Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization*

Milos Prvulovic, **María Jesús Garzarán**, **Lawrence Rauchwerger**[†], and **Josep Torrellas**
University of Illinois at Urbana-Champaign
[†]Texas A&M University
http://iacoma.cs.uiuc.edu

Abstract

Speculative thread-level parallelization is a promising way to speed up codes that compilers fail to parallelize. While several speculative parallelization schemes have been proposed for different machine sizes and types of codes, the results so far show that it is hard to deliver scalable speedups. Often, the problem is not true dependence violations, but sub-optimal architectural design. Consequently, we attempt to identify and eliminate major architectural bottlenecks that limit the scalability of speculative parallelization. The solutions that we propose are: low-complexity commit in constant time to eliminate the task commit bottleneck, a memory-based overflow area to eliminate stall due to speculative buffer overflow, and exploiting high-level access patterns to minimize speculation-induced traffic. To show that the resulting system is truly scalable, we perform simulations with up to 128 processors. With our optimizations, the speedups for 128 and 64 processors reach 63 and 48, respectively. The average speedup for 64 processors is 32, nearly four times higher than without our optimizations.

1 Introduction

While shared-memory multiprocessors have become widespread and microprocessors are being designed with multiprocessing support, many applications are still being developed with sequential machines in mind. Moreover, explicitly-parallel applications, like many web and database systems, are typically developed at a very high cost. This state of affairs is mainly due to the higher difficulty of programming, debugging, and testing parallel programs.

To address this problem, automatic compiler parallelization has been tried. Unfortunately, this approach is usually ineffective for codes with unknown or complicated dependence patterns. Examples of such codes are those with pointer-based accesses, indirect accesses to arrays, irregular control flow, accesses to structures across complicated procedure calling patterns, and accesses whose pattern depends on input data.

One way to extract parallelism from these codes is to use speculative thread-level parallelization [1, 5, 7, 8, 9, 11, 13, 15, 17, 19, 20, 21, 22, 23, 24, 26, 27]. In this technique, the computation in the program is divided into tasks and assigned to different threads. The threads execute in parallel, optimistically assuming that sequential semantics will not be violated. As the threads run, their control flow and the data that they access are tracked. If a dependence violation is detected, the offending threads are stopped and a repair action is initiated. Such a repair action involves re-executing offending tasks, possibly after recovering some old, safe state.

Speculative parallelization can be done purely in software [8, 19, 20]. In this case, the run-time checking for dependence violations is performed by code inserted by the compiler.

To reduce the software overhead of this technique, hardware support can be added to detect dependence violations, help repair the state, or speed up other operations. For example, data dependence violations may be detected by enhancing the cache coherence protocol support, which already tracks accesses from different processors to ensure data coherence. State repair may use hardware support to speed up the detection of tasks that need to be re-executed and the destruction of the incorrect state in their caches.

In recent years, many schemes with hardware support for speculative parallelization have been proposed [1, 5, 7, 9, 11, 13, 15, 21, 22, 23, 24, 26, 27]. Among other issues, they differ in their target machine size and type of code, as well as in their relative emphasis on hardware and software support.

Some of these schemes have focused on architecting a solution for scalable machines [5, 22, 26, 27]. The evaluation of such solutions for up to 16 processors has shown that it is hard to deliver scalable speedups. This is the case even for applications with large task sizes and few true cross-task dependences, which suggests that the reason may be sub-optimal architectural design. Since we believe that scalable machines will eventually incorporate some form of support for speculative parallelization, uncovering and removing the bottlenecks to the scalability of this technique is very important.

In this paper, we attempt to identify generic architectural bottlenecks to the scalability of speculative parallelization and provide general solutions to eliminate them. The solutions that we propose are: low-complexity commit in constant time to eliminate the task commit bottleneck, a memory-based set-associative overflow area to eliminate stall due to speculative buffer overflow, and exploiting high-level access patterns to minimize speculation-induced traffic. With these three supports, we find that speculative parallelization is truly scalable. To show it, we use simulations with up to 128 processors. With the optimizations, the speedups for 128 and 64 processors reach 63 and 48, respectively. The average speedup for 64 processors is 32, nearly four times higher than without the optimizations.

This paper is organized as follows: Section 2 overviews speculative parallelization and lists architectural bottlenecks to scalability, Section 3 proposes solutions to eliminate them, Section 4 discusses our evaluation setup, Section 5 evaluates the solutions, and Section 6 discusses related work.

2 Background

2.1 Speculative Thread-Level Parallelization

Speculative thread-level parallelization consists of extracting tasks of work from sequential code and executing them on parallel threads, hoping not to violate sequential semantics. The control flow of the sequential code imposes a control dependence relation between the tasks. This relation establishes an order of the tasks,

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, and EIA-0072102; by the CICYT under grant TIC98-0511-C02; and by gifts from IBM and Intel.

0-7695-1162-7/01/$10.00 (C) 2001 IEEE

and we can use the terms predecessor and successor to express this order. The sequential code also yields a data dependence relation on the memory accesses issued by the different tasks that parallel execution cannot violate.

A task is *speculative* when it may perform or may have performed operations that violate data or control dependences with its predecessor tasks. Otherwise, the task is non-speculative.

When a non-speculative task finishes execution, it is ready to *commit*. The role of commit is to inform the rest of the system that the data generated by the task are now part of the safe, non-speculative program state. Among other operations, committing always involves passing the non-speculative status to a successor task. This is because we need to maintain correct sequential semantics in the parallel execution, which requires that tasks commit in order from predecessor to successor. If a task reaches its end and is still speculative, it cannot commit until it acquires non-speculative status.

Memory accesses issued by a speculative task must be handled carefully. Stores generate *speculative versions* of data that cannot be merged with the non-speculative state of the program. The reason is that they may be incorrect. Consequently, these versions are stored in a *speculative buffer* local to the processor running the task. Only when the task becomes non-speculative can its versions be allowed to merge with the non-speculative program state.

Loads issued by a speculative task try to find the requested datum in the local speculative buffer. If they miss, they fetch the closest predecessor version from the speculative buffers of other tasks. If no such version exists, they fetch the datum from memory.

As tasks execute in parallel, the system must identify any violations of cross-task data dependences. Typically, this is done with special hardware or software support that tracks, for each individual task, the data written and the data read without first writing it. A data dependence violation is flagged when a task modifies a version of a datum that may have been loaded earlier by a successor task. At this point, the consumer task is *squashed* and all the data versions that it has produced are discarded. Then, the task is re-executed.

With less sophisticated schemes, it is possible that other types of data-dependent accesses also induce task squashes. For example, consider a system that only tracks accesses on a per-line basis. It cannot disambiguate accesses to different words in the same memory line. In this case, false sharing can also trigger squashes. An example is a write preceded by a read by a successor task to a different word in the same line. Furthermore, if there is no support to keep multiple versions of the same datum in the speculative buffers of the system, cross-task WAR and WAW dependence violations also cause squashes [5].

Finally, while speculative parallelization can be applied to various code structures, it is most often applied to loops. In this case, tasks are typically formed by a set of consecutive iterations and are dynamically scheduled on the processors of the system. In general, such an environment is mostly concerned with not violating data dependences. The only control dependence violation to check for is speculative tasks executing past the loop exit. For this reason, this paper focuses exclusively on checking for data dependences.

2.2 Scalability Bottlenecks

We have tried to identify architectural mechanisms in speculative parallelization that induce overheads that are both large and likely to increase with the number of processors. We briefly list them here.

Task Commit. When a task commits, before it passes the non-speculative status to a successor, it typically performs certain operations designed to ensure that the committing data can be later located by the cache coherence protocol. These operations depend on the protocol. For example, they may consist of writing the data generated by the task back to main memory [5, 9] or asking for its ownership [22]. In any case, since tasks must commit in order, these operations are serialized across tasks. As the number of tasks executing in parallel increases, committing them eventually becomes a bottleneck.

Speculative Buffer Overflow. Typically, if the memory state of a speculative task is about to overflow the buffer that holds it, the processor stalls. Otherwise, we could lose record of what data has been accessed by the task and could even pollute memory with incorrect data. Unfortunately, even modest stalls in an environment with many processors can cause serious slowdowns. Indeed, the stall of one task may force its successors to remain speculative for a longer time. This, in turn, causes speculative buffers to accumulate more speculative state and greatly increases the risk of further stalls.

Speculation-Induced Traffic. The false sharing of data between tasks in a speculation environment can lead to squashes, as accesses appear to violate dependences. To eliminate these unnecessary squashes, we need to disambiguate accesses at a finer grain than memory lines. For this reason, most speculation schemes keep at least some access information on a per-word basis [5, 7, 9, 13, 21, 22, 24, 26, 27] instead of only per line. Unfortunately, protocols with full per-word dependence tracking cause more traffic: an invalidation or dependence-checking message for one word does not eliminate the need for a similar message for another word in the same line. In general, as the number of processors increases, the total number of messages in the system per unit time tends to increase faster than the memory system bandwidth, eventually creating a scalability bottleneck. If, in addition, we use a protocol with per-word dependence tracking, we are likely to reach this bottleneck sooner.

Other. Other architecture-related overheads include spawning threads at the beginning of the application, barrier synchronization and related operations performed at each speculative section entry and exit, and dynamically scheduling speculative tasks to threads. For the environment considered in this paper, we find these overheads to be negligible. Specifically, for our applications running on 64 processors, the combination of these overheads accounts for an average of only 0.7% of the total execution time.

Finally, another important overhead is processor stall at the end of each speculative section due to load imbalance. While this overhead is sometimes significant, it is not speculation-specific. Furthermore, it is probably best dealt with in software, through improved task partitioning and scheduling. Therefore, in this paper, we focus only on the three architectural bottlenecks described above.

3 Removing Scalability Bottlenecks

In this section, we propose architectural mechanisms to address the three main bottlenecks identified above.

3.1 Task Commit

While tasks can execute in parallel, they commit necessarily in sequence, since a task can commit only after all its predecessors have done so. This serialization may become a bottleneck when the number of processors is large. For example, consider tasks that take E cycles to execute and C cycles to commit in the background (Figure 1). Individual processors overlap the commit of a task with the execution of the next task. When the number of processors is E/C, commit has become the bottleneck, and the application will not run faster with more processors.

Increasing the task size does not postpone this bottleneck if commit is done in a way that takes a time largely proportional to the amount of data generated by a task. Indeed, bigger tasks (longer E) will usually generate more data, which in turn will take longer to commit (longer C). The result will again be as shown in Figure 1. Instead, for true scalability, task commit needs to complete in constant time, irrespective of the task size.

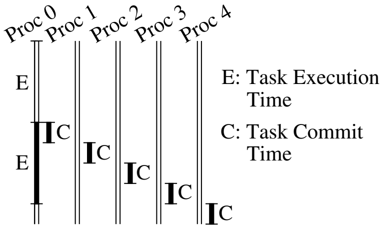


Figure 1: Commit serialization is a scalability bottleneck.

In existing speculation schemes, task commit typically consists of two steps. The first one is designed to ensure that the data generated by the task survive the commit and can be located by the coherence protocol in the future. The second step is to pass the non-speculative status to another task. Typical operations performed in the first step are writing back to memory the cache lines or data elements updated by the task [5, 9] or requesting ownership of the cache lines updated by the task that can be in other caches [22]. Unfortunately, these operations tend to grow longer with task size and, therefore, make commit not scalable.

The one proposed directory-based scheme with constant commit time is [26]. Task commit only involves advancing a shared variable (*last-commit*) that contains the ID of the last committed task. This operation informs the rest of the system of the change in non-speculative status. At all times, versions of data from committed tasks can remain in the caches and are treated as usual: a request from a reader obtains the version created by the closest producer predecessor.

Since write-backs or ownership requests at commit time are not required in [26], caches only need to write dirty data back to memory *lazily*, on displacement. In [26], main memory accepts displaced data that has been generated by committed or uncommitted tasks. This is in contrast to all other schemes, which do not allow uncommitted (and therefore, unsafe) data to be written to main memory.

One advantage of protocols with lazy write-back is that traffic from each individual processor to main memory is spread over a long time. This is unlike other schemes, which transfer data or change their state eagerly at commit [5, 9, 22]. A second advantage of lazy write-back is that the total volume of data written back to memory is potentially lower than in eager protocols. Indeed, assume that a node has two committed versions of the same variable, generated by two tasks. In this case, only the newer version needs to be sent to main memory on displacement. The older one can be silently discarded from the cache, as it is made obsolete by the newer one.

To avoid WAW violations, any speculation scheme must guarantee that the different versions of a given variable are merged with the main memory state in the correct order. In most protocols, this is trivially ensured by the eager write-back or ownership request operations, which are performed on all the variables generated by a task when the task commits. However, in a lazy protocol like [26], other mechanisms are required.

In [26], WAW dependences are satisfied by keeping at the home, for each shared word in memory, the ID of the task that generated the data version currently held in memory. With this support, the order in which data are sent back to main memory is irrelevant. The home updates memory *selectively*, i.e., only when the task ID for the arriving data is higher than the task ID for the data it already has.

Overall, the protocol in [26] commits in constant time but also has two shortcomings. First, allowing uncommitted data to be written to main memory introduces many complications, especially when recovering from dependence violations. Second, the protocol requires keeping, for each shared word in main memory, the ID of the task that produced it. More details are found in Section 6.

3.1.1 Low-Complexity Commit in Constant Time

We want to commit a task in constant time irrespective of its size, without the complexity of having to keep a version number for each location in main memory. To commit in constant time, we follow the approach in [26]: commit only involves advancing the *last-commit* shared variable; the dirty cache lines generated by the committed task are lazily written back to memory on displacement or external request, potentially much later. To eliminate the need for version numbers in memory, we guarantee that, for any given line, all its cached versions in the multiprocessor are always written back to main memory *in the correct order*. Only then can we be sure that no older version of the line overwrites a newer one already in memory.

The support required to eliminate from the commit any eager data transfers or state changes is similar to [26]. The directory keeps a bit vector of sharers for each line and, when necessary, queries them all. It uses the version numbers in the returned lines to choose the correct data version.

The support required to guarantee that the versions of any given line are always written back to main memory in order requires enforcing two new conditions:

First, only lines belonging to *committed* tasks are allowed to be displaced and written back to memory. If we allowed write-backs of lines from uncommitted tasks, we would risk out-of-order updates when a line from an older, yet-to-complete task is subsequently written back. To enforce this condition, each cache hierarchy has a copy of the *last-commit* variable, which is kept largely up-to-date by the local processor. A line displacement is allowed only if the task ID of the line is not higher than the local copy of *last-commit*.

The second condition is that, when a write-back occurs, the directory controller at the home collects from all other caches their own older versions of the line. These are all merged before updating memory. This condition ensures that the home will not later receive any write-back of an older version of the line. Consequently, when a task T suffers the displacement of a dirty line, the message sent to the home includes T's ID and the dirty words in the line. The home then asks the sharer processors to write back all the versions of the same line that were written by tasks older than T. Each reply message includes the writer task ID and the dirty words in the line. If a node has more than one version of the line that needs to be sent to memory, the node combines them so that the reply message includes, for each word, only the last version and its task ID. The home then combines the initial message with all the replies into a single line that contains the newest version of each word. Finally, all the dirty words in this line are written to memory. With this approach, memory is always updated with newer versions.

Overall, with constant-time commit, our scheme (and [26]) can potentially speed up a program over a conventional eager scheme like [5, 9, 22]. The major potential speedup comes from removing any data transfer or state change from the critical path of the commit. A second-order potential speedup comes from the benefits of lazy write-back. Specifically, the memory system contention is moderated by two effects: the write-back traffic from an individual processor is not bursty and, thanks to version combining, potentially smaller.

Implementation

To enforce in-order write-back of committed versions to memory, we use *Version Combining Registers (VCR)* in the directory controller of each node. A VCR contains as many words as a memory line (Figure 2). Each word is associated with a task ID and a validity bit. When a line write-back reaches the home (message 1), an available VCR is allocated. Then, all dirty words in the line and their task ID(s) are copied into this VCR. As the subsequent replies arrive (messages 3), the VCR is updated with the newest versions. After all the replies have been received, the valid words in the VCR are copied to memory and the VCR is released.

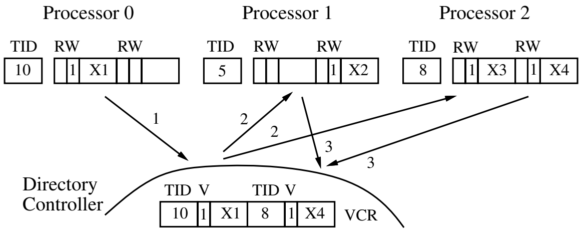


Figure 2: The eviction of a line from processor 0's cache (left) triggers a transaction in three steps (*1-3*) that fills a VCR as shown. *TID*, *R*, *W*, and *V* stand for task ID, read, write, and valid, respectively. A word is dirty if its *W* is 1.

Since a home may have to process write-backs for several lines concurrently, each directory controller contains a few VCRs, for example 8-16. To avoid complications and races, two concurrent write-backs of the same memory line are not allowed. Consequently, if two nodes displace two cache lines with the same address, only the message that arrives first at the home allocates a VCR. The second one is negatively acknowledged. After the first transaction is fully completed, the second one is undertaken if it is still necessary.

VCRs are also useful to supply data on demand in a multi-version coherence protocol. To see how, consider the case when different words of a given memory line have been written by different tasks in different caches, and a successor task reads the line. The line returned to the reader must contain the correct version of each word, namely the version generated by the closest predecessor to the reader. To combine the different word versions from potentially different caches into a single line, the home uses a VCR. Note that, in this case, the combined line may contain some words obtained from uncommitted tasks. Consequently, the contents of the VCR cannot be written to memory, nor can the versions supplied be marked as clean in the caches.

3.2 Speculative Buffer Overflow

The memory state of a speculative task is often stored in hardware buffers like caches, write buffers, or victim caches [5, 7, 9, 13, 22, 24]. This state includes versions of variables generated by the task and, often, a record of what variables the task has read. If this state is about to overflow, the task must stop to prevent the loss of information and the possible pollution of memory. Typically, the processor remains stalled until the task becomes non-speculative. Since tasks commit in order, stopping a task may force its successors to remain speculative for a longer time. In this case, speculative buffers accumulate more speculative state, which can cause further stalls. In systems with many processors, these stalls may be a serious bottleneck.

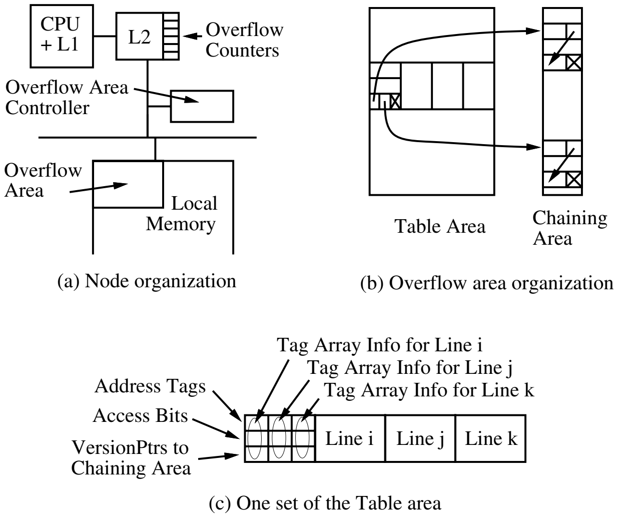
In many applications, two levels of caches and victim caches can easily hold the working set of a task. For other applications, however, this is false. For example, to amortize large communication latencies, tasks in scalable multiprocessors are likely to have coarse grains and, therefore, large working sets. Furthermore, due to commit serialization or load imbalance, individual caches may end up holding the state of several speculative tasks. This effect further increases the volume of data to be buffered.

Ideally, we would like to have an unlimited-sized area for the cache to safely overflow into, so that tasks never have to stop or overwrite memory. Such an overflow area must be able to hold state from several tasks. Further, it is possible that these several tasks have created multiple versions of the same variable. This scenario is likely in applications with privatization-like access patterns: each task that accesses the variable writes it first, therefore creating a new version, but the compiler cannot rule out the existence of true dependences. Consequently, the overflow area must be designed to hold multiple versions of the same variable.

The design proposed in [26] allocates an overflow area in the local memory of a NUMA machine. The processor can access the area through an address translation step at the page level. While such a scheme does the job, it requires significant address translation hardware. Furthermore, it uses the overflow area eagerly, which increases the overhead. More details are found in Section 6.

3.2.1 Overflowing Speculative Data Into Memory

We propose to use an unlimited-sized overflow area in the local memory of a NUMA machine that is both relatively simple and sparely accessed. The overflow area functions as a set-associative victim buffer that grows in the local memory. It stores data from uncommitted tasks that are either displaced from L2 or that are about to be overwritten by a new task. For higher speed, it is managed by a hardware cache controller, which can be either stand-alone (Figure 3-(a)) or part of the memory controller. It is organized and accessed like a cache, with line granularity and no page-level address translation. However, unlike a cache, it stores both tag and data in the same array. In addition, it cannot overflow: if it runs out of space, a software interrupt handler resizes it.



(a) Node organization

(b) Overflow area organization

(c) One set of the Table area

Figure 3: Organization of the overflow area.

The organization of the overflow area is shown in Figure 3-(b). It is composed of two data structures: the *Table* area, which functions as the victim buffer proper, and the *Chaining* area, which will be described later. When a line from an uncommitted task is displaced from L2, the Table takes it in. Then, when an access by the local or a remote processor misses in L2, the Table is accessed. If it has the requested line, it provides it. To minimize unnecessary accesses to the Table, each set of L2 keeps an *Overflow Counter* with the number of lines mapped to this set that are currently in the overflow area (Figure 3-(a)). If this counter is zero, the Table is not accessed and the miss proceeds normally.

We organize the Table as a set-associative cache where, like in S3.mp [16], the tag array information of each set is stored in an additional line. Specifically, Figure 3-(c) shows one set of a 3-way set-associative Table. The first line contains the address tags, access bits, and other information for the three lines that currently reside in the set. To be able to access the Table, the overflow area controller keeps in registers the base address and size of the Table. With these registers and the physical address of the desired line, the controller can identify the correct set in the Table. If we are attempting to insert a new line and the set is full, a software interrupt handler doubles the size of the Table and reorganizes the data in the process.

As indicated before, it is possible that several tasks executed by the processor create multiple versions of the same line. If a version created by an uncommitted task is still in the cache hierarchy of the processor when another task generates a new version, both versions have to be kept locally. Rather than custom-designing L1 or L2 to hold several lines with the same address tag and different task IDs, it is better to move any such complexity to the overflow area.

Our approach is to allow in each of L1 and L2 only one version of the line. These are the most recent local versions, which are the most likely ones to be needed in the future. Any other uncommitted versions of the line are sent to the overflow area right before the processor is about to overwrite them. In the overflow area, all the versions of the line are kept in a linked list, in order from the newest one to the oldest one. They are available in case a dependence violation occurs and a software routine needs to roll back the state to old versions. By keeping them in a linked list, we can get the most recent versions first. Furthermore, if at any time a version in the overflow area is found to belong to a task already committed, that version and all the older versions in the list can be combined and written back to the home memory, and the list can be truncated.

The linked list for a given memory line is organized as follows. The line at the head of the list is kept in the Table. As part of the tag array information for the line, we have a *VersionPtr* pointer (Figure 3-(c)). The latter points to the next version of the line in the Chaining area (Figure 3-(b)). That entry contains tag array information, the line version, and a pointer to the next version of the line, also in the Chaining area (Figure 3-(b)). Lists are truncated and freed up when versions are found to belong to tasks already committed, which keeps them short. To speed up allocation and deallocation of linked-list nodes, all free nodes in the Chaining area are linked in a free list. The overflow area controller keeps the address of the free list head.

In our design, we assume that the basic management of the overflow area is done in hardware by its controller. While it is important to have an overflow area to prevent costly processor stalls, the area is likely to be accessed infrequently compared to L2. Furthermore, most of its accesses are not in the critical path of the processor, as they are largely triggered by cache displacements. Consequently, much of the support for the overflow area could be implemented with software handlers with little performance impact.

3.3 Speculation-Induced Traffic

Speculation protocols tend to generate more traffic than plain cache coherence protocols. The two main reasons are the need to track dependences at a fine grain and, to a lesser extent, the need to identify the correct data version to access. We consider each issue in turn.

Speculation protocols typically track dependences by recording which data were written and which data were read by exposed loads in each task. An exposed load is a load not preceded by a store to the same datum by the same task. This information is often encoded in extra bits added to the cached data. Without loss of generality, we assume that it is encoded with the usual Write bit (*W*) and an additional *Exposed-Read* bit (*R*).

This access information can be kept per line or at a finer grain (per word). In protocols that keep state per line (Figure 4-(a)), tasks can falsely share data. In coherence protocols, false sharing only causes cache misses. In contrast, false sharing in speculation protocols may lead to false dependence violations and, therefore, squashes. For example, in Figure 4-(b) the two tasks access different words, but the store by T_i triggers the squash of T_{i+1}. It has been shown that false violations can be common [5].

To eliminate these unnecessary squashes, most speculation schemes keep some or all access information on a per-word basis [5, 7, 9, 13, 21, 22, 24, 26, 27]. An example is shown in Figure 4-(c). With full per-word information and multi-version support, only true violations (i.e., same-word RAW violations) need to trigger squashes [5].

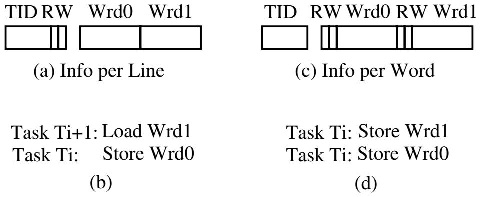


Figure 4: Keeping access information at the grain of a line (a) or a word (c). TID stands for Task ID.

Unfortunately, protocols with full per-word dependence tracking, even while keeping per-line directory state, induce extra traffic: an invalidation or dependence-checking message for one word does not necessarily eliminate the need for a similar message for another word in the same line. For example, in Figure 4-(d) T_i writes to two words. In each case, a message is issued to the directory to check for premature reads.

This additional traffic due to per-word dependence tracking can be very large. As an example, we took the applications and the baseline protocol presented in Section 4 and ran them with 16 processors, using a word- and a line-based protocol. A line has 16 4-byte words. Both protocols have multi-version support, and the line-based protocol is unrealistically enhanced to suffer only the squashes present in the word-based one. We count the messages created in the memory hierarchy below the L2 cache. The word-based protocol creates on average 5.4 times more such messages than the line-based one. As the number of processors is increased, in most systems the number of messages per unit time is likely to increase faster than memory system bandwidth. If, in addition to that, a per-word protocol is used, traffic becomes a scalability bottleneck much sooner.

The second source of additional traffic in multi-version speculation protocols is the need to identify the correct data version to access. Specifically, a node typically generates a message every time a given variable is accessed for the first time by the currently executing task. Even if the variable is found in the node's cache, most protocols cannot immediately determine if that version is the correct one, especially when it is still speculative. Thus, a message is sent to the directory to identify the correct version and, in some protocols, to record the access. Note that this additional traffic appears in both word- and line-based protocols, albeit with different intensity.

Overall, to alleviate the traffic bottleneck, we want a protocol that limits squashes to true violations (like word-based schemes) while eliminating both sources of additional, speculation-induced, traffic.

3.3.1 Exploiting High-Level Access Patterns

Our approach is to select certain very common data access patterns that, strictly speaking, do not need cross-task communication. We then enhance a word-based protocol with support to anticipate these patterns at run time. If one of these patterns is found for a *whole line*, our protocol puts the line in a *No-Traffic* state in the cache. This state allows tasks to access the line locally without inducing any traffic at all while the pattern holds.

The line can remain in caches in this state across task commits and initiations. As a result, processors can execute many tasks without the line causing even a single message since the time the pattern was identified. If, at any time, the accesses do not conform to any of the patterns that we can anticipate, the line reverts to the default word-based scheme seamlessly. In the following, we describe the access patterns that we select and the protocol support required.

High-Level Access Patterns

We select three common access patterns that, strictly speaking,

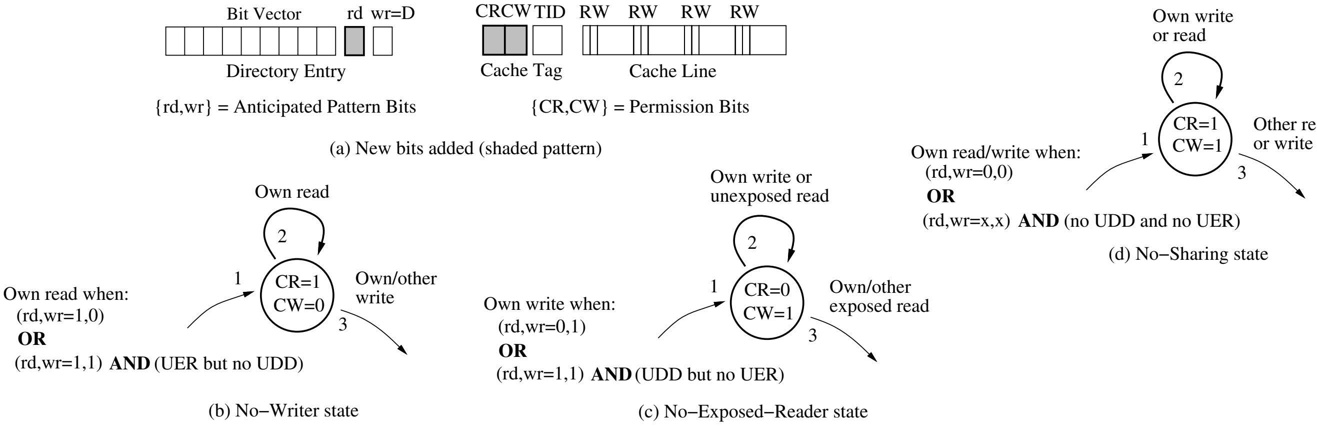


Figure 5: Exploiting high-level access patterns: new bits added to the directory and the tag array in the caches (a) and transition diagrams to lock a line in and out of the three *No-Traffic* states (b)-(d). In the figure, UDD and UER stand for Uncommitted Dirty Data and Uncommitted Exposed Reader, respectively.

do not need cross-task communication: *No-Exposed-Reader*, *No-Writer*, and *No-Sharing*. In *No-Exposed-Reader*, each task writes the data before reading them. This pattern occurs in data that are privatizable. *No-Writer* occurs in data structures that are not modified, while *No-Sharing* occurs when the data are accessed by a single processor. Of course, we are interested in the cases when the compiler cannot identify these patterns statically. Consequently, our hardware dynamically anticipates the situations when tasks start accessing lines with these patterns and when they stop.

Protocol Support

To support our protocol, we add one bit per memory line in the directory and two bits per line in the tag array of every cache (Figure 5-(a)). The new directory bit (*rd*) records whether any of the current sharers has issued an exposed read to any word of the line. We also relabel the conventional D bit in the directory to *wr* (Figure 5-(a)), which indicates whether any of the current sharers (potentially several in a multi-version speculation protocol) has written the line. These *rd* and *wr* bits we call the *Anticipated Pattern* bits. Their meaning is as follows: (*rd,wr*=1,0) denotes No-Writer and (*rd,wr*=0,1) represents No-Exposed-Reader, while a sharer bit vector in the directory entry with a single bit set indicates the No-Sharing pattern.

The two new bits per cache line in the tag array are called the *Permission* bits (Figure 5-(a)): *CanRead* (*CR*) and *CanWrite* (*CW*). They indicate in which of the three *No-Traffic* states (if any) the line is in. They are set by our hardware as it anticipates a certain access pattern for the line.

Specifically, if the No-Exposed-Reader pattern is anticipated, (*CR*,*CW*) are set to (0,1), the No-Exposed-Reader *No-Traffic* state. Tasks in this processor can issue writes and unexposed reads to any word of the line without generating traffic, for as long as the pattern holds across the machine. Likewise, if the No-Writer pattern is anticipated, (*CR*,*CW*) are set to (1,0), the No-Writer *No-Traffic* state. Tasks in this processor can read the cached line without generating traffic. Finally, if the No-Sharing pattern is anticipated, (*CR*,*CW*) get set to (1,1), the No-Sharing *No-Traffic* state. Tasks in this processor can read and write the line without generating any traffic at all.

Figures 5-(b) to (d) show the transition diagrams to lock a line in and out of the three *No-Traffic* states. Since they are symmetric, we only explain the diagram for the No-Exposed-Reader (Figure 5-(c)).

Edge 1 in Figure 5-(c) shows how a processor write (*Own write*) takes a line to the No-Exposed-Reader state. There are two cases. In the simpler case, the write finds that other processors have already anticipated the No-Exposed-Reader pattern and set the Anticipated Pattern bits in the directory to (*rd,wr*=0,1). In this case, the return message from the directory sets the Permission bits to (*CR,CW*=0,1). From now on, the processor can issue writes and unexposed reads to any word in the line without causing any traffic (Edge 2). Note that writes still set the *W* bit of the updated word. The line exits this state when any processor (*Own/other*) breaks the pattern by issuing an exposed read to a word of the line (Edge 3). In this case, the Permission bits cannot filter out the request, which has to go to the home to obtain the most recent version of the word. In the home, the directory sets the *rd* bit and sends messages to all sharers to get their version of the data. As sharers receive the message, they get the line out of the state by clearing the line's *CW* bit. We have seamlessly reverted to the baseline word-based protocol.

Suppose that, later, the task that issued the exposed read commits and the pattern returns to No-Exposed-Reader. Our line will transition to the *No-Traffic* state seamlessly. In this case, the first processor that issues a write will be the first one to anticipate the pattern. It will follow the second case in Edge 1. Indeed, the write finds (*rd,wr*=1,1) in the directory. Following the conventional protocol, the directory sends a message to all the sharers to see which ones need to be squashed. However, in the acknowledgment message, no sharer indicates that it has an uncommitted task with exposed reads (*R* bit set) to any word of the line[1]. Consequently, the directory resets the *rd* bit and replies to the initiating processor that it can take the line to the No-Exposed-Reader state.

With this support, while a line exhibits one of these patterns, it causes no traffic beyond cold misses and re-fetches after displacements. As per the previous discussion, the traffic should decrease dramatically without increasing the number of squashes.

3.4 Summary

As a way of summary, Table 1 shows which types of applications should benefit the most from our optimizations.

4 Evaluation Setup

To evaluate our optimizations, we use simulations driven by several applications. In this section, we describe the simulation environment and the applications.

[1] Of course, at least one sharer must indicate that it has an uncommitted task with dirty data (*W* bit set for at least one word). Otherwise we would be transitioning to the No-Sharing state of Figure 5-(d). This is why Figure 5-(c) has the condition *UDD but no UER*, which means Uncommitted Dirty Data, but no Uncommitted Exposed Reader.

Processor	Memory System	Support for Scalability
4-issue dynamic 1GHz Int,fp,ld/st FU: 4,2,2 Inst. window: 64 Pending ld,st: 8,16 Branch penalty: 4 cycles Int,fp rename regs: 64,64	L1: 32-KB size, 4-way assoc, 64-B line, write back L2: 512-KB size, 4-way assoc, 64-B line, write back Victim cache: full assoc, 32 64-B lines RTrip: 1 (L1), 8 (L2), 57 (local mem), 137 (neighbor mem) cycles Dir controller latency: 21 cycles (pipelined at $1/3$ CPU freq) Network: 2-D torus, virtual cut-through, msg marshaling = 60 cycles, msg transfer (line) = 4 * # hops + 14 cycles	# VCR: 16 per node, timing same as dir controller TID size: 20 bits. Overflow counter size: 4 bits Overflow area controller latency: 4 cycles Overflow area controller to overflow area: 49 cycles round trip Table area: 512-KB size, 3-way assoc # of Chaining area entries: 512

Table 2: Architectural characteristics of the modeled CC-NUMA. *RTrip* stands for contention-free round-trip latency from the processor. All cycle counts are in processor cycles.

Optimization	Types of Applications
Low-Complexity Commit in Constant Time	Tasks that generate data to commit at a high rate (measured as data to commit produced per instruction executed)
Overflow Into a Memory-Based Victim Buffer	Tasks with large working sets or severe cache conflicts. Stalls or load imbalance that cause individual caches to hold the state of several speculative tasks (especially if these tasks generate multiple versions of the same line)
Exploiting High-Level Access Patterns	Accesses with spatial locality to non-analyzable data that are mostly: read-only, per-processor private, or privatizable (no exposed reads)

Table 1: Types of applications that benefit the most from our optimizations.

4.1 Simulation Environment

We use an execution-driven simulation environment based on an extension to MINT [25] that includes a dynamic superscalar processor model [12]. The environment supports dynamic spawn, interrupt, and roll-back of light-weight threads. The architecture modeled is a CC-NUMA multiprocessor with up to 128 nodes. Each node contains a fraction of the shared memory and the directory, as well as a processor with a two-level cache hierarchy. The processor is a 4-issue dynamic superscalar with register renaming, branch prediction, and non-blocking memory operations. Each level of cache can hold only one version of a given line. However, each cache has a victim cache that can contain multiple versions of the same line. These victim caches are accessed with one additional cycle. Table 2 lists the characteristics of the architecture. All cycle counts in the table are in processor cycles. Contention in the entire system is accurately modeled.

Since many accesses to shared data are not compiler-analyzable, shared data pages are allocated round-robin in the memory modules of the participating processors. Private data are allocated locally.

The system uses a directory-based cache coherence protocol along the lines of DASH [14] with the support for speculative thread-level parallelization sketched in Section 2. In the baseline speculation protocol, task commit involves eagerly writing back to memory all the dirty lines generated by the task. Only after the operation is complete can the non-speculative status be passed on to the next task. When a line accessed by a speculative task is about to overflow both caches and victim caches, the processor stalls until the task becomes non-speculative. Finally, while the directory keeps only per-line state, caches keep per-word access information, making the protocol word-based. Other details are described in [18].

On top of this protocol, we optionally enable the optimizations presented in Section 3. For them, we use the parameters shown in the last column of Table 2. The table shows the size of the task ID field (TID), which is a design choice that we have not explored. By default, we use a value much larger than needed by our applications.

As for the support for the overflow area, we assume that the overflow counters in L2 are checked in hardware when L2 is accessed and, therefore, do not add any additional latency. The overflow area controller needs at least two accesses to the overflow area to obtain a line, since the first access reads the tags (Section 3.2.1). Every access takes a full round trip to the local memory. However, since the overflow area is accessed relatively infrequently, the overall application performance is not very sensitive to modest changes in such a round-trip latency.

The Table area is large enough (Table 2) to need no resizes. The Chaining area is large enough to hold all the lines conflicting in the Table area. The overflow area controller follows the chains in the Chaining area in hardware, paying a full round trip to memory for each link in the chain. Given the modest number of accesses to the Chaining area in our applications, if we implemented link chasing with a software handler, we would not significantly increase the overall overhead seen by the applications.

4.2 Applications

We execute a set of scientific applications where much of the code has dependence structures that are not fully analyzable by a parallelizing compiler. The reason for the non-analyzability is that the dependence structure is either too complicated or dependent on input data. Specifically, the codes often have doubly-subscripted accesses to arrays, possible dependences across complex procedure calling patterns, and complex control flow. One example of the latter is a loop with a conditional that depends on array values and that jumps to a code section that modifies the same array. Non-analyzable sections of code are not parallelized by the compiler and, therefore, we do so speculatively.

The applications that we use are: *Apsi* from SPECfp2000 [10], *Bdna* and *Track* from Perfect Club [3], *Euler* from HPF-2 [6], and *Tree* from Univ. of Hawaii [2]. We use the standard input set for the applications except for *Apsi*, where we scaled it down from $112\times112\times112$ to $64\times64\times64$ to reduce simulation time. To determine the sections of these applications that are non-analyzable, clearly parallel, or clearly serial, we use Polaris, a state-of-the-art parallelizing compiler [4].

Columns 2-4 of Table 3 show the breakdown of the sequential execution time (*Tseq*) of these applications, with I/O time excluded. The version that we profile is a fully optimized sequential version running on a single-processor Sun Ultra 5 workstation. The parallel portion includes any loop that Polaris marks as parallel, accounts for at least 1% of *Tseq*, and has at least 128 iterations.

From the breakdown, we see that the non-analyzable portion dominates the execution time of these applications. On average, it accounts for 75.6% of the sequential execution time. The parallel portion accounts for the next biggest chunk of time. On average, it accounts for 12.5% of the sequential execution time. Consequently, if these applications were executed in parallel, we expect the relative weight of the non-analyzable portion to increase.

The next column, labeled *Execution Time Simulated* shows the fraction of the execution time that we simulate in our experiments. Usually, this number is equal to the fraction of non-analyzable code

Appl.	Execution Time Breakdown (% of Tseq)			Execution Time Simulated (% of Tseq)	Names of Non-Analyzable Loops	Avg. Tasks per Invoc.	Written + Spec Read Data per Task (Lines)	Written Data per Task (Lines)	Ownership Req at Commit per Task (Lines)
	Non-Analyzable	Parallel	Serial						
Apsi	93.5	1.3	5.2	53.8	**run_do[20,30,40,50, 60,70,100]** *dtdtz_do40,dudtz_do40, dvdtz_do40,dcdtz_do40, wcont_do40, dkzmh_do[30,60]*	63	1632	1170	330
Bdna	44.2	51.1	4.8	44.2	*actfor_do240*	1499	605	582	368
Euler	89.8	5.2	5.0	89.8	*dflux_do[comb(100), comb(200)]* *psmoo_do[comb(20)]* *eflux_do[comb(100), 200,300]*	1871	287	287	256
Track	58.1	2.5	39.4	58.1	*nlfilt_do300*	502	59	33	24
Tree	92.2	2.5	5.3	92.2	*accel_do10*	4096	14	14	11
Average	75.6	12.5	11.9	67.6		1606	519	417	198

Table 3: Application characteristics. All times refer to sequential execution. Averages are arithmetic ones. In *Apsi*, the loops that we simulate are printed in boldface. In *Euler*, each task consists of 32 consecutive iterations.

in the application, since we only simulate the non-analyzable portion. However, since *Apsi* has such a large problem size, we can only simulate a part of its non-analyzable portion. In the rest of the paper, we focus exclusively on the code in this column. We apply speculative parallelization to and report *speedup numbers for only the code in this column*.

Characteristics of the Non-Analyzable Code

The non-analyzable portion of the applications consists of loops. The remaining columns in the table show the loop names, the average number of tasks per loop invocation, and some information on the working set sizes of the tasks. Unless otherwise indicated, each task is one iteration of the loop. For a given loop, each processor runs a single thread that dynamically picks up tasks to execute.

The working set size information is collected through simulation of a 64-processor system, by tracking how much data are written and speculatively read by each task. For comparison, we also give the number of cache lines that are not only dirty at task commit time, but would also induce ownership requests in a protocol like [22].

In each of these loops, the compiler identifies references to variables that are read-only, private, or profitably privatizable. The rest of the references are marked by the compiler as speculative, since they can cause violations. These references are recognized by our simulator and trigger our speculation protocol.

Dependence Structure of the Non-Analyzable Code

We end this section by describing the non-analyzable portion of each application.

Each loop in *Apsi* has multiple arrays with non-analyzable access patterns. At run time, several of these arrays turn out to have a privatization access pattern: loop iterations access overlapping data, but each iteration generates the values before using them. The rest of the arrays are either read-only or there is no overlap between accesses from different iterations. Consequently, there are no same-word RAW dependences.

Bdna has multiple arrays with non-analyzable access patterns. At run time, all of them have a privatization access pattern and, therefore, there are no same-word RAW dependences.

The loops in *Euler* perform compiler-verifiable reductions on an array. However, the access pattern to the array is very sparse. As a result, transforming the reduction at compile time for parallelization results in loop slowdowns due to the high cost of the accumulation step at the end. At run time, however, many of the accesses happen without same-word RAW dependence violations, so we run the loops under speculative parallelization. Still, the number of same-word RAW dependences in this application is very high, and many of them are violated at run time.

Euler was also analyzed in [5], where no same-word RAW dependences were reported. In [5], only inner loops were simulated. Such loops do not have same-word RAW dependences, but are too small to run with more than 16 processors. In this paper, we coalesce the inner and the outer loops. The resulting loops have more iterations but have dependences. The coverage of the non-analyzable code barely changes.

Track has one array with non-analyzable access patterns. In most cases, iterations only read some of the elements. However, sometimes they write and, in five cases, they cause a same-word RAW dependence. At run time, three of these five dependences are violated. These violations are spread over different loop invocations.

In *Tree*, the one array under test is used as a stack, with each iteration leaving the stack empty. However, the compiler cannot determine that the stack array is privatizable due to the complex control flow. At run time, there are no same-word RAW dependences.

All these applications have some false sharing. Thus, they suffer frequent squashes in line-based protocols. The number of dependences in *Bdna* and *Track* is shown in Table 4 of [5]. That table counts dependences to the same word and to different words of a line assuming *per-word disambiguation support*. In multi-version line-based protocols, all these RAWs and WAWs (both same-word and false) cause squashes if they occur out of order. In fact, some of the false WAWs in Table 4 of [5] will also appear as RAW violations due to the inability to disambiguate at word level.

5 Evaluation

5.1 Application Potential

To assess how amenable our applications are to the optimizations, we measure the characteristics listed in Table 1 for each application. The results are shown in Table 4.

Based on Table 4, we can place each application on a qualitative 3-D chart (Figure 6). Each dimension of the chart shows whether or not an optimization is expected to be beneficial. The origin corresponds to no benefit from any optimization. From the figure, we see that our applications cover a wide range of behaviors.

Optimization	Amenable Applications	Non-Amenable Applications
Low-Complexity Commit in Constant Time	*Apsi*, *Bdna*: Tasks generate data to commit at a high rate	*Track*, *Tree*: Tasks generate data to commit at a lower rate. *Euler*: Rate is unimportant; performance is limited by violations
Overflow Into a Memory-Based Victim Buffer	*Bdna*: Tasks have large working sets and individual caches often hold the state of several spec tasks. *Tree*: Cache conflicts	*Apsi*: Tasks have large working sets but individual caches rarely hold the state of several spec tasks. *Euler*, *Track*: Tasks have small working sets
Exploiting High-Level Access Patterns	*Apsi*, *Bdna*, *Tree*: No-Exposed-Reader and No-Writer. *Euler* (lesser): No-Sharing. *Track*: No-Sharing and mostly No-Writer.	

Table 4: Potential of our applications to benefit from the optimizations.

Configuration	Description
Opt	All 3 proposed optimizations are enabled
OptNoCT	Constant-time commit disabled; overflow area and exploiting access patterns enabled. At commit, all dirty lines in caches and overflow area are written back to memory. Each level of the hierarchy finds the lines using an ideal hardware table accessed in no time that points to the dirty lines
OptNoCTL1	Same as *OptNoCT*, but L1 does not have the table and needs to be traversed to find the dirty lines
OptNoCTL12	Same as *OptNoCT*, but L1 and L2 do not have the table and need to be traversed to find the lines
OptNoOvfL2	Overflow of speculative data from L2 disabled; constant-time commit and exploiting access patterns enabled
OptNoOvfL1	Overflow of speculative data from L1 disabled; constant-time commit and exploiting access patterns enabled
OptNoPat	Exploiting access patterns disabled; constant-time commit and overflow area enabled
OptNoPatA	Same as *OptNoPat*, but with *Aggressive loads* [5] implemented
OptLine	Same as *OptNoPat*, but using a line-based protocol instead of a word-based one
NoOpt	All 3 optimizations disabled. Writes back all dirty lines at commit time without any cache traversals (as *OptNoCT*), cannot overflow from L2 (as *OptNoOvfL2*), and uses *Aggressive loads* (as *OptNoPatA*)

Table 5: System configurations used in the evaluation.

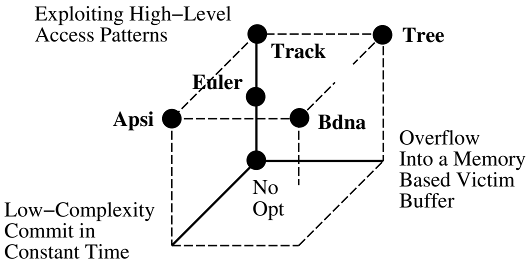

Exploiting High−Level Access Patterns
Track
Tree
Euler
Apsi
Bdna
Overflow Into a Memory Based Victim Buffer
No Opt
Low−Complexity Commit in Constant Time

Figure 6: Which applications are likely to benefit from which optimizations.

5.2 Impact of the Optimizations

To evaluate our three optimizations, we start by comparing a system without any of them (*NoOpt*) to one that supports the three of them (*Opt*). Then, we evaluate each optimization by comparing *Opt* to a system in which the optimization in question is suppressed. Table 5 describes the configurations used, ordered according to which optimization we disable.

To compare systems, we use the speedups delivered by different numbers of processors. The speedups are always relative to the execution of the plain, compiler-optimized sequential version of the code. Such a version includes no extra instructions or data copies due to parallelization. It is simulated as running on a single node of the NUMA machine, with all the data allocated locally. There are, therefore, no remote accesses.

5.2.1 Combining the Optimizations

Figure 7 compares the system with the three optimizations (*Opt*) to the one without any (*NoOpt*). We show the speedups for up to 64 or 128 processors. Note that both axes are logarithmic.

As expected from Figure 6, *Apsi*, *Bdna*, *Track*, and *Tree* benefit from the optimizations significantly. The optimizations only benefit

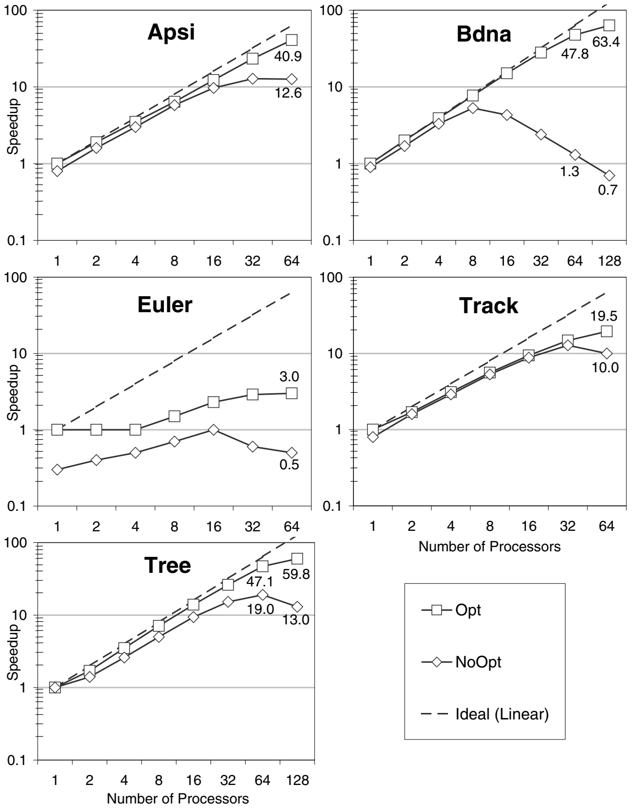


Figure 7: Comparing the system with the three optimizations (*Opt*) to the one without any (*NoOpt*).

Euler moderately because *Euler* has many dependence violations. In most applications, the impact of the scalability bottlenecks of Section 3 does not appear until 16 processors or more. From that point on, the *Opt* and *NoOpt* curves diverge, with *Opt* continuing to scale.

In the following sections, we examine which optimizations are responsible for the much higher speedups in *Opt*.

5.2.2 Low-Complexity Commit in Constant Time

We first consider eliminating the optimization of low-complexity commit in constant time. We consider three schemes from Table 5: *OptNoCT*, *OptNoCTL1*, and *OptNoCTL12*. *OptNoCT* is somewhat unrealistic for our applications. It assumes hardware tables in L1, L2, and overflow area, which know the lines to write back. As we can see from Table 3, some of our applications have tasks that need to write back (or, depending on the protocol, ask for ownership of) hundreds of lines. Therefore, the table would have to be too large.

For our applications, it is more realistic to use *OptNoCTL1*, where L1 is traversed, and L2 and the overflow area have tables. For larger applications, *OptNoCTL12* (where both L1 and L2 are traversed) is more realistic. Figure 8 compares all these scenarios.

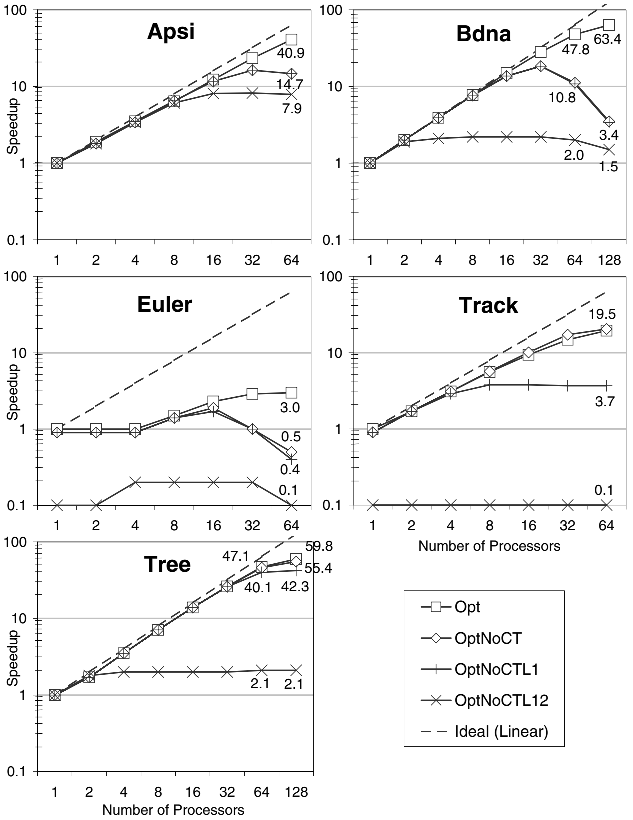


Figure 8: Effect of eliminating the optimization of low-complexity commit in constant time.

Since *OptNoCT* includes no cache traversal overhead, if we compare it to *Opt*, we see the true impact of the serialization induced by a commit that is not done in constant time. For a few processors, there is no difference. However, as the machine scales up, the two curves diverge for *Apsi* and *Bdna*. This was expected from Figure 6. For *Track* and *Tree*, we would need more processors to see a difference. For high-traffic *Euler*, the gains come indirectly from the fact that *Opt* reduces the burstiness of the traffic.

If we also include the cache traversal overheads, we get lower performance. Traversing L1 only (*OptNoCTL1*) does not slow down the system much, except for applications with short-running tasks like *Track*. Traversing both L1 and L2 (*OptNoCTL12*) can take very long due to the large size and low speed of L2.

With some hardware support, the traversal of the caches need not take as long as a full linear scan. In this case, the performance of the system would be between that of *OptNoCT* and *OptNoCTL1* (or *OptNoCTL12*, for larger working sets).

5.2.3 Overflowing Speculative Data Into Memory

We now consider eliminating only the optimization of overflowing speculative data into a memory-based victim buffer. Figure 9 compares *Opt* to a system where no speculative data can be displaced from L2 (*OptNoOvfL2*). For reference, the figure also shows a system where speculative data cannot even be displaced from L1 (*OptNoOvfL1*). Note that the number of processors in the system does not affect task sizes.

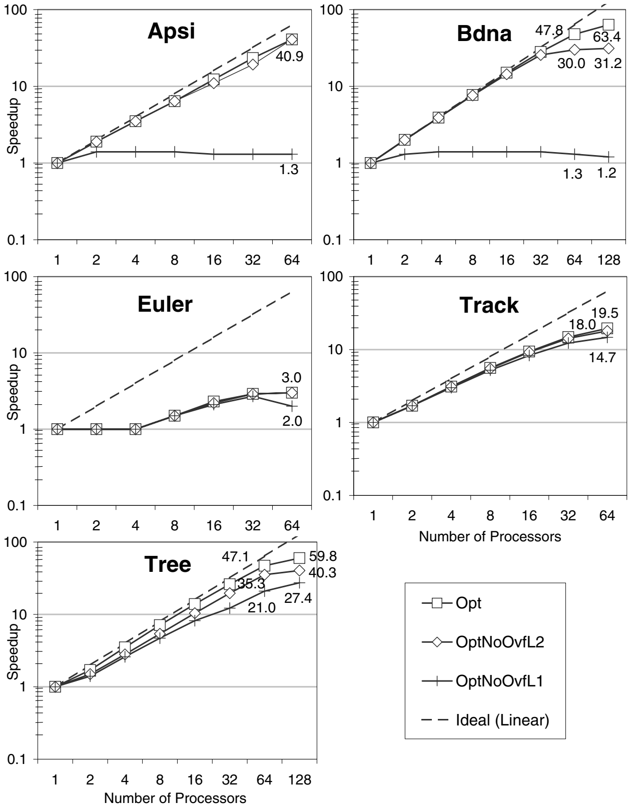


Figure 9: Effect of eliminating the optimization of overflowing speculative data into a memory-based victim buffer.

By comparing *Opt* to *OptNoOvfL2*, we see that *Bdna* benefits the most from the overflow area. For a few processors, the two curves overlap because a task in *Bdna* largely fits in L2. However, with 64-128 processors, various processor stalls force individual processors to hold the state of more than one speculative task in the cache. In these cases, L2 may overflow, which causes the processor to stall in *OptNoOvfL2*.

Apsi's tasks have a larger working set than *Bdna*'s. However, because *Apsi*'s loops have only 63 tasks, caches rarely hold the state of multiple tasks and, therefore, do not overflow. If we used *Apsi*'s standard input set, however, *OptNoOvfL2* would perform much worse because there would be more tasks and each task would have a working set of 400 Kbytes.

In *Tree*, *OptNoOvfL2* is slower due to many L2 conflicts between the state of speculative tasks. The reason is *Tree*'s privatization-like patterns, whereby each task creates its own version of the same variable. The rest of the applications have too small working sets to overflow, as indicated in Figure 6.

Overall, we conclude that our optimization can be useful for some applications. We also see that *OptNoOvfL1*, which simulates what happens in most speculative CMPs, has a low performance.

5.2.4 Exploiting High-Level Access Patterns

Finally, we consider eliminating only the optimization of exploiting high-level access patterns. The resulting system is *OptNoPat*. Figure 10 compares *Opt* and *OptNoPat*. It also shows *OptNoPatA*, which is *OptNoPat* enhanced with *Aggressive loads* [5]. These are per-task first-time accesses to a word that do not perform a version-correctness check in the directory before returning the data to the processor. Instead, the data is returned from the cache immediately and the check is performed in the background. This speeds up execution, but does not reduce traffic. Finally, we show *OptLine*, a line-based protocol.

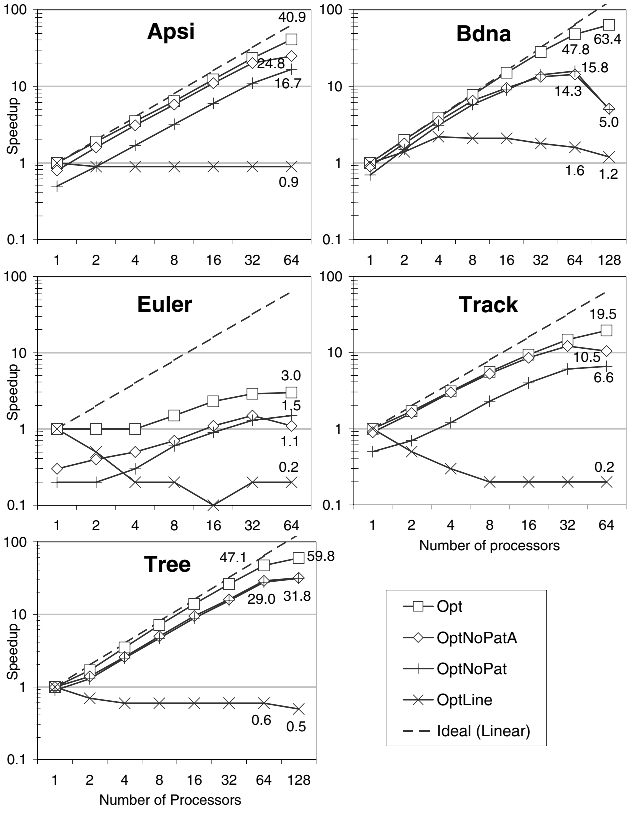


Figure 10: Effect of eliminating the optimization of exploiting high-level access patterns.

By comparing *Opt* and *OptNoPat*, we see that all applications benefit significantly from this optimization, especially *Bdna*. Although Figure 6 predicted that *Euler* is relatively less amenable to this optimization, it still benefits significantly.

Most of the overhead of *OptNoPat* comes from issuing more remote accesses than *Opt*. With *OptNoPatA*, some of these accesses are acknowledged to the processor locally, but still perform a remote check operation. However, the figure shows that, while *OptNoPatA* gets closer to *Opt* in some applications, there is still a significant gap for large numbers of processors. Consequently, our proposed optimization is required for scalability. We also note that *OptLine* performs poorly because it suffers many false dependence violations.

Overall, summarizing the whole evaluation, we conclude that we need all of the three optimizations to achieve scalability. Exploiting access patterns is the most widely-applicable optimization, while committing in constant time and overflowing speculative data into memory are important for applications that generate data to commit fast, and that have tasks with large working sets, respectively. With the three optimizations, 128-processor executions reach speedups of over 63, while 64-processor executions reach speedups of up to 48 and, on average, 32.

6 Related Work

Architectures for Speculative Thread-Level Parallelization. Many architectures have been proposed, including an early system [11], small-scale systems [1, 7, 9, 13, 15, 21, 23, 24], and scalable systems [5, 22, 26, 27]. Our work has been inspired by the bottlenecks found in scalable systems.

Low-Complexity Commit in Constant Time. The only other scheme that supports constant-time commit is [26]. It has two main drawbacks: it keeps a version number (task ID) associated with each shared word in memory, and it complicates the protocol by allowing uncommitted data in memory.

Keeping per-word version numbers in memory eliminates the need for VCRs because data can be safely sent to main memory out of order. However, per-word version numbers take space. Furthermore, memory needs logic to compare the version number of each incoming write-back to its own, and potentially discard the write-back.

Allowing the write-back of uncommitted data to main memory speeds up the eventual merging of data at the home. However, it makes memory unsafe, which complicates the recovery procedure in case of a violation. In addition, when a dirty line is written back to memory, a copy usually needs to be saved locally in the node, in case the version in memory is later overwritten by an incorrect, uncommitted version.

Overflowing Speculative Data Into Memory. The only other scheme that supports unlimited buffering in memory is [26]. It is different from our scheme in two ways: it needs more hardware support because it requires a page-level address translation step, and it uses the overflow area eagerly.

On a L2 miss, an address translation module associated with the directory controller intercepts the request and translates the shared address into a local address. The corresponding page in the overflow area is accessed. Cache displacements also follow this route. Sometimes, this scheme suffers fragmentation.

The eager use of the overflow area is mostly a result of the lazy commit protocol used, which allows uncommitted data in main memory. Since the data in main memory may be unsafe, when a processor is about to overwrite a dirty local version or to displace it to main memory, it often saves a copy of that version locally. This is true even for committed versions. The version goes to the overflow area. In our scheme, we only send data to the area if the version that is about to be overwritten or displaced is uncommitted. Therefore, our overflow area is accessed less frequently.

Exploiting High-Level Access Patterns. Our support combines even lower traffic than line-based protocols with the fine-grain dependence disambiguation of word-based protocols. Under a *No-Writer* or *No-Sharing* pattern, line-based protocols may get close to the low traffic of our scheme. Indeed, while many line-based protocols would still induce traffic every time a task accesses the line for the first time, an optimized protocol like the baseline in [22] keeps the traffic as low as in our scheme. However, under a *No-Exposed-Reader* pattern, all line-based protocols suffer much more traffic than our scheme. The reason is that any out-of-order (same-word or false) WAW present in the pattern forces them to squash threads.

Line-based protocols cannot perform fine-grain dependence disambiguation. Consequently, false sharing may cause squashes. The protocol in [22] has a clever extension where it adds some per-word state (*Fine-Grain SM* bits). The resulting hybrid protocol handles the *No-Exposed-Reader* pattern without squashes and eliminates nearly as much traffic as our scheme. However, since it is not a full word-based protocol, it still suffers some additional squashes. For example, it squashes under false RAWs, as when task *i* reads and writes word *i* and task *j* reads and writes word *j* of the same line.

7 Conclusions

The contribution of this paper is twofold: it proposes several architectural supports to eliminate key scalability bottlenecks to speculative parallelization, and reports speedups for runs on up to 128 processors. The paper shows that, by using our solutions to three bottlenecks, and by combining them into a single system, we deliver architectural scalability for 64-128 processor systems.

The speedup of our applications on 128 and 64 processors reaches up to 63 and 48, respectively. Furthermore, the average speedup for 64 processors is 32, which is nearly four times higher than without our optimizations. Of the three supports, exploiting high-level access patterns is the most widely-applicable one. The support for low-complexity commit in constant time is important for applications that generate data to commit at a high rate. Finally, the support for memory-based overflow can be useful for applications with large working sets, cache-conflicting data structures, or when individual caches end up holding the state of several speculative tasks. Overall, the three supports are necessary since, if any one of them is eliminated, at least one class of applications suffers significantly.

References

[1] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *International Symposium on Microarchitecture*, pages 226–236, December 1998.

[2] J. Barnes. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/. University of Hawaii, 1994.

[3] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[4] 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*, 29(12):78–82, December 1996.

[5] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[6] I. Duff, R. Schreiber, and P. Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice University, November 1994.

[7] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[8] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Proceedings of Supercomputing 1998*, November 1998.

[9] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[10] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[11] T. Knight. An Architecture for Mostly Functional Languages. In *ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.

[12] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[13] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, 48(9):866–880, September 1999.

[14] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[15] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 365–372, June 1999.

[16] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages I1–I10, August 1995.

[17] J. Oplinger, D. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[18] M. Prvulovic. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. Masters Thesis, Computer Science Department, University of Illinois at Urbana-Champaign, November 2000.

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

[20] P. Rundberg and P. Stenstrom. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In *Fourth Workshop on Multithreaded Execution, Architecture and Compilation*, December 2000.

[21] G. Sohi, S. Breach, and S. Vajapeyam. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[22] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[23] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. Hot Chips, August 1999.

[24] 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*, 48(9):881–902, September 1999.

[25] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, January 1994.

[26] Y. Zhang, L. Rauchwerger, and J. Torrellas. A Unified Approach to Speculative Parallelization of Loops in DSM Multiprocessors. Technical Report 1542, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, October 1998.

[27] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 162–174, February 1998.