory dependence sensitivity.

nondependence sensitive) for the medium and the extra-large configurations are presented in the Appendix. Due to space constraints, the tabulated results for the large configuration have been omitted from the Appendix, but the trends are very similar to the extra-large configuration.

We now discuss the results in detail. For the medium-sized processor configuration, Figure 7 shows that perfect/oracle-dependence prediction achieves a 4.1% speed-up over blind speculation across all applications, and 8.5% on the dependence-sensitive programs. All of the dependence predictors, which include the simple LWT as well as the more sophisticated store sets and store vector algorithms, achieve a performance level that is relatively close to that of the perfect predictor. On the dependence-sensitive programs, the LWT's 7.7% performance improvement over blind speculation is quite close to store set's 8.1%, store vector's 8.4% and the perfect predictor's 8.5% speed-ups. From these results, we conclude that for moderately sized microarchitectures, a simple dependence predictor like the LWT is sufficient. Adding a more sophisticated predictor does not provide much additional performance, and this may not justify the added complexity associated with implementing the predictor. Per-application speed-up numbers for the medium configuration further show that there is not much variation in the performance of the memory dependence predictors. There are some integer applications like "gcc-2," however, which suffer quite a significant performance loss over blind prediction when using the LWT predictor due to its conservativeness. In this particular application, we observed many missed opportunities for speculative load scheduling. Therefore, advanced memory dependence predictors may still be desirable to provide more performance overall.

For more aggressively provisioned processors, the results and conclusions are slightly different. The larger instruction window exposes more load-store re-ordering opportunities, and as a result, better dependence predictors are in fact required to manage the scheduling of these instructions. Figure 8 shows that there is considerable variation between the performance of the LWT predictor and the perfect predictor in the large configuration. The floating point applications that showed similar speed-ups for the medium configuration regardless

of the prediction algorithm used now show a slight improvement in performance when the store vectors or the perfect predictor is used. The dependence-sensitive applications are especially impacted at this larger configuration. As an example we highlight the performance of a memory dependence-sensitive application “bc-2.” For the medium configuration, the LWT predictor had a 3% speed-up over the blind predictor whereas the store vectors predictor had a 3.6% speed-up. In the large configuration, however, the speed-up of the store vectors predictor is approximately 7%, whereas the speed-up of the LWT is less than 0.5%, since the conservative LWT predictor is unable to accurately predict all of the memory dependencies that are now exposed in this application. Due to some of the problems discussed in Section 6, the store sets algorithm only provides an additional speed-up of around 1.7% over the LWT approach. Store vectors, however, achieve performance levels much closer to the oracle predictor than the other techniques, although there still remains some room for improvement.

For the extra-large configuration, the performance difference between the memory dependence predictors is even more dramatic, as shown in Figure 9. There is now a much larger performance gap between the simple LWT predictor and the oracle predictor (2.1% versus 9.2% across all benchmarks, and 6.7% versus 14.9% on the dependence-sensitive applications). Per-application performance numbers for this configuration show the additional benefit of using advanced memory dependence predictors like store sets and store vectors. Both “crc32” and “dijkstra” post considerable performance losses when using the LWT predictor. Store sets and store vectors in these cases match the performance of the oracle predictor. Additionally, among the integer applications, the “gzip” benchmarks also suffer performance degradations when the LWT predictor is used. The results, however, are quite encouraging when using the advanced memory dependence predictors like store sets and store vectors. In fact, when considering all of the application results for the extra-large configuration, we see that where speculative memory dependence prediction does not help much (low performance difference between perfect and blind) store sets and store vectors are more robust and do not cause severe performance degradation as opposed to LWT, which posts losses of more than 10% in several cases. These results highlight the fact that when there is something to be gained through prediction, store sets and store vectors do a much better job of reaching oracle performance than LWT, and when there is not much performance to be gained, they are less harmful.

To illustrate this robustness property, we present performance S-curves for the extra-large configuration for the LWT, store sets, store vectors and perfect predictors in Figure 10. The plots show the relative speed-up over blind prediction observed for the various benchmarks sorted from the lowest speed-up (or greatest slowdown) to the highest speed-up. From these curves, we can see that the LWT predictor suffers from significant performance slowdowns (greater than 1%) for 25 out of the 94 simulated benchmarks with a largest slowdown of 11.1%. Store sets do not see any major performance losses but stays flat for a considerable number of benchmarks. Among all of the proposed

Fig. 10. S-curves illustrating the range of performance behavior of the simulated memory dependence predictors across all benchmarks for the extra-large configuration.

memory dependence predictors, Store vectors follow the trend of the perfect predictor more closely than the others.

There are a few applications where LWT and store sets perform slightly better than the store vectors algorithm. In these cases, there were more memory ordering violations overall in the blind speculation case. While the LWT is conservative in its general predictions, the store sets' set merging heuristic sometimes causes store sets to also make more conservative predictions, resulting in slightly better performance than store vectors for these applications. For a small number of other applications, both store sets and store vectors are too aggressive and actually cause performance to be slightly worse than either the baseline case or the LWT. There are also a few cases where the LWT, store sets, or store vectors outperform perfect-dependence prediction due to the prefetching and early branch misprediction detection effects mentioned earlier.

Overall, these results indicate that for current processors, using a simple memory dependence predictor like the LWT should be sufficient, as evidenced by the simple PC-based dependence predictor used in the Intel Core 2 microarchitecture [Doweck 2006]. While some market segments are moving to a larger number of smaller, simpler cores, other mainstream product lines continue to invest more resources to expose more ILP (e.g., Intel's "Nehalem" core [Intel 2008]). Even in a heavily multicore world, some significant resources will still be required to attack the Amdahl's Law sequential bottleneck [Hill and Marty 2008]. For these larger instruction window processors (or cores), our results lead us to conclude that more sophisticated memory dependence prediction algorithms will be needed.

5. OPTIMIZING THE STORE VECTORS

In this section, we study the performance impact of limiting the hardware budget for the store vector predictor tables, and then explore several optimizations

Fig. 11. Speed-up over blind prediction for the memory dependence-sensitive benchmarks of proposed memory dependence predictors across different hardware budgets.

to further reduce the hardware requirements without severely impacting overall performance. Since the performance trends of the large configuration mirrors that of the extra-large configuration, all the results in this section are presented only for the the medium and the extra-large configuration.

5.1 Sensitivity to Hardware Budget

Figure 11 shows the relative performance for all memory dependence-sensitive applications across a range of hardware budgets. The results show that even for a small hardware budget, sophisticated predictors like store vectors provide a significant performance benefit. In all of the proposed predictors, exceeding the hardware budget beyond 4KB does not buy much additional performance. This is intuitive as the predictors hold dependence information for the loads. While there are many loads in a program, only few of them collide with earlier stores, and hence, only information regarding these loads affects the performance of the application. If we decrease the hardware budget below 1KB, however, we observe that the performance drops off. This is primarily a result of aliasing in the SVT. We now describe how we can reduce the hardware budget below 1KB for the store vectors predictor without significant loss of performance.

5.2 Reduction of Store Vector Length

As described, the store vector length is tied to the number of entries in the STQ. While we found that this configuration performed well, it is possible to reduce the store vector length to decrease the hardware cost. For example, reducing the store vector length to only track the 16 most recent stores (as opposed to 32 as assumed in Section 3) would reduce the space requirements of the SVT by one half. However, the sizes of the remaining hardware structures are still tied to the STQ size. The LSM still requires one column per store; therefore, the barrel shifter must also produce a bit vector matched to the number of STQ entries.

Fig. 12. The relative performance impact of reducing the length of the store vectors (dependence-sensitive applications only). Zero represents the performance with blind speculation, and 1.0 is the performance of the baseline store vector predictor.

For our processor configuration, the benefit of store vectors drops off relatively quickly with shorter store vector lengths. Figure 12 shows the relative performance for different store vector sizes, where zero is the performance of blind speculation and 1.0 is the performance of the baseline store vector technique. From these results, we conclude that the less recent stores have a greater impact on load-ordering violations. This intuitively makes sense as the compiler should choose to spill registers that will not be soon reused. Our evaluation was conducted on the Alpha ISA, which has a relatively large number of registers; the results and optimal memory dependence predictor configuration are likely to be different for an ISA like x86 where spills/fills and memory traffic in general are more frequent.

5.3 Most Recently Conflicting Store

While Figure 12 indicates that the performance of store vectors drops when fewer stores are tracked in a conventional store vector technique, if we model a different style of tracking stores, we may still be able to reduce the store vector length and not incur a great loss in performance. Our motivation in searching for different approaches to holding store information for each load, comes from the observation that each store vector generally only has a few 1s. This indicates that only a few stores ever collide with the load, and hence it may not be necessary to allocate storage for all potential stores in the store vector. The trick lies in being able to keep only relevant stores in the vector. We tried two such techniques, the first one being where only the most recent conflicting store (MRCST) would be kept in the SVT. The MRCST is that particular store that caused a memory ordering violation with the load the last time this load was issued out of order. This could potentially reduce the hardware cost of the store vector technique tremendously, since each entry in the SVT now holds just one $\lceil \log_2 n \rceil$ -bit store age-index in place of an n -bit vector. The reasoning

Fig. 13. Different store-tracking strategies. The second bar indicates the MRCST strategy and the third bar depicts the performance of the store window method.

behind this strategy is that since store-load dependencies generally follow a recurring pattern, just storing information about the most recent dependence should suffice. Another recent distance-based memory dependence predictor proposed exploits a similar observation and makes a load depend only on its most recent conflicting store [Sha et al. 2005]. The MRCST technique needs the index of the store to be kept in the SVT so that the correct store cell can be set in the matrix. The second bar in Figure 13 corresponds to MRCST technique in the medium and extra-large machines, respectively. The results are encouraging because even though the performance is lower than that of the conventional store vector technique, it still outperforms store sets (which shows 8% and 9% speed-up over Blind for sensitive applications in the medium and extra-large configurations, respectively) at less than a quarter of its hardware budget. MRCST provides a very scalable SVT, since the hardware required for it is much less than the conventional SVT. For the purpose of this experiment, to model MRCST, we used an SVT with 512 entries, each holding a store age-index of 5 bits giving a total SVT hardware of 0.3125KB. Thus for negligible loss in performance we get a 84% reduction in hardware from the original SVT (2KB) evaluated in Section 4. The implementation complexity of this technique is also much lower. We can replace the barrel shifters needed to align the vector in the matrix as explained in Section 3.1 with a narrow-width adder and decoder.

5.4 Store Window

Extending this idea our next strategy deals with storing just a window of k stores, where k is statically assigned like described before but the start of the window can be dynamically chosen. If for a particular load, its fourth most recent store to its eighth most recent store are the stores generally causing a problem then storing only these in the SVT may be enough. Each store window was chosen to start at the store which corresponded to the load's MRCST as described previously. Thus, each vector now needs to hold a store age-index indicating the MRCST and k bits after it which indicate the dependence information. In this experiment, we keep track of four stores, thus each vector holds a 5 bit age-index and 4 bits of dependence information. In both plots of Figure 13,

Fig. 14. Performance of tagged SVT of different sizes compared with conventional untagged SVT. The first bar represents an 2KB untagged SVT. The second and third bars represent 0.625KB tagged SVT and 0.3125KB tagged SVT, respectively, while the fourth bar corresponds to a 0.1526KB tagged SVT storing only the MRCST.

the “store window” bar corresponds to this technique simulated in the medium and the extra-large machine, respectively. In this method, for negligible loss in performance, we obtain nearly, 70% reduction in hardware. The store window technique, however, does not provide much performance improvement over the MRCST as indicated in Figure 13.

5.5 Tagged SVT

After considering how we could best reduce the store vector length, we then moved on to determining whether some of the vectors could be eliminated altogether. On further examination, it was found that not only were a lot of bits in some vectors zero, but many vectors only contained zeroes. This is not very difficult to comprehend as the SVT stores information for each and every static load encountered in the program. Many loads just do not have dependencies with earlier unresolved stores. Knowing that a vector is completely zero is important in order to ensure that loads are allowed to issue as soon as their addresses are ready, but storing these zero vectors is wasteful and with just a slight change in the structure of the SVT these vectors could be eliminated while still keeping track of the respective loads. We augment the SVT with tags and only store nonzero vectors. A load miss in the table indicates that its vector is completely zero and the load can speculatively issue. This strategy of eliminating zero vectors greatly reduces the hardware budget while incurring practically no loss in performance. The reason for this is quite simple. All of the necessary information that enables store vectors to be an accurate memory dependence predictor is contained only in the nonzero vectors and hence storing only these vectors had minimal impact on the performance. For this simulation, we assumed a partial tag of 8 bits and a conventional store vector of 32 bits to give a total vector size of 5 bytes. In Figure 14, the first bar shows the performance of store vectors when using a untagged SVT (2KB), the second bar shows a tagged SVT with 128 entries (0.625KB), and the third bar shows a tagged SVT with 64 entries (0.3125KB). With the 128-entry tagged SVT, we get 68% reduction in

hardware, and with the 64 entry tagged SVT, we get 84% reduction in hardware. In fact, the 128-entry tagged SVT performs as well as a 2KB store sets predictor or a 2KB LWT predictor (results shown earlier in Figures 7 and 9). These results are consistent with our observation that most vectors are zero vectors, and hence, eliminating them does not significantly affect the performance. Finally, we present the performance of the most optimized version of the store vectors algorithm, which involves using a 256-entry tagged SVT that stores only the most recently conflicting store (fourth bar). This configuration provides comparable performance as the original store vectors algorithm with a hardware budget of 0.1563KB. Additionally, this configuration provides a 92% savings compared to the LWT and the store sets predictor with a performance difference of over 5% and 3%, respectively. Even though this design stores the least amount of information as compared to the other designs, it stores the most pertinent information.

5.6 Effect of Incorporating Control-Flow Information

In this experiment we use branch history information along with the load PC to index the SVT in the store vectors algorithm. In theory, store sets should have some more tolerance to memory dependence predictions that vary depending on the control path leading up to a load because the stores are all explicitly tracked by their PCs. With store vectors, a load might have a conflict with the third previous store when the program traverses one path, and the load might conflict with the fourth oldest store on another path. In this scenario, the store vector algorithm will mark bits three and four as conflicts, which may in turn cause the load to be overly conservative. We observe negligible improvement in performance, however, by the addition of control flow information. There are a few reasons why this effect does not have a great impact on performance. First, the load is only unnecessarily delayed if the nonconflicting store address resolves after the real store dependency. Second, even if delayed, the load only impacts performance if it is on the critical path. Third, aggressive compiler optimization may reduce the effects of control flow on dependence prediction. A load aggressively hoisted and duplicated to both paths leading up to a control flow “join” effectively assigns the load to two different PCs, which allows the predictor to identify the different control flow cases. If programs other than SPEC exhibit a high degree of memory dependences within very branchy code, the SVT can be modified in a gshare fashion to make use of some branch or path history information [McFarling 1993]. A memory dependence predictor that incorporates branch history information was also separately implemented in Sha et al. [2006].

6. WHY DO STORE VECTORS WORK?

Store vectors utilize the predictability in relative distances between colliding loads and stores. Hence, even if the control flow of the program differs, the relative position of a store that a load depends on is predictable. Another proposed work by Yoaz et al. [1999] made use of distance-based information for

Fig. 15. (a) ideal load synchronization against predicted store dependencies and (b) actual serialization with store sets.

determining which loads could be speculatively disambiguated. The authors proposed using a collision history predictor where each entry would store a minimum allowable distance that the load can be advanced. This distance is based on past observed distances between conflicting loads and stores. However, the mechanism of distance information collection and load scheduling is quite different from the SVT and LSM used by the store vector algorithm. In this section, we specifically explain the advantages of store vectors over store sets and the LWT. In particular, we will discuss scenarios where the store sets algorithm results in overly conservative predictions or unnecessary delays. The first reason why store sets can delay a load from issuing actually stems from delaying the issuing of stores. The desired behavior of a load with a predicted set of store dependencies is that the load waits for all of the dependencies to be resolved. Figure 15(a) shows a load instruction with four predicted store input dependencies. Ideally, the stores should be able to execute independently, since memory write-after-write false dependencies are already properly handled by the STQ. To prevent loads and stores from having multiple direct input dependencies (which would require more CAMs), the store sets algorithm serializes execution of stores within the same store set. This is illustrated in Figure 15(b), where the load's execution has now been delayed by three extra cycles. With the store vector approach, individual stores have no knowledge about dependency relationships with other stores; in fact, the store does not even explicitly know if any loads are dependent on it. Stores may issue in any order, and they just obliviously clear their respective columns in the LSM.

The store sets merging update rules allow two or more different loads to be dependent on the same store. Consider the loads in Figure 16(a), where both loads have several predicted store dependencies and both loads are predicted to

Fig. 16. (a) ideal load synchronization for two loads against their predicted store dependencies, (b) actual serialization of merged store sets, and (c) the store vector values that avoid unnecessary serialization.

be dependent on Store χ . Without store sets merging, Store χ can only belong to the store set of Load 1, for example. In this situation, Load 2 will not wait for Store χ , which results in ordering violations. With store sets merging, all of the stores associated with both loads will be merged into the same store set. Now Load 1 and Load 2 will serialize behind Store χ . This prevents the ordering violation. Unfortunately, this can introduce substantial additional store dependencies. Load 1 must wait for all stores in Load 2's store set, and vice versa. If all Stores A through F, and Store χ are simultaneously present in the STQ, then Loads 1 and 2 will be considerably delayed, as illustrated in Figure 16(b). With the store vector approach, each load may have its own store vector that is capable of tracking dependencies independent of all other loads (modulo aliasing effects in the SVT). Figure 16(c) shows the corresponding store vectors for Loads 1 and 2 that do not result in the spurious serializations induced by store sets.

The LWT, though simple in design, does not equal the performance of an oracle predictor due to its conservative nature. As described earlier, the LWT does not store identifying information about the colliding load-store pair but only marks suspected loads. This marking then prevents the load from issuing before all previous unresolved stores. Thus LWT does not take into account the various dynamic instances of store-load pairs and makes predictions that can tend to be too conservative. Much like the untagged SVT, the LWT too has many entries that are zero and do not hold any valuable information.

7. IMPLEMENTATION COMPLEXITY

Having described the various implementations of the predictors as well as presented the performance of these predictors in different processor configurations, we now compare the implementation complexities of these predictors. The basic units that handle memory instructions are the load queue and the STQ. These structures contain CAM logic to provide memory dependence checks and store forwarding. We now discuss how the load and store queues must be modified to support the different prediction algorithms along with any additional hardware structures/overhead that are required.

The LWT is the simplest dependence predictor that we evaluated and, as such, requires the fewest modifications to the existing memory scheduling structures. In a processor without speculative load-store reordering, loads must wait until all previous (older) stores have resolved their addresses. A load's "readiness" is now effectively gated by this all-earlier-stores-resolved signal. To implement the LWT predictor, we simply compute the logic OR of this signal with the LWT's prediction (assuming a prediction of 1 indicates that the load is predicted to have no dependencies; an extra inverter takes care of the case of the reversed prediction encoding). Apart from the predictor table itself, which can be accessed early in the pipeline off the critical path, there are no other changes to the memory scheduling structures. While very simple to implement, our earlier performance results showed that the performance benefits of the LWT approach are limited when we consider larger processor configurations.

Store sets can be implemented using either a CAM, or RAM-based organization, as explained in Section 2.2. Assuming a CAM implementation, the addition of a second set of CAMs in both the load and store queues represents a very large area and power overhead that impacts the scalability of the already difficult to scale load and store queues. These CAMs need as many ports as the maximum per-cycle store issue rate. Even for current load and STQ sizes, implementing these additional CAMs will not likely be able to meet clock frequency targets or would otherwise require very deep and complex pipelined accesses. These CAMs, however, are not as bad as traditional CAMs designed for load and store queues, since they require smaller comparator widths and do not require age support.

When using a RAM-based implementation, the maximum number of loads per store set either makes the load queue difficult to scale (too many write ports, even though they are direct/RAM accessed) or requires some overflow handling. A third alternate implementation of store sets can use a matrix-based memory dependence scheduler similar to the one used in the store vectors algorithm. This matrix would track the dependencies between the instructions in a store set. The matrix, however, would have to be considerably larger as we need one row per load and per store, since stores need to wake up other stores belonging to the same store set.

As previously discussed in Section 2, branch misprediction recovery introduces further complexities for cleaning up broken store-to-load pointers/indexes. The recurrence induced by intragroup store set dependencies also makes the implementation of the LFST difficult for a wide-issue processor

without placing additional constraints of load/store dispatch/allocation rates. The speculative update of the LFST with wrong-path store instructions also requires additional complexity to repair the LFST's state after a branch mispredict detection. The same types of techniques employed for repairing the Register Alias Table (RAT) could potentially be employed here. RAT/mapper checkpointing is a commonly used technique for rapid recovery of the renaming state [Kessler 1999], but this approach is more costly for the LFST. The RAT typically only contains a number of entries equal to the size of the architected register file, whereas the LFST contains 256 entries, thereby requiring more storage per checkpoint. If the processor employs an "undo-list" approach for pipeline recovery where the recovery logic walks the ROB and incrementally undoes the RAT changes, then we can piggy-back on this logic to also repair the LFST entries at the same time. In either case, the recovery needed to support store sets adds a level of implementation complexity that store vectors does not require.

The store vectors algorithm needs a scheduling matrix that has a number of rows equal to the load queue size and a number of columns equal to the STQ size. This matrix only needs one wire per column, which is connected to the corresponding STQ entries for clearing the column when the store issues. Each row of the matrix needs a wired OR or wired AND circuit (depending on whether the logic is implemented as active-low or active-high), which indicates when the load is ready to issue. The SVT is a simple SRAM with the same port requirements as the basic LWT predictor. The SVT does not suffer from the dependency recurrence of store sets' LFST. The last significant component is a barrel shifter for rotating the vector to align it properly in the matrix. As the STQ size increases, the width of the shifter must also increase proportionately, which could make the shifter a critical timing path. Fortunately, this shifting operation can be very easily pipelined over several cycles. The SVT look-up and subsequent barrel shift can be started as soon as the decode stage has identified the instruction as a load (this may also require that the decode stage maintains a speculative STQ tail pointer, but this is also trivial to implement). For these reasons, the shifter does not impose any timing problems, although the area cost may be moderate, since we need to implement one shifter per load (up to however many loads may be decoded per cycle). The shifter requirements may be further reduced by using a tagged SVT and then allowing only one load with predicted store dependencies to proceed to the shifter per cycle. This works well, since the majority of loads are predicted to not have a dependency, which does not place much pressure on the shifter.

8. SUMMARY

Out-of-order load execution is necessary to realize the potential of large-window superscalar processors. Blindly allowing loads to execute, however, will result in ordering violations which may cancel out the benefits of supporting a large number of in-flight instructions. We have proposed a new memory dependence prediction and scheduling algorithm based on dependency vectors and scheduling matrices. Our store vectors approach yields performance results better than

the state-of-the-art store sets algorithm while maintaining a simpler implementation. While the results for current processor configurations indicate that even an LWT predictor provides sufficient performance, future processors with larger instruction windows need a more sophisticated predictor like store vectors to achieve performance close to that of an oracle predictor. Another important result in this study is that by changing the store tracking strategies (MRCST and store window), we are able to obtain nearly an 80% hardware savings as compared to the original implementation with minimal impact in performance. By using a tagged predictor, we are able to decrease the hardware required by about 70% again with minimal loss in performance, while using a tagged predictor tracking only the most recently conflicting store provides comparable performance to the original store vectors design while providing over 90% hardware savings. Thus, depending on the given hardware requirements, store vectors can be easily modified and still perform very well. Our results also show that, while performing substantially better than previous predictors, there remains some performance left between store vectors and an ideal predictor (on large machine configurations), which suggests that further innovation may help to extract the remaining performance.

APPENDIX

Here, we include additional results in Tables II through V, which provide a comparison in terms of performance of all the memory dependence predictors over the entire set of applications. Due to space limitations, we present the tabulated results only for the medium and the extra-large configurations. While the base IPC refers to blind prediction, perfect depicts the performance with an oracle predictor. The results indicate that store vectors nearly reach the performance of a perfect predictor for most applications.

Table II. The Performance of the Memory Dependence Predictors Simulated on the “Medium” Configuration for the Entire Set of Applications

Benchmark Name			Base IPC	LWT	St. Sets	St. Vectors	Perfect
adpcm-dec	M		3.83	0.0%	0.0%	0.0%	0.0%
adpcm-enc	M		1.58	0.0%	0.0%	0.0%	0.0%
ammp	F		1.32	0.0%	0.0%	0.0%	0.0%
anagram	P	x	2.06	11.5%	11.6%	11.7%	11.7%
applu	F		0.72	0.0%	0.0%	0.0%	0.0%
apsi	F	x	2.14	1.8%	1.8%	1.9%	1.9%
art-1	F		1.53	0.0%	0.0%	0.0%	0.0%
art-2	F		1.49	0.0%	0.0%	0.0%	0.0%
bc-1	P	x	2.17	5.6%	6.0%	6.8%	6.0%
bc-2	P	x	1.11	2.9%	3.2%	3.6%	3.5%
bc-3	P		1.15	0.2%	0.2%	0.2%	0.3%
bzip2-1	I		1.56	0.3%	0.5%	0.5%	0.6%
bzip2-2	I		1.59	0.0%	0.1%	0.0%	0.1%
bzip2-3	I		1.70	−0.3%	−0.3%	−0.3%	0.4%
clustalw	B		3.33	0.3%	0.3%	0.3%	0.3%
crafty	I	x	1.24	6.1%	6.3%	6.2%	6.3%
crc32	E		2.62	0.0%	0.0%	0.0%	0.0%
dijkstra	E		2.15	0.0%	0.0%	0.0%	0.1%
eon-1	I	x	1.23	17.0%	19.9%	18.8%	19.4%
eon-2	I	x	0.91	3.4%	3.6%	3.4%	3.9%
eon-3	I	x	1.00	6.6%	7.1%	7.4%	8.1%
epic	M		1.86	0.0%	0.0%	0.0%	0.0%
equake	F		0.83	0.0%	0.0%	0.0%	0.0%
facerec	F	x	2.06	6.7%	6.7%	6.7%	6.7%
fft-fwd	E		1.98	0.0%	0.0%	0.0%	0.0%
fft-inv	E		1.98	0.0%	0.0%	0.0%	0.0%
fma3d	F	x	0.97	1.9%	1.9%	2.1%	2.0%
ft	P		2.12	0.3%	0.3%	0.4%	0.4%
g721decode	M	x	1.76	1.5%	1.5%	1.6%	1.6%
g721encode	M	x	1.64	1.1%	1.1%	1.2%	1.2%
galgel	F		3.31	0.0%	0.0%	0.0%	0.0%
gap	I		1.27	0.7%	0.6%	−0.2%	0.7%
gcc-1	I	x	2.31	0.6%	0.7%	1.0%	1.0%
gcc-2	I	x	0.76	−6.3%	4.3%	4.1%	4.8%
gcc-3	I	x	0.82	2.6%	2.6%	2.7%	2.8%
gcc-4	I	x	0.96	2.3%	2.5%	2.5%	2.6%
ghostscript-1	E	x	1.40	14.7%	15.0%	15.1%	14.8%
ghostscript-2	M	x	1.40	15.3%	15.5%	16.0%	15.5%
ghostscript-3	E	x	1.67	10.1%	11.2%	12.5%	12.7%
glquake-1	G	x	1.04	4.0%	4.1%	4.2%	4.3%
glquake-2	G	x	1.17	3.3%	3.3%	3.5%	3.6%
gzip-1	I		1.79	0.0%	0.0%	0.0%	0.1%
gzip-2	I		1.37	5.0%	0.3%	0.3%	0.3%
gzip-3	I	x	1.49	1.1%	1.2%	1.2%	1.3%
gzip-4	I		1.80	0.0%	0.0%	0.0%	0.0%
gzip-5	I		1.34	0.0%	0.3%	0.3%	0.3%
jpegdecode	M		2.12	0.5%	0.5%	0.7%	0.7%
jpegencode	M	x	1.72	3.7%	3.7%	3.8%	3.7%
ks-1	P		1.22	0.0%	0.0%	0.0%	0.0%
ks-2	P		0.78	0.0%	0.0%	0.0%	0.0%
lucas	F		1.00	0.0%	0.0%	0.0%	0.0%
mcf	I		0.58	0.0%	0.0%	0.0%	0.0%

An “x” signifies that the benchmark is dependency sensitive (>1% performance change between blind prediction and perfect prediction).

Table III. Continuation of Data from Table II

Benchmark Name		Base IPC	LWT	St. Sets	St. Vectors	Perfect
mesa-1	M	1.49	0.0%	0.0%	0.0%	0.0%
mesa-2	F x	1.50	17.9%	18.0%	18.2%	18.4%
mesa-3	M x	1.06	1.3%	1.3%	1.3%	1.3%
mgrid	F	1.26	0.1%	0.1%	0.1%	0.1%
mpeg2decode	M	2.26	0.2%	0.2%	0.3%	0.4%
mpeg2encode	M	3.12	0.1%	0.1%	0.1%	0.1%
parser	I x	1.26	6.9%	6.9%	6.9%	7.1%
patricia	E	1.09	0.1%	0.1%	0.3%	0.1%
perlbmk-1	I x	0.95	10.0%	10.5%	11.6%	11.3%
perlbmk-2	I x	1.22	1.7%	1.8%	2.4%	1.9%
perlbmk-3	I x	2.37	2.5%	2.1%	2.6%	2.7%
phylip	B	1.42	0.0%	0.0%	0.0%	0.0%
povray-1	G x	0.76	18.3%	18.6%	18.7%	19.1%
povray-2	G x	0.98	8.8%	8.7%	9.1%	9.3%
povray-3	G x	1.01	20.3%	20.8%	20.8%	21.3%
povray-4	G x	1.03	11.8%	11.8%	11.5%	12.0%
povray-5	G x	0.84	10.7%	10.9%	11.0%	11.2%
rsynth	E	1.62	0.1%	0.1%	0.1%	0.1%
sha	E x	2.88	8.8%	9.0%	8.8%	9.0%
sixtrack	F	1.30	0.2%	0.2%	0.2%	0.3%
susan-1	E	2.27	0.2%	0.2%	0.2%	0.2%
susan-2	E	1.64	0.0%	0.0%	0.0%	0.0%
swim	F	0.80	0.0%	0.0%	0.0%	0.0%
tiff2bw	E	1.51	0.0%	0.0%	0.0%	0.0%
tiff2rgba	E	1.63	0.2%	0.2%	0.2%	0.2%
tiffdither	E x	1.59	3.6%	3.7%	3.7%	3.9%
tiffmedian	E	2.46	0.1%	0.1%	0.1%	0.1%
twolf	I x	0.83	1.3%	1.2%	1.3%	1.3%
unepic	M	0.40	0.0%	0.0%	0.0%	0.0%
vortex-1	I x	1.56	31.0%	30.4%	31.7%	32.4%
vortex-2	I x	1.06	37.6%	37.6%	41.0%	39.9%
vortex-3	I x	1.45	35.0%	35.2%	36.7%	37.2%
vpr-1	I x	1.06	5.0%	5.0%	5.2%	5.2%
vpr-2	I x	0.65	3.5%	3.5%	3.5%	3.6%
wupwise	F	2.29	0.4%	0.5%	0.5%	0.5%
x11quake-1	G x	1.44	4.4%	4.4%	4.2%	4.7%
x11quake-2	G	2.74	−0.1%	−0.1%	−0.2%	0.1%
x11quake-3	G x	1.42	4.5%	4.5%	4.3%	4.9%
xanim-1	G	3.27	0.0%	0.0%	0.0%	0.0%
xanim-2	G x	2.86	0.9%	0.9%	1.0%	1.0%
xdoom	G x	2.06	4.7%	4.9%	5.3%	5.4%
yacr2	P x	2.10	2.5%	2.5%	2.6%	2.5%
ALL		1.44	3.7%	4.0%	4.2%	4.2%
SpecINT	I	1.22	5.3%	6.1%	6.3%	6.5%
SpecFP	F	1.37	0.7%	0.7%	0.7%	0.7%
MediaBench	M	1.69	2.8%	2.8%	2.8%	2.8%
MiBench	E	1.84	2.4%	2.5%	2.6%	2.6%
Graphics	G	1.41	6.7%	7.0%	7.0%	7.3%
PtrDist	P	1.49	2.8%	2.9%	3.1%	3.0%
BioBench	B	2.17	0.1%	0.1%	0.1%	0.1%
Dep. Sensitive	x	1.31	7.7%	8.1%	8.4%	8.5%

Table IV. The Performance of the Memory Dependence Predictors Simulated on the “Extra-Large” Configuration for the Entire Set of Applications

Benchmark Name			Base IPC	LWT	St. Sets	St. Vectors	Perfect
adpcm-dec	M		3.83	−0.02%	0.0%	0.0%	0.0%
adpcm-enc	M		1.58	−3.3%	0.0%	0.0%	0.0%
ammp	F	x	1.60	0.8%	1.34%	2.3%	4.4%
anagram	P	x	2.05	10.6%	12.1%	12.7%	19.5%
applu	F	x	0.92	−0.3%	0.0%	10.11%	12.5%
apsi	F	x	2.38	2.7%	1.1%	1.1%	1.5%
art-1	F	x	1.70	1.9%	0.0%	3.9%	3.8%
art-2	F	x	1.66	1.9%	0.0%	4.2%	4.1%
bc-1	P	x	2.18	2.2%	6.6%	8.7%	9.9%
bc-2	P	x	1.11	−2.2%	3.0%	5.7%	8.0%
bc-3	P		1.27	−1.9%	0.3%	0.7%	0.7%
bzip2-1	I	x	1.56	−2.3%	1.4%	2.2%	3.5%
bzip2-2	I	x	1.57	−3.0%	0.7%	5.36%	2.5%
bzip2-3	I	x	1.71	−3.1%	2.2%	3.1%	4.7%
clustalw	B		3.33	0.0%	0.2%	0.4%	0.3%
crafty	I	x	1.24	5.2%	7.0%	8.1%	7.5%
crc32	E		2.62	−11.0%	0.0%	0.0%	0.2%
dijkstra	E		2.15	−7.82%	0.0%	0.5%	0.4%
eon-1	I	x	1.26	13.9%	16.4%	19.8%	26.8%
eon-2	I	x	0.93	1.4%	4.19%	5.8%	29.0%
eon-3	I	x	1.02	5.2%	8.1%	10.7%	10.5%
epic	M		1.86	−1.3%	0.0%	0.0%	0.0%
equake	F	x	0.99	−0.4%	0.0%	19.4%	21.0%
facerec	F	x	2.13	9.0%	9.3%	16.1%	16.0%
fft-fwd	E		2.24	−2.5%	0.0%	0.4%	0.4%
fft-inv	E		2.24	−2.5%	0.0%	0.7%	0.4%
fma3d	F	x	1.09	5.3%	6.1%	25.0%	25.1%
ft	P	x	1.88	−1.9%	0.3%	0.1%	14.9%
g721decode	M	x	1.76	−0.3%	1.6%	1.8%	1.9%
g721encode	M	x	1.64	1.0%	1.1%	1.9%	1.9%
galgel	F		3.34	0.0%	0.0%	0.0%	0.0%
gap	I	x	1.27	−9.1%	0.5%	3.4%	4.2%
gcc-1	I	x	1.7	37.3%	35.8%	61.5%	61.7%
gcc-2	I	x	0.76	−9.2%	4.3%	5.7%	5.7%
gcc-3	I	x	0.82	−0.2%	2.7%	4.3%	4.1%
gcc-4	I	x	0.96	−0.2%	2.5%	4.4%	4.1%
ghostscript-1	E	x	1.40	17.5%	15.8%	15.6%	16.9%
ghostscript-2	M	x	1.66	18.0%	15.6%	14.8%	16.6%
ghostscript-3	E	x	1.40	12.2%	12.7%	12.7%	12.8%
glquake-1	G	x	1.04	2.6%	4.8%	5.6%	6.0%
glquake-2	G	x	1.17	2.8%	4.7%	5.3%	5.1%
gzip-1	I		1.79	−3.0%	0.0%	0.8%	0.6%
gzip-2	I		1.37	−11.1%	0.4%	0.4%	0.2%
gzip-3	I	x	1.49	−4.8%	1.1%	1.5%	1.6%
gzip-4	I		1.80	−3.7%	0.0%	0.6%	0.6%
gzip-5	I		1.34	−7.5%	0.3%	0.5%	0.8%
jpegdecode	M	x	2.21	−4.9%	1.2%	2.3%	2.3%
jpegencode	M	x	1.72	−0.6%	3.7%	4.5%	4.4%
ks-1	P		1.23	−9.9%	0.0%	0.0%	0.0%
ks-2	P		0.78	−10.5%	0.0%	0.0%	0.0%
lucas	F	x	1.05	−0.2%	0.0%	22.7%	22.4%
mcf	I	x	0.70	−3.8%	0.0%	12.5%	11.8%

An “x” signifies that the benchmark is dependency sensitive (>1% performance change between blind prediction and perfect prediction).

Table V. Continuation of Data from Table IV

Benchmark Name			Base IPC	LWT	St. Sets	St. Vectors	Perfect
mesa-1	M		1.49	−0.9%	−0.1%	0.4%	0.7%
mesa-2	F	x	1.60	37.3%	37.5%	40.5%	40.3%
mesa-3	M	x	1.06	−0.3%	1.07%	1.98%	45.8%
mgrid	F		1.41	−0.3%	0.0%	− 0.5%	0.0%
mpeg2decode	M		2.85	−0.1%	−0.4%	− 0.1%	0.1%
mpeg2encode	M		3.17	−0.2%	0.0%	− 0.6%	0.8%
parser	I	x	1.26	2.2%	7.4%	9.1%	8.9%
patricia	E		1.09	−0.8%	0.5%	0.0%	0.3%
perlbmk-1	I	x	0.95	5.9%	9.9%	13.9%	14.6%
perlbmk-2	I	x	1.22	−6.6%	−0.4%	4.3%	3.1%
perlbmk-3	I	x	2.36	−0.6%	2.2%	7.6%	9.1%
phylip	B	x	1.42	−1.0%	0.0%	1.5%	1.5%
povray-1	G	x	0.77	16.6%	20.7%	22.0%	22.5%
povray-2	G	x	1.02	7.3%	8.6%	10.2%	10.2%
povray-3	G	x	1.06	19.2%	22.8%	24.2%	24.62%
povray-4	G	x	1.04	11.2%	13.6%	14.4%	14.9%
povray-5	G	x	0.88	6.8%	9.2%	10.2%	10.1%
rsynth	E	x	2.59	−0.2%	0.9%	1.7%	1.7%
sha	E	x	2.85	12.3%	11.9%	11.8%	13.6%
sixtrack	F		1.38	0.0%	0.0%	0.4%	0.0%
susan-1	E	x	2.39	−0.5%	1.1%	4.7%	4.7%
susan-2	E		1.85	−0.1%	0.0%	0.0%	0.0%
swim	F		0.86	0.2%	0.0%	2.5%	0.0%
tiff2bw	E		1.52	−0.1%	0.0%	0.8%	0.8%
tiff2rgba	E	x	1.53	−0.2%	0.1%	30.6%	30.3%
tiffdither	E	x	1.59	1.6%	3.8%	4.3%	7.3%
tiffmedian	E		2.47	−0.6%	0.0%	0.5%	0.4%
twolf	I	x	0.86	−0.7%	1.4%	3.1%	1.3%
unepic	M		0.40	0.0%	0.0%	1.27%	0.0%
vortex-1	I	x	1.53	30.7%	30.3%	42.9%	46.3%
vortex-2	I	x	1.08	31.0%	33.4%	45.0%	46.3%
vortex-3	I	x	1.42	34.8%	36.6%	47.9%	55.2%
vpr-1	I	x	1.08	1.9%	5.4%	6.3%	6.1%
vpr-2	I	x	0.66	−0.8%	3.0%	16.9%	16.3%
wupwise	F	x	2.45	0.1%	0.4%	11.3%	12.1%
x11quake-1	G	x	1.58	2.6%	5.3%	6.2%	6.4%
x11quake-2	G		2.89	−0.4%	0.3%	− 2.5%	−2.7%
x11quake-3	G	x	1.57	2.6%	5.4%	6.4%	6.6%
xanim-1	G		3.27	−0.82%	0.0%	0.1%	0.2%
xanim-2	G	x	2.86	0.9%	1.3%	1.7%	1.6%
xdoom	G	x	2.07	4.4%	5.3%	6.4%	10.7%
yacr2	P	x	2.14	0.1%	1.7%	1.1%	1.7%

Benchmark Group			Base IPC	LWT	St. Sets	St. Vectors	Perfect
ALL			1.51	2.2%	4.7%	7.7%	8.8%
SpecINT	I		1.22	2.8%	7.2%	11.4%	12.7%
SpecFP	F		1.51	1.5%	1.3%	8.1%	8.4%
MediaBench	M		1.72	1.7%	3.9%	4.4%	7.2%
MiBench	E		1.92	0.9%	3.0%	5.3%	5.7%
Graphics	G		1.48	5.5%	7.5%	8.1%	8.6%
PtrDist	P		1.49	−1.9%	2.9%	5.31%	6.6%
BioBench	B		2.17	−0.5%	0.2%	0.9%	0.9%
Dep. Sensitive	x		1.35	6.7%	9.2%	11.8%	13.0%

REFERENCES

- AKKARY, H., RAJWAR, R., AND SRINIVASAN, S. T. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 423–434.
- ALBAYRAKTAROGLU, K., JALEEL, A., WU, X., FRANKLIN, M., JACOB, B., TSENG, C.-W., AND YEUNG, D. 2005. BioBench: A benchmark suite of bioinformatics applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, Los Alamitos, CA, 2–9.
- AUSTIN, T., LARSON, E., AND ERNST, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Micro Mag.* 59–67.
- AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. 1994. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 290–301.
- BREKELBAUM, E., II, J. R., WILKERSON, C., AND BLACK, B. 2002. Hierarchical scheduling windows. In *Proceedings of the 35th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 27–36.
- BROWN, M. D., STARK, J., AND PATT, Y. N. 2001. Select-free instruction scheduling logic. In *Proceedings of the 34th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 204–213.
- CAIN, H. W. AND LIPASTI, M. H. 2004. Memory ordering: A value-based approach. In *Proceedings of the 31st International Symposium on Computer Architecture*. ACM, New York, 90–101.
- CHRYSOS, G. Z. AND EMER, J. S. 1998. Memory dependence prediction using store sets. In *Proceedings of the 25th International Symposium on Computer Architecture*. ACM, New York, 142–153.
- CRISTAL, A., SANTANA, O. J., VALERO, M., AND MARTÍNEZ, J. F. 2004. Toward kilo-instruction processors. *ACM Trans. Archit. Code Optim.* 1, 4, 389–417.
- DOWECK, J. 2006. Inside Intel core microarchitecture and smart memory access. White paper, Intel Corporation. <http://download.intel.com/technology/architecture/sma.pdf>.
- GOSHIMA, M., NISHINO, K., NAKASHIMA, Y., ICHIRO MORI, S., KITAMURA, T., AND TOMITA, S. 2001. A high-speed dynamic instruction scheduling scheme for superscalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 225–236.
- GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Workshop on Workload Characterization*. IEEE, Los Alamitos, CA, 83–94.
- HILL, M. D. AND MARTY, M. R. 2008. Amdahl's law in the multicore era. *IEEE Comput.*
- INTEL. 2008. First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). White paper, Intel Corporation. <http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf>.
- KESSLER, R. E. 1999. The Alpha 21264 microprocessor. *IEEE Micro Mag.* 19, 2, 24–36.
- LARSON, E., CHATTERJEE, S., AND AUSTIN, T. 2001. MASE: A novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, Los Alamitos, CA, 1–9.
- LEBECK, A. R., KOPPANALIL, J., LI, T., PATWARDHAN, J., AND ROTENBERG, E. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th International Symposium on Computer Architecture*. 59–70.
- LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings of the 30th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 330–335.
- McFARLING, S. 1993. Combining branch predictors. Tech. rep. TN 36, Compaq Computer Corporation Western Research Laboratory. <http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf>
- MICHAUD, P. AND SEZNEC, A. 2001. Data-flow prescheduling for large instruction window in out-of-order processors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 27–36.
- MOSHOVOS, A. 1998. Memory Dependence Prediction. Ph.D. thesis, University of Wisconsin.

- MOSHOVOS, A., BREACH, S. E., VIJAYKUMAR, T. N., AND SOHI, G. S. 1997. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*. ACM, New York, 181–193.
- PALACHARLA, S. 1998. Complexity-Effective Superscalar Processors. Ph.D. thesis, University of Wisconsin.
- PATT, Y. N., PATEL, S. J., EVERS, M., FRIENDLY, D. H., AND STARK, J. 1997. One billion transistors, one uniprocessor, one chip. *IEEE Comput.*, 51–57.
- PERELMAN, E., HAMERLY, G., AND CALDER, B. 2004. Picking statistically valid and early simulation points. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. ACM, New York, 244–255.
- ROTH, A. 2005. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *Proceedings of the 32nd International Symposium on Computer Architecture*. ACM, New York, 458–468.
- SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECKLER, S. W., AND MOORE, C. R. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th International Symposium on Computer Architecture*. ACM, New York, 422–433.
- SASSONE, P. G., RUPLEY, J., BREKELBAUM, E., LOH, G. H., AND BLACK, B. 2007. Matrix scheduler reloaded. In *Proceedings of the 34th International Symposium on Computer Architecture*. ACM, New York, 335–346.
- SATO, T. AND ARITA, I. 2001. Revisiting direct tag search algorithm on Superscalar processors. In *Proceedings of the Workshop on Complexity-Effective Design*. IEEE, Los Alamitos, CA.
- SHA, T., MARTIN, M. M. K., AND ROTH, A. 2005. Scalable store-load forwarding via store queue index prediction. In *Proceedings of the 38th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 159–170.
- SHA, T., MARTIN, M. M. K., AND ROTH, A. 2006. NoSQ: Store-load communication without a store queue. In *Proceedings of the 39th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 159–170.
- SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. 1995. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*. ACM, New York, 414–425.
- SRINIVASAN, S. T., RAJWAR, R., AKKARY, H., GANDHI, A., AND UPTON, M. 2004. Continual flow pipelines. In *Proceedings of the 11th Symposium on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 107–119.
- SUBRAMANIAM, S. AND LOH, G. H. 2006. Store vectors for scalable memory dependence prediction and scheduling. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 64–75.
- SWANSON, S., MICHELSON, K., SCHWERIN, A., AND OSKIN, M. 2003. WaveScalar. In *Proceedings of the 36th International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 291–302.
- YOAZ, A., EREZ, M., RONEN, R., AND JOURDAN, S. 1999. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th International Symposium on Computer Architecture*. ACM, New York, 42–53.
- ZILLES, C. B. AND SOHI, G. S. 2000. Understanding the backward slices of performance degrading instructions. In *Proceedings of the 27th International Symposium on Computer Architecture*. ACM, New York, 172–181.

Received August 2006; revised June 2007, July 2008, January 2009; accepted June 2009