Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors*

Marcelo Cintra†
Division of Informatics
University of Edinburgh
mc@dcs.ed.ac.uk

Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
torrellas@cs.uiuc.edu

ABSTRACT

With speculative thread-level parallelization, codes that cannot be fully compiler-analyzed are aggressively executed in parallel. If the hardware detects a cross-thread dependence violation, it squashes offending threads and resumes execution. Unfortunately, frequent squashing cripples performance.

This paper proposes a new framework of hardware mechanisms to eliminate most squashes due to data dependences in multiprocessors. The framework works by learning and predicting violations, and applying delayed disambiguation, value prediction, and stall and release. The framework is suited for directory-based multiprocessors that track memory accesses at the system level with the coarse granularity of memory lines. Simulations of a 16-processor machine show that the framework is very effective. By adding our framework to a speculative CC-NUMA with 64-byte memory lines, we speed-up applications by an average of 4.3 times. Moreover, the resulting system is even 23% faster than a machine that tracks memory accesses at the fine granularity of words – a sophisticated system that is not compatible with mainstream cache coherence protocols.

1 INTRODUCTION

Despite advances in compiler technology, compilers still fail to automatically parallelize many codes. Typically, compilers abstain from parallelizing codes with complex or unknown data dependences, such as those that contain pointer accesses, references to arrays with non-linear subscripts, very irregular control flow, or accesses across complicated procedure calling patterns.

To extract parallelism in such codes, speculative thread-level parallelization has been proposed [1, 4, 6, 7, 8, 10, 12, 18, 19, 20, 22, 23, 25, 26, 32]. In this approach, potentially dependent threads are speculatively executed in parallel, hoping not to violate dependences. If a cross-thread dependence is violated at run time, a corrective action is triggered to repair the state. Such an action often involves squashing one or several threads.

Proposed schemes for speculative parallelization differ in many ways. For example, some schemes rely on support code inserted by the compiler to check for dependence violations and to perform corrective actions [7, 19, 20]. Other schemes rely on special hardware to perform some or all of these operations [1, 4, 6, 8, 10, 12, 18, 22, 23, 25, 26, 32].

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, and EIA-0072102; by DARPA under grant F30602-01-C-0078; and by gifts from IBM and Intel.

†Work conducted in part while the author was with the Department of Computer Science at the University of Illinois at Urbana-Champaign.

In most schemes, however, squashing a thread due to a dependence violation and restarting it is a costly operation. Typically, the cost includes the overhead of the squash operation itself, the loss of the work performed, and the cache misses necessary to reload state after restart. Squashes are especially costly in large multiprocessors, where the overhead of squashing distributed threads is high and the typical coarse grain of the threads likely implies more wasted work.

Unfortunately, squashes due to data dependence violations can be frequent. One reason is that the coherence protocol of multiprocessors typically operates at the granularity of memory lines. This is because word-based accesses usually exhibit suboptimal locality and result in increased traffic. However, line-based systems are subject to false sharing, which may appear to violate dependences.

Another reason for frequent dependence violations may be the early stage of development of compilation support for speculative parallelization [15, 16, 24, 28]. Specifically, compilers may occasionally make poor assessments of data dependences and attempt to speculatively parallelize codes with many dependences.

One approach to reduce the number of squashes is to attempt to learn at run time where the cross-thread dependence violations occur. Then, when we predict that one such violation is about to occur, we can prevent it from taking place. Past work has used these ideas to dynamically synchronize dependent load-store pairs in a uniprocessor [5, 21, 31], or even to dynamically synchronize dependent threads in a tightly-coupled multiscalar processor [14].

In this paper, we focus on how to eliminate squashes through run-time dependence learning in the distributed architecture of a directory-based CC-NUMA. Clearly, the problem that we address is different than the one addressed by previous work. Indeed, we cannot afford any centralized learning structure and there is no global context readily available. Moreover, we must largely rely on memory access information that is only available at the grain size of memory lines.

We propose a new framework of hardware mechanisms to eliminate most squashes due to data dependences in directory-based multiprocessors. The framework works by learning and predicting violations, and applying delayed disambiguation for false dependences, value prediction for same-word dependences, and stall and release for unpredictable, same-word dependences.

The framework works with multiprocessors that track system-level memory accesses at the coarse granularity of memory lines. Simulations of a 16-processor machine show that the framework is very effective. By adding the framework to a speculative CC-NUMA with 64-byte memory lines, we speed-up applications by an average of 4.3 times. Moreover, the resulting system is even 23% faster than a machine that tracks memory accesses at the fine granularity of words. Such fine-grain access tracking is not compatible with mainstream cache coherence protocols.

Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)
1503-0897/02 $17.00 © 2002 IEEE

This paper is organized as follows: Section 2 discusses background concepts; Section 3 presents the proposed framework; Section 4 outlines its implementation; Section 5 discusses the evaluation methodology; Section 6 evaluates the framework; Section 7 discusses related work; and Section 8 concludes the paper.

2 SPECULATIVE PARALLELIZATION

2.1 Basic Concepts

Speculative thread-level parallelization consists of extracting threads from sequential code and running them in parallel, hoping not to violate sequential semantics. Threads in a speculative system are typically classified into a single *non-speculative* thread and a set of *speculative* ones that can generate unsafe state. In speculative threads, stores generate speculative versions of variables, and loads that do not find a local version must get the most up-to-date one from a predecessor thread or memory. In such a system, a thread *commits* when it has finished its execution and is non-speculative. In the speculation protocol that we use in this paper, when a thread commits, it writes-back to main memory all the dirty lines in its cache [4]. Furthermore, a speculative thread becomes non-speculative only when all its predecessors have committed.

As execution proceeds, the system tracks memory accesses to identify any cross-thread data dependence violation. If one is found, the offending thread is *squashed*. Typically, to simplify the protocol, its successor threads are also squashed. Thread squash is a very costly operation. The cost is three-fold: overhead of the squash operation itself, loss of the work already performed by the offending thread and successors, and cache misses in the offending thread and successors needed to reload state after restarting. The latter overhead appears because, as part of the squash operation, the speculative state in the cache is invalidated. Figure 3a shows an example of a RAW violation across threads i and $i+j+1$. The consumer thread and its successors are squashed.

2.2 General Architectural Model

A few designs have been proposed to support speculative thread-level parallelization in directory-based architectures [4, 18, 23, 32]. In the discussion of our framework, we assume a model where a Speculation Module is added to the directory controller of the CC-NUMA machine (e.g., as in [4]). The module keeps track of the mapping of threads to processors and their ordering.

The speculation module is also responsible for detecting any data dependence violation. It does so by recording what data has been speculatively accessed by what thread with an exposed load or a store. An exposed load is a load to a location by a thread before the same thread has updated the location. If a violation is detected, the module squashes all the necessary threads [4].

2.3 Granularity of Access Tracking

Speculation protocols can be classified depending on whether they track speculative accesses with word or line granularity. Per-word protocols only need to squash threads in cross-thread RAW violations on the same word. However, they require costly per-word state support in caches, network messaging, and directory modules. They also tend to generate higher traffic.

Per-line protocols are more cost-conscious and are compatible with mainstream cache coherence protocols. However, they cannot disambiguate accesses at word level. Furthermore, they cannot combine different versions of a given line that have been updated in different words. Consequently, cross-thread RAW and WAW violations, on both the same and different words of a line, cause squashes.

A per-line protocol with per-word extensions in the cache hierarchy is a compromise that effectively reduces squashing to the case of RAW violations on same or different words of a line. Each cache line is augmented with one *Store*, one *exposed Load*, and one *Valid* bit per word. The first bit indicates that the local thread has updated the word. The second bit, that the local thread has issued an exposed load to it. The Valid bits indicate which words of the line are valid and allow for invalidation of individual words. Line write-backs to memory piggy-back the Store bits of the line, so that the directory controller can successfully combine different versions of the line. Hence, all cross-thread WAW dependences are supported without causing squashes. Still, the directory state is kept per line. In this paper, we use this protocol as our baseline.

In all the protocols considered in this paper, we disable support for the forwarding of uncommitted dirty data between caches. We do this to simplify the implementation of the protocols. As a result, the violations described above include both out-of-order dependent accesses and in-order dependent accesses that need forwarding of uncommitted dirty data.

3 PROPOSED FRAMEWORK

Squashes due to cross-thread data dependence violations can cause major overheads under speculative parallelization in multiprocessors. One possible way to remove many such squashes is through learning where violations occur, predicting them, and then preventing them from taking place. The motivation for this general approach is that some form of learning, prediction, and synchronization of data dependences have been successfully used in uniprocessors and tightly-coupled chip multiprocessors [1, 5, 11, 13, 14, 21, 29, 30, 31].

In our work, we target directory-based CC-NUMA architectures. These architectures have distributed processors, caches, and directories, which require different solutions. In particular, we exploit the fact that, as indicated in Section 2.2, some speculative CC-NUMA designs have a speculation module in the directory controller (module 1 in Figure 1). Such a module can see all the accesses that are involved in dependence violations [4]. Consequently, we can build support for learning, predicting, and eliminating violations around that module.

In this section, we present a novel framework that uses these ideas. We first present an overview of the mechanisms (Section 3.1) and learning heuristics (Section 3.2) used. We then describe each mechanism in detail (Section 3.3) and some possible extensions (Section 3.4).

3.1 Overview of the Mechanisms

In conventional speculative systems, when the speculation module at the directory controller observes the two accesses involved in a RAW violation, it triggers a squash. In our framework, we try to eliminate such a squash with four mechanisms.

First, some squashes are unnecessary because there is no true data transfer between the threads involved: there is only false sharing. Since our baseline speculation protocol tracks accesses at the granularity of lines at the directory, false sharing causes squashes. To eliminate these squashes, we optimistically let the consumer thread proceed. However, before we allow the thread to commit, we use the per-word access bits in its cache hierarchy to check whether or not it was false sharing. We call this mechanism *Delay&Disambiguate*.

Secondly, even when there is true data transfer between threads, a squash can be avoided with effective use of value prediction. Specifically, we predict the value that the producer will produce, speculatively provide it to the consumer, and let the latter proceed. Before we allow the consumer to commit, we check whether or not the value used was correct. We call this mechanism *ValuePredict*.

In cases where value prediction fails, we can avoid the squash

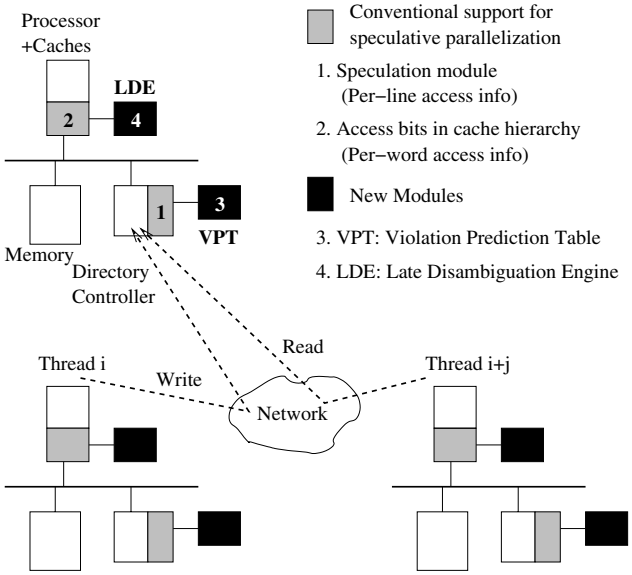

Figure 1: Model of speculative CC-NUMA used in this paper. The shaded areas are support that has been proposed elsewhere. The black areas are our proposed additions.

by stalling the consumer thread until the producer has produced the value. At that point, the consumer reads the produced value and can resume. This case has two possible mechanisms. An aggressive approach is to release the consumer thread as soon as the first producer thread commits. In this case, if an intervening thread between the first producer and the consumer later writes the line, the consumer will be squashed. We call this mechanism *Stall&Release*. A more conservative approach is not to release the consumer thread until it becomes non-speculative. In this case, the presence of multiple predecessor writers will not squash the consumer. We call this mechanism *Stall&Wait*.

3.2 Learning and Prediction Heuristics

To learn and predict what memory accesses cause violations, we use the line address requested by the access. Specifically, we add an extension to each speculation module called the *Violation Prediction Table (VPT)* (module 3 in Figure 1). The VPT dynamically keeps the address and other information for the lines that have been involved in potential or actual violations in the past. When the VPT observes an access to one such line, it triggers one of our four mechanisms.

To decide what mechanism to use for a particular line, the VPT uses the finite state machine of Figure 2. Initially, a line accessed speculatively is in the Plain Speculative state in the VPT, and the VPT takes no action. When the line appears to be involved in a potential violation, the VPT transitions to the Delay&Disambiguate state for the line (transition 1 in Figure 2). This engages the mechanism for handling false dependences. Note that, by default, we predict that the dependence will turn out to be false. We make the assumption that, in line-based protocols, false dependences are more likely than same-word dependences.

If the dependence turns out to be a same-word dependence, a squash is triggered and the VPT transitions to the ValuePredict state for the line (transition 2 in Figure 2). This engages the mechanism for value prediction. If further same-word violations occur in the line, the VPT transitions to the Stall&Release state for the line (transition 3 in Figure 2). This engages the mechanism for consumer stall and restart as soon as the first producer commits.

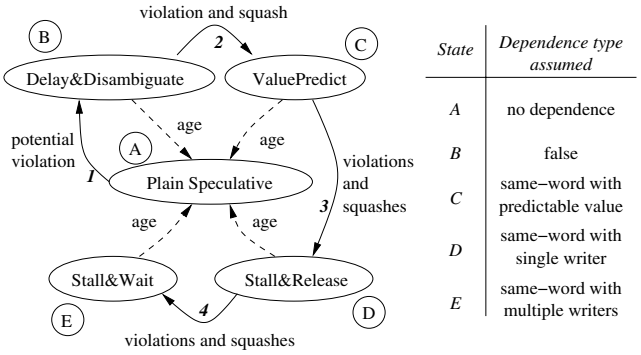

Figure 2: Finite state machine followed by the VPT for individual memory lines. Plain Speculative is the initial state.

Finally, if further violations occur in the line, the VPT transitions to the Stall&Wait state (transition 4 in Figure 2). This engages the mechanism for consumer stall and restart only when it becomes non-speculative. Any of these states can directly age back to the Plain Speculative state (dashed transitions in Figure 2).

3.3 Mechanisms Used

3.3.1 Delay&Disambiguate: False Dependences

For lines under the Delay&Disambiguate state, potential violations detected by the speculation module at the directory controller are assumed to be false. No squash is generated. Instead, the per-line speculation protocol operates mostly unaltered. Later, before the consumer thread is allowed to commit, access information from the producer and consumer threads are compared to perform word-address disambiguation.

Figures 3b and 3c show how a RAW violation is handled in this case. If the disambiguation shows the dependence to be false (Figure 3b), the consumer thread only sees a small overhead while the disambiguation is performed. However, if the dependence turns out to be for the same word (Figure 3c), the consumer thread and successors are squashed and restarted.

Implementing the Delay&Disambiguate mechanism requires the following: identifying potential violations that are likely to be false, remembering delayed unresolved violations, performing the delayed disambiguation, squashing threads when violations are confirmed, and learning which lines are involved in same-word violations.

This mechanism is triggered when the speculation module detects the second access of a RAW access pair to a line. At this point, the VPT checks if it has learned that the line has caused same-word dependences in the past. If it has not, the speculation module ignores the potential violation. At this point, the VPT transitions its state for the line to Delay&Disambiguate (Figure 2) and records the tag of the line involved and the ID of the consumer thread. The VPT also prepares a bit mask that will identify the words of the line that are actually written by predecessor threads. This mask is called the *Modified* mask, and is updated every time that a predecessor thread writes back the line to main memory at commit time. Recall that write-backs include bits that identify which words were written (Section 2.3).

When the consumer thread is about to become non-speculative, the Modified mask contains the record of the modifications by all the predecessor threads since the time of the potential violation. The VPT then sends the Modified mask to a new module associated with the cache hierarchy of the consumer node, along with a request for late disambiguation. Such a module is called the *Late Disambiguation Engine (LDE)* (module 4 in Figure 1). Before the

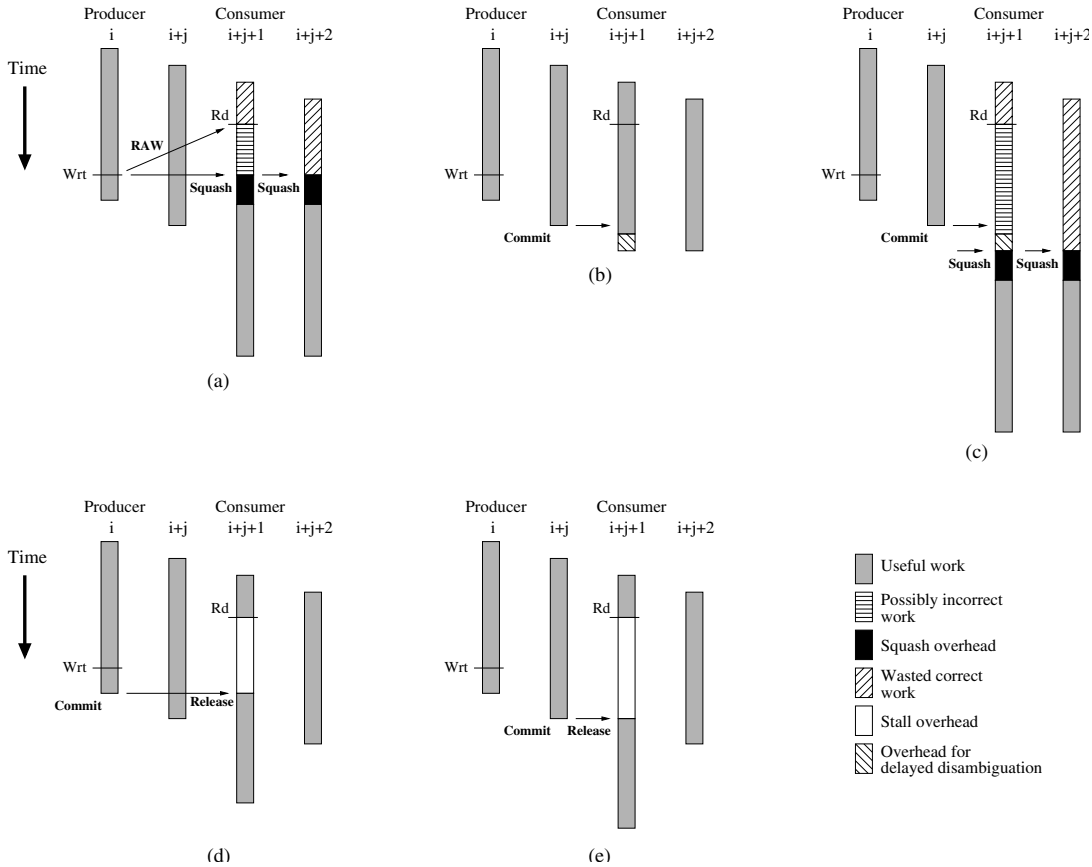

Figure 3: RAW violation with a squash (a); a successful delayed disambiguation (false RAW) or value prediction (b); a failed delayed disambiguation (same-word RAW) or value prediction (c); a stall with early release (d); and a stall with no early release (e).

consumer is allowed to commit, the LDE compares the Modified mask against the per-word Load and Store access bits of the corresponding line in the cache hierarchy of the consumer node (module 2 in Figure 1).

If there is an intersection between the Load bits and the mask, a same-word violation occurred and the thread is squashed. The VPT then transitions its state for the line to ValuePredict (Figure 2), effectively learning that the line exhibits same-word dependences. If no intersection is detected, the state in the VPT is kept as Delay&Disambiguate. Furthermore, any words in the line that both have the Store bit clear and are marked in the Modified mask, have become stale. Therefore, they are invalidated from the consumer cache hierarchy.

The effectiveness of Delay&Disambiguate is related to the fraction of delayed disambiguations that do not cause late squashes. We define such a fraction as the *selectivity* of the mechanism. Higher selectivity is better.

3.3.2 ValuePredict: Predicting Consumed Values

For lines under the ValuePredict state, the VPT provides a predicted value to the consumer thread on demand. The VPT remembers this value, which is finally compared with the correct value in the system when the consumer thread becomes non-speculative. If the prediction was correct the thread continues (Figure 3b). Otherwise, the thread and its successors are squashed (Figure 3c).

This mechanism requires: learning when to apply value prediction, predicting a value based on past observed values, comparing predicted consumed values with the actual values, and squashing threads if necessary.

When Delay&Disambiguate identifies a same-word violation, a squash is triggered and the VPT transitions immediately to ValuePredict for the line. Alternatively, we could wait and observe the predictability of the values before attempting value prediction. The choice here depends on the history depth required by the prediction mechanism. In this paper, we only investigate an *approximate last-value* prediction, which allows for immediate transition.

We design the mechanism as follows. When a thread attempts to consume data from a line marked ValuePredict in the VPT, the VPT provides a predicted value of the line. For simplicity, in our implementation the predicted value of the line is the last value written-back to main memory. Consequently, the VPT provides the current version of the line in memory. This prediction technique has also been called *value reuse* or *silent store*. The VPT also records the line tag, the value of the line provided, and the ID of the consumer thread.

When the thread finally becomes non-speculative, the VPT compares the prediction it made to the final value of the line. Recall that, in our protocol, such a value is the one currently in memory (Section 2.1). For words whose values differ, the VPT sets the corresponding bit in the Modified mask described in Section 3.3.1. The VPT then sends the Modified mask to the LDE in the consumer node, which performs local late disambiguation as described in Section 3.3.1. The state of the line in the VPT remains in ValuePredict unless a squash is required and such a squash increases the squash count for the line over a certain threshold.

The effectiveness of the ValuePredict mechanism depends on

the predictability of the values and on the accuracy of the predictor used.

3.3.3 Stall&Release: Waiting for the First Writer

For lines under the Stall&Release state, the VPT stalls the consumer thread when it tries to load the line. Later, when the producer thread commits, the consumer thread finally gets the line and continues. Figure 3d shows how a potential RAW violation is handled under the Stall&Release mechanism.

This mechanism requires: identifying the lines that cause same-word violations where value prediction fails, stalling threads before they are allowed to consume unsafe data, and releasing threads when the data is considered safe.

Before transitioning to Stall&Release for a line, the VPT counts the number of squashes caused by the line under ValuePredict. When a threshold is exceeded, the VPT learns that violations cannot be successfully handled with value prediction. At that point, after the squash is complete, the VPT transitions the state of the line to Stall&Release (Figure 2) and issues cache invalidations for the line to all speculative threads. These invalidations are selective in that, in each cache, they only invalidate the words in the line that have no local modified version.

We can now see how the Stall&Release mechanism works. When a consumer thread issues an exposed read to one of the words in the line, the request misses in the cache and is propagated to the directory. Since the VPT knows that the line is in state Stall&Release, it buffers the request and does not reply. Therefore, the consumer thread will eventually stall when it runs out of instructions to execute.

The safe time to release a stalled thread is when it becomes non-speculative. However, releasing it earlier may speed-up execution. The earliest time when a line can be forwarded and the consumer released is when the producer thread commits and writes back the modified line to memory. At this time, the line can be forwarded provided that no other predecessor of the stalled consumer thread has created a newer version of the line in its cache. If no such version exists, the VPT releases the buffered request and allows it to proceed, therefore effectively releasing the stalled thread. This is shown in Figure 3d, where the consumer thread is released as soon as the producer thread commits. However, if any predecessor of the released thread now writes to the line, a squash will occur.

The effectiveness of Stall&Release is measured in terms of coverage and selectivity. *Coverage* is the fraction of the squashes that are successfully avoided. *Selectivity* is the fraction of the stalls that actually eliminate squashes. Ideally, both should be high.

3.3.4 Stall&Wait: Waiting for Multiple Writers

For lines under the Stall&Wait state, the VPT also stalls the consumer thread when it tries to load the line. However, the VPT does not allow the thread to resume until the thread becomes non-speculative. At that point, it is completely safe for the thread to resume. Figure 3e shows how a potential RAW violation is handled under the Stall&Wait mechanism.

For a given line, the VPT reaches Stall&Wait from Stall&Release. While in Stall&Release, the VPT counts the number of squashes caused by the line. Such squashes are nearly always due to premature early releases. Specifically, after the consumer has been released, other predecessor threads also write the line. If the number of such squashes reaches a certain threshold, the VPT transitions from Stall&Release to Stall&Wait for the line (Figure 2), effectively learning that the line has multiple writers.

We design the Stall&Wait mechanism like the Stall&Release one. The only difference is that the VPT keeps the consumer read buffered until the consumer thread becomes non-speculative.

3.4 Advanced Learning Heuristics

3.4.1 Violation Predictors

Our violation prediction heuristics are based on counting the number of violations caused by accesses to a particular memory line. Such a scheme is easy to implement in the memory subsystem of CC-NUMA multiprocessors, as it only requires monitoring in the directory controller the addresses of the accesses that cause violations.

Prediction of violations can instead be based on the addresses of the instructions causing the violations, as in [14]. However, this approach is a bit harder to implement in CC-NUMA multiprocessors as instruction addresses are not usually visible at the main memory system.

We have also experimented with more advanced prediction mechanisms based on history tables [17]. Such mechanisms can exploit correlation between related memory operations and thus predict violations better. However, our experiments showed little performance improvement. Due to the additional complexity of such predictors, we do not pursue them further.

3.4.2 Value Predictors

Our framework can accommodate more complex value predictors than the one investigated in this paper. Indeed, once a particular line has been identified as causing same-word violations, the VPT can keep a history of the actual values produced. This history can be updated every time that a thread commits and writes back to memory a new version of the line. Then, when the line is read, we can make a prediction based on this history.

Our experiments with a suite of floating-point applications (Section 5.1) show that values are either highly unpredictable or do not change because they are accessed by silent stores. For this reason, we do not pursue more complex value predictors. A more comprehensive investigation of value prediction under speculative parallelization for multiprocessors is beyond the scope of this paper.

4 IMPLEMENTATION

In this section, we show an implementation of the VPT and LDE modules. As an example, we implement them on top of the CC-NUMA speculation protocol and the speculation module presented in [4]. That module was called *Global Memory Disambiguation Table* (GMDT). In the following, we first describe the GMDT (Section 4.1) and then the VPT and LDE (Sections 4.2 and 4.3). We also outline implementations under other speculative CC-NUMA configurations (Section 4.4).

4.1 Global Memory Disambiguation Table (GMDT)

The GMDT tracks speculative accesses in a CC-NUMA somewhat like the directory tracks regular coherent accesses [4]. The GMDT is coupled with the directory and, like such, it is physically distributed across nodes based on data address ranges. We show the GMDT as module 1 in Figure 1.

More specifically, the GMDT records the subset of lines that receive speculative exposed loads or stores from currently-active speculative threads (and stores from the non-speculative thread). The GMDT also knows about the relative ordering of the threads. While lines read speculatively can be displaced from caches, the GMDT cannot forget any of the exposed reads that took place. Unlike in [4], the GMDT design that we use tracks speculative accesses at the granularity of *lines*, not words.

The GMDT in a node is organized as a set-associative SRAM table where rows are dynamically allocated per memory line upon

a speculative access (or a write by the non-speculative thread) [4]. This SRAM table is allowed to overflow into memory. Each row corresponds to a line (Figure 4). It contains a line address tag, a Valid bit, and a pair of Load and Store bits for the non-speculative thread and for each of the speculative threads that can be active at a time. The Load and Store bits indicate whether individual threads have issued an exposed load or a store, respectively, to the line. When a thread commits, all its cached dirty lines are written back to memory, and its GMDT bits are cleared and reassigned to another thread. If all Load and Store bits for an entire row become zero, the entry is deallocated.

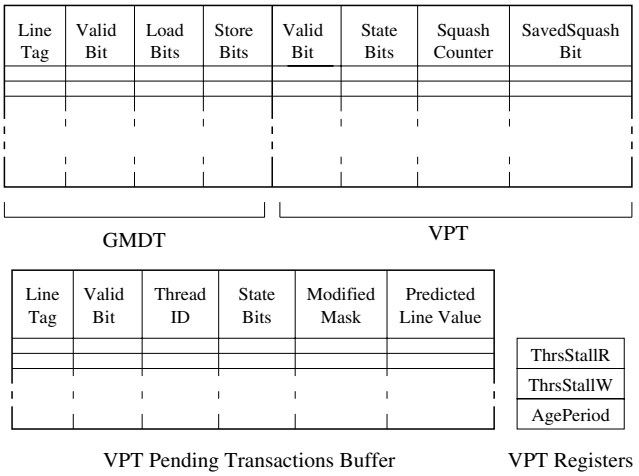

Figure 4: Hardware structures for speculation (GMDT) and for learning (VPT, VPT Pending Transactions Buffer, and VPT Registers).

4.2 Violation Prediction Table (VPT)

The VPT is a table that keeps dynamic information on the memory lines that have recently been involved in potential or actual violations. The main VPT structure is an extension to every row of the GMDT (Figure 4). A VPT entry is allocated when its associated GMDT entry is allocated (Section 4.1). Initially, the VPT entry simply records that the line is in the Plain Speculative state of Figure 2. However, as potential or actual violations on the line occur, the VPT entry changes state as shown in Figure 2. The corresponding GMDT and VPT entries are routinely looked up when the directory controller receives a transaction that involves the line. Depending on the state of the VPT entry, the VPT triggers the actions discussed in Section 3.3. Finally, the associated GMDT and VPT entries are deallocated together, only when neither entry has any useful information. For the VPT entry, this occurs when the information about the involvement of the line in violations has aged out and the line is back in Plain Speculative state.

As shown in Figure 4, each row of the main VPT structure contains a *Valid* bit, 3 *State* bits that encode which state of Figure 2 the VPT entry is in, a *Squash* counter, and a *SavedSquash* bit. The last two will be described later. Overall, if we use 3 bits for the counter, a VPT row takes only 1 byte. We set the number of rows in the per-node GMDT and VPT to be 2048. Such a number was shown to be enough in [4], mostly because the compiler carefully marks the data that is accessed speculatively. As a result, the per-node VPT takes 2 Kbytes.

The VPT has two helper structures, namely the *VPT Registers* and the *VPT Pending Transactions Buffer* (Figure 4). The former contain three settable values called *ThrsStallR*, *ThrsStallW*, and *AgePeriod*. They affect state transitions for VPT entries. We will discuss them later.

The VPT Pending Transactions Buffer allocates one entry for each exposed load in progress to a line that is in the VPT in state other than Plain Speculative. The entry contains temporary state for the transaction and is deallocated as soon as the transaction completes. Each entry effectively corresponds to one potential violation that has not yet been resolved. The buffer is implemented as a small SRAM table. When the buffer is full, accesses that can potentially cause violations cannot allocate entries and, therefore, are allowed to proceed unhindered. Fortunately, our experiments show that, at any time, only a few accesses that can potentially cause violations overlap with each other (Section 6).

In the following, we show how VPT entries change state, how Pending Transactions Buffer entries are allocated and deallocated, and how VPT entries age.

4.2.1 Changing State of VPT Entries

When a VPT entry is initially allocated, its State bits are set for the Plain Speculative state. As potential or actual violations on the line occur, possibly followed by squashes, the State bits in the entry change to follow the state diagram of Figure 2. At each state, the VPT entry triggers the corresponding actions discussed in Section 3.3.

Some of the state changes in Figure 2 depend on the number of squashes seen. For them, the VPT registers keep two squash thresholds, namely ThrsStallR and ThrsStallW (Figure 4). They keep the number of squashes that a line must cause before its VPT entry transitions from ValuePredict to Stall&Release, and from Stall&Release to Stall&Wait, respectively (Figure 2). At any time, the Squash counter in the VPT entry counts the number of squashes caused by the line. Consequently, when a VPT entry transitions to ValuePredict, the Squash counter is cleared. When the line causes a squash, the Squash counter is incremented and compared to ThrsStallR. If the counter exceeds the threshold, the VPT entry transitions to Stall&Release. A similar process occurs for ThrsStallW and the transition from Stall&Release to Stall&Wait.

4.2.2 Allocation & Deallocation of Buffer Entries

Every exposed load to a line that is in the VPT in state other than Plain Speculative triggers the allocation of an entry in the VPT Pending Transactions Buffer. The entry remains allocated until the transaction completes. As shown in Figure 4, an entry contains a line address tag, a *Valid* bit, the *Thread ID* of the consumer thread, 3 *State* bits for the state in Figure 2 that the corresponding VPT entry is in when the entry is allocated in the buffer, the *Modified Mask*, and the *Predicted Line Value*.

Consider first an entry in the buffer in state Delay&Disambiguate. Every time that a predecessor commits and writes back to memory a dirty copy of the line, the Modified mask updates its bit-map according to the dirty words in the line. When the consumer thread is finally about to become non-speculative, the Modified mask contains a bit-map of all the modifications since the load. At this point, the mask is sent to the consumer node's LDE for late disambiguation and the buffer entry is deallocated.

When an entry in state ValuePredict is allocated in the buffer, the value of the line provided to the consumer is copied to the Predicted Line Value field. When the consumer thread is about to become non-speculative, the current value of the line in memory is compared to the value in the Predicted Line Value field. Words with mismatching values are marked in the Modified mask. The mask is then sent to the consumer node's LDE for late disambiguation like in Delay&Disambiguate and the buffer entry is deallocated.

An entry in state Stall&Release is kept in the buffer only until a committing predecessor writes back to memory a dirty copy of the line, and there are no other predecessors of the stalled thread with

dirty versions of the line in their caches. At that point, the entry is deallocated, allowing the original consumer load to proceed with the line read. Finally, an entry in state Stall&Wait is kept until the consumer thread is about to become non-speculative. Only then is the entry deallocated and the original consumer load allowed to proceed.

4.2.3 Aging VPT Entries

Given a VPT entry in state other than Plain Speculative, we want it to age back to Plain Speculative when the corresponding memory line is no longer involved in potential violations. The desired transitions are shown as dashed lines in Figure 2.

To support these transitions, we use the SavedSquash bit of each VPT entry (Figure 4). This bit is set every time that any of the mechanisms in our framework (Section 3.3) saves a violation on the corresponding line and, therefore, a squash. Consequently, at regular intervals, the directory controller scans all the local VPT entries. For a given entry, if the SavedSquash bit is clear, the state is set back to Plain Speculative. Otherwise, it means that at least one squash has been saved. In this case, the state is kept as it is and the SavedSquash bit is cleared. Note that, before doing this scanning pass, the directory controller checks the entries in the VPT Pending Transactions Buffer. Any memory lines that have entries there cannot have their VPT entries aged back to Plain Speculative state. The reason is that these buffer transactions are still pending under one mechanism.

The time interval between these scanning passes on the VPT is given by the value stored in the AgePeriod register. Such a value is given in terms of number of thread commits observed. The size of the interval determines how fast entries age.

We decide whether or not to set the SavedSquash bit every time that we deallocate an entry from the VPT Pending Transactions Buffer. At that point, an exposed load transaction is fully completed, and we can know whether our support indeed eliminated a squash relative to a plain speculative system. To know whether or not it did, we reuse the Modified mask of the buffer entry. Recall that, for entries in Delay&Disambiguate state, the mask records all the words in the line that are being updated by predecessors since the exposed load. To support aging, we simply use the Modified mask in the same way for all buffer entries, irrespective of their state. Then, right before the entry is about to be deallocated, we check the mask. If it is not clear, at least one predecessor wrote the line and, therefore, a squash would have been generated in a plain speculative system. Consequently, we set the SavedSquash bit. Otherwise, the bit is left unmodified. Note that, for entries in the ValuePredict state, after this operation is done, the mask is cleared and we proceed to use the mask as indicated in Section 4.2.2.

4.3 Late Disambiguation Engine (LDE)

The LDE is associated with the cache hierarchy of a node, although it is located outside the processor chip. It performs late disambiguation for exposed loads issued by the local node to lines that are in state Delay&Disambiguate or ValuePredict in their home VPT.

The LDE receives the Modified mask of any line for which it has to perform late disambiguation. The mask indicates what words in the line have potentially changed (in Delay&Disambiguate state) or indeed changed (in ValuePredict state) since the line was originally provided to the consumer thread. The LDE needs to compare the mask against the per-word exposed Load and Store bits that record the accesses of the consumer thread to the line. Such access bits are represented as module 2 in Figure 1 and are kept somewhere in the local cache hierarchy.

The LDE operation has two steps. First, it performs a bit-wise AND between the mask and the Load bits. If the result is not zero, the consumer has consumed incorrect data and has to be squashed. Second, the LDE performs a bit-wise AND between the mask and the negated Store bits. If a resulting bit is set, the corresponding word was changed by predecessors and not overwritten by the consumer. Consequently, the word is stale and the LDE has to invalidate it from the local cache hierarchy.

While the LDE could perform the two AND operations as soon as it receives the mask, we choose a simpler implementation to minimize races with the local processor. Specifically, the LDE waits until the consumer thread finishes execution before performing the operations. After that, the thread can safely commit. Note that, in general, the access bits of a consumer thread need to be available after the thread has finished and until late disambiguation can be performed. In addition, the bits need to be accessible from outside the processor chip. To accomplish this, these bits may be temporarily buffered in the LDE.

4.4 Implementation Variations

The implementation that we have presented for our framework implicitly assumes the speculation protocol proposed in [4]. In this section, we briefly outline some changes necessary to accommodate the framework to other protocols.

Some scalable speculation protocols keep the speculation information only in the cache hierarchy of the processing nodes [23]. A thread communicates a commit or a squash operation only to its immediate successor. There is no speculation module attached to the directory controller such as the GMDT that knows about all such operations.

In such systems, the VPT will be coupled with the speculation engines in the cache hierarchies of the processing nodes. This VPT must be made to work with only the partial information available locally. For example, nodes that are not involved in the squash may not learn that the line is causing violations. Similarly, aging may be based on fairly limited information about the line's behavior. Consequently, our learning heuristics may have to change.

Other scalable protocols do not eagerly merge the state of committing threads with main memory. For example, dirty lines are not written back to main memory as the thread commits [18]. Instead, they are lazily merged with main memory on demand, often on a cache displacement. The implementation that we presented here relied on eager write-backs at commit time to quickly resolve pending exposed loads.

In such lazy systems, the VPT can be extended to proactively request write-backs from committing threads. These write-backs are for lines that have pending transactions in the VPT Pending Transactions Buffer and may be dirty in the cache of the committing thread. In this way, the few lines that are actively experiencing potential violations are written back eagerly at commit time, while all the other lines are written back lazily.

5 EVALUATION METHODOLOGY

5.1 Applications

To evaluate our framework, we choose one Perfect Club application (*TRACK*), two SPECfp2000 applications (*EQUAKE* and *WUPWISE*), and two HPF-2 applications (*EULER* and *DSMC3D*). The input sets used are the standard ones except for *EQUAKE* and *WUPWISE*, which use the *train* inputs. All applications spend a large fraction of their time on loops that cannot be fully analyzed by state-of-the-art compilers. The reason for the non-analyzability is that the dependence structure is either too complex or dependent on input data. Specifically, the codes often have array accesses with subscripted subscripts, procedure calls inside the loops, and complex control flow. Consequently, we use speculative parallelization for these loops. We use the Polaris parallelizing compiler [3] to identify and instrument such loops, mark the speculative variables, and privatize variables whenever safe and convenient. All the loops

Application	Loop to Parallelize	% of Seq. Time	Avg. Iterations per Invocation	RAW Dependences
TRACK	nlfilt_300	58	502	Same−word and False
DSMC3D	move3_100	41	758972	Same−word and False
EULER	dflux_[100,200] psmoo_20 eflux_[100,200, 300]	90	2494	False
EQUAKE	smvp_1195	45	7294	Same−word
WUPWISE	muldeo_200' muldoe_200'	67	8000	Same−word and False

Table 1: Characteristics of the applications studied.

Processor Param.	Value
Issue width	4
Instruction window size	64
No. functional units (Int,FP,Ld/St)	3,2,2
No. renaming registers (Int,FP)	32,32
No. pending memory ops. (Ld,St)	8,16

Memory Param.	Value
L1,L2,VC size	32KB,1MB, 64KB
L1,L2,VC assoc.	2−way,4−way, 8−way
L1,L2,VC line size	64B,64B,64B
L1,L2,VC latency	1,12,12 cycles
L1,L2,VC banks	2,3,2
Local memory latency	75 cycles
2−hop memory latency	290 cycles
3−hop memory latency	360 cycles
GMDT size	2K entries
GMDT assoc.	8−way
GMDT/VPT lookup	20 cycles
Pend. Trans. Buffer size	128 entries
Pend. Trans. Buffer scan	3 cycles/entry

Table 2: Parameters of the 16-processor CC-NUMA architecture modeled.

considered exhibit cross-iteration dependences, either to the same word or to different words of the same memory line (false dependences).

For each application, Table 1 shows the loops that we attempt to parallelize speculatively, the fraction of the sequential execution time taken by these loops on a Sun server excluding initialization and I/O, the average number of iterations executed per loop invocation, and the type of cross-iteration RAW dependences that exist.

These loops are dynamically scheduled into processors. The loops in *TRACK*, *DSMC3D*, *EULER*, and *EQUAKE* are unrolled three times to exploit data locality. In the case of *WUPWISE*, we obtain loops muldeo_200' and muldoe_200' by merging the three outer loops in loop nests muldeo_200 and muldoe_200, respectively. For that, it is necessary to hoist some induction variables and compute the loop indices appropriately, which is within the capabilities of compilers[1].

In Section 6, we present results and speedups for the loops in Table 1 only. We do not estimate overall application speedups because they are dependent on the efficiency of the parallel execution of the rest of the code.

5.2 Architecture Simulated

The evaluation is based on execution-driven simulations. Our simulation environment uses an extension to MINT [27] that includes a superscalar processor model [9], and supports dynamic spawn, squash, restart, and retire of light-weight threads. The processor model is a 4-issue dynamic superscalar with register renaming, branch prediction, and non-blocking memory operations. Some of its parameters are shown in the left portion of Table 2.

The memory system models the speculative CC-NUMA of Figure 1. However, as in [4], each node is a speculative chip multiprocessor (CMP). Each CMP includes 4 processors with their private L1 caches and an on-chip speculation engine that keeps speculation state at the granularity of words. This on-chip speculation engine only triggers squashes on same-word out-of-order RAW dependences inside the chip [4]. These squashes are beyond the control of the VPT and, therefore, not amenable to the mechanisms of our framework. Nevertheless, they are visible to the VPT and do increment the Squash counter like the squashes due to dependences across chips.

Each node in the machine has a CMP, an L2 cache, a victim cache (VC) for dirty lines evicted from L2, one local GMDT, VPT, and LDE module, a portion of the global memory and directory, and a network controller. The local cache hierarchy holds the per-word Load and Store access bits. However, it discards the Load access bits immediately when a thread finishes, in order to reallocate these bits to a newly-scheduled thread [4]. Thus, the local LDE is augmented to capture this information when a thread finishes, for possible disambiguation when the thread finally becomes non-speculative. We dynamically assign iterations to CMP nodes in chunks of four consecutive iterations [4]. In this way, the chunk appears to the GMDT and VPT as a single bigger thread, no different than if a single processor per node was used and assigned a block of unrolled iterations. The machine is equipped with a DASH-like directory-based cache coherence protocol and the GMDT-based speculation protocol outlined in Section 4.1 and discussed in [4]. For simplicity, when a thread is squashed, all its successors are also squashed.

The right part of Table 2 lists the main parameters used. L1, L2, VC, and memory latencies are round-trip times from the processor, without contention. Contention is modeled everywhere except in the interconnect, where a fixed time is assumed for each hop. We model all protocol transactions and messages in detail, as well as all GMDT, VPT, and LDE overheads. The GMDT and VPT Pending Transactions Buffer parameters in the table correspond to a single node. In all experiments, we use the same static round-robin page allocation policy across the nodes. Our *Baseline* system corresponds to a CC-NUMA with *per-line* speculation state in the GMDT. The machine has 4 CMPs, for a total of 16 processors. Unless otherwise indicated, we set ThrsStallR, ThrsStallW, and AgePeriod to 1, 4, and 4, respectively. Note that an AgePeriod of one corresponds to one commit seen by the VPT, which in our case is the commit of a chunk of 4 iterations.

6 EVALUATION

To evaluate the framework, we first examine the squash behavior of the applications. Then, we examine the mechanisms in the framework individually and in combination.

6.1 Squash Behavior

To assess the potential of the mechanisms in the framework, we characterize the squash behavior of the applications on the *Baseline* architecture. For each application, we count the number of squashes induced by each memory line. Squashes are classified based on their source: false dependences, same-word dependences where the store generates a new value (non-silent), and same-word dependences due to a silent store.

Figure 5 shows the squash counts. For each application, the figure groups the memory lines into those that generate 1, 2-10,

[1]Recently, as part of the SPEC OMP parallelization effort [2], loops similar to muldeo_200 and muldoe_200 have been parallelized with help from hand analysis. Such analysis is still beyond the capabilities of automatic parallelization alone.

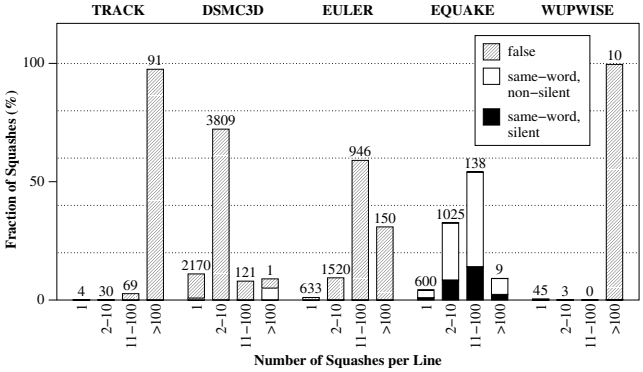

Figure 5: Distribution of the number of times that the same memory line causes a squash in *Baseline*. The numbers on top of the bars are the number of different lines in each bin.

11-100, and more than 100 squashes during the execution of the application. The actual number of lines in each bin is shown on top of each bar. The height of each bar is the fraction of total squashes in the application that fall in that bin. For example, in *TRACK*, 91 memory lines cause more than 100 squashes each, and their combined effect accounts for nearly 100% of the total squashes in the application.

The figure shows that, except in *EQUAKE*, the large majority of squashes are due to false dependences. Consequently, the Delay&Disambiguate mechanism has the potential to save many squashes. We also see that, of the squashes due to same-word dependences in *EQUAKE*, about one quarter are caused by silent stores. Consequently, the ValuePredict mechanism with the simple last-value prediction scheme that we use also has the potential to be beneficial. The figure also shows that, in general, individual memory lines cause many squashes. The best examples are *WUPWISE* and *TRACK*. Consequently, learning what lines cause squashes and then applying our mechanisms looks generally promising, the start-up cost of learning is likely to be amortized. Finally while it looks like the squashes are caused by a large number of lines, our experiments show that the squashes caused by a line are often clustered in time. Therefore, at any time, many fewer lines are actively involved in squashes. As a result, we may not need large VPT Pending Transactions Buffers.

6.2 Plain Speculation

We first compare a system with per-line speculation state in the GMDT and none of our mechanisms, to a system with full per-word speculation state in the GMDT. We call these systems *Baseline* and *Word*, respectively. Per-line schemes can suffer from false sharing. The latter can create false dependences and, therefore, squashes. However, per-word schemes tend to induce more traffic. Indeed, an invalidation or a dependence-checking message for one word does not usually eliminate the need for a similar message for another word in the same line.

The first two bars for each application in Figure 6 compare the execution time of the applications on these systems. For each application, the bars are normalized to *Baseline* and broken down into the following categories: execution of instructions and stall due to memory accesses (*Busy+Mem*); overhead associated with squash operations, including draining pending transactions and waiting for synchronization messages (*Squash*); other speculative execution overheads [4] plus conventional pipeline hazards (*Ovhd+Other*); and stall on exposed loads forced by our Stall&Release and Stall&Wait mechanisms (*Stall*). Note that the total cost of squashes shows up as *Squash* time and as additional *Busy+Mem* and *Ovhd+Other* time due to the reexecution of threads. The numbers on top of the bars show the speedups over the sequential execution.

The figure shows that *Baseline* is usually much slower than *Word*. The large slowdowns of *Baseline* in *TRACK*, *DSMC3D*, *EULER*, and *WUPWISE* are mostly due to the additional squashes caused by false sharing. As shown in Figure 5, the squashes in these four applications come mostly from false dependences. Consequently, they appear in *Baseline* but not in *Word*. The higher traffic of *Word*, while probably slowing down the applications to some extent, has a much lower impact.

EQUAKE has only same-word violations and, therefore, the number of squashes in *Baseline* and *Word* is about the same. These squashes are very frequent and determine the execution time. Both *Baseline* and *Word* take the same time to execute, showing slowdowns with respect to sequential execution.

6.3 Individual Mechanisms

6.3.1 Delayed Disambiguation Only

We now augment *Baseline* with support for our Delay&Disambiguate mechanism only. The resulting system is called *Delay*. Such a system only implements states *A* and *B* in Figure 2. Specifically, once a VPT entry gets to Delay&Disambiguate state, it remains there unless it ages back to Plain Speculative state. For comparison, we also implement an ideal system called *Oracle_Delay*. Such a system is like *Delay* except that a VPT entry in Delay&Disambiguate state will not perform the actions for delayed disambiguation (Section 3.3.1) if the end result is that the thread will later get squashed anyway. Instead, it will trigger the squash immediately, as soon as the second access in the RAW dependence is received.

Figure 6 shows the performance of *Delay* and *Oracle_Delay*. We focus first on the applications with mostly false dependences (*TRACK*, *DSMC3D*, *EULER*, and *WUPWISE*). These applications are sped-up by *Delay* significantly. The reason is that the Delay&Disambiguate support eliminates practically all squashes. This can be seen from the negligible *Squash* time in *Delay*. As a result, *Delay* is much faster than *Baseline* and, typically, even faster than *Word*. It outperforms *Word* because it suffers no more squashes than *Word* and creates less traffic than it.

For these same applications, *Oracle_Delay* is no better than *Delay*. The reason is that most squashes come from false dependences and, therefore, *Oracle_Delay* will typically work as *Delay*.

For *EQUAKE*, *Delay* outperforms both *Baseline* and *Word*. This is unintuitive since all squashes in *EQUAKE* come from same-word dependences. We would expect *Delay* to be slower than *Baseline* because it delays the resolution of transactions that will cause squashes anyway.

In fact, in our architecture, delaying transaction resolution can help in applications with many dependences across threads. To see why, recall that our architecture uses CMP nodes and that dependences between threads running on different processors of the same CMP are not subject to our squash-removing mechanisms. Consequently, consider a thread with two dependences, one with a far-away thread in another CMP and one with a close-by thread in the same CMP. In *Baseline* and *Word*, as soon as the inter-chip dependence is detected, the thread is eagerly squashed. Then, the thread is re-started, only to be later squashed again by the intra-chip violation. In *Delay*, the resolution of the inter-chip dependence is delayed. Eventually, the intra-chip violation will trigger a squash, therefore avoiding the need for the first squash. The result is fewer squashes and less overhead. This is why *Delay* outperforms both *Baseline* and *Word*. Note that *Delay* still has significant *Squash* time.

Oracle_Delay performs no different than *Delay* in *EQUAKE*. The reason is that inter-chip dependences do not end up causing

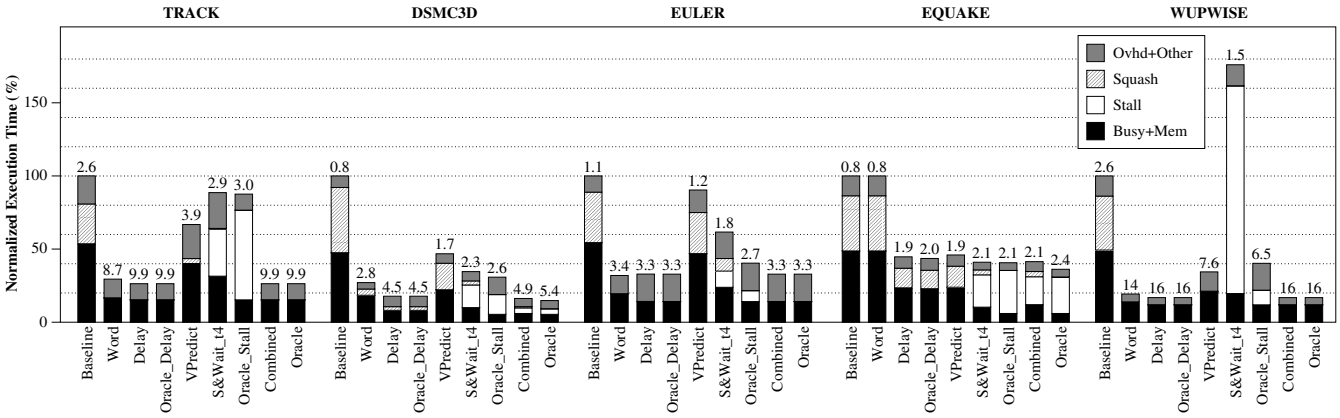

Figure 6: Execution time of the applications on different systems. The numbers on top of the bars are the speedups of the applications over the sequential execution.

Application	*Delay&Disambiguate*		*Stall&Release (ThrsStallR=1)*		
	Selectivity (%)	*Buffer entries Avg. / Max.*	*Coverage (%)*	*Selectivity (%)*	*Buffer entries Avg. / Max.*
TRACK	100.0	0.6 / 9	85.9	100.0	1.5 / 12
DSMC3D	99.8	0.9 / 14	89.3	97.1	1.8 / 12
EULER	100.0	4.1 / 77	74.2	100.0	0.7 / 12
EQUAKE	99.8	0.2 / 15	76.7	100.0	2.1 / 12
WUPWISE	99.1	0.2 / 3	97.8	100.0	3.3 / 15
Average	99.7	1.2 / 23.6	84.8	99.4	1.9 / 12.6

Table 3: Performance and usage statistics for different systems. The numbers for the VPT Pending Transactions Buffer are *per node*.

squashes and, as a result, *Oracle_Delay* follows *Delay*. Moreover, intra-chip squashes are outside the control of *Oracle_Delay*.

Finally, Table 3 shows some statistics that give insight into the behavior of *Delay*. The second column shows the selectivity of its Delay&Disambiguate mechanism. The selectivity is always close to 100%, which means that there are very few late squashes. This is consistent with the fact that *Delay* and *Oracle_Delay* have practically the same performance.

The third column in the table shows that the Delay&Disambiguate mechanism keeps very few entries in the VPT Pending Transactions Buffer at a time. On average, the buffer in a node keeps only 1.2 entries. This means that, at any given time, there are only very few concurrent transactions to lines being monitored for squashes.

6.3.2 Value Prediction Only

We augment *Baseline* with support for the ValuePredict mechanism only. Such a system only has states *A* and *C* in Figure 2. For a VPT entry to transition to the ValuePredict state, the VPT must first observe a squash to the line. As usual, the VPT entry remains in this state until it ages back to Plain Speculative. The resulting system is called *VPredict*.

Note that the simple value predictor that we use would not strictly need to squash to transition to ValuePredict: the value predicted is simply the current version of the line in memory and we can save it in the VPT Pending Transactions Buffer at the time that the potential violation is detected. However, we choose to need one squash to assess the impact of the start-up cost of a more realistic value predictor. Such a predictor would need to see more than one value before predicting.

Figure 6 shows the performance of *VPredict*. In all the applications, *VPredict* performs better than *Baseline*. The reason is that it intrinsically supports delayed disambiguation of false dependences. However, it does not compare favorably to *Word* or *Delay*. The reason is that, in *VPredict*, each VPT entry needs one squash to transition out of the Plain Speculative state. These additional squashes make *VPredict* much slower than *Word* and *Delay* in *TRACK*, *DSMC3D*, *EULER*, and *WUPWISE*.

For the application with only true dependences and some silent stores (*EQUAKE*), *VPredict* performs much better than *Word*, but no faster than *Delay*. *VPredict* would be able to eliminate some squashes due to inter-chip dependences. However, it has little opportunity to succeed because threads are often squashed before the late disambiguation takes place. Its major effect, like *Delay*, comes from simply deferring the squashes due to inter-chip dependences until another squash occurs. Overall, therefore, we do not see any advantages of *VPredict* over *Delay* for our combination of applications and architecture.

6.3.3 Stall Only

We augment *Baseline* with support for the Stall&Release mechanism only, and with support for both the Stall&Release and Stall&Wait mechanisms. We call the former *S&Release* and the latter *S&Wait*. In these schemes, a VPT entry transitions from Plain Speculative to Stall&Release when the line has caused ThrsStallR squashes. In *S&Wait*, the entry further transitions to Stall&Wait when ThrsStallW additional squashes occur. For *S&Release*, we use a ThrsStallR of 1 and 4 (*S&Release_1* and *S&Release_4*). For *S&Wait*, we use a ThrsStallR of 1 and a ThrsStallW of 1 and 4 (*S&Wait_1* and *S&Wait_4*).

Figure 7 shows the execution times of the applications on these systems normalized to *Baseline*. We see that these systems eliminate most of the *Squash* time in *Baseline*. However, they trade it for the *Stall* category. This is because threads eliminate squashes by waiting. In general, these systems are faster than *Baseline* because many of the squashes have disappeared. However, in some cases such as *WUPWISE*, the performance is much worse. In *WUPWISE*, threads are very long, and the relative position of the conflicting loads and stores is such that trading off squashes for stall time hurts performance. Depending on the parameters used, stall-only mechanisms can backfire and lead to slowdowns compared to *Baseline*.

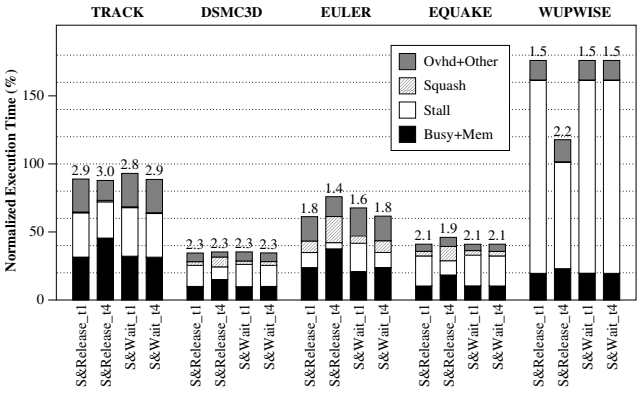

Figure 7: Performance of the Stall&Release and Stall&Wait mechanisms alone. The numbers on top of the bars are the speedups over sequential execution.

In general, these schemes tend to have good coverage and selectivity, and use the VPT Pending Transactions Buffer little. This is confirmed for *S&Release_t1* in the last three columns of Table 3.

Comparing the different schemes for the applications that improve, we see that they tend to have similar performance. However, *S&Wait* schemes tend to be faster than *S&Release*. Among the *S&Release* schemes, a low ThrsStallR (*S&Release_t1*) seems to be better. This suggests reacting fast to squashes as they initially occur. Among the *S&Wait* schemes, a high ThrsStallW (*S&Wait_t4*) seems to be better. This suggests aggressively releasing threads despite occasional squashes. Overall, therefore, *S&Wait_t4* seems to be best.

Finally, we place *S&Wait_t4* as the sixth bar in Figure 6. We can see that, in most applications, this system is not competitive with *Word* or *Delay*. Relative to these systems, *S&Wait_t4* suffers from much *Stall* time. The exception is *EQUAKE*, where *S&Wait_t4* is slightly faster than the other schemes. In applications with squashes due to same-word dependences, *S&Wait_t4* may have an edge over all the other schemes.

For comparison purposes, Figure 6 also includes an ideal stall-only system called *Oracle_Stall*. This is an oracle system that only stalls a thread when the exposed load would cause a squash (due to a false or a same-word dependence) and releases the thread as soon as the correct version is produced. Except in two applications, the performance of *S&Wait_t4* is close to this ideal system.

6.4 Combining all Mechanisms

Finally, we augment *Baseline* with support for our complete framework as shown in Figure 2. The resulting system is called *Combined*. For each application, it is shown as the last but one bar in Figure 6.

The figure shows that *Combined* is always as fast, or faster than, each of the systems with only a single mechanism. Furthermore, it successfully adapts to the behavior of the application. Indeed, consider first the applications with no or very few same-word dependences, namely *TRACK*, *EULER*, and *WUPWISE*. For these cases, VPT entries rarely transition to the ValuePredict and Stall states. The figure shows *Combined* to perform as well as *Delay*.

For applications with only same-word dependences where value prediction often fails, such as *EQUAKE*, VPT entries under *Combined* transition to the Stall&Wait state. As a result, Figure 6 shows that *Combined* performs as well as *S&Wait_t4*. More interestingly, consider applications with a mix of same-word and false dependences such as *DSMC3D*. In this case, VPT entries adapt to the dependence patterns, and *Combined* is shown to outperform both *Delay* and *S&Wait_t4*.

Overall, to speed up a wide range of applications we recommend adding our complete framework to *Baseline*. For applications with mostly false dependences, the system will run as well as a system with only Delay&Disambiguate support. For applications with same-word dependences where value prediction often fails the system will run as well as with only Stall&Wait support. Finally, for applications that have mixed dependence patterns, the framework will adapt and perform better than all the other systems. On average, *Combined* runs the applications 4.3 times as fast as *Baseline*. Moreover, the average execution time of the applications is 23% lower than under *Word*.

Finally, the last bar of each application in Figure 6 (*Oracle*) corresponds to an ideal environment. This system is like *Combined* with full knowledge of all dependences. Therefore, *Oracle* can appropriately decide when to use delayed disambiguation, predict a silent store, stall a thread, and release a stalled thread. From the figure, we see that *Combined* always gets very close to *Oracle*. Consequently, we conclude that the performance of *Combined* is very close to its upper bound.

7 RELATED WORK

There are many proposals for architectures that support speculative thread-level parallelization [1, 4, 6, 8, 10, 12, 18, 22, 23, 25, 26, 32]. We focus on directory-based systems [4, 18, 23, 32].

While we use a framework of mechanisms to eliminate squashes under a per-line protocol, the other directory-based systems have tried other approaches to limit the impact of data dependence violations. Specifically, some systems provide per-word protocol support [4, 32], while another employs compiler-generated synchronization instructions plus some local per-word access information [23].

The system in [18] uses support for *high-level access patterns*. This support performs speculation at the per-line granularity when applications have no dependences. Consequently, under these conditions, it works like our system. However, when applications exhibit false dependences, the support in [18] simply reverts to a per-word protocol for lines exhibiting such dependences. Our system, instead, thanks to the Delay&Disambiguate mechanism, continues to operate with a per-line protocol even for these lines. While we save traffic over [18] for these lines, we do not expect our system to have a noticeable performance advantage unless many lines exhibit false dependences at the same time. However, our main advantage is that we only need to support a per-line protocol.

Dynamic prediction and synchronization of load-store pairs in a uniprocessor has been widely investigated in the past (e.g., [5, 21, 31]). Dynamic prediction and synchronization of cross-thread dependences has been investigated in the context of a tightly-coupled multiscalar processor in [14] and, concurrently with our work, in the context of a chip multiprocessor in [24]. Our solution and those of these two works are different. Indeed, we focus on a distributed directory-based multiprocessor. Moreover, while [14] uses learning mechanisms based on program counter values, we use mechanisms based on memory line addresses.

Value prediction within a single thread of control has been investigated in the past (e.g., [11, 29]). Value prediction in the context of multiple concurrent threads has been investigated in [1, 13, 24, 30]. These works have concentrated on integer applications and, except for [24], rely on little compiler support to eliminate largely statically-predictable values. We investigate value prediction for floating-point applications and use the compiler to eliminate easily-predictable values and to limit prediction to memory locations. We are then left with hard-to-predict floating-point values and our mechanism does not achieve the same level of gains as these other works.

8 CONCLUSIONS

We have proposed a new framework of hardware mechanisms to eliminate most squashes due to data dependences under speculative parallelization. The framework works by learning and predicting cross-thread violations. It is suited for directory-based multiprocessors with protocols that track speculative memory accesses at the system level with the coarse granularity of memory lines.

Simulations of a 16-processor machine showed that the framework is very effective. It can quickly and accurately track the violation behavior of applications and gets very close to an oracle system. We have taken a CC-NUMA that tracks memory accesses at the system level with the granularity of a 64-byte line and added our framework. The resulting system runs a set of applications with dependence violations on average 4.3 times faster. Moreover, the system is even 23% faster than a CC-NUMA that tracks accesses at the system level with the fine granularity of a word – a sophisticated system that is not compatible with mainstream cache coherence protocols.

For numerical applications with mostly false dependences such as ours, we found that the delayed disambiguation mechanism is responsible for most of the performance gains. Moreover, whenever same-word dependences occur, the stall and wait mechanism can complement it and improve performance. Finally, for our applications and architecture, a simple value prediction mechanism does not improve performance much.

REFERENCES

[1] H. Akkary and M. A. Driscoll. "A Dynamic Multithreading Processor." *Intl. Symp. on Microarchitecture*, pages 226-236, December 1998.

[2] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance." *Wksp. on OpenMP Applications and Tools*, pages 1-10, July 2001.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. "Advanced Program Restructuring for High-Performance Computers with Polaris." *IEEE Computer*, Vol. 29, No. 12, pages 78-82, December 1996.

[4] M. Cintra, J. F. Martínez, and J. Torrellas. "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors." *Intl. Symp. on Computer Architecture*, pages 13-24, June 2000.

[5] G. Chrysos and J. Emer. "Memory Dependence Prediction Using Store Sets." *Intl. Symp. on Computer Architecture*, pages 142-153, June 1998.

[6] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. "Speculative Versioning Cache." *Intl. Symp. on High Performance Computer Architecture*, pages 195-205, February 1998.

[7] M. Gupta and R. Nim. "Techniques for Run-Time Parallelization of Loops." *Supercomputing*, November 1998.

[8] L. Hammond, M. Wiley, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 58-69, October 1998.

[9] V. Krishnan and J. Torrellas. "A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors." *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 286-293, October 1998.

[10] V. Krishnan and J. Torrellas. "A Chip-Multiprocessor Architecture with Speculative Multithreading." *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, Vol. 48, No. 9, pages 866-880, September 1999.

[11] M. H. Lipasti and J. P. Shen. "Exceeding the Dataflow Limit via Value Prediction." *Intl. Symp. on Microarchitecture*, pages 226-237, December 1996.

[12] P. Marcuello and A. González. "Clustered Speculative Multithreaded Processors." *Intl. Conf. on Supercomputing*, pages 365-372, June 1999.

[13] P. Marcuello, J. Tubella, and A. González. "Value Prediction for Speculative Multithreaded Architectures." *Intl. Symp. on Microarchitecture*, pages 230-237, December 1999.

[14] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. "Dynamic Speculation and Synchronization of Data Dependences." *Intl. Symp. on Computer Architecture*, pages 181-193, June 1997.

[15] C.-L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. "Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip Multiprocessor." *Intl. Conf. on Supercomputing*, pages 368-380, June 2001.

[16] J. Oplinger, D. Heine, and M. Lam. "In Search of Speculative Thread-level Parallelism." *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 303-313, October 1999.

[17] S.-T. Pan, K. So, and J. T. Rahmeh. "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 76-84, October 1992.

[18] M. Prvulovic, M. J. Garzaran, L. Rauchwerger, and J. Torrellas. "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization." *Intl. Symp. on Computer Architecture*, pages 204-215, June 2001.

[19] L. Rauchwerger and D. Padua. "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization." *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 218-232, June 1995.

[20] P. Rundberg and P. Stenström. "A Software Approach to Thread-Level Data Dependence Speculation for Multiprocessors." *Ninth ISCA Wksp. on Scalable Shared Memory Multiprocessors*, June 2000.

[21] Y. Sazeides and J. E. Smith. "The Predictability of Data Values." *Intl. Symp. on Microarchitecture*, pages 248-258, December 1997.

[22] G. Sohi, S. Breach, and T. Vijaykumar. "Multiscalar Processors." *Intl. Symp. on Computer Architecture*, pages 414-425, June 1995.

[23] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "A Scalable Approach to Thread-Level Speculation." *Intl. Symp. on Computer Architecture*, pages 1-12, June 2000.

[24] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "Improving Value Communication for Thread-Level Speculation." *Intl. Symp. on High-Performance Computer Architecture*, February 2002.

[25] M. Tremblay. "MAJC: Microprocessor Architecture for Java Computing." Presentation at *Hot Chips*, August 1999.

[26] J.-Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.-C. Yew. "The Superthreaded Processor Architecture." *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, Vol. 48, No. 9, pages 881-902, September 1999.

[27] J. Veenstra and R. Fowler. "A Front End for Efficient Simulation of Shared-Memory Multiprocessors." *Intl. Wksp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201-207, January 1994.

[28] T. N. Vijaykumar and G. Sohi. "Task Selection for a Multiscalar Processor." *Intl. Symp. on Microarchitecture*, pages 81-92, December 1998.

[29] K. Wang and M. Franklin. "Highly Accurate Data Value Prediction Using Hybrid Predictors." *Intl. Symp. on Microarchitecture*, December 1997.

[30] F. Warg and P. Stenström. "Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms." *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 221-230, September 2001.

[31] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. "Speculation Techniques for Improving Load Related Instruction Scheduling." *Intl. Symp. on Computer Architecture*, pages 42-53, May 1999.

[32] Y. Zhang, L. Rauchwerger, and J. Torrellas. "Hardware for Speculative Run-time Parallelization in Distributed Shared-Memory Multiprocessors." *Intl. Symp. on High-Performance Computer Architecture*, pages 161-173, February 1998.