

# Store Vectors for Scalable Memory Dependence Prediction and Scheduling

Samantika Subramaniam   Gabriel H. Loh  
Georgia Institute of Technology  
College of Computing  
{samantik,loh}@cc.gatech.edu

## Abstract

*Allowing loads to issue out-of-order with respect to earlier unresolved store addresses is very important for extracting parallelism in large-window superscalar processors. Blindly allowing all loads to issue as soon as their addresses are ready can lead to a net performance loss due to a large number of load-store ordering violations. 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. These techniques tend to employ relatively complex CAM-based designs, which we believe have been obstacles to the industrial adoption of these algorithms. In this paper, we use the idea of dependency vectors from matrix schedulers for non-memory instructions, and adapt them to implement a new dependence prediction algorithm. For applications that experience frequent memory ordering violations, our “store vector” prediction algorithm delivers an 8.4% speedup over blind speculation (compared to 8.5% for perfect dependence prediction), achieves better performance than store sets (8.1%), and the store vector algorithm’s matrix implementation is considerably simpler.*

## 1. Introduction

Each successive technology generation provides more transistors to processor designers. However, an increase in the device count does not automatically 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) [22, 25, 7]. 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 [15, 18, 4],

creating large *effective* instruction windows [1, 26], or using non-conventional processor organizations [24, 27]. The effectiveness of a large instruction window for mining ILP is quickly limited by memory level parallelism (MLP) [9]. To maximize the effectiveness of a large instruction window, loads must be able to execute out-of-order with respect to unresolved store addresses.

Out-of-order load instruction scheduling is a non-trivial problem. Load instructions may have data dependencies through memory where an earlier store instruction writes a value to an address and the load instruction reads that value 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 issued. 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 relationship between loads and stores. However, a load instruction that issues too early due to a dependence misspeculation 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 [28] 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 been resolved. The communication of this dependency resolution typically requires hardware-intensive CAM logic. The load-store queue (LSQ) 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 a second set of CAMs to support memory dependence prediction is not practical because of the negative consequences on critical path latency of the LSQ and overall processor clock frequency.

Conventional implementations of non-memory instruction schedulers are also CAM-based. The associative logic causes the scheduler to be a timing critical path [21]. Dependency-vector/matrix scheduler organizations have been proposed for faster and more scalable implementations [10, 5]. In this paper, we propose a new memory dependence prediction and scheduling algorithm based on this idea of dependency vectors. Besides providing a more scalable hardware implementation, our “store vector” prediction algorithm also results in higher overall performance than the previous state-of-the-art CAM-based store sets approach.

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 of our store vector algorithm compared to naive speculation, store sets, and perfect dependence prediction. Section 4 also provides more detail on why store vectors perform better than store sets. Section 5 concludes the paper with a discussion on possible design alternatives if microarchitectural parameters were changed.

## 2. Background

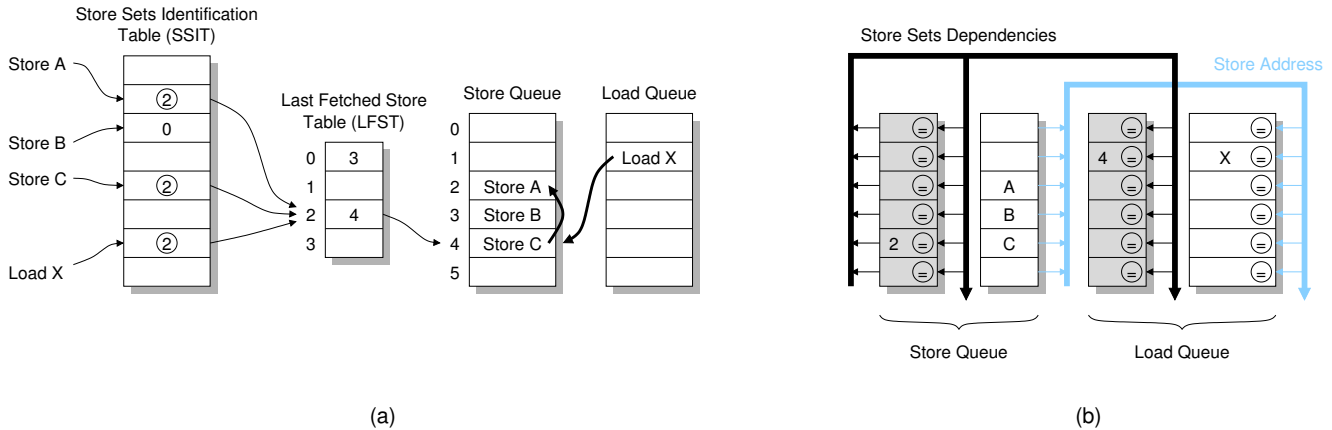
### 2.1. Memory Dependence Predictors

Like many other properties in microprocessors, memory dependencies exhibit a form of temporal locality. The concept of *memory dependence locality* was introduced by Moshovos [19]. 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 stores* 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 or not 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 *naive* 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. The Alpha 21264 employed a “store wait table” that tracks all loads that experienced ordering violations [13]. When a store executes and exposes a load ordering violation, the load’s address indexes into the store wait table where a bit is set. On subsequent instances of the load, the store wait table 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.

Status-based prediction schemes do not take instruction timing into account. A load that is predicted to have a dependence 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 non-dependent 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 a “timely” issuing of the load. 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 [19, 20]. When a load speculatively issues and violates a true dependency with an older store, the program counters (PC) of the load and the store are recorded in a table. Subsequent executions of the load instruction will check the table to see if they have conflicted with a store in the past. If so, the load will also check the store queue (using the store PC recorded in the table) to see if that store is present. If the store is present in the store queue, then the load must wait until the store has issued. This load-store pair approach has the advantage that mis-speculations 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 [19]. Chrysos and Emer generalized this approach with their “store sets” algorithm that allows one or more load in-



**Figure 1. (a) Store sets data structures, (b) LSQ hardware modifications (shaded) required to support store sets.**

structions to be associated with one or more stores [6]. Previous work showed that a store sets memory dependence predictor provides performance close to that of an oracle predictor. From our own simulations we observed that store sets delivers about 94% of the benefit that an oracle predictor provides over blind speculation. We use store sets as our baseline for comparison in this paper, and therefore we describe the algorithm in greater detail in the next section.

## 2.2. Store Sets

The store sets algorithm groups a load’s conflicting stores into one logical group or a *store set* [6]. Each store set has a unique identifier, called the store set identifier (SSID). Each load and each store may belong to one store set. Figure 1(a) 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 store instruction. The example in the figure shows Stores A and C and Load X belonging to store set number 2, while Store B belongs to store set number 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 store queue (STQ), each store updates the last fetched store table (LFST) which indicates the store queue 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 indi-

cated 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 Store A and Store C in Figure 1(a). 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. 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. Compared to a non-memory instruction scheduler, the load and store queue CAMs are much larger because they must deal with 64-bit addresses rather than 7-8 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 there exist multiple older stores to the same address, a load needs the age information to make sure that it receives its data from the more recent store. Implementing store sets requires a second set of

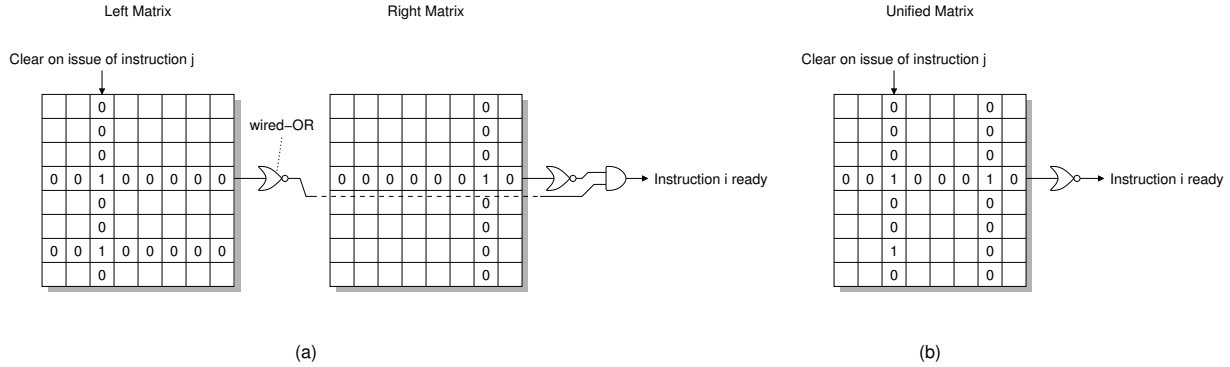


Figure 2. (a) Two-matrix scheduler, (b) single-matrix unified scheduler.

CAM logic and the associated broadcast buses to track the store sets dependencies, as illustrated by the shaded blocks in Figure 1(b).

Conventional non-memory instruction scheduling logic is already a timing critical path in modern processors [21]. The load and store queue circuits have similar complexity<sup>1</sup>, and therefore adding even more logic to implement store sets will likely have severe negative consequences on the processor clock frequency or the maximum load and store queue sizes. As processors increase the number of instructions in flight, the load and store queues will not scale well [8] and a CAM-based store sets implementation would only make the poor scaling worse. Furthermore, the LFST creates a new critical loop similar to the register renamer. 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 intra-group 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.

### 2.3. Scheduling Structures

Instruction schedulers typically use content addressable memory (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 that the critical path delay increases quadratically with the issue width and the number of entries [21].

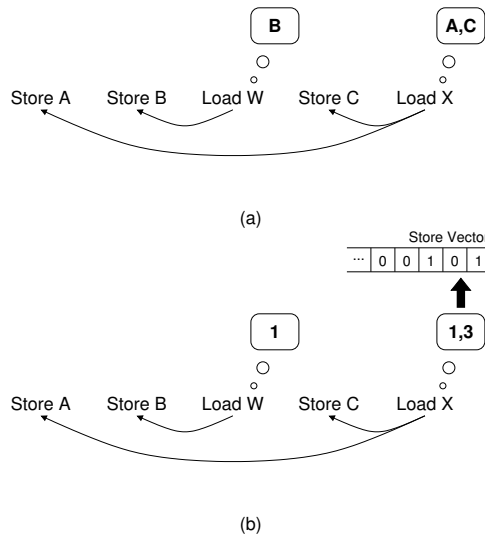
The dependency vector or dependency matrix scheduler organization is an alternative scheduler topology designed to be significantly more scalable. Goshima et al. proposed

<sup>1</sup>The previous discussion would make it seem like the load and store queues are *significantly* more complex, but the issue width of these structures tend to be less than that of non-memory instruction schedulers which somewhat balances out the complexity.

to replace the left and right CAM banks of a conventional scheduler with left and right dependency matrices as illustrated in Figure 2(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 multi-bit 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 2(b) illustrates a single-matrix implementation, suggested by Brown et al. Each matrix row can contain multiple non-zero entries to denote all of the dependencies [5], 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 program counters (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 3. The five load and store instructions are listed in program order, and Load



**Figure 3. (a) Store-address tracking of dependencies, and (b) store position or age tracking of dependencies.**

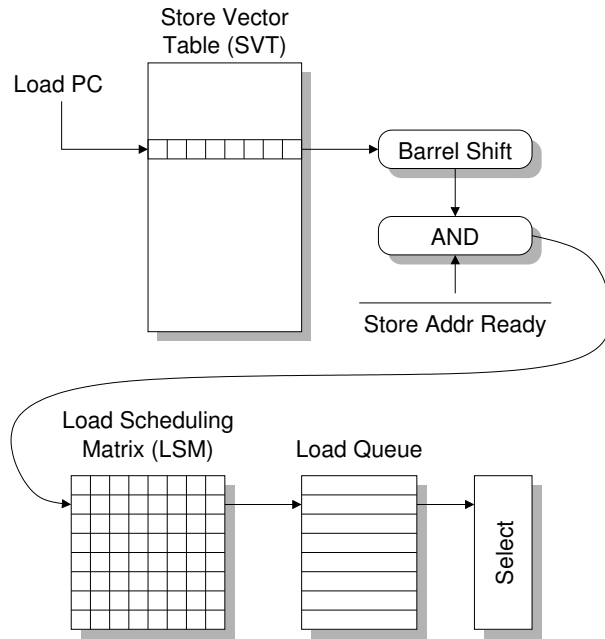
W has had ordering violations with Store B in the past, and Load X has had ordering violations with both Stores A and C. Figure 3(a) illustrates the PC-based dependency information used by previous pair or set-based approaches. In contrast, an age-based approach illustrated in Figure 3(b) stores the 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 3(b). In the general case, the length of the store vector will be equal to the number of store queue entries. The store vector could be optimized to only track store dependencies if the violating store was one of the  $k$  most recent stores with respect to the misspeculating load.

### 3.1. Step-by-Step Operation

The store vector algorithm has three main steps: lookup/prediction, scheduling, and update due to ordering violations. These are described in turn below, followed by an example.

**Lookup/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 4. 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,



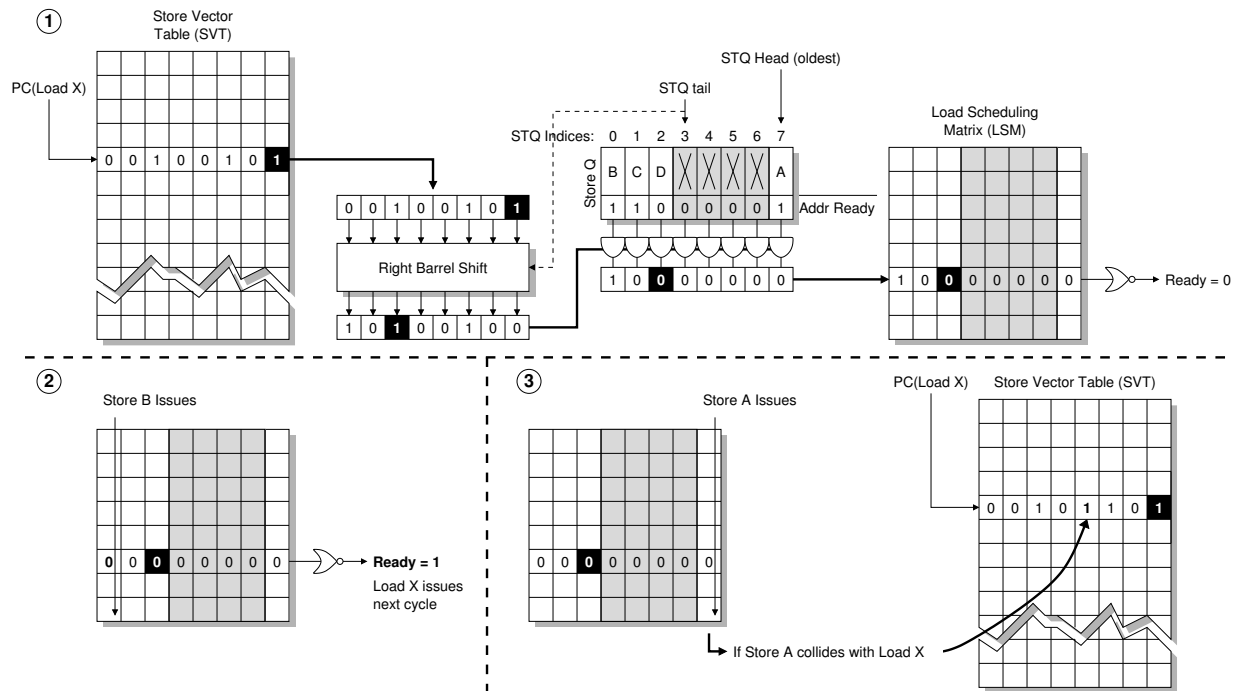
**Figure 4. Store vectors data structures and interaction with a conventional load queue.**

and one column for each store queue 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 bit-wise AND of the store vector with the bits from each store queue entry that indicates if the store's address is *not* ready. Finally, the vector is written into the matrix.

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



**Figure 5. Example store vector operation for the ② prediction phase, ③ scheduling phase, and ④ update phase.**

matrix is identical to the single-matrix scheduler described earlier in Figure 2(b). A wired-NOR determines if any unresolved predicted dependencies remain.

Each store queue 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.

**Update** — Initially, vectors in the SVT are initialized to all zeros. In this fashion, all load instructions will initially execute as if under a naive/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 store queue indexes of the most recent store and the store involved in the 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 21264 store wait table 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 5.

① **Prediction:** After decoding the instruction Load X, a hash of the load’s PC is used to select a vector from the Store Vector Table. The store vector bit that corresponds to the most recent store fetched before this load is indicated with shading. The 1’s 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 store queue (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 bar-

rel 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 non-zero bits in load’s store vector, the most recent store is Store A, the third most recent is Store C, 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 A had already issued and so Load X should not wait on Store A. Each entry of the store queue 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 A). This final store vector is written into the Load Scheduling Matrix 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 Load Scheduling Matrix. 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 1’s, and 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 store queues 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 re-fetched. 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 store queue 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 our Store Vector dependence prediction algorithm.

### 4.1. Evaluation Methodology

We use cycle-level simulation to evaluate the performance of memory dependence predictors. In particular,

Processor Width	6	IL1 Cache	16KB, 4-way, 3-cycle
Scheduler Size (RS)	64	DL1 Cache	16KB, 4-way, 3-cycle
Load Queue Size	32	L2 Cache	512KB, 8-way, 10-cycle
Store Queue Size	32	L3 Cache	4MB, 16-way, 30-cycle
ROB Size	256	Main Memory	250 cycles
Integer ALU/Mult	4/1	ITLB	64-entry, FA
FP ALU/Mult	2/1	DTLB	64-entry, FA
Memory Ports	2	L2 TLB	1024-entry, 8-way, 7-cycle

**Table 1. Simulated processor configuration.**

we use the MASE simulator [14] from the SimpleScalar toolset [2] 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 load-store queue), store sets and the store vector algorithms.

Our experiments use a 6-wide superscalar processor. Table 1 lists the processor parameters which were loosely based on the Intel Netburst microarchitecture (Pentium 4) [12]. We are not trying to model an Intel processor with an Alpha-based simulator; rather, we use Netburst as a configuration starting point to baseline the aggressiveness of our microarchitecture. The minimum latency from fetch to execution is twenty cycles, and we simulate a full in-order front-end pipeline as opposed to the default SimpleScalar behavior of simply stalling fetch for some number of cycles.

We simulated a variety of applications from SPEC2000, MediaBench [16], MiBench [11], Graphics applications including 3D games and ray-tracing, and pointer-intensive benchmarks [3]. 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 1.2 toolset to choose representative samples of 100 million instructions [23]. Using SimPoint avoids simulating non-representative portions of an application such as start-up and initialization code which can sometimes take up the first several billion instructions of a program’s execution. All “average” IPC speedups are computed using the geometric mean.

### 4.2. Performance Results

Our baseline processor configuration uses naive/blind speculation. We also use a perfect oracle dependence predictor as an “upper bound.” The oracle guarantees perfect dependency prediction, but this does not necessarily result in maximum performance because misspeculated loads can still have performance benefits due to prefetching effects or early branch misprediction detection. Table 2 lists the raw IPC rates for all of the benchmarks with blind speculation, and the relative performance impact of the different memory dependence predictors.

Note that the IPC rates of several applications are not sensitive to memory dependence prediction. In most of

Benchmark Name		Base IPC	St.Sets	St.Vectors	Perfect
adpcm-dec	M	3.83	0.0%	<b>0.0%</b>	0.0%
adpcm-enc	M	1.58	0.0%	<b>0.0%</b>	0.0%
ammp	F	1.32	0.0%	<b>0.0%</b>	0.0%
anagram	P x	2.06	11.6%	<b>11.7%</b>	11.7%
applu	F	0.72	0.0%	<b>0.0%</b>	0.0%
apsi	F x	2.14	1.8%	<b>1.9%</b>	1.9%
art-1	F	1.53	0.0%	<b>0.0%</b>	0.0%
art-2	F	1.49	0.0%	<b>0.0%</b>	0.0%
bc-1	P x	2.17	6.0%	<b>6.8%</b>	6.0%
bc-2	P x	1.11	3.2%	<b>3.6%</b>	3.5%
bc-3	P	1.15	0.2%	<b>0.2%</b>	0.3%
bzip2-1	I	1.56	0.5%	<b>0.5%</b>	0.6%
bzip2-2	I	1.59	0.1%	<b>0.0%</b>	0.1%
bzip2-3	I	1.70	-0.3%	<b>-0.3%</b>	0.4%
crafty	I x	1.24	6.3%	<b>6.2%</b>	6.3%
crc32	E	2.62	0.0%	<b>0.0%</b>	0.0%
dijkstra	E	2.15	0.0%	<b>0.0%</b>	0.1%
eon-1	I x	1.23	19.9%	<b>18.8%</b>	19.4%
eon-2	I x	0.91	3.6%	<b>3.4%</b>	3.9%
eon-3	I x	1.00	7.1%	<b>7.4%</b>	8.1%
epic	M	1.86	0.0%	<b>0.0%</b>	0.0%
equake	F	0.83	0.0%	<b>0.0%</b>	0.0%
facerec	F x	2.06	6.7%	<b>6.7%</b>	6.7%
fft-fwd	E	1.98	0.0%	<b>0.0%</b>	0.0%
fft-inv	E	1.98	0.0%	<b>0.0%</b>	0.0%
fma3d	F x	0.97	1.9%	<b>2.1%</b>	2.0%
ft	P	2.12	0.3%	<b>0.4%</b>	0.4%
g721decode	M x	1.76	1.5%	<b>1.6%</b>	1.6%
g721encode	M x	1.64	1.1%	<b>1.2%</b>	1.2%
galgel	F	3.31	0.0%	<b>0.0%</b>	0.0%
gap	I	1.27	0.6%	<b>-0.2%</b>	0.7%
gcc2k-1	I x	2.31	0.7%	<b>1.0%</b>	1.0%
gcc2k-2	I x	0.76	4.3%	<b>4.1%</b>	4.8%
gcc2k-3	I x	0.82	2.6%	<b>2.7%</b>	2.8%
gcc2k-4	I x	0.96	2.5%	<b>2.5%</b>	2.6%
ghostscript-1	E x	1.40	15.0%	<b>15.1%</b>	14.8%
ghostscript-2	M x	1.40	15.5%	<b>16.0%</b>	15.5%
ghostscript-3	E x	1.67	11.2%	<b>12.5%</b>	12.7%
glquake-1	G x	1.04	4.1%	<b>4.2%</b>	4.3%
glquake-2	G x	1.17	3.3%	<b>3.5%</b>	3.6%
gzip-1	I	1.79	0.0%	<b>0.0%</b>	0.1%
gzip-2	I	1.37	0.3%	<b>0.3%</b>	0.3%
gzip-3	I x	1.49	1.2%	<b>1.2%</b>	1.3%
gzip-4	I	1.80	0.0%	<b>0.0%</b>	0.0%
gzip-5	I	1.34	0.3%	<b>0.3%</b>	0.3%
jpegdecode	M	2.12	0.5%	<b>0.7%</b>	0.7%
jpegencode	M x	1.72	3.7%	<b>3.8%</b>	3.7%
ks-1	P	1.22	0.0%	<b>0.0%</b>	0.0%
ks-2	P	0.78	0.0%	<b>0.0%</b>	0.0%
lucas	F	1.00	0.0%	<b>0.0%</b>	0.0%
mcf	I	0.58	0.0%	<b>0.0%</b>	0.0%

Benchmark Name		Base IPC	St.Sets	St.Vectors	Perfect
mesa-1	M	1.49	0.0%	<b>0.0%</b>	0.0%
mesa-2	F x	1.50	18.0%	<b>18.2%</b>	18.4%
mesa-3	M x	1.06	1.3%	<b>1.3%</b>	1.3%
mgrid	F	1.26	0.1%	<b>0.1%</b>	0.1%
mpeg2decode	M	2.26	0.2%	<b>0.3%</b>	0.4%
mpeg2encode	M	3.12	0.1%	<b>0.1%</b>	0.1%
parser	I x	1.26	6.9%	<b>6.9%</b>	7.1%
patricia	E	1.09	0.1%	<b>0.3%</b>	0.1%
perlbmk-1	I x	0.95	10.5%	<b>11.6%</b>	11.3%
perlbmk-2	I x	1.22	1.8%	<b>2.4%</b>	1.9%
perlbmk-3	I x	2.37	2.1%	<b>2.6%</b>	2.7%
povray-1	G x	0.76	18.6%	<b>18.7%</b>	19.1%
povray-2	G x	0.98	8.7%	<b>9.1%</b>	9.3%
povray-3	G x	1.01	20.8%	<b>20.8%</b>	21.3%
povray-4	G x	1.03	11.8%	<b>11.5%</b>	12.0%
povray-5	G x	0.84	10.9%	<b>11.0%</b>	11.2%
rsynth	E	1.62	0.1%	<b>0.1%</b>	0.1%
sha	E x	2.88	9.0%	<b>8.8%</b>	9.0%
sixtrack	F	1.30	0.2%	<b>0.2%</b>	0.3%
susan-1	E	2.27	0.2%	<b>0.2%</b>	0.2%
susan-2	E	1.64	0.0%	<b>0.0%</b>	0.0%
swim	F	0.80	0.0%	<b>0.0%</b>	0.0%
tiff2bw	E	1.51	0.0%	<b>0.0%</b>	0.0%
tiff2rgba	E	1.63	0.2%	<b>0.2%</b>	0.2%
tiffdither	E x	1.59	3.7%	<b>3.7%</b>	3.9%
tiffmedian	E	2.46	0.1%	<b>0.1%</b>	0.1%
twolf	I x	0.83	1.2%	<b>1.3%</b>	1.3%
unepic	M	0.40	0.0%	<b>0.0%</b>	0.0%
vortex-1	I x	1.56	30.4%	<b>31.7%</b>	32.4%
vortex-2	I x	1.06	37.6%	<b>41.0%</b>	39.9%
vortex-3	I x	1.45	35.2%	<b>36.7%</b>	37.2%
vpr-1	I x	1.06	5.0%	<b>5.2%</b>	5.2%
vpr-2	I x	0.65	3.5%	<b>3.5%</b>	3.6%
wupwise	F	2.29	0.5%	<b>0.5%</b>	0.5%
x11quake-1	G x	1.44	4.4%	<b>4.2%</b>	4.7%
x11quake-2	G	2.74	-0.1%	<b>-0.2%</b>	0.1%
x11quake-3	G x	1.42	4.5%	<b>4.3%</b>	4.9%
xanim-1	G	3.27	0.0%	<b>0.0%</b>	0.0%
xanim-2	G x	2.86	0.9%	<b>1.0%</b>	1.0%
xdoom	G x	2.06	4.9%	<b>5.3%</b>	5.4%
yacr2	P x	2.10	2.5%	<b>2.6%</b>	2.5%

Benchmark Group		Base IPC	St.Sets	St.Vectors	Perfect
ALL		1.44	4.0%	<b>4.2%</b>	4.2%
SpecINT	I	1.22	6.1%	<b>6.3%</b>	6.5%
SpecFP	F	1.37	1.9%	<b>1.9%</b>	1.9%
MediaBench	M	1.69	0.7%	<b>0.7%</b>	0.7%
MiBench	E	1.84	2.5%	<b>2.6%</b>	2.6%
Graphics	G	1.41	7.0%	<b>7.0%</b>	7.3%
PtrDist	P	1.49	2.9%	<b>3.1%</b>	3.0%
<b>Dep. Sensitive</b>	x	1.31	8.1%	<b>8.4%</b>	8.5%

**Table 2. The performance of the memory dependence predictors. The base IPC is for a blind/naive predictor, and the IPC speedups of the remaining configurations are all relative to blind speculation. The letter after each benchmark denotes its application group, and an 'x' signifies that the benchmark is dependency sensitive (> 1% performance change between blind and perfect).**

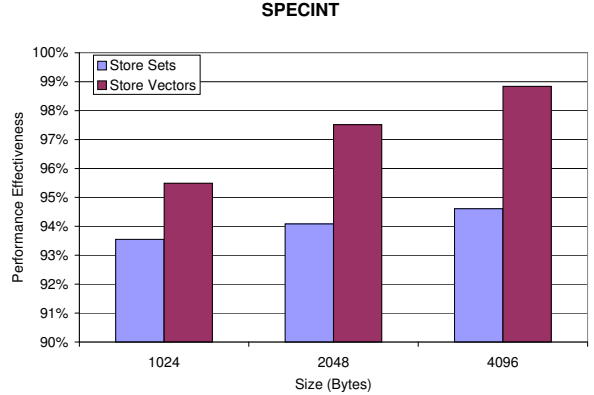
these cases, even blind speculation only results in a few (single digits) ordering violations over the course of 100 million instructions. We call an application *dependence-sensitive* if perfect dependence prediction results in at least a 1% IPC speedup over blind speculation. Each dependence-sensitive application is marked with an ‘x’.

Table 2 lists the performance of the store sets and store vector algorithms over blind speculation. Both predictors use 2KB of state in their prediction tables; we do not count the storage for the LFST against store sets. The bottom of Table 2 lists the overall speedups by application group as well as the speedup for those benchmarks categorized as being dependence-sensitive. The non-sensitive applications are not of great interest in this context because even perfect dependence prediction does not have any significant impact. Perfect prediction achieves an 8.5% speedup over blind speculation on the dependence-sensitive programs. Consistent with previously reported results, the store sets algorithm’s speedup of 8.1% is very close to the performance of perfect speculation. However, this performance comes with the price of complex hardware support. Our store vector prediction algorithm has a simpler implementation, and it also attains a speedup of 8.4% which is better than store sets.

On a per-group basis, there are clearly some program classes that are more sensitive to ordering violations than others. For example, the floating point and media programs typically have very regular data access patterns (vector/matrix or streaming) that result in few store-to-load forwardings. It may be surprising that the pointer-intensive benchmarks are not more sensitive to memory dependencies, however, these applications spend much more time traversing pointer-based data structures (loads) rather than modifying the structures (stores).

There are a few applications where store sets performs slightly better than the store vector algorithm. In these cases, there were more memory ordering violations overall in the blind speculation case. Store sets’ set merging heuristics cause store sets to be slightly more conservative, which is why it results 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 the baseline case. There are also a few cases where either store sets or store vectors outperform perfect dependence prediction due to the prefetching and early branch mispredict detection effects mentioned earlier.

The performance speedup of store sets and store vectors is very close to that of perfect dependence prediction. To make it easier to distinguish between the algorithms, we define the *performance effectiveness* of a memory dependence



**Figure 6. Percent of perfect prediction’s performance achieved by store sets and store vectors for 1KB, 2KB and 4KB hardware budgets.**

predictor  $P$  as:

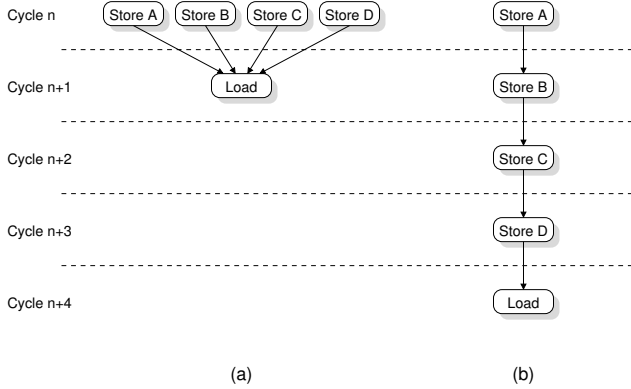
$$\text{Performance Effectiveness}(P) = \left( \frac{1 - \frac{\text{IPC}_P}{\text{IPC}_{\text{blind}}}}{1 - \frac{\text{IPC}_{\text{perfect}}}{\text{IPC}_{\text{blind}}}} \right)$$

This metric has no physical significance; it simply measures the performance benefit of a prediction algorithm  $P$  as a fraction of what is achievable with perfect dependence prediction. For example, if  $\text{IPC}_{\text{blind}} = 1.0$ ,  $\text{IPC}_{\text{perfect}} = 1.2$  and  $\text{IPC}_P = 1.15$ , then  $P$ ’s performance effectiveness is 75%. Figure 6 shows the performance effectiveness of store sets and store vectors for three different predictor sizes on the SPECint benchmarks. As the hardware budget increases from 1KB to 4KB, the performance effectiveness of store sets ranges from 93.5% up to 94.7%. For our store vector algorithm, the performance effectiveness ranges from 95.5% to almost 99%. For both algorithms, further increases in predictor hardware budgets do not result in any significant increases in performance.

### 4.3. Why Do Store Vectors Work?

In this section, we explain the advantages of store vectors over store sets. In particular, we will discuss scenarios where the store sets algorithm results in overly conservative predictions or unnecessary delays.

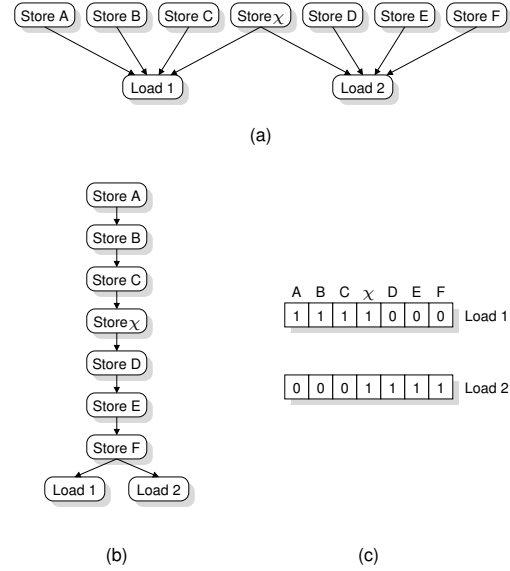
The first reason why store sets can delay a load from issuing actually stems from delaying the issuing of stores. The desired behavior of a load with a predicted set of store dependencies is that the load waits for all of the dependencies to be resolved. Figure 7(a) shows a load instruction with four 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 store queue. To prevent loads



**Figure 7. (a) Ideal load synchronization against predicted store dependencies, and (b) actual serialization with store sets.**

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 7(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 8(a), where both loads have several predicted store dependencies, and both loads are predicted to be dependent on Store- $\chi$ . Without store sets merging, Store- $\chi$  can only belong to the store set of Load-1, for example. In this situation, Load-2 will not wait for Store- $\chi$  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- $\chi$ . 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 visa-versa. If all Stores A through F, and Store- $\chi$  are simultaneously present in the store queue, then Loads 1 and 2 will be considerably delayed as illustrated in Figure 8(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 8(c) shows the corresponding store vectors for Loads 1 and 2 which do not result in the spurious serializations induced by store sets.

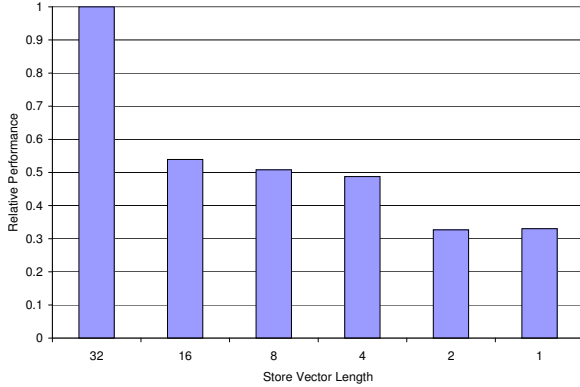


**Figure 8. (a) Ideal load synchronization for two loads against their predicted store dependencies, (b) actual serialization of merged store sets, and (c) the store vector values which avoid unnecessary serialization.**

#### 4.4. Design Alternatives

As described, the store vector length is tied to the number of entries in the store queue. 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 the rest of the paper) would reduce the space requirements of the Store Vector Table (SVT) by one half. However, the sizes of the remaining hardware structures are still tied to the store queue size. The Load Scheduling Matrix still requires one column per store, and therefore the barrel shifter must also produce a bit vector matched to the number of store queue entries.

For our processor configuration, the benefit of store vectors drops off relatively quickly with shorter store vector lengths. Figure 9 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. If a data dependency exists between two nearby instructions, it is likely that the value will be kept in a register as opposed to being pushed out to memory. Even for subroutine calls, most functions have only a few argu-



**Figure 9. 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.**

ments which can be passed through the registers. However, 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 is much more frequent.

For our simulations, the effects of control flow on dependence prediction performance were negligible. 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. The store vector will mark bits three and four as conflicts, which may in turn cause the load to be overly conservative. 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 non-conflicting 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 dependencies within very branchy code, the store vector table can be modified in a gshare fashion [17] to make use of some branch or path history information.

## 5. 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.

The results presented in this study are for a specific microarchitecture configuration. The performance benefits of store vectors (and store sets) may change given different assumptions about the underlying microarchitecture. A larger instruction window may increase the potential cases for ordering violations. A deeper pipeline increases the penalty for misspeculating load-store dependencies. However, if the relative performance cost of being too conservative or too aggressive changes, the store vector algorithm can be easily modified and re-tuned for a new microarchitecture. For example, adding tags or partial tags to the SVT could improve prediction accuracy. In the case of a SVT miss, the default prediction could be set to blind speculation, no speculation (wait on all unresolved stores), or the predictor could even dynamically choose the default prediction based on the recent frequency of mispredictions. For our microarchitecture, the 2KB store vector predictor was able to deliver 97.5% of the performance benefit of an oracle dependence predictor, and so we did not place too much effort into further optimizing the prediction algorithm.

## Acknowledgments

This research was sponsored in part by equipment and funding donations from Intel Corporation. We are grateful for the constructive feedback provided by the anonymous reviewers.

## References

- [1] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 423–434, San Diego, CA, USA, May 2003.
- [2] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.
- [3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, FL, USA, June 1994.

- [4] Edward Brekelbaum, Jeff Rupley II, Chris Wilkerson, and Bryan Black. Hierarchical Scheduling Windows. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 27–36, Istanbul, Turkey, November 2002.
- [5] Mary D. Brown, Jared Stark, and Yale N. Patt. Select-Free Instruction Scheduling Logic. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 204–213, Austin, TX, USA, December 2001.
- [6] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 142–153, Barcelona, Spain, June 1998.
- [7] Adrián Cristal, Oliverio J. Santana, Mateo Valero, and José F. Martínez. Toward Kilo-Instruction Processors. *Transactions on Architecture and Code Optimization*, 1(4):389–417, December 2004.
- [8] Amit Gandhi, Haitham Akkary, Ravi Rajwar, Srikanth T. Srinivasan, and Konrad Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 446–457, Madison, Wisconsin, June 2005.
- [9] Andy Glew. MLP Yes! ILP No! Memory Level Parallelism, or, Why I No Longer Worry About IPC. In *Proceedings of the ASPLOS Wild and Crazy Ideas Session*, San Jose, CA, USA, October 1997.
- [10] Masahiro Goshima, Kengo Nishino, Yasuhiko Nakashima, Shin ichiro Mori, Toshiaki Kitamura, and Shinji Tomita. A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 225–236, Austin, TX, USA, December 2001.
- [11] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the 4th Workshop on Workload Characterization*, pages 83–94, Austin, TX, USA, December 2001.
- [12] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyler, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 2001.
- [13] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro Magazine*, 19(2):24–36, March–April 1999.
- [14] Eric Larson, Saugata Chatterjee, and Todd Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, pages 1–9, Tucson, AZ, USA, November 2001.
- [15] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 59–70, Anchorage, AK, USA, May 2002.
- [16] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, NC, USA, December 1997.
- [17] Scott McFarling. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.
- [18] Pierre Michaud and André Seznec. Data-Flow Prescheduling for Large Instruction Window in Out-of-Order Processors. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 27–36, Monterrey, Mexico, January 2001.
- [19] Andreas Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin, 1998.
- [20] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 181–193, Boulder, CO, USA, June 1997.
- [21] Subbarao Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin, 1998.
- [22] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One Billion Transistors, One Uniprocessor, One Chip. *IEEE Computer*, pages 51–57, September 1997.
- [23] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, New Orleans, LA, USA, September 2004.
- [24] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 422–433, San Diego, CA, USA, May 2003.
- [25] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, Santa Margheruta Ligure, Italy, June 1995.
- [26] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual Flow Pipelines. In *Proceedings of the 11th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Boston, MA, USA, October 2004.
- [27] Steve Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 291–302, San Diego, CA, USA, May 2003.
- [28] Craig B. Zilles and Gurindar S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 172–181, Vancouver, Canada, June 2000.