5-1-2000

Hardware Support for Data Dependence Speculation in Distributed Shared-Memory Multiprocessors Via Cache-block Reconciliation

Renato J. 0. Figueiredo
Purdue University School of ECE

Jose A.. B. Fortes
Purdue University School of ECE

Figueiredo, Renato J. 0. and Fortes, Jose A.. B., "Hardware Support for Data Dependence Speculation in Distributed Shared-Memory Multiprocessors Via Cache-block Reconciliation" (2000). *ECE Technical Reports.* Paper 21.
http://docs.lib.purdue.edu/ecetr/21

HARDWARE SUPPORT FOR DATA DEPENDENCE SPECULATION IN DISTRIBUTED SHARED-MEMORY MULTIPROCESSORS VIA CACHE-BLOCK RECONCILIATION

RENATO J. O. FIGUEIREDO
JOSÉ A. B. FORTES

TR-ECE 00-6
MAY 2000

SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

Hardware Support for Data Dependence Speculation in Distributed Shared-Memory Multiprocessors Via Cache-block Reconciliation

Renato J. O. Figueiredo and José A. B. Fortes
School of Electrical and Computer Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285
{figueire,fortes)@purdue.edu

[0] This work was partially funded by the National Science Foundation under grants CCR-9970728 and EIA-9975275 Renato Figueiredo is supported by a CAPES grant.

Contents

List of Tables

List of Figures

Abstract

Data dependence speculation allows a compiler to relax the constraint of data-independence to issue tasks in parallel, increasing the potential for automatic extraction of parallelism from sequential programs. This paper proposes hardware mechanisms to support a data-dependence speculative distributed shared-memory (DDSM) architecture that enable speculative parallelization of programs with irregular data structures and inherent coarse-grain parallelism. Efficient support for coarse-grain tasks requires large buffers for speculative data; DDSM leverages cache and directory structures to provide large buffers that are managed transparently from applications. The proposed cache and directory extensions provide support for distributed speculative versions of cache blocks, run-time detection of dependence violations, and program-order reconciliation of cache blocks. This paper describes the DDSM architecture and presents a simulation-based evaluation of its performance on five benchmarks chosen from the Spec95 and Olden suites. The proposed system yields simulated speedups of up to **12.5 in** a 16-node configuration for programs with coarse-grain speculative windows (millions of instructions and hundreds of KBytes of speculative data).

1 Introduction

Modern high-performance computers exploit parallelism across instructions of a sequential stream (instruction-level parallelism), as well as parallelism across tasks executing in distributed processing units (thread-level parallelism). The former type of parallelism - ILP - is often achieved transparently from a programmer via compiler and hardware techniques. The latter - TLP - is currently achieved either via explicit parallel programming, or with the aid of parallelizing compilers [5, 4].

Currently, parallelizing compilers must assume that tasks are data-dependent when a compile-time analysis cannot prove data-independence for all possible dynamic (run-time) instances of the tasks. This assumption is conservative, since tasks that cannot be proven to be data-independent statically may indeed be independent at run-time. The availability of system hardware and/or software to support data dependence speculation allows a compiler to relax the constraint that parallel tasks must be provably data-independent, thus increasing the potential for automatic extraction of TLP from sequential codes [17, 21].

The main contribution of this paper is a hardware-based data dependence speculation scheme for distributed shared-memory (DSM) multiprocessors that (1) requires no application-managed buffering for speculative data and (2) uses a distributed, directory-based protocol to detect and recover from dependence violations in speculatively parallelized programs. The proposed mechanism allows for automatic extraction of TLP from sequential programs with irregular data structures and inherent coarse-grain parallelism that cannot be detected statically.

Hardware-based data dependence speculation solutions have been proposed in previous work for tightly-coupled designs [8, 10]. However, these techniques do not apply efficiently to distributed multiprocessors, since they rely on the availability of a low-latency interface to speculative versions of blocks across all processing units (on-chip bus).

Previous work on data dependence speculation for coarse-grain parallelism in DSMs has considered distributed solutions [23], under the assumption of software-controlled buffering: the compiler must be able to explicitly copy all speculative data prior to the beginning of speculative execution. However, this assumption limits the scope of programs that may be automatically parallelized to those that operate on static, regular data structures (such as arrays) that can be identified and copied by a compiler. Programs with dynamic data structures (such as pointel*-based trees and linked lists) are difficult to be parallelized automatically under this model, since addresses are not generally known at compile time.

Concurrently with the research presented in this paper, related hardware solutions for speculative distributed systems have been proposed [2, 20]. These architectures rely on the design of dedicated hardware data structures to hold part of the speculative state; in [2], memory disambiguation tables (LMDT, GMDT) are used to enforce sequential ordering of speculative accesses, while in [20] an ownership buffer (ORB) is used to track speculatively modified blocks. In contrast, the solution proposed in this paper encodes the speculative state entirely in cache blocks and directory entries, leveraging existing hardware mechanisms of buffering data in coherent cache blocks (potentially backed up by main-memory node caches [6]) to provide large speculative buffers for coarse-grain tasks.

This paper proposes novel extensions to existing L2 caches and directory protocols to support a hardware-based Data Dependence Speculative DSM - DDSM. The proposed extensions to L2 caches include extra per-block state and coherence messages. The proposed directory protocol extensions implement a priority-encoding reconciling function [12] to commit speculative versions of cache blocks to main memory, and provide run-time detection of data dependence violations. These extensions allow DDSMs to handle accesses issued by speculatively-parallelized programs, in addition to supporting conventional DSM accesses to shared-memory (issued by non-speculative code).

This paper describes the DDSM architecture and presents a simulation-basecl evaluation of its performance for a set of five speculatively parallelized programs from the Spec95 [19] and Olden [1] benchmark suites. These Fortran and C programs operate on both static and pointer-based data structures (integer and floating-point). Similarly to chip-multiprocessor designs, speculative thread-level parallelism is obtained from loop iterations and subroutine calls [15].

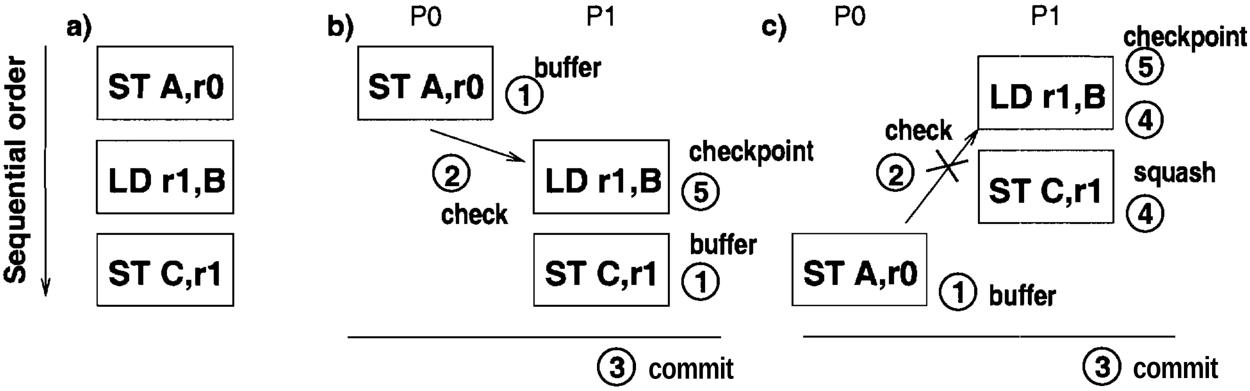


Figure 1: Data dependence speculation example: a) sequential program; **b)** successful speculation; c) data dependence violation.

The performance analysis presented in this paper is based on a modified version of the RSIM simulator [16] that models a DDSM with up to 16 out-of-order ILP processing nodes. The analysis shows that the system delivers speedups of up to 12.5 for the studied programs. It also determines how its performance is affected by the size of the speculative window, number of processors, and mis-speculation frequency for a sparse matrix-based kernel.

This analysis shows that DDSMs with up to 16 processors efficiently support coarse-grain windows with millions of instructions and speculative data sets of hundreds of kilobytes, for applications with low occurrence ($\sim$ 10%) of dynamic data dependence violations. While tightly-coupled designs target fine-grain tasks with hundreds to thousands of instructions and few kilobytes of speculative data, DDSMs must provide larger speculative buffers to allow execution of coarser-grain windows and tolerate the large memory access latencies of distributed multiprocessors.

The rest of this paper is organized as follows. Section 2 introduces a classification of five basic mechanisms to support data dependence speculation. Section 3 describes the DDSM programming and execution models. Section 4 describes the DDSM structures that implement the methods of Section 2 and support the models of Section 3. Section 5 describes the experimental methodology used in the performance analysis; the results of this analysis are summarized in Section 6. Section 7 compares the DDSM approach with previous work, and Section 8 presents conclusions.

2 Dependence speculation methods

This section presents a classification of five mechanisms necessary to support data-dependence speculation in multiprocessors. The goal of this classification is to provide a reference for the description of the DDSM design, and for the comparison with related work.

Consider the series of memory operations of the sequential program shown in Figure la). In this example, the program stores the contents of register $r0$ to memory location A, and later (in sequential order) reads from memory location B to r1. It then copies the contents of r1 to memory location C. Without loss of generality, assume that one wishes to execute this sequence in parallel using two processors: P_0 executes the first store, while P_1 executes the other two instructions.

Addresses A, B, and C may not be known at compile time. Hence, the dependences among these instructions may not be known statically. During execution, however, the dependences among these instructions are determined. If the addresses A, B, and C correspond to different memory locations, the instructions may be executed in parallel. If two or more of the instructions access the same location, they become data-dependent, and need to be synchronized [14] to enforce the original sequential semantics of the program.

Assume that A = B $\neq$ C; hence, there is a read-after-write (RAW) dependence between the

Method	Description
1	Buffer speculative data
2	Check for dependence violations
3	Commit data from buffers to memory
4	Discard squashed data from buffers
5	Checkpoint speculative tasks

Table 1: Summary of methods required to support memory dependence speculation.

Method	Units	Actions
1 buffer	L2 cache	Request and hold speculative blocks from directory.
2 detect	Directory	Serialize access to memory blocks; enforce ordering
3 commit	L2 cache/ Directory	Cache flushes spec. blocks, directory reconciles, commits.
4 squash	L2 cache	Cache gang-invalidates squashed blocks
5 checkpt	CPU, software	System calls mark begin/ end of speculation.

Table 2: DDSM support to enable data dependence speculation. Methods 1 through 5 are described in detail in Sections 4.1 through 4.5

first store and the load. Two possibilities arise with respect to the dynamic ordering of these two instructions.

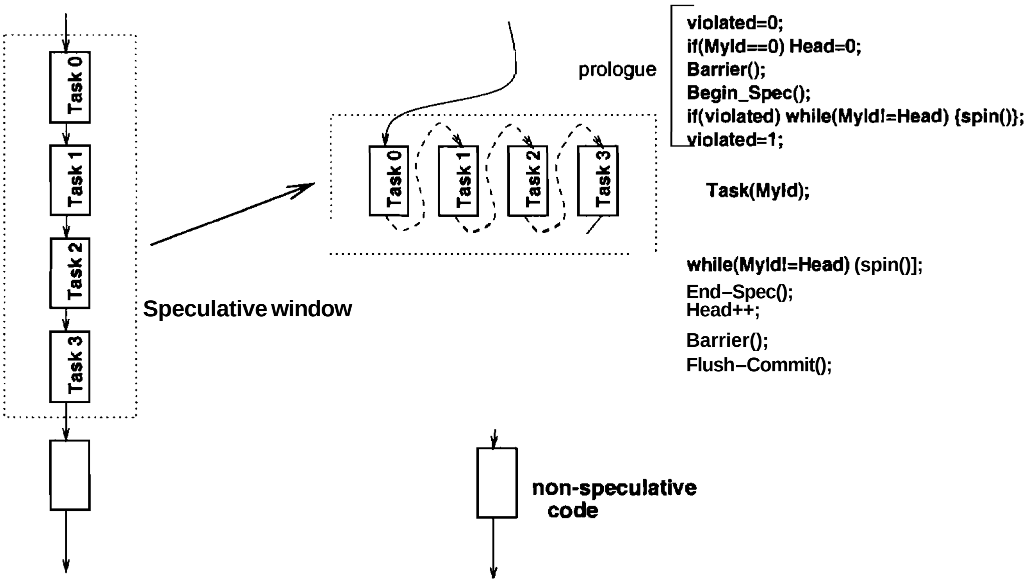
First, if P_0 completes the store before P_1 executes the load (Figure lb)), the sequential order is preserved, as long as there is a mechanism to forward the stored value from P_0 to P_1. However, if P_1 executes the load prior to P_0's store, there is a RAW data dependence violation: P_1 has used a value before its definition (Figure lc)). In order to preserve the sequential semantics, P_1 must re-issue the load, as well as all subsequent data-dependent operations that have already executed.

This example shows the necessity of five basic mechanisms to support data dependence speculation across multiple processors. These are summarized in Table 1.

Method **1** ensures that a value speculatively stored does not overwrite non-speculative data, until it is known that the value is no longer speculative and is safe to be written to memory (method 3). Method **2** is necessary to detect when a data dependence violation occurs, in order to trigger the re-issue of dependent instructions (checkpointed by method 5) and the squashing of all incorrectly generated speculative data (method 4). Table **2** presents an overview of the methods provided by a DDSM to support data dependence speculation, following the convention of Table 1. Section 4 presents a detailed description of the data structures and actions associated with each method, and Section 7 compares DDSMs to related approaches based on the classification of Table 1.

3 Programming and execution models

The programming model assumed for DDSMs in this paper extends the single-program, multiple-data (SPMD) model to support speculative accesses to shared-memory. In SPMD, processors execute the same program on multiple data during parallel execution, and communicate and synchronize via shared-memory accesses. Parallel tasks in SPMD may be identified manually by a programmer, or automatically by a compiler; they must be data-independent to ensure correct execution. The SPMD model is currently supported by extensions to existing programming languages, such as OpenMP [3]



```
violated=0;
if(MyId==0) Head=0;
Barrier();
Begin_Spec();
if(violated) while(MyId!=Head) {spin()};
violated=1;

Task(MyId);

while(MyId!=Head) (spin()];
End-Spec();
Head++;
Barrier();
Flush-Commit();
```

Figure 2: Example of sequential code speculatively executed in parallel by each of the 4 processors.

for Fortran and C, and by parallelizing compilers [4, 5].

Data-dependence speculation allows a compiler or programmer to relax the constraint of data-independence to issue SPMD tasks in parallel. Speculatively parallelized tasks are able to safely access shared-memory in parallel, since mechanisms to detect and recover from data dependence violations are provided.

The execution model supported by many speculative multiprocessor designs proposed to date [8, 10, 2] is based on a hierarchical enforcement of sequential semantics [18]: intra-task sequential ordering of memory accesses is enforced by each execution unit, independently, while inter-task sequential ordering is enforced by an arbiter that tracks accesses to global memory. To facilitate the bookkeeping of data-dependent tasks, proposed implementations of this model conservatively assume that all instructions in a task are data-dependent on the first instruction of the task.

Figure 2 shows an example of a speculatively parallel execution under the hierarchical execution model. Consider a window of instructions consisting of four tasks of a sequential program. Assume that control-flow dependences known at compile-time among tasks 0 through 3 imply the sequential order of execution shown to the left of the figure. Tasks 0 through 3 may be speculatively executed in parallel: if the tasks in the speculative window are data-independent, their parallel execution preserves the original sequential semantics. If they are not data-independent, the dependence violation(~)must be detected, and the violated tasks must be re-issued to preserve sequential ordering.

In the DDSM realization of the hierarchical execution model, speculative tasks are issued in parallel, commit in sequential order, and are synchronized via a barrier at the end of the speculative execution. Speculative tasks that violate dependences during speculative execution are blocked by software and are re-executed in sequential order. Sequential order is implied by the unique identifiers of the processors executing speculative tasks. During speculative execution, the *Head* task is defined as the earliest task (in sequential order) that has yet to commit.

The program segments that are amenable to speculative parallelization under the DDSM model include loops and sequences of subroutine calls [15] with statically known control dependences. These segments are assigned to speculative execution through the addition of prologue and epilogue wrappers to the source code (Figure 2). Section 4.5 describes the DDSM software interface in detail.

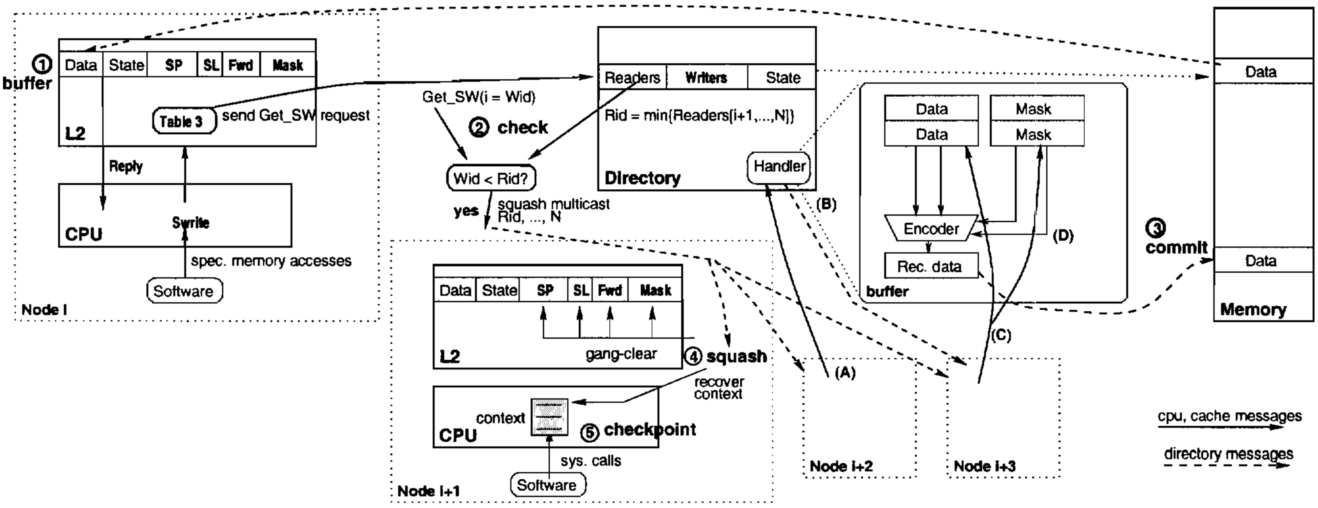


Figure 3: Overview of DDSM speculation: ① extensions to L2 cache allow for buffering of speculative shared-memory data; ②**upon** receipt of a speculative write request (GetSW), the directory checks the list of read sharers for RAW violations. When a violation occurs, the data-dependent tasks are squashed ④ : speculative cache blocks transition to the squashed state, and the processor context - checkpointed at the beginning of speculation ⑤- is restored. At the end of speculation, caches flush committed speculative blocks to the directory, which employs a reconciling function ③ to commit the program-order version of speculative blocks to memory.

State	Description
SP_NO	Not speculative
SP_IN	Speculative, in-flight
SP_CO	Speculative, committed
SP_SQ	Speculative, squashed

Table 3: Speculative states encoded in the SP-bits of a DDSM cache line.

4 DDSM speculation methods

This section describes the support provided by DDSMs to implement the dependence speculation methods summarized in Table 2. Figure 3 presents an overview of the five DDSM speculation mechanisms.

4.1 Speculative buffers

Speculative data in DDSMs is buffered in L2 caches. The motivations are twofold: first, hardware-based buffering simplifies the programming model and allows speculation across dynamic data structures. Second, caches are used in existing DSM implementations; their design can be leveraged to also hold speculative data.

In order to leverage conventional cache organizations to hold speculative data, extra state information must be appended to the state of conventional cache blocks, and the cache controller must be extended to handle speculative accesses. The novel state extensions proposed in this paper encode the speculative state of a memory block in two bits (SP-bits).

A cache block in DDSM may be in one of the following four states, with respect to its speculative status (Table 3). A block is in the S P l N state if it has been accessed speculatively by a task, and the block has not been squashed nor committed. A block is in the **SP_CO** state if its contents are safe to be committed to main memory. A block is in the **SPSQ** state if it has been squashed. A block is in the **SP_NO** state if it is not speculative[1]. Figure 4 shows the state diagram for speculative DDSM cache blocks. Transitions in the diagram are triggered by speculative reads and writes (Sreads, Swrites), commit, squash and flush requests.

In addition to SP-bits, each cache block has an SL bit (flags that the block has been speculatively loaded), and a Fwd bit (flags that the block has been forwarded to one or more later tasks).

Each DDSM cache block also has a write mask. The mask allows for accesses at a finer granularity than a cache block [8, **10**, 2]: it determines which words of the block have been speculatively written. This information is required to reduce the occurrence of speculative false-sharing, and to allow the reconciliation of multiple speculative versions by the directory. The resulting cache block is depicted in Figure 3①.

Although the buffering of speculative state takes place in L2 cache blocks, DDSMs must allow caching of speculative data in L1 caches for high performance. For conventional write-through L1 caches, only the SP and SL bits need to be added to the state of cache blocks, since all writes are seen by the L2 cache. Interfacing with write-back L1 caches requires that write masks and the Fwd bit be also present in L1 blocks. For simplicity, this paper assumes a write-through L1 cache.

The SP-bits allow the cache controller to determine whether speculative accesses must generate messages to the home node of a block. Tables 4 and 5 show how speculative memory accesses to a sub-word i of a cache block B are resolved by the cache controller. The first speculative read (write)

[1] **Conventional cache-coherent block states, such as shared-clean and dirty, apply to non-speculative (SPNO) blocks.**

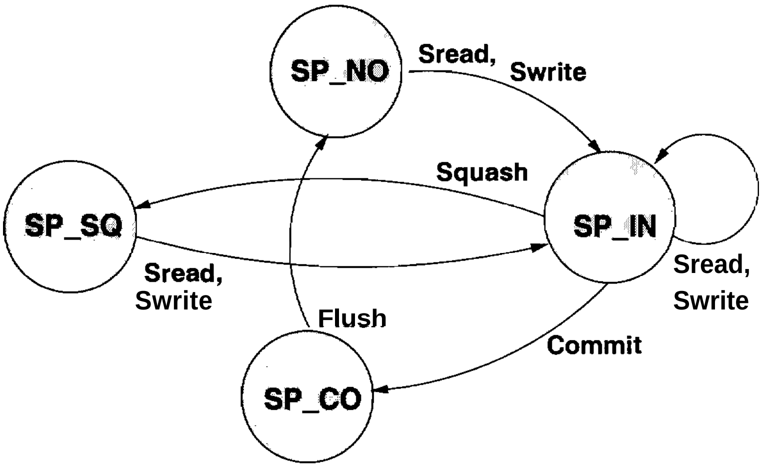


Figure 4: **State machine for speculative L2 cache blocks.**

Message	Description
Get_SR	Request spec. block from directory, which marks requester as reader
Get_SW	Request spec. block from directory, which marks requester as writer
Recflush	Flush spec. block to directory to be reconciled
Rec_squash	Request a partially reconciled copy of block from directory

Table 4: **Speculative messages of DDSM L2 caches.**

access to B triggers the sending of a Get-SR (Get-SW) message to B's home node. A first-read is detected by testing the value of SL; a first-write is detected by testing the result of'an OR operation across the block's write-mask bits.

The first-read/write messages are used by the directory to dynamically build a record of all speculative readers and writers of a block and track dependence violations. If a block is both read and written speculatively, the cache will notify the directory of both first-read and first-write events. Subsequent reads and/or writes are satisfied by the L2 cache without notifying the directory, until the end of speculative execution.

There are two special cases that need to be handled by the controller. First, if a speculative read is to a word previously written speculatively by the same processor, the home node is not notified; according to the hierarchical execution model, this read access cannot violate inter-task dependences. Second, if a speculative write accesses a forwarded block, the system squashes subsequent speculative tasks. Section 4.6 describes this special case in detail.

4.2 Ordering violation detection

Out-of-order memory accesses issued by DDSM may violate data dependences imposed by the sequential semantics of a program. In order to detect violations, the system must keep track of the ordering of accesses to memory. A DDSM maintains ordering information of speculative data at a coherence block granularity at the directory.

The main motivation for using the directory to track memory ordering and detect memory violations stems from the fact that accesses to a memory block are serialized by the directory controller. Zhang et al. [23] first proposed the use of this property of directories to track ordering violations in DSMs. However, their approach detects violations on explicit shadow copies of speculative data structures, while DDSM detects violations on arbitrary shared-memory blocks.

The DDSM directory extends conventional CC-NUMA protocols with extra states and transactions to track memory access ordering. Conventional protocols store the state and one read-sharers bit-vector for each memory block [13]. The DDSM directory protocol uses two bit-vectors to record both speculative readers and speculative writers of a block, and defines one extra block state (Speculative). Figure 3 (Directory) depicts the format of the directory entry for a memory block.

The DDSM protocol operates with blocks that can be either non-speculative or speculative. Blocks become speculative when the directory receives any Get-SR or *Get_SW* request. Blocks

Access	**SP-bits**	**Action**
Sread (B[i])	**SP_NO**	Send **Get_SR** to home(B)
	SP_IN	If SL = 1 or $Mask(B[i]) = 1$, request is satisfied by L2; else, send **Get_SR** to home(B)
	SPSQ	Send **Rec_squash** to home(B)
Swrite (B[i])	**SP_NO**	Send **GetS W** to home(B)

	SP_IN	If $Fwd = 1$, multicast violation;
		else, if $OR(Mask) = 1$, then
		request is satisfied by L2;
		else, send **Get_SW** to home(B)

	SPSQ	Send **Rec_squash** to home(B)
Flush	**SP_CO**	Send **Recflush** to home(B)
Commit	**SP_IN**	Change B's state to **SP_CO**
Squash	**SP_IN**	Change B's state to **SPSQ**

Table 5: **Actions performed by the cache controller for speculative reads/writes to sub-word i of block B (Flush, Commit and Squash requests apply to the entire block B).**

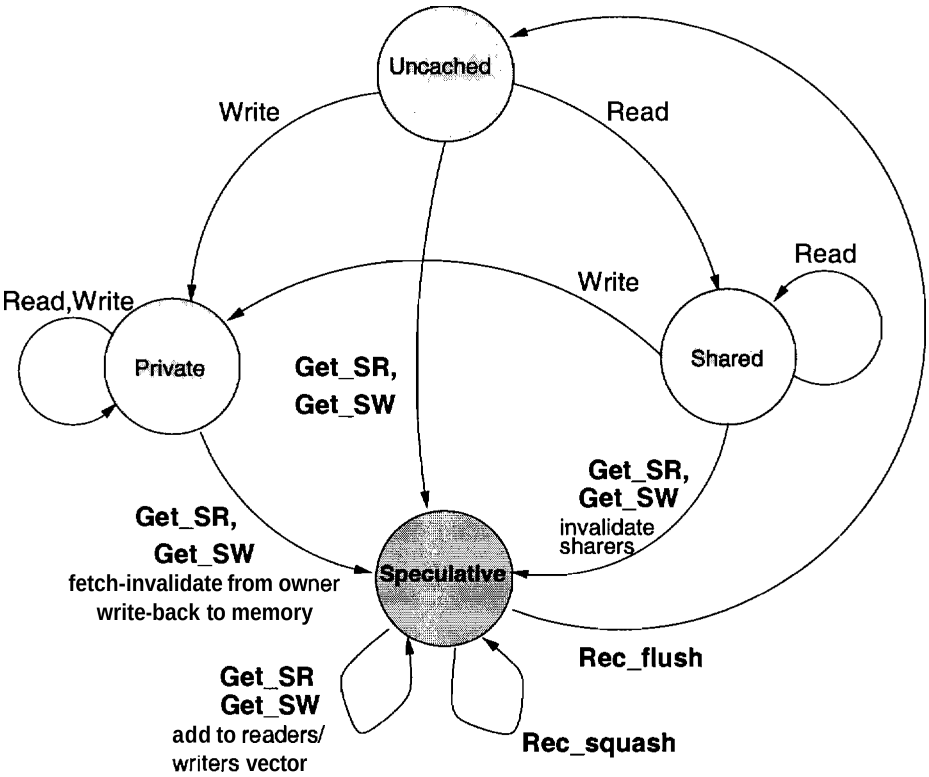


Figure 5: **Extended** DDSM **state machine for cache blocks at the directory. Extended states and transitions are shown in boldface.**

become non-speculative only after a reconciling request (*Rec_flush*) is satisfied by the directory. The extended protocol transactions are summarized in the state-machine diagram of Figure 5.

When a speculative request for a non-speculative block is received by the DDSM directory controller, all sharers of the block are invalidated. If the block is exclusive (dirty), it is written back to main memory. The block then becomes speculative, and its memory contents are sent to the requester's cache. Subsequent speculative accesses to the block are serialized by the directory controller, and the requester's identifier is recorded in the readers/writers bit-vectors. The directory includes the memory contents for the block in the reply, unless the request requires forwarding.

Blocks in the *Speculative* state may be modified in the L2 caches of multiple processors; DDSM produces a single, consistent version of the block at the end of speculation (Section 4.3). This support for multiple writable copies across distributed caches allows for automatic privatization of speculative blocks. Output- and anti-dependences (WAW, WAR) are resolved via this renaming mechanism, and thus do not cause data dependence violations. Violations of true (RAW) data dependences, however, are detected and trigger recovery mechanisms.

The directory checks for a RAW violation every time a speculative write (*Get_SW*) request is received. The check consists of comparing the processor identifier of the writer (Wid) to the identifier of the earliest speculative reader that may be data dependent on the writer ($R_{id} = min\{Readers[(W_{id} + \mathrm{I}),...,N]\}$). If $W_{id} < R_{id}$, a RAW violation is detected, and the directory multicasts a squash message to all processors with identifiers greater than or equal to R_{id} (Figure 3 ②).

System call	Action
Begin_Spec	Save processor context.
End_Spec	Set cache SP-bits to *SP_CO*.
Flush_Commit	Flush *SP_CO* blocks to directory

Table 6: DDSM **software interface.**

4.3 Task commits

The DDSM directory protocol allows multiple outstanding writable copies of a shared-memory block to reside in L2 caches during speculative execution. At the end of speculative execution, however, there should only be one program-order consistent copy of the block. The DDSM directory applies a reconciliation operation to all speculative versions of a block at the end of speculative execution to commit the program-order version of the block to main memory.

Committing speculative data to global memory is performed in two steps, initiated by system calls issued by the speculative application. In the first step, all SP-bits of speculative cache blocks are marked as committed (SP-CO) in the L2 cache, similarly to the gang-commit operation in SVC [8]. This operation is local to a node and does not require the sending of write-back messages to the distributed directories. This step is performed in sequential order with respect to the speculative tasks.

In the second step, SP-CO blocks across distributed caches are globally reconciled to produce the program-order version of each block. This step is performed in parallel by the caches and directory controllers.

The current implementation of the second step requires that caches flush *SP_CO* blocks to their respective home nodes. When the home node directory controller receives the first reconciliation request for a given block, it allocates a temporary buffer to hold all possible speculative versions of the block. The controller then multicasts fetch-and-invalidate requests to the speculative sharers of the block. It also sets a reply counter with the expected number of replies.

When a remote cache receives a fetch-and-invalidate request, it replies with the block's contents and its write mask, and invalidates the block. For each such reply received by the directory, the block's contents and mask are copied to the temporary buffer at the position given by the sender's identifier. When all versions are received (i.e., the reply counter reaches zero), a priority encoding operation, equivalent to the one performed by Hydra [10], produces the final version of the block. The priority encoding enforces that the final version of each sub-word of a block is the one written by the latest processor (in sequential order).

An example of the reconciling protocol transaction is discussed next and illustrated in Figure 3. The transaction is based on the read-exclusive operation to a shared block of existing directory-based protocols [13], where multiple invalidations are sent to read-sharers of a block. It begins when a reconciling request (*Rec_flush*) is received from a remote cache (node $i + 2$, **(A.)).**The directory handler then allocates a buffer for this transaction, copies contents and mask of the requester to the buffer, and multicasts fetch-invalidate request to the sharers of the block (B).When the directory receives all outstanding replies for the block *(C)*,the priority encoding function is applied to the reconciling buffer (D) and the final version of the block is committed to memory. The hardware requirements to support the extended reconciling transaction are that (1)fetch-and-invalidate acknowledgments be appended with contents and write mask for the block, (2) space for a data buffer indexed by processor identifiers be allocated by the directory at the beginning of reconciliation, and **(3)** a priority-encoding function be programmed in the directory controller (Figure 3).

4.4 Task squashes

The DDSM caches are responsible for squashing all data produced by a speculative task that has violated a data dependence. Squashing is achieved by globally setting all SP-bits of speculative cache blocks to the state SPSQ, and resetting the respective Fwd, SL and Mask bits, as shown in Figure 3④. When a restarted task accesses a SPSQ block, the cache controller requests a partially reconciled copy of the block from the directory (*Rec_squash*, Table 5). The partial reconciliation transaction is similar to the one described in the previous section, except that it only reconciles versions produced by earlier tasks than the requester i in program order (i.e. tasks $0, ..., i-1$).

When a task is squashed due to a data dependence violation, its execution is restarted by recovering the processor context (saved in the checkpoint phase at the beginning of speculative execution) via an interrupt. Restarted tasks execute in sequential order.

4.5 Checkpointing

The DDSM software/hardware interface is provided by a set of system calls (Table 6). The interface requires two extensions to the CPU. First, the processor must be able to save its context before speculation begins, and retrieve it when a violation interrupt is received. Second, the instruction set needs to support speculative instances of all memory operations. This can be achieved by assigning one bit in the instruction opcode to determine whether the access is speculative or not. This bit is not used during the decoding of memory instructions; it is passed on to the data cache controllers to determine whether the access is subject to the conventional directory-based protocol (non-speculative) or to the DDSM reconciling protocol (speculative).

When a compiler for DDSMs speculatively parallelizes a sequence of tasks, it (1) generates prologue code to set up the beginning of speculation, (2) marks the memory instructions inside the body of the task as speculative, and (3) generates epilogue code to set up the end of speculation. The prologue and epilogue consist of barriers, the system calls of Table 6, accesses to a local variable (violated),and non-speculative accesses to a shared-memory variable that stores the identifier of the Head task, as shown in Figure 2[2].

The prologue code initializes the Head variable and enforces that the re-issue of violated tasks is performed sequentially. The epilogue code enforces in-order commits of speculative blocks to local L2 caches (End-Spec, synchronized by *Head*) and initiates the global reconciliation of committed blocks to main memory (Flush-Commit, synchronized by a barrier) as described in Section 4.3.

4.6 Forwarding optimization

Some dynamic RAW dependences can be resolved via data-forwarding without incurring violations, as long as the writer and the reader are appropriately synchronized [14]. The basic DDSM reconciling directory does not provide automatic support for data forwarding. However, full support for forwarding across distributed nodes can substantially increase protocol complexity. This subsection presents a mechanism that extends the basic DDSM speculation system to provide support for a simple form of forwarding, denominated write-violate forwarding. The implementation of this extension is based on existing cache-to-cache transactions of conventional DSM directory protocols.

The write-violate forwarding scheme allows a processor P_i to forward a speculatively written block to later processor(s) (in sequential order) via cache-to-cache transactions, as long as the forwarded block is not re-written speculatively by P_i. Forwarding of a block is initiated by the directory controller (upon receipt of a speculative Get-SR or Get-SW request) if there is any earlier (program-order) writer recorded in the block's bit vector. The latest of such writers provides the data for the

[2] **Since a vendor compiler was used in this paper, the distinction between speculative and non-speculative memory operations was not implemented in the generated instruction set, but emulated via a CPU state bit that is set/reset via system calls. Accordingly, the non-speculative Head variable was represented as a special CPU register in the simulations, with access time equal to the average latency of incrementing a global variable with 16 processors. The performance impact of this simplification is negligible for the coarse-grain speculative windows of the studied benchmarks.**

cache-to-cache transaction.

If a forwarded block is speculatively re-written by a task, all later tasks are squashed and restarted. The implementation of this scheme relies on the *Fwd* flag introduced in Section 4.1. This flag determines whether a speculative block has been forwarded or not to 'other speculative tasks. Speculative reads are allowed to complete without generating violations, independently of the value of this flag; speculative writes to forwarded blocks, however, trigger the restart of later tasks.

4.7 Event ordering and consistency

In DDSM, a speculative load observes the effect of speculative stores from other tasks only when a block is forwarded. In this case, any further stores to the forwarded cache block squash all dependent loads, and force them to re-execute sequentially. Speculative writes issued by a processor are thus observed in sequential order by other processors.

The DDSM techniques apply to multiprocessors that support out-of-order memory accesses and relaxed consistency models such as release-consistency [7]. For release-consistent machines, the *End-Spec* system call is preceded by a memory fence. Such fence ensures that all squashes initiated by a task's speculative write are acknowledged (globally performed) before the task commits.

4.8 Cache replacements

The reconciliation operation requires that blocks from all speculative writers be available in their respective caches to generate the program-order version of the block [12]. When a speculatively written block is replaced from a cache, it cannot be written back to memory, since it is not guaranteed to be valid; nor can it simply be discarded, since its contents are needed to perform reconciliation. Therefore, a DDSM must conservatively squash all tasks data-dependent on a replaced block to ensure that the block's contents are re-generated.

When the DDSM directory receives a replacement request for a speculatively written block B from processor i, it multicasts squash requests for processor i and all later processors. The tasks executing in these processors are restarted sequentially; when they access squashed blocks, they request a partially reconciled, program-order copy of the block from the directory (*Rec_squash*, Table 4).

Blocks that are re-generated sequentially by squashed tasks are guaranteed to be program-order consistent, and are safe to be committed to main memory without generating violations in the event of a cache replacement, These blocks can be identified by the directory via an additional per-block bit (R) that is set upon receipt of a *Rec_squash* request.

Since the replacement of speculatively written lines causes false dependence invalidations and performance degradation, it is important to reduce the probability of their occurrence. Although not studied in this paper, the use of main-memory node caches, in addition to L2 caches [6, 12], can potentially provide a large, associative cache for speculative data and reduce the probability of capacity- or conflict-induced restarts. Since speculative state encoding is restricted to cache blocks and directory entries, existing node-caching mechanisms can be leveraged in the design of speculative node caches for DDSMs.

5 Experimental methodology

5.1 Machine model and configuration

The machine model assumed in this work is a release-consistent CC-NUMA multi-processor with hardware support for DDSM speculation. The system consists of up to 16 identical nodes connected by a 2-D mesh network. Each node has a memory bus connecting a single processor with on-chip L1 and L2 caches, main memory, directory controller and network interface (Figure 6).

Table 7 shows the parameters assumed for processors, caches and memory inside each node of the machine. The average simulated remote vs. local memory read latency ratio between adjacent

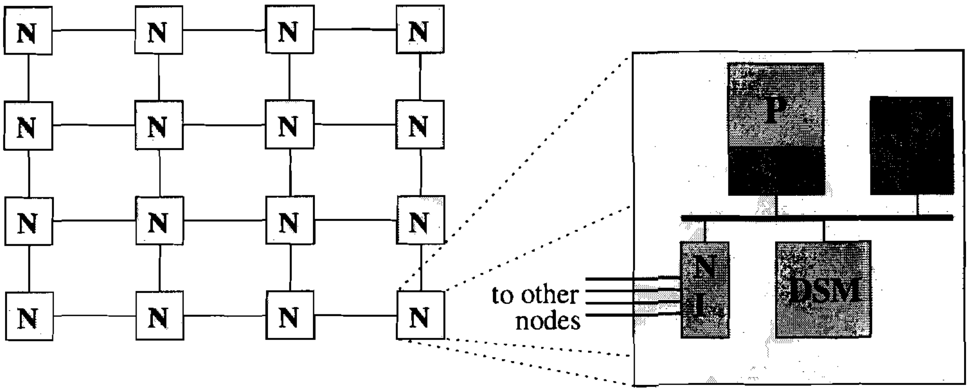


Figure 6: **Machine model: each node (N) has a memory bus connecting a single processor (P) with on-chip L1+L2 caches ($), memory (M), directory controller (DSM) and network interface (NI).**

Unit	Configuration
CPU	300MHz, 4-way out-of-order
Ll cache	16KB, write-through, direct-mapped
L2 cache	2MB, write-back, 8-way associative
Memory	60ns DRAM latency

Table 7: **Model parameters (all caches have 64B blocks).**

Bench.	Working set	Tasks
Turb3d	32x32x32, 3 iter.	loop/subr.
Health	7 levels, 24 iter.	recursive subr.
Power	3200 customers	loop/subr.
Perimeter	4K x 4K, 8 levels	recursive subr.
TreeAdd	256K nodes	recursive subr.
Ssm	1K-64K integers	loop iter.

Table 8: **Benchmarks used in the performance analysis.**

nodes is 2.8. The assumed configuration of the L2 cache allows the studied speculative programs to execute without the occurrence of replacement-induced restarts.

5.2 Workloads and simulation model

The performance analysis of Section 6 is based on the simulation of the speculatively parallelized benchmarks listed in Table 8 (with respective working sets) from the Spec95 and Olden suites, in addition to a sparse-matrix kernel developed by the authors for this study.

Turb3d is a benchmark from the Spec95 suite that simulates isotropic, homogeneous turbulence in a cube[3]. Turb3d has available parallelism across procedures that compute Fast-Fourier Transforms (FFTs) along distinct dimensions. Data dependence speculation allows parallelization across FFT subroutine calls without the need for inter-procedural analysis.

The Olden benchmarks Health, Power, Perimeter and TreeAdd are written in C and repre-

[3] **This benchmark's dataset has been scaled down from its original 64x64x64 size because the base RSIM simulator is not able to simulate more than 2 billion processor cycles; with the original dataset, RSIM reaches this maximum before the conclusion of the first iteration.**

sent codes with dynamic data structures, where parallelism is hard to be detected automatically at compile-time. Olden benchmarks are distributed in two versions: the sequential version, and a hand-parallelized version based on futures [9]. Hand-parallelized Olden programs have been studied previously in systems without data dependence speculation [1]. In contrast, this paper begins with the sequential version of the Olden programs.

The sequential version of these programs are manually prepared for speculative parallelization in this paper, due to the unavailability of a speculative parallelizing compiler. Speculative parallelization of the sequential programs of Table 8 for DDSMs consists of the addition of prologue/epilogue wrappers, partial inlining of procedures and reduction of summation variables. The reduction techniques performed manually in this study are supported by existing compilers [5, 4]; the support for inlining of existing compilers can be extended to allow partial inlining.

The benchmark Power solves a power system optimization problem. Parallelism in Power is difficult to detect automatically, due to the pointer structures that are used in its computation. However, Power can be speculatively-parallelized in the outermost loop, as long as reduction is applied to two summation variables (tmp.P and tmp.Q).

The Health benchmark simulates the Columbian health care system. Health utilizes pointer-based lists with elements that are dynamically inserted/removed, and traversed recursively. It is speculatively parallelized by partially inlining recursive calls and speculating across.subroutine calls.

TreeAdd is a benchmark that computes the summation of values in a tree. Perimeter computes the perimeter of a set of quad-tree encoded raster images. Both TreeAdd and Perimeter use pointer-based tree structures. Similarly to Health, partial inlining is used in these two benchmarks to expose subroutine-level parallelism to the speculation engine.

For systems without hardware support for speculative buffering [23], it is difficult to speculatively parallelize the programs that operate on dynamic data structures. Systems with fine-grain support for speculative parallelization [8, 10] may not be able to parallelize these benchmarks in the outermost loops and procedure calls due to the limited space available for speculative buffering. DDSM allows speculative parallelization of these programs due to its large, hardware-based speculative L2 buffers.

Procedure calls provide a potential source of speculative parallelism [15]; except for Ssm, all benchmarks under study exploit this type of parallelism. Procedure-level parallelism is exploited in the benchmarks with recursive subroutine calls (Health, TreeAdd, and Perimeter) by applying partial inlining (Figure 7).

Conventional compilers use inlining to both reduce procedure call overhead and enlarge the window of operations that can be inspected for global optimizations. In DDSM, partial inlining does not target either of these goals: it is used to expose speculative parallelism by increasing the number of procedure calls.

During speculative execution, these procedures are issued in parallel (Figure 7c)). After their execution have completed, the "combine" phase of the recursion may require either code serialization (Figure 7a)), or, when applicable, a reduction (Figure 7b)) to avoid data dependence violations. Serialization at the end of the recursion is applied to Health, while reductions are applied to TreeAdd and Perimeter.

The remaining benchmark, Ssm, is a kernel that models indirect addressing of array elements. The pseudo-code for this kernel is as follows:

```
for(i=0;i<Niterations;i++)
  for(j=0;j<Nitems;j++)
    Array[j] += Array[indirect(i,j)];
```

This program is not parallelizable automatically unless all data dependences implied by the indirect() function are known at compile time. However, it can be speculatively parallelized across the j loop. This program is used to study the behavior of DDSMs under several speculative window sizes (controlled by the parameter **Nitems**), mis-speculation frequencies (function **indirect()**) and number of processors.

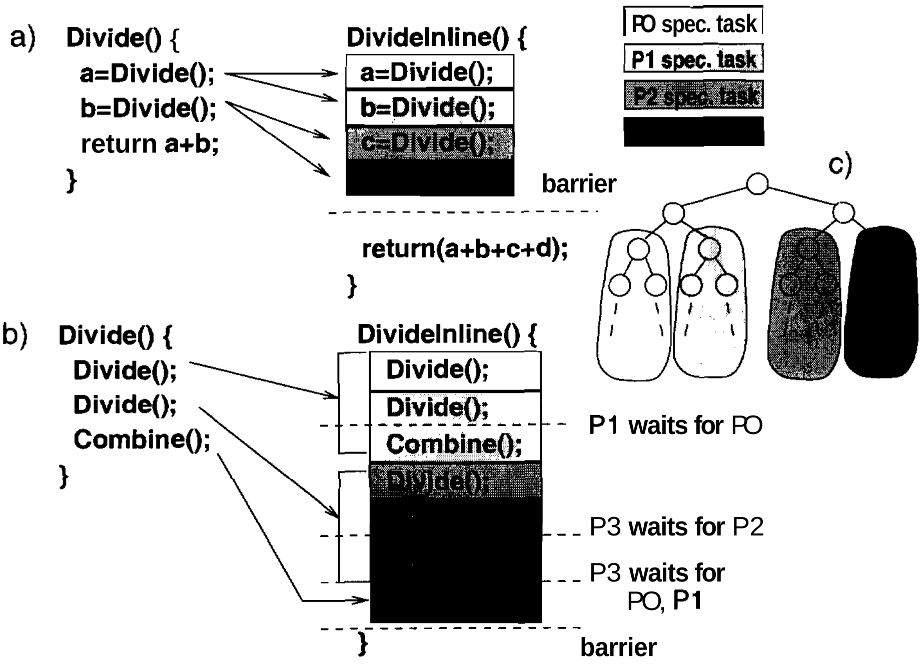


Figure 7: **Partial inlining of recursive calls to expose subroutine parallelism across 4 processors (c). The combine step of the recurrence uses partial-sum reductions (a) or explicit serialization (b).**

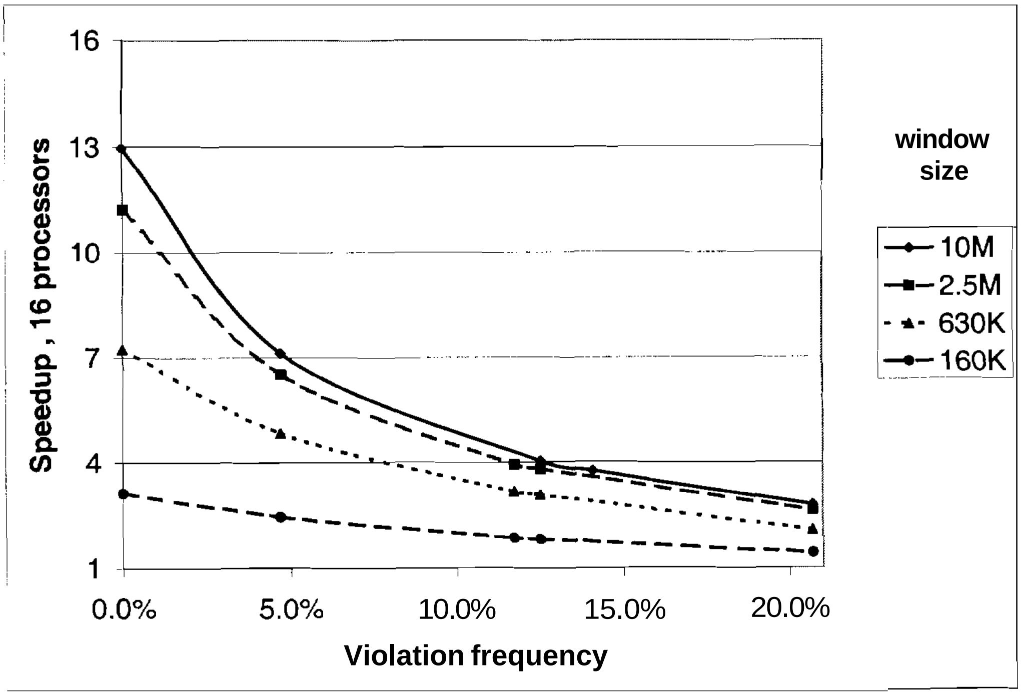


Figure 8: **16-processor parallel speedup for Ssm.**

All programs are compiled with Sun's Workshop C Compiler[4], version 4.2, with optimization

[4] The **benchmark** Turb3d **is converted from Fortran to** C **with** f2c.

level -x04. Two versions of each program are simulated: the sequential version (without speculation and synchronization code, executing in a single node) and the speculative version. Speedups are measured as execution time ratios (sequential/parallel) after data initialization.

The programs are simulated by a modified version of the RSIM simulator [16] that models the DDSM methods described in this paper. The original RSIM simulator models a release-consistent DSM with out-of-order uniprocessor nodes. The modifications applied to the original simulator provide support for: data buffering on L2 caches; processor context saving and recovery; reconciling directory operations, and cache flushing.

6 Performance Analysis

The performance analysis is divided in two parts. In the first part, the DDSM performance for the Ssm kernel is studied in detail. The second part analyzes the performance of the remaining benchmarks of Table 8. The following terms are used throughout the performance analysis of this section:

- *Window size:* total number of instructions, executed by the *original* sequential program, that are speculatively executed in parallel.

  Task size: number of instructions executed speculatively by a task - approximately (window size/number of tasks) for load-balanced tasks.

- *State size:* maximum amount of data stored in speculative L2 cache blocks of a single node.
- *Violation frequency:* ratio of restarted speculative tasks versus total number of tasks.
- *Flushing overhead (F_{ov}):* ratio of stall time due to flushing over total execution time for a speculative task.

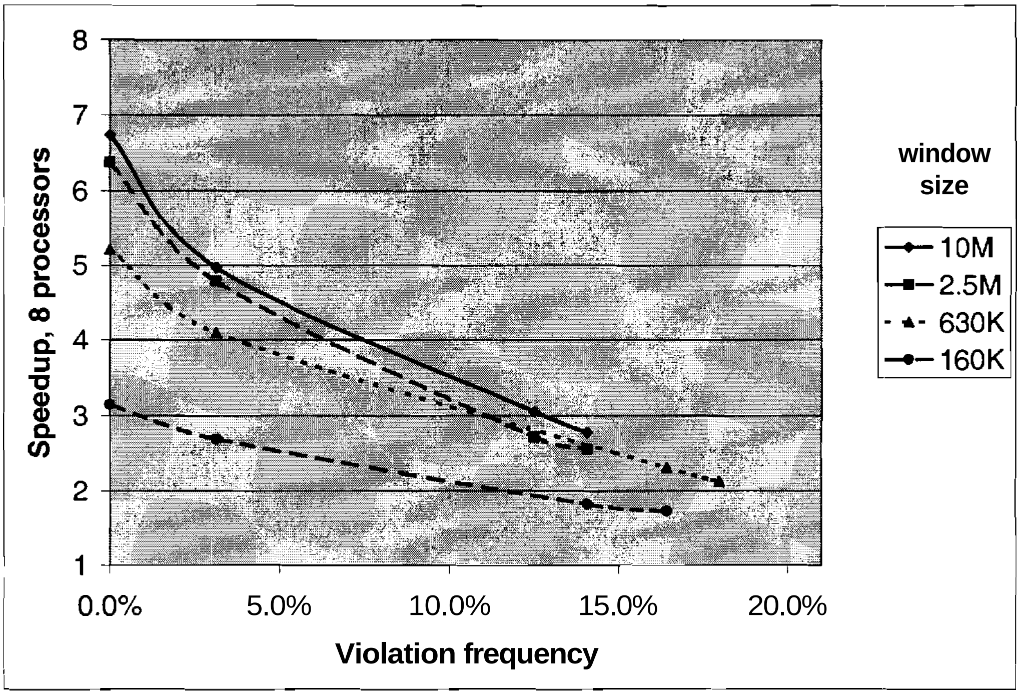


Figure 9: **8-processor parallel speedup for Ssm.**

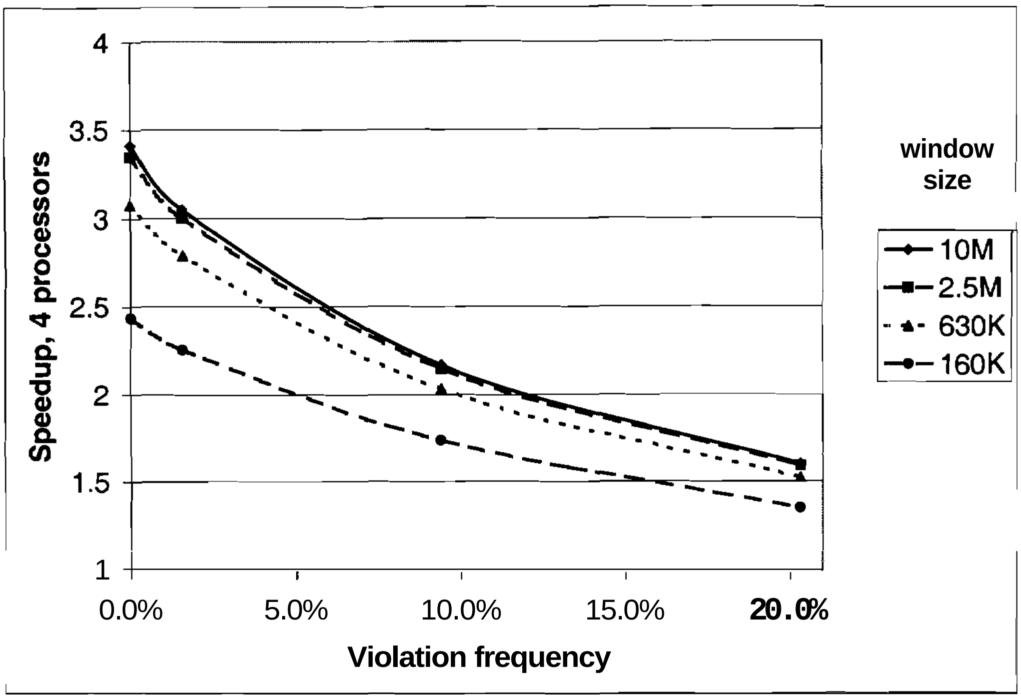


Figure 10: 4-processor parallel speedup for Ssm.

6.1 Kernel performance

In this analysis, the performance of DDSMs for the benchmark Ssm is analyzed as a function of the following parameters: number of speculative tasks (4, 8, and 16); window size (160K, 630K, 2.5M, and 10M instructions); state size (256 Bytes - 128 KBytes); and violation frequency (0% - 20%). This program implements the algorithm described in Section 5.2; in this study, the shared data structure Array[0..Nitems] is block-mapped onto the distributed memories.

Figures 8, 9 and 10 show plots of parallel speedup versus violation frequency for 16, 8 and 4 processors, respectively, and four window sizes. These plots show that DDSMs achieve good speedups for large task granularities and 0% violation frequency. The plots also show that, as window sizes decrease and violation frequencies increase, performance degrades.

The performance degradation due to both smaller windows and larger number of violations is more severe for DDSMs with larger numbers of processors. A window size reduction from 2.5M to 630K instructions yields speedup reductions of 55% and 9% for 16 and 4 processors, respectively. At 10% violation frequency, the performance degradations for the largest window size and 16 and 4 processors are 63% and 38%.

In addition to window sizes and violation frequency, the speculative state size impacts the performance of applications executing on DDSMs. In particular, at the end of the execution of a window, where all speculative blocks are flushed and reconciled.

Figure 11 plots the minimum and maximum flushing overhead for different speculative window sizes (and respective state sizes) and number of processors. The figure shows that the flushing overheads decrease as the speculative state size increases; the directory controllers can overlap a larger number of flushed blocks with increased efficiency via pipelining. For the largest window size, the maximum overhead is below 10%. The figure also shows that, for a given window size, the maximum flushing overhead increases as the number of processors increase.

In summary, this analysis shows that, for the Ssm kernel, the DDSM machine under study delivers parallel efficiency of 50% or more for up to 16 processors (i.e. speedups of 8 or more with

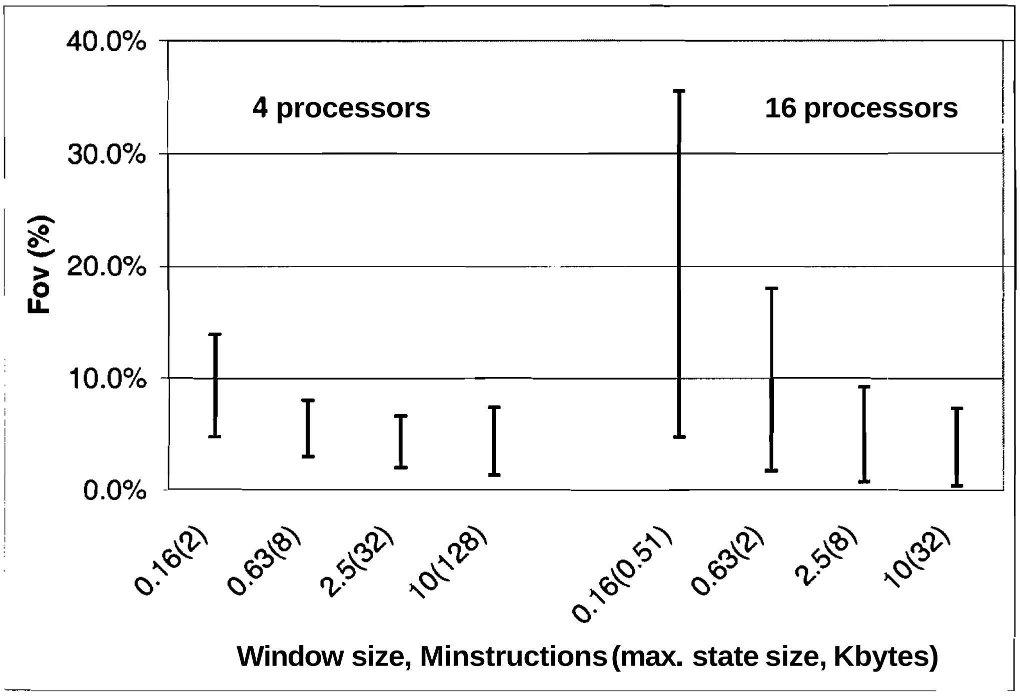


Figure 11: Minimum/maximum F_{ov} for Ssm.

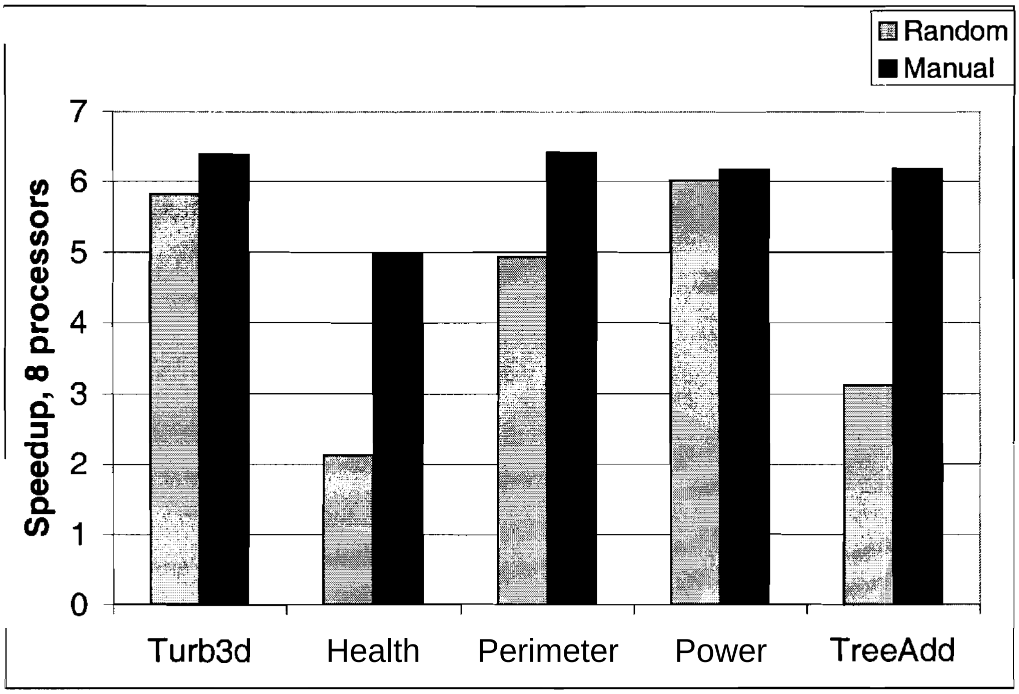


Figure 12: 8-processor DDSM speedup for two different data distribution policies: random and manual.

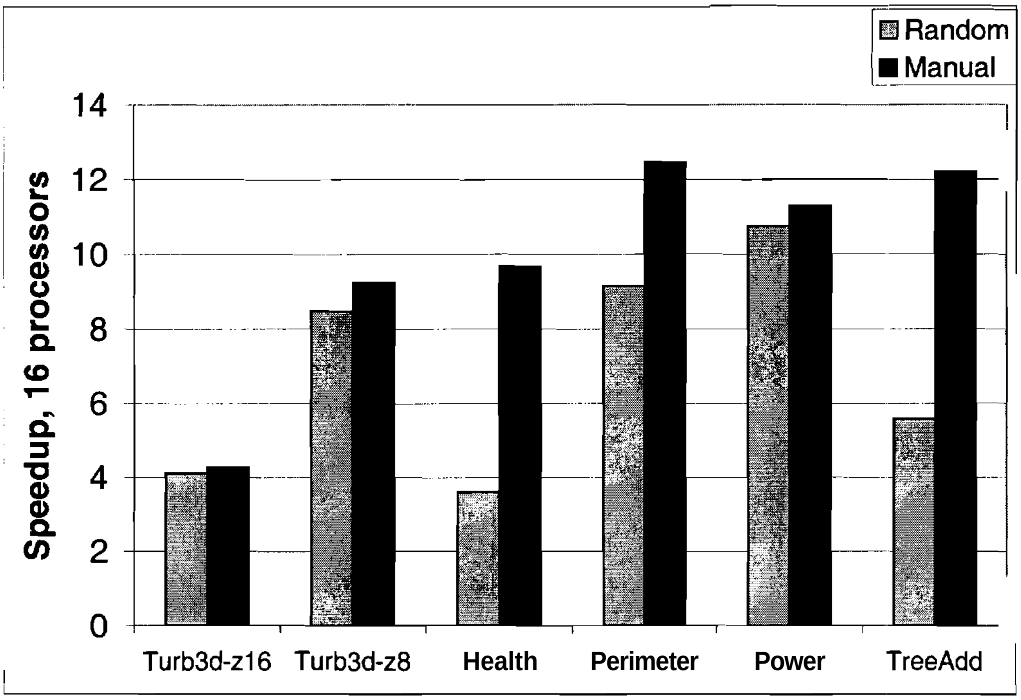


Figure **13:** 16-processor DDSM speedup for two different data distribution policies: random and manual. **Turb3d-z8** uses speculation across only 8 processors in the `zfft()` loop to avoid false-sharing violations.

respect to the unspeculative sequential code), if speculative windows sizes are of the order of millions of instructions and if the restart frequency is 4% or less. Configurations with smaller number of processors deliver better parallel efficiency for smaller window sizes and/or larger violation frequencies (50% efficiency at 10% violation frequency for 4 processors). The analysis also shows that the overhead of speculative state flushing is small for speculative windows of the order of millions of instructions.

6.2 Performance of speculative benchmarks

In this subsection, the remaining benchmarks of Table 8 are analyzed. The goal of this analysis is to determine the performance of DDSMs for applications with dynamic data structures and inherent coarse-grain parallelism - the speculative tasks of these applications have large window sizes and do not violate true data dependences.

The DDSM support for speculative parallelization allows for the distribution of code across processors. However, the distribution of data across memories also impacts the performance of distributed-memory machines. The techniques presented in this paper are orthogonal to data-distribution schemes: DDSMs can leverage proposed user-transparent locality enhancement techniques applied to conventional DSM mechanisms, such as migration [22] and prediction/speculation [11].

None of these techniques are used in the simulations presented in this paper. In order to study the sensitivity of DDSM performance for different data distributions, two different scenarios are considered. In the first scenario, data is distributed randomly across memories, modeling a naive, user-transparent distribution. In the second scenario, data is distributed manually via program annotation[5]. The results presented for the second scenario provide an estimate of the potential performance benefits of improved data distribution.

[5] **Both distributions are implemented at a cache block granularity in the RSIM simulator.**

Benchmark	Window size (instructions)	State size (KBytes) N=8	Task/state size (instr./Byte) N=8	F_{ov} (%) N=8	F_{ov} (%) N=16
Turb3d	21.5M - 45.0M	210	12.8 - 26.7	1.4% - 7.4%	1.6% - 4.9%
Health	895K - 1.36M	106 - 662	0.26 - 1.06	13.0% - 25.0%	12.8% - 25.0%
Perimeter	20M	559 - 731	3.42 - 4.47	5.0% - 6.7%	5.9% - 8.5%
Power	2.3M	86	3.35	0.4% - 5.3%	0.4% - 11.0%
TreeAdd	5.0M	525	1.19	9.3% - 10.0%	8.8% - 10.4%

Table 9: Minimum and maximum window size, state size, task/state size ratio, and flush overheads (N nodes).

Figure 12 shows the parallel speedups of an 8-processor DDSM. Except for Health, all programs achieve speedups in excess of 6.0 under the manual distribution scenario. On average, the random and manual speedups across the five benchmarks are 4.4 and 6.0, respectively.

Figure 13 shows the parallel speedups of a 16-processor DDSM . For this configuration, two different speculatively parallelized versions of the benchmark Turb3d are studied: one where all speculative windows are executed in parallel by 16 processors (Turb3d-z16), and a second version, where the 16-processor assignment applies to all windows, except for the FFT calculations along the *z* dimension (*zfft()*), which is executed by 8 processors only (Turb3d-z8).

The poorer performance of Turb3d-z16 is due to speculative false-sharing. Although the tasks in Turb3d are data-independent, they operate on sub-words of cache blocks. When speculative tasks are distributed across 16 processors, some accesses to different sub-words of the same cache block are issued by different tasks during speculative execution of *zfft()* calls. The pattern of these accesses yields multiple read-write operations, which are not supported by the write-violate forwarding optimization. The DDSM protocol must conservatively squash and restart tasks sequentially. This access overlapping does not occur when speculation of the FFTs across the *z* dimension is performed by 8 tasks.

The analysis of the Ssm kernel in Section 6.1 shows that both state size and window size impact the performance of DDSMs. Table 9 shows a summary of the speculative window and state sizes for each benchmark simulated with 8 processors.

These applications exhibit coarse-grain speculative windows of the order of millions of instructions, and speculative states of the order of hundreds of kilobytes. The simulation data shows that the speculative state of the coarse-grain tasks amenable to speculative parallelization in DDSM is unlikely to fit in L1 caches.

Table 9 also shows that Health and TreeAdd have the smallest ratios of speculative task size versus state size. These programs perform less computation per speculative block fetched from the memory system, and therefore are more sensitive to data placement.

Table 9 also shows the minimum and maximum flushing overheads (as defined in Section 6.1) for the benchmarks under study. With the exception of Health, the execution time overheads due to flushing range from 1% to 11%.

7 Related Work

Related data-dependence speculative designs have been recently proposed in the literature for several target architectures: chip-multiprocessors (CMP), ILP processors, and DSM machines. On-chip designs exploit fine-grain parallelism by relying on aggressive hardware support and low-latency communication. Distributed designs target coarser-grain parallelism, since inter-processor latencies are orders-of-magnitude larger. Table 10 presents a summary of how the five speculation methods of Table 1 are enforced in some of the related designs.

	Multiscalar-SVC Fine-grain ILP		Hydra Fine-grain CMP		Stampede Fine-grain CMP	
1	HW	L1 cache	HW	spec. buffers	HW	L1 cache cache ctrl
2	HW	VCL	HW	cache ctrl	HW	cache ctr
3	HW	on-demand	HW	buffer draining	HW	cache writeback
4	HW	gang-clear	HW	gang-clear	H/SW	gang-clear
5	H/SW	task descriptor	H/SW	co-proc. handlers	H/SW	handlers
	IACOMA Coarse-grain DSM				DDSM Coarse-grain DSM	
1	SW	shadow copies			HW	caches
2	HW	directory			HW	caches+directory
3	SW	copying			HW	reconciliation
4	H/SW	gang-clear			HW	cache flush
5	H/SW	unspecified			H/SW	system calls

Table 10: Comparison of different data dependence speculation proposals in terms of the 5 methods summarized in Table 1. HW and SW entries correspond to hardware and software solutions, respectively; H/SW corresponds to hybrid solutions.

A related speculative solution for Multiscalar [18] processors manages speculative data on modified L1 caches, called Speculative Versioning Caches (SVC [8]). In SVC, speculative data is buffered in the processor cache and co-resides with unspeculative data. A centralized hardware structure, the Versioning Control Logic (VCL), tracks dependence violations among blocks across the L1 caches of multiple processing units.

The methods for buffering speculative data and enforcing access ordering in DDSM bear similarities with the SVC approach. As is SVC, a DDSM cache contains both unspeculative and speculative data, which are differentiated via extra state information, and memory access ordering is enforced by hardware mechanisms.

Although the state extensions to cache blocks proposed for DDSM are similar to those of SVC, there are two main differences among the two designs: DDSM uses a simpler, 2-bit encoding of the state, and uses the L2 cache as a speculative buffer. SVC employs a more complex encoding of speculative state information to support on-demand, per-word commits.

The Hydra [10] design targets CMPs and speculatively parallelized thread-based code. DDSM uses a priority encoding function similar to Hydra to commit speculative blocks to main memory. However, unlike Hydra and other on-chip designs, DDSM employs a distributed structure to track memory access ordering.

The solution proposed by the IACOMA group in [23, 24] targets DSMs and (coarse-grain,partially parallel loops. Hardware support is provided for the detection of data dependence violations, while software explicitly manages buffering of speculative data in shadow memory copies.

The DDSM design proposed in this paper also targets coarse-grain tasks with low occurrence of mis-speculations. However, DDSM differs from the IACOMA solution in a very important aspect. In DDSM, the software overhead of maintaining shadow copies of speculative data is removed by using caches as buffers. This simplifies the programming interface, specially when speculation is performed on irregular data structures, since no explicit copying is required in the application code.

Concurrently with the research presented in this paper, related hardware solutions for distributed systems have been proposed in [2] and [20]. Both solutions consider systems with a small number of chip-multiprocessor building blocks and target fine-grain speculative parallelism. In contrast, DDSM supports efficient speculative execution of coarse-grain speculative windows for machine with larger number of distributed nodes.

The use of reconciling functions to allow loosely-coherent copies of shared-memory data (LCM) was proposed in [12]. Although the LCM scheme allows for buffering and reconciliation of shared-memory data, it does not provide speculation mechanisms to support data-dependence violation detection and recovery (methods 2, 4 and 5 of Table 1).

8 Conclusions and outlook

This paper proposes and evaluates a novel hardware-based approach for data-dependence speculation in DSM multiprocessors. The proposed DDSM architectural support allows for automatic extraction of speculative coarse-grain, thread-level parallelism from codes with dynamic data structures and statically unanalyzable shared-memory data dependences.

The architecture is based on hardware extensions to the cache coherence communication layer of DSMs that allow buffering of speculative data of large windows in L2 caches, and violation detection and reconciliation of speculative versions of memory blocks at the directory.

The simulation-based performance analysis presented in this paper shows parallel speedups of up to 6.5 and 12.5 for speculatively parallelized benchmarks with dynamic, pointer-based data structures, executing in 8- and 16-node configurations, respectively. It also shows that the design efficiently supports coarse-grain speculative windows (of the order of a million instructions) and state sizes (hundreds of KBytes per processor) when the frequency of data dependence violations is low (up to 4% for 50% parallel efficiency).

The DDSM hardware support for data dependence speculation is based on conventional, directory-based cache coherence transactions. It may be possible to optimize the base design described in this paper in order to improve overall system performance, at the expense of more complex protocol support. Research directions for optimizations include support for on-demand, per-block reconciliation (to eliminate the necessity of a global flushing operation) and use of main-memory node caches to hold speculative data.

References

[1] Carlisle, M. and Rogers, A. Software caching and computation migration in Olden. In *Proc. 5^{th} ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 29--38, 1995.

[2] Cintra, M., Martinez, J. F., and Torrellas, J. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *To appear, Proceedings of the 27th Annual International Symposium on Computer Architecture,* 2000.

[3] Dagum, L. and Menon, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science &' Engineering,* 5(1), 1998.

[4] Blume, W. et al. Parallel Programming with Polaris. *IEEE Computer,* Dec 1996.

[5] Hall, W. W. et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer,* pages 84–89, Dec 1996.

[6] B. Falsafi and D. A. Wood. Reactive numa: A design for unifying S-COMA and CC-NUMA. In *Proc. 24th International Symposium on Computer. Architecture,* 1997.

[7] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17^{rth} International Symposium Computer Architecture,* June 1990.

[8] Gopal, S., Vijaykumar, T. N., Smith, J. E., and Sohi, G. S. Speculative versioning cache. In *Proc. 4^{th} International Symposium on High-Performance Computer Architecture,* Feb 1998.

[9] R. *H.* Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems,* 7(4):501–538, Oct. *1985.*

[10] Hammond, *L.,* Willey, M., and Olukotun, K. Data speculation support for a chip multiprocessor. In *Proc. of the* 8^{th} *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pages *58–69,* Oct *1998.*

[11] Lai, A-C. and Falsafi, B. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999.*

[12] Larus, J., Richards, B., and Viswanathan, G. LCM: memory system support for parallel language implementation. In *Proc. Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pages *208–218, 1994.*

[13] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. The Stanford DASH Multiprocessor. *IEEE Computer,* Mar 1992.

[14] Moshovos, A., Breach, S., Vijaykumar, T., and Sohi, G. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture,* pages *181–193, 1997.*

[15] Oplinger, J. *T.,* Heine, D. L., and Lam, M. S. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architecture:: and Compilation Techniques,* Oct *1999.*

[16] Pai, V. S., Ranganathan, P., and Adve, S. V. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In *Proc.* 3^{rd} *International Symposium on High-Performance Computer Architecture,* Feb *1997.*

[17] Rauchwerger, L. and Padua, D. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In ACM Press, editor, *Proc. SIGPLAN'95,* pages *218–232, 1995.*

[18] Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture,* pages *414–425,* June *1995.*

[19] Standard Performance Evaluation Corporation. Spec newsletter, Sep *1995.*

[20] Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. A scalable approach to thread-level speculation. In *To appear, Proceedings of the 27th Annual International Symposium on Computer Architecture, 2000.*

[21] Steffan, J.G. and Mowry, T. C. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc.* 4^{th} *International Symposium on High-Performance Computer Architecture,* Feb *1998.*

[22] Verghese, B., Devine, S., Gupta, A., and Rosenblum, M. Operating systern support for improving data locality on CC-NUMA compute servers. In *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1996.*

[23] Zhang, *Y.,* Rauchwerger, L., and Torrellas, J. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Proc.* 4^{th} *International Symposium on High-Performance Computer Architecture,* Feb *1998.*

[24] Zhang, *Y.,* Rauchwerger, L., and Torrellas, J. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proc.* 5^{th} *International Symposium on High-Performance Computer Architecture,* Feb *1999.*