Scalable Speculative Parallelization on Commodity Clusters

Hanjun Kim Arun Raman Feng Liu Jae W. Lee† David I. August
Departments of Electrical Engineering and Computer Science
Princeton University
Princeton, USA
{hanjunk, rarun, fengliu, august}@princeton.edu
† Parakinetics Inc.
Princeton, USA
leejw@parakinetics.com

Abstract

While clusters of commodity servers and switches are the most popular form of large-scale parallel computers, many programs are not easily parallelized for execution upon them. In particular, high inter-node communication cost and lack of globally shared memory appear to make clusters suitable only for server applications with abundant task-level parallelism and scientific applications with regular and independent units of work. Clever use of pipeline parallelism (DSWP), thread-level speculation (TLS), and speculative pipeline parallelism (Spec-DSWP) can mitigate the costs of inter-thread communication on shared memory multicore machines. This paper presents Distributed Software Multi-threaded Transactional memory (DSMTX), a runtime system which makes these techniques applicable to non-shared memory clusters, allowing them to efficiently address inter-node communication costs. Initial results suggest that DSMTX enables efficient cluster execution of a wider set of application types. For 11 sequential C programs parallelized for a 4-core 32-node (128 total core) cluster without shared memory, DSMTX achieves a geomean speedup of 49×. This compares favorably to the 15× speedup achieved by our implementation of TLS-only support for clusters.

Keywords

loop-level parallelism; multi-threaded transactions; pipelined parallelism; software transactional memory; thread-level speculation; distributed systems

1. Introduction

Clusters of commodity servers and switches are deployed to speed up the execution of programs beyond the performance achievable on a single-board computer. Commodity clusters typically have high inter-node communication cost and lack globally shared memory. For these reasons, clusters are primarily used for large-scale scientific programs and network services like web search and mail. These programs consist of units of work that are mostly independent, allowing parallel execution with little inter-process communication or with predictable, regular communication among contexts. Consequently, the high inter-node communication latency typical of commodity clusters does not necessarily impact such programs significantly. In contrast, general-purpose applications are characterized by irregular data access patterns and complex control flow. Executing such programs on a cluster platform poses several challenges. The high inter-node communication cost for remote accesses of shared data means that an intelligent partitioning of the code and data is needed. Extracting parallelism to scale to the parallel hardware resources afforded by the cluster is difficult. For these reasons, such programs have been deemed unsuitable for execution on clusters.

Recent research has demonstrated that a combination of pipeline parallelism (DSWP), thread-level speculation (TLS), and speculative pipeline parallelism (Spec-DSWP) can unlock significant amounts of parallelism in general-purpose programs on shared memory systems [6, 24, 27, 29, 30, 32, 36, 37]. Pipeline parallelization techniques such as DSWP partition a loop in such a way that the critical path is executed in a thread-local fashion, with off-critical path data flowing unidirectionally through the pipeline. The combination of these two factors makes the program execution highly *insensitive* to inter-thread communication latency. TLS and Spec-DSWP use speculation to break dependences that rarely manifest at run-time. Doing so exposes more parallelism and allows the parallelization to scale.

The scalability and inter-thread communication latency tolerance of these techniques warrant a re-visit of the assumption about the unsuitability of clusters as a first-class platform for executing general-purpose programs. Referring back to the factors that enabled scalable execution on shared memory machines, the main unconventional requirement is support for speculative execution. Speculative execution requires the memory system to provide a means to checkpoint and rollback program state in the event of misspeculation. Most existing proposals for transactional memories or TLS memory systems target only small-scale parallel computers with tens of cores. They require specialized hardware [27, 28, 31, 33, 35] or rely on cache-coherent shared memory [20, 21, 24, 30]. These requirements are not met on a cluster platform. There have been some proposals for transactional memories on clusters [5, 10, 16, 19]. Because they implement single-threaded transactional semantics, these systems may be used to support TLS after modification to enforce an ordering among the transactions. However, the single-threaded transactional semantics implemented by these systems are insufficient to support Spec-DSWP which requires Multi-threaded Transactions (MTXs) [3, 24, 31, 32]. Current implementations of MTX require specialized hardware [31] or hardware with cache-coherent shared memory [24].

This paper proposes the Distributed Software Multi-threaded Transactional memory system (DSMTX), a software-only runtime system that enables both TLS and Spec-DSWP on commodity clusters. DSMTX implements the MTX concept originally proposed in a patent [3] and described in prior

work [24, 31]. For clusters, DSMTX supports the same APIs as those systems, allowing existing programs parallelized using MTX to be scaled up to hundreds of cores in a cluster without having to be re-written. An initial evaluation of the DSMTX system on a 128-core cluster suggests that application types heretofore considered unsuitable for parallel execution in a cluster environment can benefit from the large number of cores afforded by a cluster.

The contributions of this paper are:

1) A runtime system called Distributed Software Multi-threaded Transactional memory (DSMTX) that enables both TLS and Spec-DSWP on both shared memory machines and message-passing machines
2) A description of a DSMTX prototype built on top of the OpenMPI communication library [8] deployed on a commodity cluster with 32 nodes (128 cores), with a focus on the communication optimizations necessary for efficient parallelization in the face of high inter-thread communication cost
3) An in-depth evaluation of DSMTX using applications from the SPEC CPU92, SPEC CPU95, SPEC CPU2000, SPEC CPU2006, and PARSEC benchmark suites

Section 2 motivates this work along two orthogonal axes—support of hardware platforms and support of parallelization techniques. Section 3 presents the design of DSMTX. Section 4 describes the implementation of DSMTX and the code transformations and optimizations needed to hide the high cost of inter-thread communication. Section 5 presents the performance improvements obtained when DSMTX is used to support TLS and Spec-DSWP parallelizations. Section 6 contextualizes related work, and Section 7 concludes the paper.

2. Background & Motivation

Existing proposals for runtime systems to support speculative parallelization are limited both in the range of parallelization paradigms supported and in the types of hardware platforms supported. This work addresses these limitations. Section 2.1 provides background on TLS and Spec-DSWP. Section 2.2 motivates Multi-threaded Transactions (MTX) as an enabling mechanism for both TLS and Spec-DSWP. Section 2.3 presents machines without shared memory as first-class target platforms for these parallelization paradigms.

2.1. Loop Parallelization

Many scientific and numeric applications are embarrassingly parallel; they typically consist of counted loops that manipulate regular structures, accesses to which can be precisely analyzed statically. DOALL parallelization allows these applications to be executed in a very scalable manner [1]. DOALL partitions the iteration space into groups that are executed concurrently with no inter-thread communication. Consequently, DOALL often results in speedup that is proportional to the number of threads.

However, DOALL is not applicable when a loop has inter-iteration dependences. The code in Figure 1(a) shows such an example. The Program Dependence Graph (PDG) in Figure 1(b) has inter-iteration dependences (indicated by cycles in the graph) that prevent the loop from being parallelized using DOALL.

To parallelize such loops, either DOACROSS [11] or Decoupled Software Pipelining (DSWP) [22] can be applied. Both techniques handle inter-iteration dependences by means of communications among threads, and Figure 1(c) shows their respective execution plans. Each node represents a dynamic instance of a statement in Figure 1(a), where the number indicates the iteration to which it belongs. DOACROSS schedules the entire loop body iteration by iteration on alternate threads. DSWP partitions the loop body into multiple pipeline stages with each stage executing within a thread over all iterations. Figure 1(c) shows, ignoring the pipeline fill time, both DOACROSS and DSWP yield a speedup of 2× using 2 threads in the steady state when the inter-thread communication latency is one cycle.

Although the two techniques are comparable in applicability, DSWP provides more robust performance than DOACROSS because DSWP is more tolerant to increases in inter-core communication latency. This difference is attributed to their inter-core communication patterns: DOACROSS exhibits a cyclic communication pattern between threads, while DSWP exhibits an acyclic, or unidirectional, communication pattern. The pipeline organization of DSWP keeps dependence recurrences local to a thread, avoiding communication latency on the critical path of program execution. When the inter-core latency is increased from one to two cycles as in Figure 1(d), the speedup with DOACROSS reduces to 1.33×, but the speedup with DSWP remains 2× in the steady state.

Poor scalability of DSWP due to limited number of and imbalance among pipeline stages is not a problem. Huang et al. proposed DSWP+ that intentionally creates unbalanced pipeline stages to expose opportunities for scalable parallelization such as DOALL [15]. For example, `256.bzip2` cannot be parallelized directly with DOALL due to inter-iteration dependences from reading and writing files. DSWP+ can extract a DOALL stage (compressing blocks) that accounts for most of the execution time, and can exploit the scalability of DOALL. As more threads are assigned to this stage, the pipeline balance naturally improves. The *DSWP+[...]* notation describes the hybrid parallelization technique. Within square brackets, parallelization techniques applied to each stage are specified. `256.bzip2` is expressed as Spec-DSWP+[S, DOALL, S]. Here, *S* indicates a stage that is squentially executed, whereas *Spec-* indicates speculationi between stages.

Non-speculative loop transformations must respect all dependences in a loop, which significantly limits their applicability and scalability due to the conservative nature of static dependence analysis. Speculating that some dependences will not manifest at run-time allows these techniques to extract significantly greater amounts of parallelism. Referring to Figure 1, speculating that the loop will execute many times allows

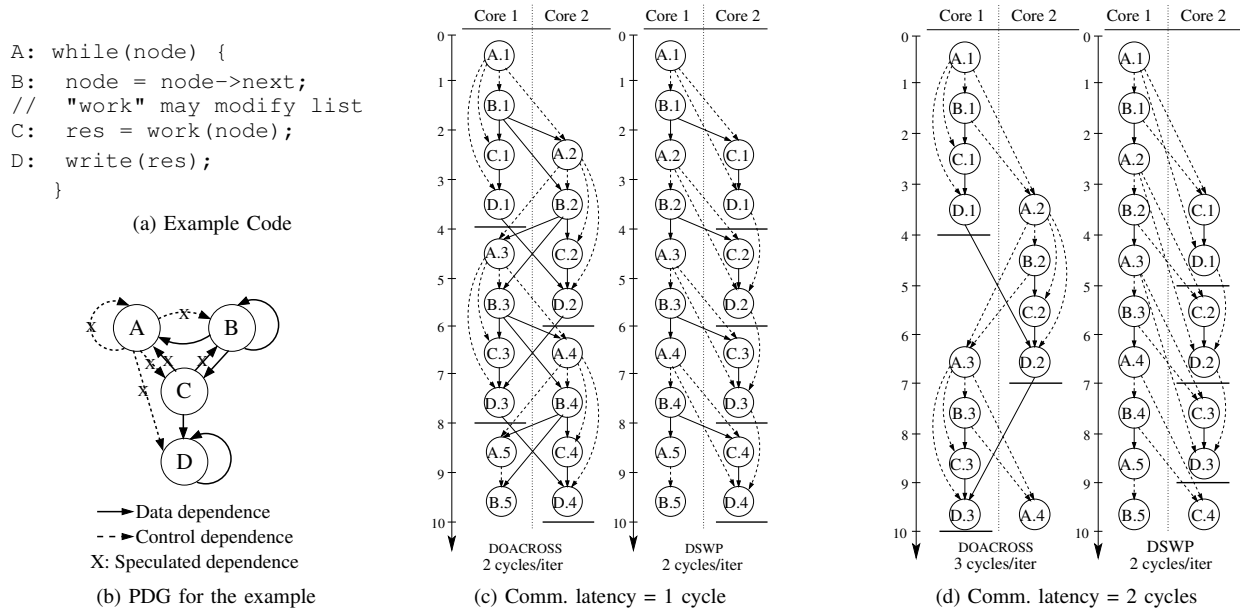


Figure 1: DSWP is more tolerant than DOACROSS to increases in inter-core communication latency.

the control dependences to be removed. Further, speculating that `work` does not modify the linked list allows memory dependences to be removed. Combined, speculation allows significant portions of different iterations to be overlapped in execution. All speculative parallelization techniques are iteration-centric in that they remove inter-iteration dependences. Consequently, a loop iteration is the unit of atomic work in both TLS and Spec-DSWP. The memory system must support the atomic execution of memory operations in a loop iteration.

2.2. Multi-threaded Transactions to Enable TLS and Spec-DSWP

In TLS, each loop iteration is executed by a single thread. Consequently, the unit of atomic execution is single-threaded. Conventional transactional memory or TLS memory systems that guarantee single-threaded atomicity may be used to support TLS. However, in Spec-DSWP, each loop iteration is executed in a staged manner by multiple threads, making the atomic unit multi-threaded. To support these atomic units, Multi-threaded Transactions (MTXs) [3, 24, 31] are required. An MTX represents an atomic set of memory accesses like its single-threaded counterpart, but may contain many sub-transactions (subTXs) each of which is executed by only one thread. subTXs are ordered by the program order of the sequential loop. Typically, a subTX corresponds to the execution of a pipeline stage on each iteration. All threads executing the subTXs of an MTX can see the results of uncommitted speculative stores executed by earlier subTXs within the MTX. Support for MTXs enables Spec-DSWP to extract deeper and wider pipelines. An MTX with only one subTX degenerates to a single-threaded transaction. This allows a system that implements MTX to implicitly support TLS in addition to Spec-DSWP.

2.3. Parallel Computers without Cache-Coherent Shared Memory

Despite the relative ease and simplicity of programming shared memory machines, commodity clusters dominate for large-scale parallel processing. A cluster node, with one or more processor cores, typically has its own private physical memory address space and constitutes an independent domain of cache coherence; multiple nodes communicate via explicit message passing through I/O channels. Unlike Symmetric Multiprocessors (SMP), clusters have scalable per-processor memory and I/O bandwidth. In addition, commodity clusters have a cost advantage over cache-coherent Non-Uniform Memory Access (ccNUMA) multiprocessors whose cost for cache coherence in hardware increases dramatically as the number of nodes increases. Because of their low cost, commodity clusters are the most widespread example of large-scale parallel computers that enable network services and high-performance computing today [23].

Since the memory system of a cluster is physically distributed across multiple nodes without a globally shared address space, remote data must be explicitly sent and received between a producer-consumer pair using a message passing protocol such as MPI. The difficulty of message passing-style programming, combined with high inter-node communication latency, has limited the use of clusters to applications such as scientific codes, many of which have little communication and can be easily parallelized. Although distributed transactional memory systems [5, 10, 16, 19] can be modified to enable TLS on clusters, they do not support MTXs and hence cannot be used by Spec-DSWP. Prior proposals to support MTXs [24, 31] require either hardware modifications or cache-coherent shared memory, so they cannot run on commodity clusters.

In addition to clusters, there are emerging multicore architectures targeted for certain workloads that discard even

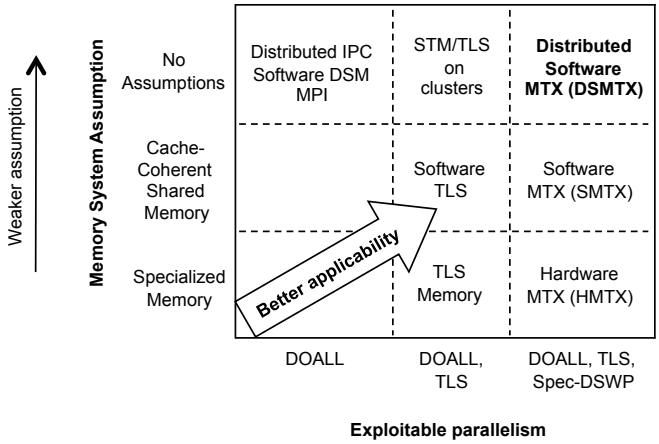


Figure 2: DSMTX enables the widest variety of parallelization paradigms, while making the fewest assumptions about the underlying hardware.

chip-wide cache coherence to minimize hardware cost and maximize energy efficiency. For example, Intel's new 48-core architecture does not support hardware cache coherence [14]. Such processors rely on explicit message passing for efficient inter-core communication and face the same programming challenges as clusters, with the main difference being lower communication latency. A runtime system that exposes additional parallelization opportunities adds great value to these platforms that lack shared memory.

Figure 2 summarizes the motivation of this paper. DSMTX aims to improve the applicability and scalability of parallelization technology for general-purpose programs by supporting the widest variety of parallelization techniques while making the fewest assumptions about the underlying hardware.

3. Distributed Software Multi-threaded Transactional Memory

Distributed Software Multi-threaded Transactional memory (DSMTX) enables both TLS and Spec-DSWP on both shared memory machines and message-passing machines. Section 3.1 describes the execution of MTXs on message-passing machines that do not have shared memory. As in SMTX, a prior implementation of MTX in software [24], DSMTX moves the speculation management overheads off the critical path by executing the MTX validation and commit operations in separate pipeline stages; Section 3.2 provides the details. Section 3.3 describes how DSMTX provides a unified virtual address space on top of the message-passing machine in order to allow each thread to initialize its memory state without an address translation, as on a shared memory machine.

3.1. Multi-threaded Transactions on a Message-Passing Machine

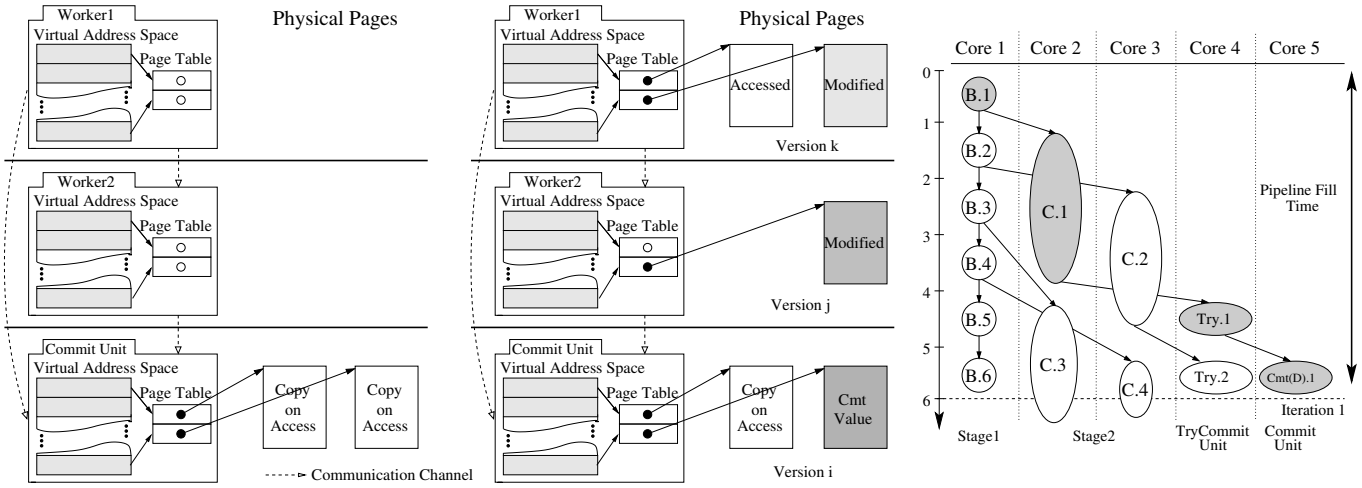
Most of the dependences that are speculated by Spec-DSWP are cross-iteration dependences; breaking these dependences allows code on different iterations to be executed in parallel. Consequently, an iteration is the unit of speculative execution, and updates to shared memory in an iteration must be executed atomically. Referring to the code in Figure 1(a) and the execution model in Figure 3(c), it can be seen that Spec-DSWP splits an iteration of the loop across multiple threads; this makes the unit of speculative execution multi-threaded. Each iteration may be wrapped in a multi-threaded transaction or MTX. As Figure 3(c) shows, each MTX is composed of one or more sub-transactions or subTXs. Each stage of a loop iteration executes in a subTX within the same MTX as the other stages of the iteration. subTXs are executed in the original sequential program order. To support loop parallelization, DSMTX orders MTXs according to the sequential loop iteration order.

In order to support the simultaneous execution of multiple MTXs, DSMTX creates worker threads that have access to different physical memory spaces. These "workers" are launched as POSIX processes, potentially on different nodes of a cluster. To take the overhead of speculation management off the critical path, DSMTX creates a separate "try-commit unit" that is responsible for validating MTXs and a "commit unit" that is responsible for committing MTXs. These also execute in different physical memory spaces. The memory updates by a worker in an MTX are forwarded to other workers that participate in the MTX via communication channels. Combined, the physical memories of the workers and the try-commit and commit units and the communication channel buffers allow each MTX to have the illusion of a private memory, and also allow multiple MTXs to be outstanding in the system.

The life cycle of an MTX in the context of loop parallelization, from initialization to commit, is described in detail below.

MTX Initialization: The first MTX in the program is initialized with the non-speculative memory state. This state is generated by the sequential, non-transactional code prior to the parallel section. One option is to have each worker generate the state by redundant execution of the sequential code. However, this may result in replicated side effects. DSMTX uses *Copy-On-Access* to initialize an MTX. Only the commit unit executes the sequential, non-transactional code to generate the initial non-speculative memory state. The memory state in a worker is initially marked as uninitialized (this is done at the memory page granularity). On the first access of a location on an uninitialized page, DSMTX implicitly copies the physical page from the commit unit to the worker's memory. The worker uses the values on this page to execute the MTX. Figure 3(a) and Figure 3(b) show this operation.

MTX Execution: All speculative loads and stores in an MTX happen in the private memories of the workers. Stores by an earlier subTX in an MTX must be visible to loads in a later subTX so that there is no intra-MTX misspeculation. Since subTXs are executed by different worker threads in different memories, each worker must forward its speculative stores to workers executing later subTXs. This is called *uncommitted value forwarding*, and it happens via the communication channels shown in Figure 3(b). *Only those workers that*



(a) Program State before Start (b) Program State during Execution (c) Execution Model

Figure 3: DSMTX overview

participate in the same MTX are connected. This ensures that the number of communication channels in the system does not grow quadratically in the number of threads. The uncommitted values are explicitly forwarded at the end of a subTX. A later subTX refreshes its memory with the uncommitted values before commencing execution.

The communication channels also serve to decouple the workers and the commit unit. Workers can execute subTXs from different MTXs, and can do so without waiting for validation and commit of the prior MTX. Figure 3(b) illustrates the decoupled execution. Worker1 is executing a subTX of MTX_k, while Worker2 is executing a subTX of MTX_j, and the commit unit is still committing MTX_i ($k > j > i$).

MTX Validation: An MTX is deemed to be free of conflict by means of a unified value prediction and checking mechanism. For control dependences, misspeculation is detected if the predicted value of the branch condition does not match the actual value at run-time. False memory dependences are automatically broken by means of memory versioning, so there is no need to check for their manifestation. A true memory dependence between a load and a store operation is checked by comparing the speculatively loaded value (predicted value) with the actual value stored by the store operation when that store is ready to be committed. This check is done by the try-commit unit. In all misspeculation cases, a signal is sent to the commit unit which orchestrates recovery.

MTX Committing: After all the subTXs in an MTX are deemed to be free of conflicts, the commit unit commits the entire MTX atomically. Through the same mechanism as uncommitted value forwarding, all stores in subTXs are forwarded from the workers to the commit unit. The commit unit commits the subTXs in a transaction by updating its memory with the forwarded values. The updates are done in order of subTX (which is the program order); if a memory location is updated in two different subTXs, the last update takes effect. This is called *group transaction commit*. Reiterating, since the commit unit's operations are decoupled from the worker threads by means of the communication channels, the overhead of commit does not impact the workers' execution. In the event of a conflict between MTXs, the commit unit signals the workers to restart the logically later MTX. The MTX is reinitialized with the committed memory state as before, and speculative execution resumes.

3.2. Speculation Management Via Pipelining

Figure 3(c) shows the execution model when DSMTX is used by Spec-DSWP to parallelize the example in Figure 1(a). Like SMTX [24], DSMTX pipelines the operations needed for managing the transactional/speculative execution. DSMTX moves the MTX validation and commit operations off the critical path of execution by executing the try-commit and commit units in their own pipeline stages. All stages of the pipeline are decoupled from each other by means of inter-core communication queues. This decoupled execution scheme allows the worker threads shown executing Stage1 and Stage2 in Figure 3(c) to run ahead and execute later MTXs, while the try-commit and commit units validate and commit earlier MTXs. Although the speculation management operations execute in parallel with the execution of the original loop in the workers, their serialization in the try commit and commit units may result in a performance bottleneck as the number of workers increases. While this did not happen for most of the benchmarks studied on the evaluation platform as described in Section 5, it may be noted that the algorithms of the try-commit unit and the commit unit are parallelizable and can alleviate this serialization concern.

3.3. Unified Virtual Address Space

DSMTX provides a Unified Virtual Address space (UVA) to all the threads in the system; a pointer to a memory location allocated by thread 1 is valid in thread 2 without the need for an address translation. UVA allows workers in DSMTX to share the same virtual address view across threads in different machines. UVA works by statically assigning ownership of

different virtual address regions to different threads. Most of the time, a memory allocation request is satisfied by a thread by allocating from the virtual address region that it owns. Synchronization among threads is required only when the memory requirements of a thread exceed the size of the region that it owns. Ownership information is encoded in the upper bits of the virtual address. This information is used to fetch a page from a remote node if the page does not exist locally. MTX initialization via Copy-On-Access relies on the UVA implementation to load committed data. Finally, because DSMTX implements the same interface as SMTX, existing codes parallelized using the SMTX library that relies on shared memory can be executed without any modification (with the exception of worker initialization, described in the next section) on machines without shared memory.

4. Implementation

DSMTX is implemented as a user-land library on top of OpenMPI [8]. Section 4.1 explains the difference between the DSMTX API and the SMTX API proposed in [24]. Section 4.2 describes the low-level implementation details of the various MTX operations, highlighting the different communication mechanisms for MTX initialization and MTX execution. Finally, Section 4.3 explains recovery from misspeculation in greater detail.

4.1. DSMTX API

Table 1 shows the API of the DSMTX runtime system. The API is identical to that of the SMTX system, with the addition of two functions: `DSMTX_Init` and `DSMTX_Finalize` that must be called at the beginning and the end of programs, respectively. This is an artifact of the MPI-based implementation since MPI requires `MPI_Init` and `MPI_Fini` to be invoked. As an implementation detail, unlike SMTX, `mtx_spawn` in DSMTX does not create a new worker because DSMTX spawns all the workers at the beginning of the program. Instead, `mtx_spawn` causes the worker whose thread ID matches `tid` to execute `function` with argument `arg`. Notably missing from the API are custom `malloc` and `free` functions. This is because DSMTX hooks the system `malloc` and `free` calls in order to implement the Unified Virtual Address (UVA) abstraction (explained in Section 3.3).

4.2. DSMTX Communication Support for MTX Operations

Virtually all MTX operations require communication among the workers, try-commit unit, and commit unit, albeit in a unidirectional way. MTX initialization requires data to be sent from the commit process to the workers executing the MTX; uncommitted value forwarding during MTX execution requires the communication of speculative stores to the later subTXs; speculative stores and loads must be forwarded for MTX validation and commit. Since the threads executing the MTXs do so in different physical memory spaces, all the communication must be explicitly managed by the runtime system. To minimize the cost of such communications, DSMTX implements two different mechanisms.

Copy-On-Access (COA): DSMTX allows only the commit unit to execute the sequential, non-transactional portions of a parallelized program. The program state generated by these sequential portions may be used by the worker threads in the parallel portion of the program. All the data that is live into an MTX must be transferred from the commit unit to the workers participating in that MTX.

A naïve solution is to send the entire live-in data set to all the workers when parallel execution begins. However, finding the exact minimal live-in set is difficult, and may require the parallelizer to be conservative and send the entire program state which could be very costly. Even if there is exact information about the live-in data, it may not be required by each worker thread since code in the parallel section is split across the many workers. A better solution is to send data only when it is required (Copy-On-Access). At the beginning of parallel execution, each worker thread adds access protections to its heap space. When a thread accesses a memory location, the protection results in a page fault, causing a transfer of data from the commit unit to the worker. This mechanism transfers only data that is really needed by each worker, thereby avoiding the transfer of excessive, unnecessary data.

On a cluster, the round-trip latency induced by COA can be prohibitive if COA is done at a word granularity. DSMTX implements COA at the memory page granularity (4096 bytes on our experimental platform). Words on memory pages in the commit unit are not modified once the sequential portion of the program has ended and parallel execution has begun. By sending a memory page in response to a request for a word, DSMTX aggressively speculates that words near the original location will be accessed by the worker in the future. This serves as a constructive prefetching mechanism that amortizes the round-trip latency cost over accesses to multiple locations in the same page.

Message Passing: COA is suitable for transferring data that will not be modified by the sender. However, during parallel execution, memory locations may be accessed and modified multiple times in highly irregular patterns by different worker threads. Because of the asymmetric fashion in which Spec-DSWP partitions a loop between the worker threads, memory locations on the same page may be updated by threads executing different stages of the pipeline. This calls for a communication mechanism at a finer granularity than an entire page. DSMTX supports communication of such data by means of message passing queues over which data is communicated at the word granularity. Consider uncommitted value forwarding in an MTX. When a worker executes `mtx_write`, the address and value of the location to which the speculative store occurred is sent over the message queue between the worker and the commit unit.

Instead of directly using the MPI primitives for sending and receiving data, DSMTX builds an enhanced message queue on

Operation	Description
One-time Operations	
`DSMTX_Init(&argc, &argv)`	Initialize MPI and Unified Virtual Address space
`DSMTX_Finalize()`	Finalize MPI and Unified Virtual Address space
`system = mtx_newDSMTXsystem(n, configuration)`	Initialize `system` of `n` threads with the given pipeline `configuration`; create queues, etc.
`mtx_deleteSMTXsystem(system)`	Finalize `system`; delete various data structures
`mtx_spawn(function, tid, argument)`	Execute `function` with the provided `argument` if `tid` matches ID of self
`mtx_commitUnit(system, recovery_fun, commit_fun, arg)`	Encapsulates the functionality of the commit unit; executes `commit_fun` when an MTX successfully commits, and executes `recovery_fun` to generate correct program state following misspeculation
`mtx_tryCommitUnit(tid, arg)`	Encapsulates the conflict detection mechanism that is executed by the try-commit unit
Running Operations for Workers	
`mtx_produce(queue, value)`	Enqueue `value` in specified `queue`
`value = mtx_consume(queue)`	Dequeue and return `value`
`state = mtx_begin(tid)`	Enter an MTX by updating memory with stores in this MTX by earlier subTXs; notify commit unit that a new MTX has been entered; returns the `state` of the system to check for misspeculation or termination
`state = mtx_end(tid)`	Exit the current MTX and notify later stages including the commit unit of the same; returns the `state` of the system to check for misspeculation or termination
`mtx_writeTo(tid, dest, addr, value)`	Forward an $\langle$`addr`, `value`$\rangle$ tuple to the specified `dest`ination
`mtx_writeAll(tid, addr, value)`	Forward an $\langle$`addr`, `value`$\rangle$ tuple to all later stages in the pipeline including the try-commit unit and the commit unit
`mtx_read(tid, addr, value)`	Forward an $\langle$`addr`, `value`$\rangle$ tuple to the try-commit unit
`mtx_misspec(tid)`	Notify the commit unit of misspeculation
`mtx_terminate(tid)`	Notify the commit unit of termination of the parallel section
`mtx_doRecovery(tid)`	Handle recovery from misspeculation

Table 1: DSMTX Library Interface

top of the MPI library. While pipelined execution is insensitive to inter-thread communication latency, it is sensitive to the overhead of the operations required to send a datum [25]. The MPI send (`MPI_Send`) and receive (`MPI_Recv`) primitives execute 500 to 2,295 instructions, respectively, to send and receive 8 bytes [8]. DSMTX avoids using `MPI_Send` and `MPI_Recv` for every produce and consume by buffering the produced values in the message queue, and invoking `MPI_Send` only when the buffer fills up to a predetermined size. This reduces the operational overhead of most produces and consumes, and amortizes the cost of the underlying MPI primitives. Note that unlike `MPI_Bsend`, a buffered send operation in MPI, the message queue automatically manages the buffer space so programmers need not worry about explicitly allocating, deallocating, or overwriting it.

4.3. MTX Rollback

When an MTX is detected to conflict with an earlier MTX, it must be re-executed. After receiving a misspeculation signal from a worker thread or the try-commit unit, the system goes into recovery mode. First, all threads hit a barrier to ensure that others have also entered recovery mode. Second, message queues containing speculative state are flushed. Third, all threads again hit a barrier. Fourth, all threads but the commit unit reinstate the access protection to the heap area, effectively discarding the remaining speculative state. The commit unit executes the loop iteration corresponding to the aborted MTX in single-threaded fashion. During the execution of this iteration, the commit unit may produce data via the message queues to the workers; this explains why the barrier in step three is necessary. Finally, all threads hit a barrier to ensure that parallel execution may recommence. New MTXs are spawned and COA ensures that the data they access will be the new, committed data from the commit unit's memory space.

5. Evaluation

5.1. Evaluation Platform

The DSMTX system is evaluated on a 128-core cluster (32 nodes $\times$ 4 cores/node). Each node is a Dell PowerEdge 1950 with two dual-core processors (Intel Xeon 5160 @ 3.00GHz) and 16GB of RAM. The nodes are interconnected by an InfiniBand network, and OpenMPI [8] (v1.3.0 with gcc v3.4.6, -O3) is used as the underlying communication layer.

CPU-intensive benchmarks that require speculation for efficient parallelization are selected from the SPEC CPU92, CPU95, CPU2000, CPU2006 [26], and PARSEC benchmark suites [4]. Codes are manually parallelized in a systematic manner. Due to the difficulty of manual application of compiler algorithms, benchmark selection was influenced by source code tractability. The selected benchmarks exhibit diversity in terms of parallelization paradigms, types of speculation required, and communication characteristics. To find candidates for speculation, we used loop-level profiling and analysis information from the VELOCITY compiler [7] and the LLVM compiler infrastructure [17].

Table 2 lists the selected benchmarks along with information such as benchmark description, parallelization paradigm, and the speculation required for effective parallelization. Details of each benchmark can be found in [4, 26]. In Table 2, when pipeline parallelism is exploited, the parallelization technique applied to each stage is indicated within square brackets in the expression DSWP+[...]. S indicates that a stage executes

sequentially. A parallelization that uses speculation is prefixed by *Spec-*. Spec-DSWP+[...] indicates that speculation spans the entire pipeline. For such parallelizations, MTXs are necessary.

Depending on application structure, an appropriate parallelization paradigm is chosen. By supporting both TLS and Spec-DSWP, DSMTX achieves the best speedup provided by either. DSMTX is compared against our implementation of TLS-only support for clusters. TLS parallelizations are done according to the algorithms in [27, 34]. Where possible, optimizations such as minmax reduction and accumulator expansion are applied in both sets of parallelizations. Dependences in functions such as `rand` are relaxed to allow reordering of calls to the function due to the commutativity of such functions. In two benchmarks, `052.alvinn` and `swaptions`, the DSMTX and TLS-only parallelizations are the same—both are Spec-DOALL with no communication among the threads except in the event of misspeculation.

5.2. Performance Scalability

Figure 4 shows the speedup over sequential execution of each benchmark. In each graph, the x-axis shows the number of cores, and the y-axis shows full application speedup. All execution times were averaged over five runs. This section discusses the parallelization and scalability bottlenecks of each application.

`052.alvinn`: The loop that is parallelized is at the second level in a loop nest. All the threads must be initialized with data from the commit unit at the beginning of each invocation of the loop and must communicate data for a reduction operation over many arrays at the end of each invocation. These synchronizations limit the speedup.

`130.li`: The parallelization speculates that each script is independent of the others and does not change the interpreter's environment or cause the interpreter to exit altogether. Accesses to the memory state corresponding to the interpreter's environment are executed transactionally. Control flow speculation is used to break the dependences from the program exit condition. In TLS, speedups are limited due to synchronization arising from the `print` instruction.

`164.gzip`: Compression works in three stages: 1) read block from input file, 2) compress block in parallel, and 3) write compressed block. `164.gzip` uses a variable block size, with the starting point of the next block being known only after the current block is compressed. This dependence prevents parallelization. The Y-branch [6] is used to break the dependence and new blocks are started at fixed intervals. Multiple versions of the arrays used for holding the blocks are automatically created by DSMTX. Speedup is limited by communication bandwidth.

`179.art`: The execution times of iterations in the parallelized loop are highly unbalanced due to the varying trip count of the inner loops. A round-robin assignment of iterations to threads results in wasted execution time due to the imbalance. To address this, the first stage distributes work based on queue occupancy as a proxy for load on each parallel-stage thread. As the number of threads increases, the round-trip communication cost causes TLS speedup to grow slower than DSMTX using Spec-DSWP.

`197.parser`: The values of various global data structures are speculated to be reset at the end of each iteration and control flow speculation for error cases is applied. An entire dictionary must be copied from the commit unit on access by the worker threads, and sentences must be transferred from the first stage to later stages. As a result, the communication bandwidth becomes a performance bottleneck as the number of threads increases beyond 32.

`256.bzip2`: Like `164.gzip`, the second stage compresses blocks in parallel. Unlike `164.gzip`, the Y-branch is not necessary because the block size is known in the first stage. DSMTX creates multiple versions of the block array. Control flow paths to handle error conditions are speculated as not taken. While Spec-DSWP sends the whole input file to each DOALL thread, TLS sends only the file descriptor of the input file to each thread. In this case, communication bandwidth is the main factor that affects performance, so TLS shows slightly better performance than Spec-DSWP.

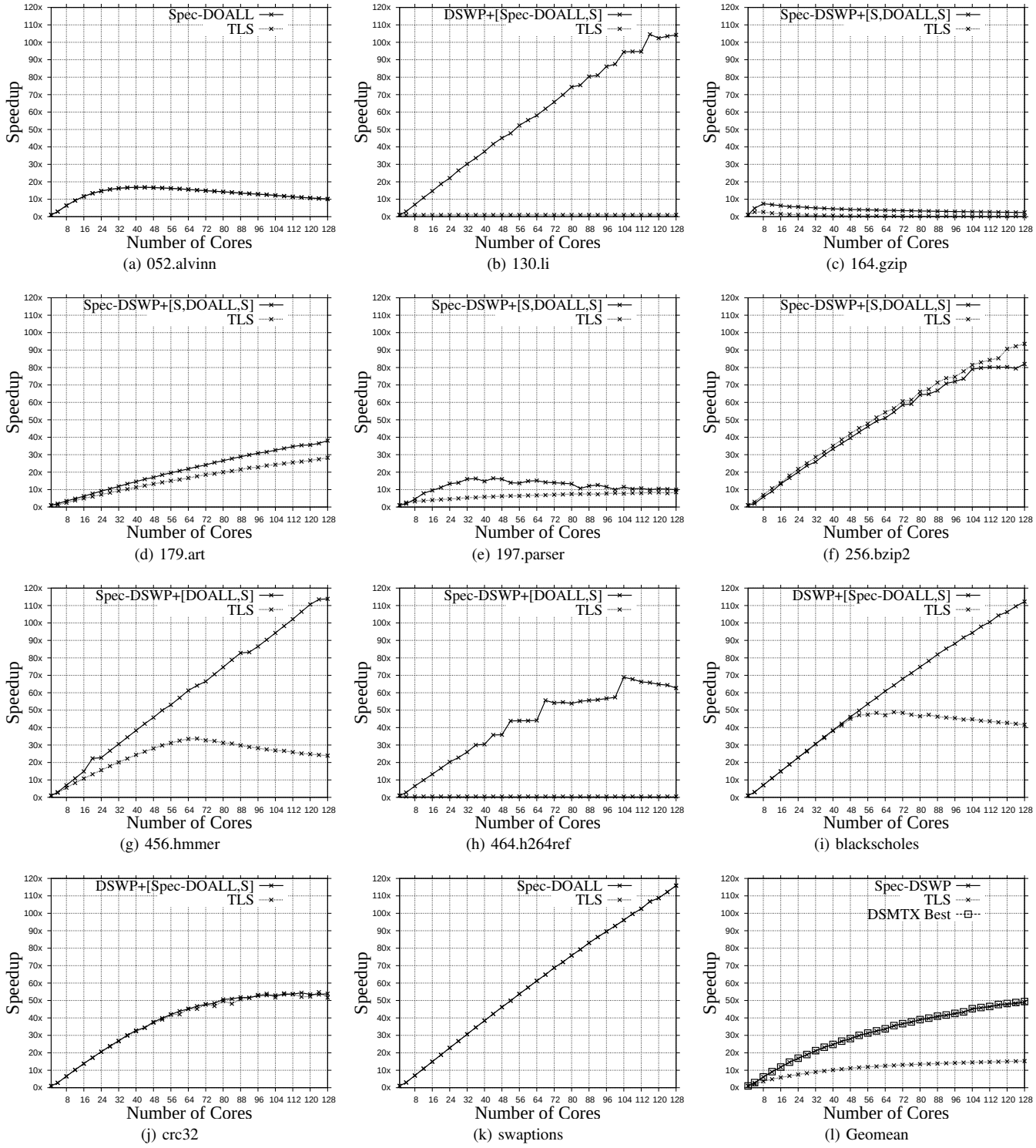
`456.hmmer`: The first stage calculates scores in parallel, while the second stage computes a histogram of the scores sequentially. Max-reduction is applied to compute the maximum score. Spec-DSWP scales to a higher core count than TLS because the cyclic dependence of TLS puts inter-thread communication latency on the critical path of program execution, which becomes the bottleneck as the number of threads increases.

`464.h264ref`: Groups of Pictures (GoPs) are encoded in parallel. Dynamic memory versioning enabled by DSMTX breaks false memory dependences in the parallel stage and allows the parallel encoding of GoPs. The source and destination of the synchronized dependences are inside an inner loop, effectively serializing the TLS execution. Spec-DSWP moves the dependence cycle to a separate stage allowing other stages to execute without waiting. Speedup is limited primarily by the number of GoPs available.

`crc32`: On a cluster with a network file system, the original program spends most of its execution time reading the files. To reduce this effect, block read is used instead of character read by replacing `getc` with `fread`. The program is speculatively parallelized assuming errors do not occur in the CRC computation. Its speedup is limited by the number of input files.

`blackscholes`: Speculation is applied on an error condition. As in `456.hmmer`, the TLS parallelization peaks at 52 cores due to increased inter-thread communication latency at higher numbers of cores.

`swaptions`: As in `blackscholes`, the outermost loop is parallelized with speculation on an error condition during price calculation. Scalability is limited by the input size.



(a) 052.alvinn (b) 130.li (c) 164.gzip

(d) 179.art (e) 197.parser (f) 256.bzip2

(g) 456.hmmer (h) 464.h264ref (i) blackscholes

(j) crc32 (k) swaptions (l) Geomean

Figure 4: Performance scalability of Spec-DSWP and TLS using DSMTX

Benchmark	Source Suite	Description	Parallelization Paradigm	Speculation Types
052.alvinn	SPEC CFP 92	neural network	Spec-DOALL	MV
130.li	SPEC CINT 95	lisp interpreter	DSWP+[Spec-DOALL,S]	CFS,MVS,MV
164.gzip	SPEC CINT 2000	file compressor	Spec-DSWP+[S,DOALL,S]	MV
179.art	SPEC CFP 2000	image recognition	Spec-DSWP+[S,DOALL,S]	MV
197.parser	SPEC CINT 2000	English parser	Spec-DSWP+[S,DOALL,S]	CFS,MVS,MV
256.bzip2	SPEC CINT 2000	file compressor	Spec-DSWP+[S,DOALL,S]	CFS,MV
456.hmmer	SPEC CINT 2006	gene sequence database search	Spec-DSWP+[DOALL,S]	MV
464.h264ref	SPEC CINT 2006	video encoder	Spec-DSWP+[DOALL,S]	MV
crc32	Ref. Impl.	polynomial code checksum	DSWP+[Spec-DOALL,S]	CFS,MV
blackscholes	PARSEC	option pricing	DSWP+[Spec-DOALL,S]	CFS
swaptions	PARSEC	portfolio pricing	Spec-DOALL	CFS

CFS = Control Flow Speculation, MVS = Memory Value Speculation, MV = Memory Versioning

Table 2: Benchmark Details

5.3. Communication Characteristics and Optimization

Figure 5(a) presents each application's bandwidth requirements when parallelized using Spec-DSWP. Bandwidth is computed by dividing the total data transferred via DSMTX by the application's execution time. To show how the bandwidth requirement increases as more cores are used, data is presented for three consecutive core counts starting from the number of pipeline stages in the parallelization.

The figure shows that `164.gzip` has very high bandwidth requirements that grow as the number of threads is increased, explaining its limited scalability. Interestingly, the amount of data transferred by `256.bzip2` is similar to `164.gzip`; however, the amount of computation in `256.bzip2` is much more resulting in longer execution time and much lower bandwidth. This explains the vast difference in their performance improvements.

Figure 5(a) also explains the plateauing of the speedups of `052.alvinn` and `197.parser`. In both programs, as the number of threads increases, the application bandwidth increases much faster when compared to the other programs. At a large number of threads, the bandwidth requirements limit the speedup.

Although Spec-DSWP is communication latency tolerant, it is sensitive to the overhead of the operations required to send a datum [25]. Since a single invocation of a send or receive function can take as many as 2,295 instructions in OpenMPI [8], DSMTX coalesces multiple data transfer requests, thereby amortizing the costs. Communication using DSMTX queues can sustain a bandwidth of 480.7 MBps, whereas communication using `MPI_Send`, `MPI_Bsend`, or `MPI_Isend` directly provides 13.1, 12.7 and 8.1 MBps of bandwidth respectively. The increased bandwidth boosts program performance as shown in Figure 5(b). In `052.alvinn`, `164.gzip` and `256.bzip2`, array data is already explicitly produced in the form of chunks. Consequently, they do not benefit from this optimization.

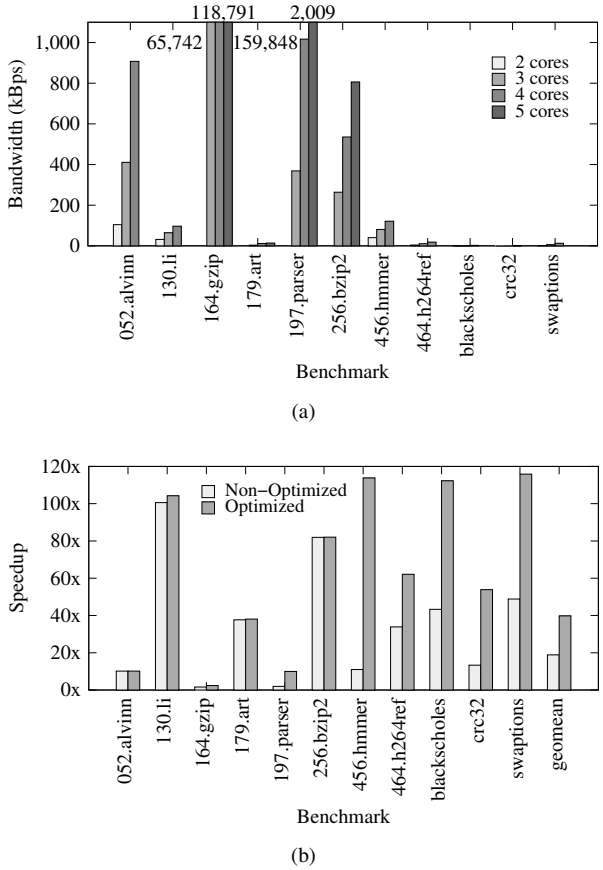


Figure 5: (a) Bandwidth requirement for each application. As more cores are used, the required communication bandwidth increases. Some programs have very high bandwidth requirements. Both factors limit performance scalability; (b) Effect of communication optimization on 128 cores. Batched communication in DSMTX yields much better speedup compared to communication using `MPI_Send` and `MPI_Recv` directly.

5.4. Recovery Overhead

To analyze the overhead of recovery from misspeculation, inputs to each benchmark are modified to cause misspeculation at a rate of 0.1%. `052.alvinn`, `164.gzip`, `179.art`, `456.hmmer` and `464.h264ref` are not evaluated because they do not have input-dependent misspeculation.

Figure 6 illustrates the overhead due to recovery from misspeculation. Recovery is done at multiple thread counts to illustrate overhead trends. Each full bar shows speedup when there is no misspeculation. **`MIS`** shows the speedup when iterations misspeculate at a rate of 0.1%. The difference

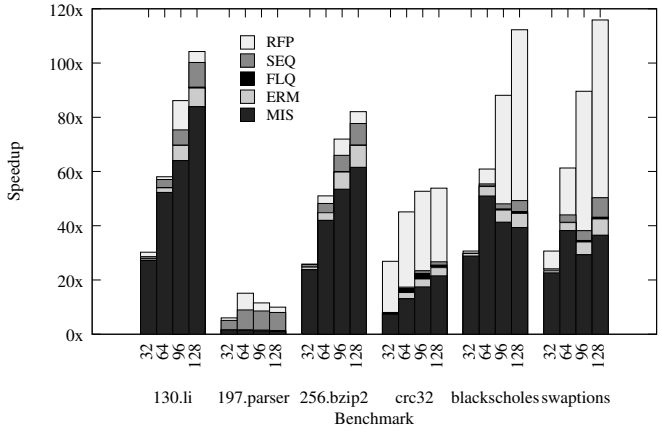


Figure 6: Recovery overhead analysis with a misspeculation rate of 0.1%. (RFP: ReFill Pipeline, SEQ: SEQuential execution, FLQ: FLush Queue, ERM: Enter Recovery Mode, MIS: speedup with MISspeculation)

between the two bars is the recovery overhead which may be broken down as follows. **ERM** is the time taken to synchronize workers before recovery starts; **FLQ** is the time taken to flush the communication queues and reinstall page protections; **SEQ** is the time taken to re-execute the misspeculated iteration, and **RFP** is the time taken to fill the pipeline again. The `RFP` phase has the highest overhead. Since DSMTX processes iterations in order, it squashes all iterations higher than the one in which misspeculation occurred causing the work done in the later iterations to go to waste. This process empties the pipeline after misspeculation. `RFP` accounts for the time to refill the pipeline again. Interestingly, the communication optimization described in Section 5.3 may cause many iterations worth of work to be wasted. This is the reason for the high `RFP` overhead in `197.parser`, `crc32`, `blackscholes` and `swaptions`. Reducing the communication batch size can help reduce `RFP` overhead, but it may degrade performance during normal execution.

6. Related Work

Distributed Software Transactional Memory (DSTM): Distributed software transactional memory systems [5, 10, 16, 19] aim to support transactional execution on systems without shared memory. None of these systems implement MTX semantics. DSMTX extends them with MTX to support Spec-DSWP, and this is the main difference between DSMTX and other distributed software transactional memory systems.

Distributed Multiversioning (DMV) [19] modifies a software distributed shared memory system (SDSM) to support transactions. DMV and DSMTX expose a unified virtual address space to programs. DMV performs transaction validation and commit at the page granularity by means of page diffing. This static batching of spatially adjacent words may result in unnecessary diffs and excessive communication for memory access patterns that access pages in a sparse fashion. DSMTX eliminates this problem by performing transactional operations at the word granularity and batches up the words according to dynamic access patterns.

Cluster-STM [5] is an STM system for large-scale clusters. Cluster-STM introduces new memory allocation and deallocation functions such as `stm_alloc`, `stm_all_alloc`, and `stm_free`, and forces programmers to use these functions. By contrast, DSMTX overrides the default `malloc` and `free` functions, which make program modification unnecessary. Like Cluster-STM, DSMTX exposes a globally shared address space to programs across the distributed memory system.

DiSTM [16] is a DSTM system which builds on Java Remote Method Invocation (RMI). DiSTM detects and resolves conflicts at object granularity. In DiSTM, the main node keeps the committed state of a program, and worker nodes execute transactions using private cached data. DSMTX has a commit unit that keeps committed data, and workers which execute MTXs in their private physical memories. Although DiSTM allows parallel commits using the multiple leases protocol, the workers are tightly coupled through the validation/commit process because a worker cannot start the next transaction until the current transaction commits. By contrast, DSMTX decouples transaction execution from validation/commit, allowing a worker to start a new transaction before the commit process of the current one is completed.

With the exception of Cluster-STM [5], these systems have not been evaluated and their scalability has not been demonstrated on platforms with over 100 cores. DSMTX allows more applications to be parallelized with robust and scalable performance under high inter-core communication latency by enabling speculative and pipelined execution. Unlike these proposals, DSMTX moves the validation/commit operations into separate stages, allowing fast execution of the critical path.

Partitioned Global Address Space (PGAS) and Software Distributed Shared Memory (SDSM): The Unified Virtual Address (UVA) provided by DSMTX is influenced by PGAS [9, 13, 18] and SDSM [2]. Like PGAS and SDSM, UVA provides a unified virtual address space to all threads. Like PGAS, UVA partitions the address space into several non-overlapping regions each of which is associated with a different worker. However, while PGAS is a language-based approach to concurrency in distributed systems, UVA is a library-based approach which does not require code modifications. Compared to SDSM, UVA is customized and optimized to support DSMTX by selectively supporting coherence through Copy-On-Access.

Pipeline Parallelism on Distributed Systems: The three-stage parallelization paradigm of some benchmarks resembles Google MapReduce [12]. The first stage dispatches jobs to threads in the second parallel stage ("map"), which send results to the third stage ("reduce"). MapReduce is a programming model that requires a program to be rewritten as a sequence of map and reduce operations. DSMTX enables multiple parallelization paradigms and allows the most well-performing one to be applied to a program, rather than enforcing a single paradigm as MapReduce does.

7. Conclusion

This paper proposes Distributed Software Multi-threaded Transactional memory (DSMTX), a software runtime system that enables both the DSWP family of parallelization techniques (Spec-DSWP) and the thread-level speculation (TLS) techniques on both shared memory systems and message-passing systems. DSMTX is implemented on a commodity cluster with 32 nodes (128 cores), and 11 applications from the SPEC CPU benchmark suites [26] and PARSEC benchmark suite [4] are parallelized using Spec-DSWP and TLS. DSMTX achieves a geomean speedup of 49×.

DSMTX may also be useful for emerging manycore architectures that discard chip-wide cache coherence [14]. These architectures offer challenges similar to those found in clusters. DSMTX can add value to these platforms by enabling scalable parallelization of a variety of codes.

Acknowledgment

We thank the Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. The evaluation presented in this paper were performed on computational resources supported by the PICSciE-OIT High Performance Computing Center and Visualization Laboratory. This material is based upon work supported by the National Science Foundation under Grant No. CCF-0811580 and OCI-0849512, and the Air Force Research Laboratory.

References

[1] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[3] D. I. August, N. Vachharajani, and M. J. Bridges. *System and method for supporting multi-threaded transactions*. United States Patent Application 12/380677. March 2008.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[5] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[6] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[7] M. J. Bridges. *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Codes*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[8] D. Buntinas, G. Mercier, and W. Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the nemesis communication subsystem. *Parallel Computing, North-Holland*, 33(9):634–644, 2007.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2005.

[10] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. *Pacific Rim International Symposium on Dependable Computing, IEEE*, 0:307–313, 2009.

[11] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, August 1986.

[12] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[13] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. John Wiley and Sons, 2005.

[14] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 7-11 2010.

[15] J. Huang, A. Raman, Y. Zhang, T. B. Jablin, T.-H. Hung, and D. I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization*, April 2010.

[16] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, 2008.

[17] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[18] K. Y. Luigi, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. pages 10–11, 1998.

[19] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006.

[20] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[21] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE '08: Proceedings of the 1st International Workshop on Multicore Software Engineering*, 2008.

[22] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO '05: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[23] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, CA, 4th edition, 2008.

[24] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative Parallelization Using Software Multi-threaded Transactions. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010.

[25] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. N. Cai. Support for high-frequency streaming in CMPs. In *Proceedings of the 39th International Symposium on Microarchitecture*, December 2006.

[26] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[27] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, February 2005.

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

[29] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[30] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO '08: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[31] N. Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[32] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[33] R. M. Yoo and H.-H. S. Lee. Helper transactions: Enabling thread-level speculation via a transactional memory system. In *PESPMA '08: Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, June 2008.

[34] A. Zhai. *Compiler Optimization of Value Communication for Thread-Level Speculation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, United States, January 2005.

[35] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *The 5TH International Symposium on High-Performance Computer Architecture*, February 1999.

[36] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA '08: Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

[37] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *MICRO '02: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, 2002.