Speculative Parallelization Using Software Multi-threaded Transactions

Arun Raman Hanjun Kim Thomas R. Mason Thomas B. Jablin David I. August

Departments of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, USA

{rarun,hanjunk,tmason,tjablin,august}@princeton.edu

Abstract

With the right techniques, multicore architectures may be able to continue the exponential performance trend that elevated the performance of applications of all types for decades. While many scientific programs can be parallelized without speculative techniques, speculative parallelism appears to be the key to continuing this trend for general-purpose applications. Recently-proposed code parallelization techniques, such as those by Bridges et al. and by Thies et al., demonstrate scalable performance on multiple cores by using speculation to divide code into atomic units (transactions) that span multiple threads in order to expose data parallelism. Unfortunately, most software and hardware Thread-Level Speculation (TLS) memory systems and transactional memories are not sufficient because they only support single-threaded atomic units. Multi-threaded Transactions (MTXs) address this problem, but they require expensive hardware support as currently proposed in the literature. This paper proposes a Software MTX (SMTX) system that captures the *applicability* and *performance* of hardware MTX, but on *existing multicore machines*. The SMTX system yields a harmonic mean speedup of 13.36x on *native* hardware with four 6-core processors (24 cores in total) running speculatively parallelized applications.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Algorithms, Design, Languages, Performance

Keywords automatic parallelization, loop-level parallelism, multi-threaded transactions, pipelined parallelism, software transactional memory, thread-level speculation

1. Introduction

Until recently, the computing industry relied on clock frequency scaling and uniprocessor microarchitectural enhancements to improve the performance of a wide range of applications. Unfortunately, power and thermal issues have rendered this approach infeasible.

Meanwhile, Moore's law continues to double the number of transistors per unit area. Processor designers leverage these additional transistors by placing multiple cores on the same die. To a first-order approximation, *only* multi-threaded applications will enjoy a performance boost.

Consequently, producing parallel programs is the primary concern of the multicore era. Scientific applications generally have regular control flow and memory access patterns. Often, they can be parallelized non-speculatively using DOALL (execute loop iterations independently of each other) and DOACROSS (execute loop iterations in parallel and synchronize the dependences to ensure that later iterations get the correct values from earlier ones) [3]. In contrast to scientific applications, general-purpose programs generally contain irregular dependences that manifest infrequently or statically-unresolvable dependences that may not manifest at runtime. Removing these dependences speculatively can dramatically improve parallelization. Of course, if a speculatively-removed dependence manifests at runtime, the program must undo the effects of speculative execution, but this is rare if good speculation decisions are made.

To provide improved levels of coarse-grained loop-level parallelism, Bridges et al. [7] and Thies et al. [29] use pipeline transformations such as Decoupled Software Pipelining (DSWP) [21], enhanced with speculation and replication of stages without any cross-iteration dependences. Instead of executing each iteration in a different thread as in DOALL and DOACROSS, DSWP executes parts of the loop body in parallel, with dependences flowing unidirectionally in a pipelined fashion. Bridges et al. and Thies et al. demonstrate scalable performance on many cores by dividing atomic units (loop iterations) across multiple threads and speculating across those units to expose data parallelism.

Combining speculation with pipelining in this manner requires a notion of *Multi-threaded Transactions* (MTXs), so that speculative work done in different pipeline stages (by different threads) can be committed together. Most Thread-Level Speculation (TLS) memory systems and transactional memory systems do not support this paradigm because they guarantee only single-threaded atomicity [11, 31]. Vachharajani introduced a *hardware* memory system that can support MTXs [31]. The proposed implementation requires changes to the cache coherence protocol in order to buffer speculative state and recover from misspeculation.

The primary contribution of this paper is the introduction of the notion of a Software Multi-threaded Transaction (SMTX) to the literature. SMTXs are efficiently executed by a novel software runtime system that uses memory versioning. The SMTX system maintains speculative state and non-speculative state separately using process-separation, with non-speculative state owned by a *commit* process. This allows all the processes to share the same view of the virtual address space while the virtual memory system transparently manages the privatization of memory with Copy-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

Copyright © 2010 ACM 978-1-60558-839-1/10/03. . . $10.00

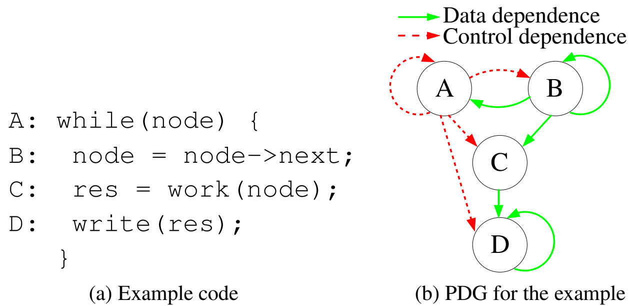

Figure 1: Running example: The code in (a) has the dependence pattern shown in (b), and is amenable to both DOACROSS and DSWP parallelizations.

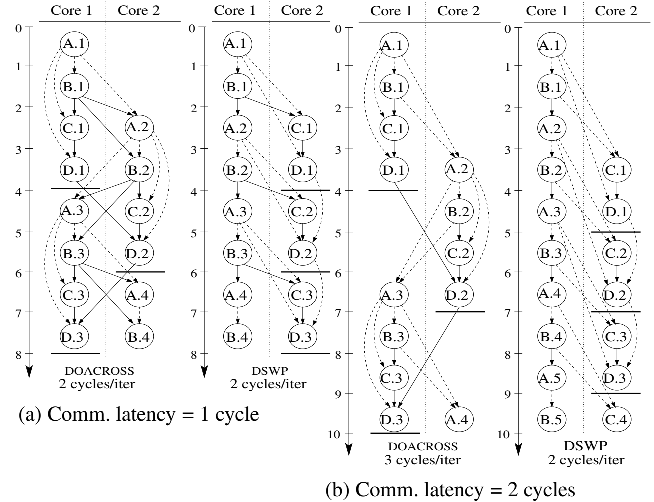

Figure 2: Advantage of pipelining: Pipelined parallelism keeps critical-path dependences thread-local and communication unidirectional; thus it is tolerant to increase in communication latency.

On-Write semantics. To recover from misspeculation, the SMTX system remaps the virtual address space of the worker threads to the committed memory state.

Section 2 motivates the need for SMTXs. Section 3 discusses the design and implementation of the SMTX system. Section 4 demonstrates how this system can be used for speculative parallelization. It also presents a value-based conflict detection algorithm for loop parallelizations. Section 5 presents the results of parallelization and the overheads of the system. Section 6 discusses the advantages of the SMTX system over cutting-edge software techniques that extract parallelism speculatively from general-purpose applications [10, 18, 30]. Section 7 concludes the paper.

2. Motivation

Scientific and numerical applications typically consist of counted loops that manipulate regular structures, accesses to which can be precisely analyzed statically. Techniques such as DOALL and DOACROSS can be used to good effect in these domains [3]. DOALL partitions the iteration space into groups that are executed concurrently with no inter-thread communication. DOALL often results in speedup that is proportional to the number of threads; however, it is inapplicable when the loop has cross-iteration dependences. Consider the example code in Figure 1(a). As the Program Dependence Graph (PDG) in Figure 1(b) shows, there are cross-iteration dependences that inhibit DOALL. However, both DOACROSS and DSWP [21] are applicable, and their respective schedules are shown in Figure 2(a). Each node represents a dynamic instance (as indicated by the iteration number following the dot) of a statement in Figure 1(a). DOACROSS schedules the entire loop body iteration by iteration on alternate threads. DSWP partitions the loop body across the execution threads, and each thread is responsible for all iterations of its piece of the loop body. As Figure 2(a) shows, ignoring the pipeline fill time, both DOACROSS and DSWP yield a speedup of 2x using 2 threads at steady state.

As a result of the manner in which DOACROSS schedules the iterations, cross-iterations dependences that form recurrences (strongly-connected components in the PDG) must be communicated from thread to thread. This puts the inter-thread (and potentially inter-core) communication latency on the critical path. In contrast, by keeping dependence recurrences thread-local, DSWP is tolerant to increase in inter-thread communication latency. So, when the inter-core latency is increased to 2 cycles as in Figure 2(b), the speedup with DOACROSS reduces to 1.33x whereas the speedup with DSWP remains 2x at steady state.

While DOACROSS and DSWP speed up many programs, most general-purpose programs are characterized by complicated control flow and irregular memory access patterns. For example, assume that statement C in Figure 1(a) may modify the linked-list. This introduces cross-iteration data dependences through memory from statement C to A and B. Respecting all the potential dependences severely limits the achievable speedup: DOACROSS suffers because the iterations are essentially serialized, while DSWP suffers because three of the four statements now participate in a dependence recurrence thereby skewing the pipeline balance.

The compiler is conservative either because it respects an infrequent dependence or because it cannot determine that a dependence does not actually exist. Either way, speculation allows the compiler to overcome these limitations. Most speculative parallelization proposals are iteration-centric in that they try to break dependences that go around the loop's back-edge. Such dependence edges result in cycles in the PDG as shown in Figure 1(b).

One approach called Spec-DOALL breaks all the cycles using speculation. Speculating all loop-carried dependences typically results in the misspeculation of many iterations since it is not easy to correctly predict all the edges. Another approach called Spec-DOACROSS breaks only *some* of the cycles using speculation, while other cycles are respected and the corresponding dependences are synchronized as in DOACROSS. However, as in DOACROSS, this strategy puts the inter-core communication latency on the critical path thus negating most parallelism benefits on multicore architectures which have non-unit communication latency. Together, these schemes are called Thread-Level Speculation (TLS) [5, 24, 28, 34]. A third approach called Spec-DSWP also breaks only some cycles. In contrast to Spec-DOALL, this approach selectively breaks those cycles that can be broken with high confidence, thereby resulting in higher success rates. In contrast to Spec-DOACROSS, this approach selectively allows those cycles that contain hard-to-predict edges to remain thread-local, and is thus not penalized by inter-core communication latency [7, 29, 32].

The memory dependences from C to A and B, and the control dependences from A to itself, B, C, and D are easy to predict and can be speculated (Figure 3(a)). To apply Spec-DOALL, all cycles must be broken; so, in addition to dependences that are easily predicted, those that are hard to predict such as the self-dependences of statements B and D must be speculated. Predicting the values of `node` on each iteration is very difficult in general. Ap-

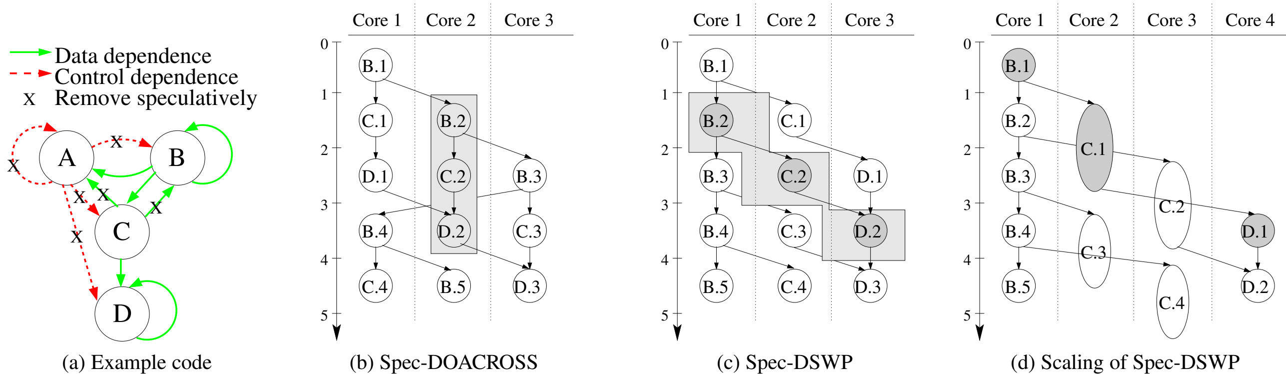

Figure 3: The need for Multi-threaded Transactions (MTXs): With several dependences speculatively removed (a), the loop in Figure 1(a) can be parallelized with Spec-DOACROSS (b) and with Spec-DSWP (c). While an atomic unit (shown as a lightly-shaded region) is confined to a single thread in (b), it spans multiple threads in (c). The darkly-shaded regions correspond to sub-transactions in an MTX. Spec-DSWP can help replicate stages without any cross-iteration dependences to use additional cores (d).

plying Spec-DOALL can lead to high misspeculation rate and poor performance. Applying Spec-DOACROSS would synchronize the hard-to-predict dependences as in Figure 3(b). However, the resulting cyclic communication pattern adversely affects performance. In contrast, Spec-DSWP keeps the dependence cycles thread-local as in Figure 3(c). Put another way, Spec-DSWP has the freedom to either break a dependence cycle or keep it thread-local. It breaks cycles to create more pipeline stages (for example, the B-C cycle), and also to create stages with no cross-iteration dependences (with respect to the parallelized loop) so that such stages (that dominate execution time) may be replicated as in Figure 3(d) (this is like applying Spec-DOALL to a part of the loop body).

In TLS schemes, a loop iteration is the unit of atomicity; in other words, each iteration is executed in a transaction. A traditional three-thread Spec-DOACROSS parallelization would schedule the loop iterations as shown in Figure 3(b) (Statement A is transformed to `while(true)` and is not shown). Observe that each transaction (shown as a lightly-shaded box) is executed in a single thread. Consequently, conventional transactional semantics that guarantee single-threaded atomicity suffice. Similar to TLS techniques, Spec-DSWP also uses loop iterations as the unit of atomic work. In order to handle misspeculation, the system conceptually checkpoints the state of the single-threaded loop on each iteration. However, because the unit of atomicity is still the iteration, the atomic unit gets split up across multiple threads as shown by the lightly-shaded region in Figure 3(c). Thus, conventional TLS epochs or transactions are insufficient to execute these multi-threaded atomic units.

Multi-threaded Transactions (MTXs) can be used to support these atomic units. Conceptually, MTXs provide the illusion of a private memory for the threads participating inside these transactions. An MTX may contain many sub-transactions (subTXs). (Note that an MTX with only one subTX is equivalent to a single-threaded transaction or TLS epoch.) Only one thread executes each subTX. Figure 3(c) shows how each stage of a loop iteration can execute in a subTX *within* an MTX (shown as darkly-shaded circles in the figure). The stores by earlier subTXs are visible in later subTXs within the same MTX. This allows for synchronization of these subTXs, thus preventing intra-MTX misspeculation. Vachharajani proposed specialized hardware to support MTXs. It includes a new invalidation-based cache coherence protocol and a set of point-to-point queues [31]. This motivates an investigation of whether special hardware is *necessary* to efficiently support MTX-enabled speculative parallelization.

3. Design & Implementation

The paper describes a *software* system that generalizes existing software TLS memory systems to support speculative pipelining schemes, and is efficiently tuned for loop parallelization. Conceptually, an MTX provides a private memory (or *memory version*) for the threads participating in the MTX. This memory is initialized with the contents of committed memory at the time of creation of the MTX. Loads and stores interact with this *memory version* (Section 3.1). At the end of the MTX, if no conflicts are detected (Section 3.2), this version of memory becomes the committed version; otherwise, the MTX is rolled back (Section 3.3).

3.1 Atomicity and Isolation / Memory Versioning

To support simultaneous execution of multiple MTXs, the SMTX system must be able to create multiple versions of memory. There are two approaches. First, in-place or eager versioning directly updates the shared memory after locking the memory location, and stores the old value in an *undo-buffer*. Second, buffered or lazy versioning buffers speculative writes in a *write-buffer*, and updates the shared memory on successful commit [16]. In both cases, writing to shared memory involves acquiring locks on the locations being written to so that program state remains consistent. This can add considerable overhead. In contrast to distributed update of speculative state, if the non-speculative state is *owned* by a *commit unit*, then there is no need to lock the memory locations thereby eliminating the overhead. Section 5 shows that, for the number of threads studied, this centralization does not cause scalability problems. Therefore, the proposed SMTX system uses lazy versioning with a centralized commit unit. Figure 4 presents the commit unit's algorithm. In this model, each transaction writes to a write-buffer that is part of the transaction's private memory. Once the transaction is known to be free of conflicts (Figure 4: line 4), the *commit-unit* replays the writes in the transaction's write-buffer in its own committed memory (Figure 4: line 7).

While the locking overhead of transactional writes is eliminated in this model, a new overhead is imposed on transactional reads. All loads must read from the transaction's private memory. Consequently, the write-buffer must now be checked each time a transactional read happens in order to determine whether the transaction has written to that location in the past. This overhead can be significant particularly for long-running transactions such as outer-loop iterations [9]. This write-buffer scan cannot be eliminated in thread-based approaches that share the logical and physical address

```
1  commitUnit () {
2    version = 0;
3    while (TRUE) {
4      status = CONFLICT_DETECTOR(version);
5      switch (status) {
6        case NO_CONFLICT:
7          COMMIT_ALL_WRITES(version); break;
8        case CONFLICT:
9          DISCARD_ALL_WRITES(version);
10         REEXECUTE_MTX();
11     }
12     version++;
13   }
14 }
```

Figure 4: Commit Unit Algorithm

spaces. However, if the transactions are executed inside UNIX processes, they can rely on the underlying virtual memory system to transparently create a private physical copy (at the page granularity) of the memory location of the transactional write. A load from the same location, that happens later on in the transaction, is ordinary in the sense that there is no need to scan a write-buffer.

While the above suffices for single-threaded transactions (TLS epochs), it is insufficient to support multi-threaded transactions (MTXs). This is because stores executed speculatively by an earlier subTX must be visible to later subTXs within the same MTX (this allows the subTXs to execute cooperatively). In other words, all subTXs within an MTX must access the *same* version of memory. Vachharajani calls this *uncommitted value forwarding* [31]. To implement this, the hardware MTX system augments each cache line with a version ID (VID) and buffers speculative state in the cache. A read request carries a VID, and the hardware is responsible for returning the value from the appropriate cache line. In the SMTX system, prior to executing "real work", each subTX replays the stores in the write-buffers of earlier subTXs, thus updating its view of memory and effectively entering the same memory version as its predecessor subTXs.

In addition to *uncommitted value forwarding*, stores emanating from all the subTXs inside an MTX must appear to execute atomically. This is called *group transaction commit* [31]. As described earlier, stores inside the subTXs are forwarded over the write-buffers to the commit unit. If the commit unit determines that the MTX does not conflict with other MTXs, all the stores in the write-buffers are executed in the non-speculative memory state in order of subTX (subTXs are ordered a priori); otherwise, all of them are discarded (Figure 4: line 9). This guarantees multi-threaded atomicity.

3.2 Conflict Detection

Conflicts between MTXs arise when they access the same location and one of them writes to it. Conflict detection may be performed on each transactional load and store (*eager* conflict detection) or at commit time (*lazy* conflict detection). As mentioned earlier, the success of speculative loop parallelization depends on minimizing the penalty on load operations. So, conflict detection is decoupled from a transaction's execution and is done lazily at commit time by a *try-commit unit*.

Any software conflict detection mechanism can be used with the SMTX runtime system. Conflict detection based on mirroring the cache-coherence protocol [25], using load vectors and version numbers to detect flow dependences while committing writes [19], comparing read and write signatures to determine whether they overlap [18], etc. may all be used in conjunction with the SMTX

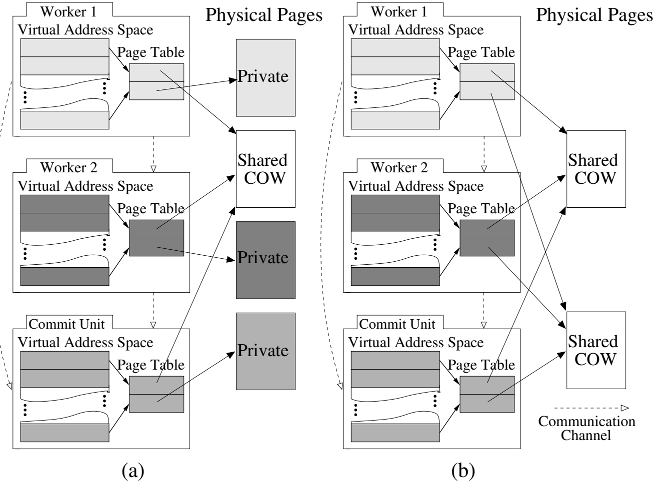

Figure 5: (a) SMTX system design: The page tables map the same virtual address range to either private or shared physical pages. (b) After recovery, the page tables of all the worker threads point to the non-speculative pages which are marked as COW.

system. Indeed, the right granularity/precision of tracking varies, often within the same application [2, 19]. Thus, it appears to be a good idea to leave the choice of dependence checking mechanisms to the client (programmer and smart compiler) of the SMTX system. Section 4 describes conflict detection mechanisms optimized for loop parallelization of the target applications using SMTX.

3.3 Conflict Resolution / Rollback

If a conflict is detected, the commit unit flushes the entries in the write-buffers of the MTXs that are to be rolled back, and then restarts the MTXs on the worker threads with a copy of its own committed version of memory. Since the focus is on loop parallelization, the total ordering determined by the chronological order of the loop iterations in the program's non-speculative sequential execution dictates which MTXs will be rolled back.

3.4 Implementation

Figure 5(a) illustrates the SMTX design with two speculative worker threads and the commit unit. Initially, the workers and the commit process share all the physical pages in memory by virtue of identical but separate page tables. However, the workers' pages are marked as Copy-On-Write (COW). The state of the system shown in the figure is such that each worker has written to parts of memory resulting in private physical copies of the *same* virtual address range. The COW semantics ensure that only the modified pages are copied. Other pages will remain shared (as shown). As mentioned earlier, speculative writes are buffered in write-buffers. These and other messages are communicated through highly optimized single-producer/single-consumer lock-free queues in shared memory. The queue structure preserves the ordering of the events occurring in the workers, and allows workers executing later subTXs to update their memory while the earlier subTX is executing. The queue implementation is similar to the FastForward queue developed by Giacomoni et al. [12]. Communication channels exist between each worker and the commit unit. For uncommitted value forwarding, communication channels are created between *only* those workers that may execute subTXs of an MTX. The parallelization strategy described later in Section 4 ensures that the number of communication channels is typically linear, not quadratic in the number of worker threads.

When misspeculation (inter-transaction conflict) is detected, the commit process is notified of this condition. It then takes the following actions:

1. The relevant pages are marked as COW.
2. For each worker, the page table entries of the commit process are copied into the worker's page tables.
3. The TLB of the core on which the worker is executing is flushed so that stale virtual-to-physical address mappings are not reused.

Loads in subsequently executed MTXs will transparently read committed, non-speculative values. Figure 5(b) illustrates this memory rollback operation. Observe that rollback does not involve any costly application-data copying operations. The speculative data is simply *discarded*. Comparing with Figure 5(a), the page tables have been remapped and the private pages have been discarded.

4. Loop Parallelization With SMTX

The SMTX system developed in the preceding section is implemented as a library. Table 1 presents the interface in two sections. The first section consists of one-time setup and finalization primitives. The second section consists of primitives that are invoked during the execution of SMTXs. The library may be used in conjunction with many parallelization schemes; its use with one scheme is described below.

4.1 Parallelization Scheme

Recent work has shown that there is significant loop-level parallelism in the outermost loops of applications [7, 29, 30, 33]. In most cases, the loop body is decomposed into three *phases*; dynamic instances of each phase are called *tasks*. Each phase exhibits a different dependence pattern (ignoring the speculated dependences). The first phase depends only on prior tasks of the same phase. The second phase depends only on the corresponding task of the first phase. The third phase depends on the corresponding task of the second phase *and* prior tasks of itself. Such a partitioning of the loop body is shown in Figure 3(a); statements B, C, and D constitute the three phases.

Bridges et al. and Thies et al. present a parallelization scheme which incorporates pipelining into the basic loop partitioning discussed above [7, 29]. By inserting decoupling buffers in between the stages of the pipeline, the different stages could be executing simultaneously on different iterations. This technique is called Decoupled Software Pipelining (DSWP)[21]. This base technique is extended with speculation to break dependence-recurrences to create a long pipeline in Spec-DSWP [32]. Speculation is further used to break inter-iteration dependences to expose data-level parallelism, allowing the replication of pipeline stages with no loop-carried dependences. This is called Speculative Parallel-Stage DSWP (Spec-PS-DSWP)[8]. Figure 6 shows this execution model. As described in Section 2, this pipelined parallelization has benefits over TLS techniques. Thus, Spec-PS-DSWP is used as the general parallelization strategy. Note that if the pipeline consists of just one stage and it is a speculatively parallel (DOALL) stage, the execution model degenerates to Speculative DOALL [18, 33].

Loop parallelization is done as follows: The speculative part of the loop is wrapped in an MTX, with each iteration split into multiple subTXs (shown in Figure 6 as similarly shaded ovals). These subTXs are executed by the stages of the Spec-PS-DSWP pipeline. To orchestrate speculative execution, Spec-PS-DSWP relies on a commit stage (Figure 6, Commit Stage). The functionality needed in the commit stage is already encapsulated by the commit unit's functionality in the SMTX system described in the previous section.

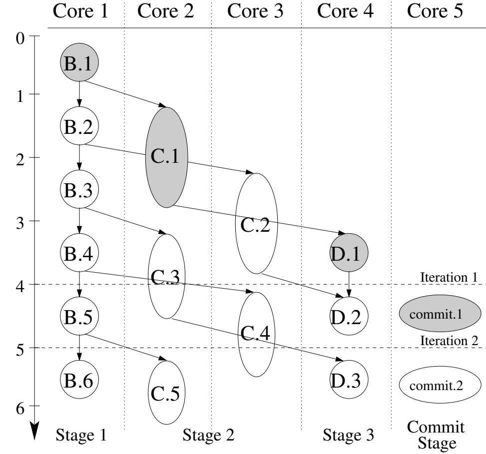

Figure 6: Pipelined Execution with Stage Replication: A dashed horizontal line indicates the time at which all subTXs in an MTX have finished and the MTX is ready to commit.

```
 1 status CONFLICT_DETECTOR(version) {
 2   for (each worker that entered this version) {
 3     do {
 4       status = ver_checkForConflict(worker);
 5     } while (status == CONTINUE);
 6     if (status == MISSPEC)
 7       return CONFLICT;
 8   }
 9   return NO_CONFLICT;
10 }
11
12 status ver_checkForConflict (worker) {
13   token = sq_consume(q[worker]);
14   value = sq_consume(q[worker]);
15   if (token) {
16     addr = mask1(token);
17     isRead = mask2(token);
18     if (isRead) {
19       if (*addr != value) return MISSPEC;
20     } else {
21       *addr = value;
22     }
23     return CONTINUE;
24   }
25   return BREAK;
26 }
```

Figure 7: A value-based conflict detector is used to detect memory conflicts between transactions.

4.2 Conflict Detection

In the context of loops, there are flow, anti, or output dependences between the loop iterations. The use of memory versioning ensures that MTXs write to different physical versions of the same logical memory address. This removes all false (anti and output) dependences without code transformations.

Loads and stores that may alias forward the ⟨`addr`, `value`⟩ tuple by calling `ver_read` and `ver_writeTo` (Table 1). The conflict detector does a sequential replay of this memory trace in its private memory to detect whether flow-dependence violations have occurred (Figure 7: lines 2-8). For example, let a store operation in MTX_1 store value v_1 in location l, and a load operation in MTX_2

Operation	Description
One-time Operations	
`system = ver_newSMTXsystem(n, configuration)`	Initialize `system` of `n` threads with the given pipeline `configuration`; create queues etc.
`ver_deleteSMTXsystem(system)`	Finalize `system`; delete various data structures
`ver_spawn(function, tid, argument)`	Spawn a new worker with thread id `tid` that will execute `function` with the provided `argument`
`ver_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 non-speculatively execute the misspeculating iteration
`ver_tryCommitUnit(tid, arg)`	Encapsulates the conflict detection mechanism that is executed by a try-commit unit
Running Operations for Workers	
`sq_produce(queue, value)`	Enqueue `value` in specified `queue`; block if `queue` is full.
`value = sq_consume(queue)`	Dequeue and return `value`; block if `queue` is empty.
`state = ver_begin(tid)`	Enter a new *version* by updating memory with stores in this *version* by workers that modified this *version* earlier; notify commit unit that a new *version* has been entered; returns the `state` of the system to check for misspeculation or termination
`state = ver_end(tid)`	End the current *version* and notify later stages including the commit unit of the same; returns the `state` of the system to check for misspeculation or termination
`ver_writeTo(tid, dest, addr, value)`	Forward an $\langle$`addr, value`$\rangle$ tuple to the specified `destination`
`ver_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
`ver_read(tid, addr, value)`	Forward an $\langle$`addr, value`$\rangle$ tuple to the try-commit unit
`ver_misspec(tid)`	Notify the commit unit of misspeculation
`ver_terminate(tid)`	Notify the commit unit of termination of the parallel region
`ver_doRecovery(tid)`	Handle recovery from misspeculation

Table 1: SMTX Runtime Library Interface For Loop Parallelization By Programmer and Optimizing Compiler

load value v_2 from location l concurrently. Since the conflict detector executes loads and stores sequentially, it first writes value v_1 to location l in its own physical copy of the virtual address space (Figure 7: line 21). When it encounters the later (in terms of original loop sequentiality) load operation, it compares the value that was loaded speculatively (v_2) with the value in its memory (v_1) in the same location l (Figure 7: line 19). If they are different, then misspeculation has occurred since the dependence was not respected; otherwise, speculation has succeeded. Value-based checking subsumes multiple load-store alias checks. Speculation will succeed so long as the load operation reads the correct value; the value may have been written by any store operation.

The commit unit cannot be used to perform this checking because it involves updating memory with the values of speculative writes. If a conflict is detected midway into a transaction's checks, then the earlier updates to memory cannot be undone, and atomicity cannot be guaranteed. Thus, a separate try-commit unit executing inside a UNIX process is used to do the conflict checking. Referring back to Figure 4, the call of CONFLICT_DETECTOR on line 4 blocks until the try-commit unit returns the status of the MTX.

This decoupling of conflict detection, transaction execution, and transaction commit allows the try-commit unit to do conflict detection *in parallel* with the speculative workers which could be executing other MTXs, and also *in parallel* with the commit unit which could be committing the writes by earlier MTXs that have been deemed conflict-free by the try-commit unit.

In many cases, a simpler variant of this general checking scheme may be employed. A load L is predicted to read the same value at the *beginning* of each iteration. At the *end* of each iteration, when the correct value has been computed, the memory location is checked to ensure that the speculation is correct. This scheme is very useful in cases of loads from data structures that may be modified heavily during an iteration but are reset to the value that they had at the beginning of the iteration. This kind of speculation has been used in other systems [8, 10, 20].

4.3 Putting It All Together

Figure 8 shows how the single-threaded loop in Figure 3(a) is transformed into a multi-threaded one using the SMTX library. The loop body is divided into three stages. The statements corresponding to the original loop are shaded in Figure 8. Two threads are needed to execute `stage 1` and `stage 3` (Figure 8(d): lines 3, 4), and two more are needed for the commit unit and the try-commit unit (Figure 8(d): lines 8, 9). The remaining threads are assigned to the parallel stage (`stage 2`) that does the bulk of the work (Figure 8(d): lines 5-7). This partitioning is informed by the dependence pattern exhibited by the loop statements. `ver_begin` and `ver_end` are used to enter and leave memory versions. As mentioned in Table 1, `ver_begin` makes the executor enter a new `version` of memory: this is done by updating memory with values forwarded by workers that have entered and left that `version` before. `ver_end` notifies other workers that the executor has finished all its updates to the current `version` of memory, and that others may now enter that `version`. All dependences that are removed speculatively (see Figure 3 (a)) must be checked for manifestation. The statement on line 8 in Figure 8 (a) checks the control dependences. Reads and writes are instrumented and forwarded to the try-commit unit for memory conflict detection. The instrumentation must be done interprocedurally. For example, since `work` may update `node`, the write operation(s) inside `work` must be instrumented. Figure 8(e) shows the additions to the algorithm presented in Figure 4. After committing a version, `commit_fun` is executed (Figure 8(e): line 9). Typically, `commit_fun` performs some I/O operations that presently cannot be performed inside speculative regions. Upon misspeculation, `recovery_fun` is executed (Figure 8(e): line 12) by the commit stage. This is a single-threaded execution of the loop body that *respects all dependences*, using the non-speculative program state. Following this, speculative execution resumes with the *new*, non-speculative program state. The `ver_terminate` call on line 9 in Figure 8 (a) informs all threads of loop termination. At this point, there are two cases possible. First, all MTXs commit successfully, allowing the worker executing `stage1` to exit. Second, an MTX in

```
1  void stage1 (tid, arg){
2    version = 0;
3    while(TRUE){
4      if (ver_begin(tid))
5      {ver_doRecovery(tid);continue;}
6      ver_read(tid, &node−>next, node−>next);
7      node = node−>next;
8      if (!node) {
9        ver_terminate(tid);
10       ver_doRecovery(tid); continue;
11     }
12     dest = 1 + (version % np);
13     ver_writeTo(tid, dest, &node, node);
14     ver_writeTo(tid, try_commit_unit, &node, node);
15     ver_writeTo(tid, commit_unit, &node, node);
16     if (ver_end(tid))
17     {ver_doRecovery(tid);continue;}
18     version++;
19   }
20 }
```

(a) Stage 1 (Sequential Stage / Read)

```
1  void stage2 (tid, arg){
2    version = 0;
3    while(TRUE){
4      if (ver_begin(tid))
5      {ver_doRecovery(tid);continue;}
6      if ((version % np) == (tid − 1)) {
7        ver_read(tid, &node, node);
8        res = work(node);
9        ver_writeAll(tid, &res, res);
10     }
11     if (ver_end(tid))
12     {ver_doRecovery(tid);continue;}
13     version++;
14   }
15 }
```

(b) Stage 2 (Parallel Stage / Work)

```
1  void stage3 (tid,arg){
2    version = 0;
3    while(TRUE){
4      if (ver_begin(tid))
5      {ver_doRecovery(tid);continue;}
6      ver_read(tid, &res, res);
7      write(res);
8      if (ver_end(tid))
9      {ver_doRecovery(tid);continue;}
10     version++;
11   }
12 }
```

(c) Stage 3 (Sequential Stage / Write)

```
1  system = ver_newSMTXsystem(n,config);
2  np = n−4; //number of parallel stage threads
3  ver_spawn(stage1, tids[0], arg);
4  ver_spawn(stage3, tids[n−3], arg);
5  for (i = 1; i <= np; i++){
6    ver_spawn(stage2, tids[i], arg);
7  }
8  ver_spawn(ver_tryCommitUnit, tids[n−2], arg);
9  ver_commitUnit(system, recovery_fun, commit_fun, arg); //
       implementation shown on the right
10 ver_deleteSMTXsystem(system);
```

(d) Main Thread / Commit Stage

```
1  void ver_commitUnit (system, recovery_fun, commit_fun, arg)
2  {
3    version = 0;
4    while (TRUE) {
5      status = CONFLICT_DETECTOR(version);
6      switch (status) {
7        case NO_CONFLICT:
8          COMMIT_ALL_WRITES(version);
9          commit_fun(arg); break;
10       case CONFLICT:
11         DISCARD_ALL_WRITES(version);
12         recovery_fun(arg);
13         RECOVER_WORKERS(system);
14     }
15     version++;
16   }
17 }
```

(e) Modified Commit Unit

Figure 8: The sequential loop in Figure 3(a) is transformed into a three-stage Spec-PS-DSWP pipeline with the second stage being a parallel stage. Stage 1 (a) is executed on core 1, stage 2 (b) on cores 2 and 3, and stage 3 (c) on core 4 as shown in Figure 6. The main thread (d) executes code outside the parallel region. Inside the parallel region, it doubles up as the commit stage on core 5 as shown in Figure 6.

which this worker participated aborts at some later time due to a memory conflict; in this case, the `continue` statement on line 10 in the same figure allows the worker executing `stage1` to resume execution of MTXs in the loop. Handling loop termination in the commit unit is straightforward and is not shown in Figure 8 (e).

5. Evaluation

5.1 Experimental Setup

The SMTX system is evaluated on two machines, called M1 and M2, with both running Linux 2.6.24. Table 2 gives the details of their hardware configurations.

CPU-intensive benchmarks requiring speculation were selected from the SPEC CINT and CFP benchmark suites [27], the PARSEC benchmark suite [6], and GIMP [13]. Code transformations (as shown in Figure 8) were done manually in a systematic manner as a modern compiler would do. We used loop-level profiling and analysis information from the LLVM infrastructure [17] and the VELOCITY compiler [8] to determine candidates for speculation. Since the transformations were done manually, tractability in terms of source code size and loop-spread across functions also influenced the loop and benchmark selection process.

Table 3 gives detailed information about each application's characteristics including parallel coverage, parallelization paradigm and speculations. A detailed description of each application can be found in [6, 13, 27]. As the last column of the table indicates,

	Intel Xeon E5310, M1	Intel Xeon X7460, M2
Processor	Intel Core, 64-bit	Intel Core, 64-bit
# Sockets	2	4
Cores per Socket	4	6
Threads per Core	1	1
Total # Threads	8	24
Clock Speed	1.6 GHz	2.66 GHz
Total RAM	8 GB	24 GB

Table 2: Hardware Platforms

Benchmark	Source Suite	Function	% of Runtime	Parallelization Paradigm	Speculation Types	Source Code Modified
052.alvinn	SPEC CFP	`main`	85.5	Spec-DSWP	MV	No
130.li	SPEC CINT	`main`	100.0	Spec-DOALL	CFS,MVS,MV	No
164.gzip	SPEC CINT	`deflate`	98.4	Spec-PS-DSWP	MV	Yes
181.mcf	SPEC CINT	`primal_net_simplex`	82.2	Spec-DSWP	CFS,SSS,MV	No
197.parser	SPEC CINT	`batch_process`	100.0	Spec-PS-DSWP	CFS,MVS,MV	No
256.bzip2	SPEC CINT	`compressStream`	98.5	Spec-PS-DSWP	CFS,MV	No
456.hmmer	SPEC CINT	`main_loop_serial`	100.0	Spec-PS-DSWP	MV	No
crc32	Ref. Impl.	`main`	100.0	Spec-DOALL	CFS,MV	No
blackscholes	PARSEC	`main`	100.0	Spec-DOALL	CFS	No
gimp-oilify	GIMP	`oilify`	98.9	Spec-DOALL	MVS/AS	No
gimp-nova	GIMP	`nova`	91.8	Spec-DOALL	MVS/AS	No

AS = Alias Speculation, CFS = Control Flow Speculation, MVS = Memory Value Speculation, MV = Memory Versioning, SSS = Silent Store Speculation

Table 3: Benchmark Details

the source code of 164.gzip was modified to make it amenable to parallelization; the modification is described in Section 5.3.2.

As a general note, by supporting Memory Versioning (MV), the SMTX system automatically breaks loop-carried false memory dependences. Because the memory pages are dynamically privatized, writes go to different physical locations. As Table 3 shows, MV is enough to parallelize some applications.

Implementation dependences in the memory allocator and random number generator were ignored because of the *commutativity* property of these functions [7, 15].

5.2 SMTX System Overhead

The overhead of the SMTX system is composed of one-time and running costs. The one-time costs are initializing the system, spawning the speculative worker threads (once per application run), and dedicating threads to the commit unit and the try-commit unit. The running costs are:

1. communicating speculative writes
2. refreshing local state with writes by other workers
3. in case of memory dependence speculation
   - executing transactional reads
   - executing transactional writes
4. recovering from misspeculation
5. copying a memory page when it is written to (COW)

Items 1, 2, and 3 depend on the performance of the software queues. Particularly in the case of transactional reads, the number of cycles to produce an $\langle$`addr, value`$\rangle$ tuple falls on the critical path, and hence producing to queues must be a fast operation. Table 4 shows the costs of performing the above operations. The cost of copying a page when it is written to is amortized over all accesses to the copied page. The absolute costs shown in the table were measured using the average across ten runs of micro-benchmarks.

Overhead	Metric	Cost on M1	Cost on M2
COW	cycles/page	9613	12018
ver_read	cycles/read	18	67
ver_write	cycles/write	22	64
Recovery	#workers/sec	4820	4336

Table 4: SMTX System Overhead

5.3 Results and Analysis

gcc (v4.2.4, -O3) was used to generate the native x86 binaries of both the sequential and parallel versions of the applications below. For each application, the baseline is the execution time of the sequential, unmodified code. All execution times were averaged across ten runs.

5.3.1 Applications without a parallel stage

`052.alvinn` trains a neural network using back propagation. Memory Versioning is used to break false memory dependences. The parallelized loop consists of inner loops. These inner loops are put in their own stages, creating a balanced two-stage pipeline: this yielded a speedup of 88%.

`181.mcf` is used to solve the single-depot vehicle scheduling problem. A pipeline consisting of two stages is extracted yielding a speedup of 40.73%. A store in the later stage is speculated to write the same value as the one already existing at that location (*silent store speculation*). Memory Versioning allows the first stage to execute later iterations while the second stage is still working on earlier iterations.

5.3.2 Applications with a parallel stage

Figure 9 shows *full application* speedup on the 24-core Intel Xeon X7460 machine. The x-axis shows the number of parallel stage threads: the size of the parallel stage indicates the scalability of the parallelization.

`130.li` is an implementation of lisp with object-oriented programming. The outermost loop processes the set of lisp scripts provided on the command line. Typically, each script can be processed independently of the other; this requires value speculation of various global environment-related variables, and a branch speculation that a script will not instruct the interpreter to exit. If the speculation fails in the latter case, the effects of all later scripts are squashed. The speedup is limited only by the input size.

`164.gzip` performs data compression using Lempel-Ziv coding. Only the compression part is parallelized. In 164.gzip, a three-stage pipeline is formed, with the first stage reading from the input file and storing the data in a block, the second stage compressing the block, and the third stage writing to the output buffer. Putting the loop-carried dependences in their own stages allows the second stage to potentially become a parallel (DOALL) stage. However, the choice of when to terminate a block and start a new one is related to the compression achieved on the current block. This dependence prevents parallelism across blocks; to break it, the code is modified to start a new block at fixed intervals. This results in a tradeoff between performance and compression level. In this application, the input file is read into a buffer, and the entire compression and decompression happens in memory. At higher thread counts, many of the threads do not fully utilize the CPU cycles; it appears that the memory bandwidth becomes the bottleneck causing them to stall on memory operations.

`197.parser` is a syntactic parser of the English language based on link grammar. The parsing of a sentence is typically independent of the others. In the parallel version, a sequential stage reads the sentences and presents them to a parallel stage for parsing. There

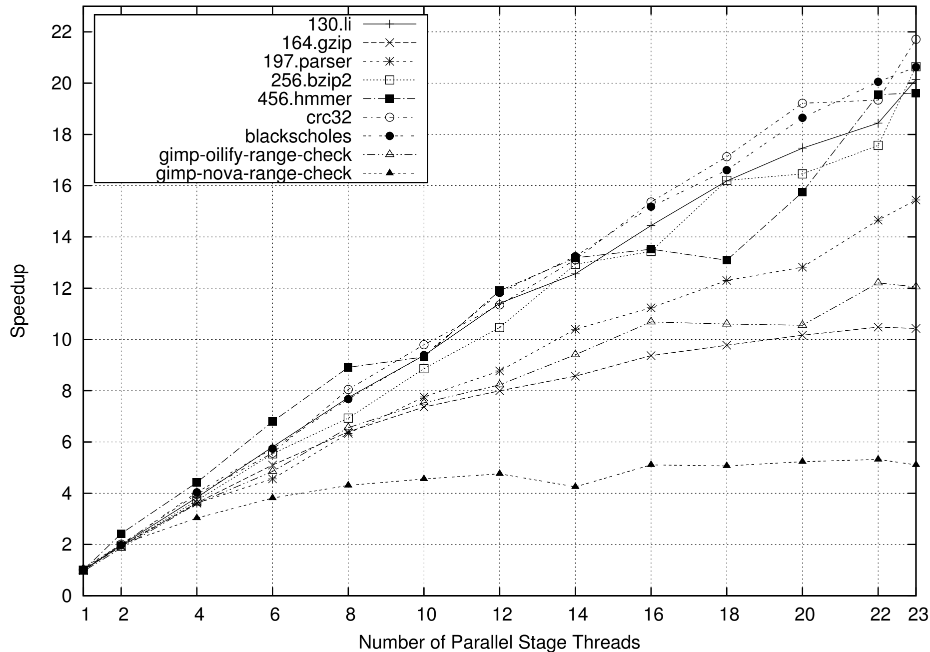

Figure 9: Scaling of application performance on machine M2: Parallel stage threads are executed on up to 23 cores, after leaving cores free for execution of sequential stages.

are several global data structures that are modified inside an iteration, but they are reset at the end of the iteration to the same values that they had at the beginning of that iteration. The value speculation described in Section 4.2 is used to break the dependences arising out of the stores to these data structures. Also, some branches that are taken under error conditions are speculated to not be taken. The loop speedup is affected by the variability in sentence length and is mainly limited by the number of sentences to parse.

`256.bzip2` performs data compression using the Burrows-Wheeler transform. Again, only the compression part is parallelized. The parallelization is similar to that of 164.gzip. In both cases, a variable-sized *block* array (produced by applying Run-Length Encoding to the input data) is the data structure used to store the blocks on which compression happens. Since the size of this array is unknown statically, very complicated whole-program analyses would be required for static privatization of the data structure. Memory Versioning allows the dynamic privatization of these data structures thus enabling parallelization.

`456.hmmer` is a computational biology application that searches for patterns in DNA sequences using profile Hidden Markov Models (HMMs). The loop in main_loop_serial is parallelized. Scores are calculated in parallel on sequences which are randomly selected. The Commutative annotation is used to break the dependence inside the random number generator. Following this parallel stage is a sequential stage in which a histogram is computed and the maximum score selected by using max-reduction. The speedup is limited by this sequential phase.

`crc32` computes the 32-bit CRC of files specified on the command line. Very little speculation is needed: error conditions that occur during the computation are speculated to not occur. The speedup is limited by the number of files and the variability in the file sizes.

`blackscholes` is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. As in `crc32`, the only speculation needed is that an error will not occur in the pricing. The speedup is limited only by the number of options to price.

5.3.3 An in-depth look at GIMP

`gimp-oilify` and `gimp-nova` are artistic transformation filters that are part of GIMP- the GNU Image Manipulation Program. The *oilify* filter makes an image look like an oil painting. The *nova* filter inserts a big star and associated light effects in the image. Rows are processed in parallel.

Figure 10 shows the speedup of `gimp` using Spec-DOALL on machine M1. The try-commit unit based checking algorithm scales up to 5 worker threads. Beyond that, the try-commit unit becomes the bottleneck. There is not enough computation happening in the worker threads in parallel with the replay of loads and stores in the try-commit unit. Micro-architectural profiling reveals that the try-commit unit spends a significant fraction of its execution time stalling due to the number of load-store instructions in the processor pipeline reaching the limit the processor can handle. This overhead is so much on machine M2 that there is no speedup to be gained. Parallelizing the execution of the

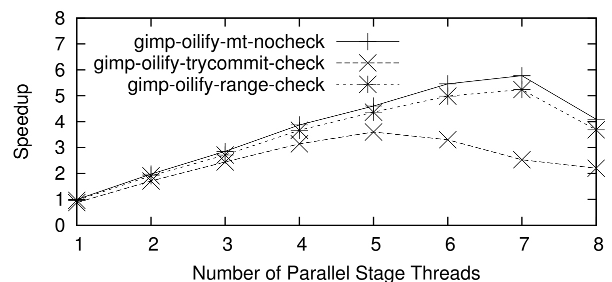

Figure 10: Overhead of the checking algorithm on M1

Benchmark	Speedup with		SMTX Recovery Components			Pages Versioned
	0% Misspec	1% Misspec	ERM	FLQ	RW	
130.li	6.36x	4.67x	50.2%	14.1%	29.5%	57.61%
197.parser	4.68x	2.78x	15.4%	59.4%	25.2%	54.67%
256.bzip2	5.25x	3.81x	64.6%	17.1%	15.6%	8.41%
crc32	5.68x	4.87x	10.1%	0.0%	86.4%	32.47%
blackscholes	6.91x	6.34x	8.4%	0.0%	91.5%	54.68%
164.gzip	5.77x	NotID	NotID	NotID	NotID	10.90%
456.hmmer	6.94x	NotID	NotID	NotID	NotID	38.79%
gimp-oilify	5.24x	NotID	NotID	NotID	NotID	63.55%
gimp-nova	5.49x	NotID	NotID	NotID	NotID	5.59%

Table 5: Overheads of Recovery and Versioning. NotID (Not Input Dependent) means speculation never fails.

try-commit unit itself would alleviate the bottleneck leading to better scaling. This is left to future work. Other checking algorithms, such as Bloom filter based signature matching [18], may be more suitable for such loops. Manually applying an address-range comparison based checking algorithm results in greatly improved performance scaling comparable to the multi-threaded parallelization with no checking. This is shown in Figures 9 and 10 as `gimp-oilify-range-check` and `gimp-nova-range-check`. Developing an analysis to insert such checking is left to future work. The scalability of `gimp-oilify` is limited by the low loop iteration count of 64 (because pixel regions are 64x64), and load balancing among the workers. Memory performance is the limiting factor for `gimp-nova`'s speedup. Multi-threading using pthreads with no checking yielded similar speedup, showing that the observed scaling is inherent to the parallelization and is not limited by SMTX overheads.

5.3.4 SMTX overheads in each application

Table 5 presents the overhead of recovery from misspeculation and the fraction of the non-speculative thread's memory pages that is versioned during the course of parallel execution. To determine the recovery overhead uniformly across applications, inputs were provided that caused the same fraction of iterations to misspeculate in each application. To determine the number of versioned pages, page fault statistics were gathered at the beginning and end of parallel execution. The two experiments were performed independent of each other on machine M1. Except for two applications, a misspeculation rate of 1% was injected. In `256.bzip2` and `crc32`, the misspeculation rate is 3.2% and 1.6% respectively since they iterate fewer number of times. Some entries are marked as "Not Input Dependent" because in these applications, speculation is used to overcome memory analysis limitations; in practice, the speculation will never fail. The recovery overhead may be split into the following items:

- SEQ [SEQuential execution]: Time taken to execute the misspeculating iteration non-speculatively. This varies from one application to another, and typically constitutes most of the recovery time. It is not shown in the table since it is around 99%.
- FLQ [FLush Queues]: Time taken to flush the inter-process queues. This depends on the queue size.
- ERM [Enter Recovery Mode]: Time taken by the workers and the commit unit to enter the recovery mode. This varies from one application to another because each speculative worker receives notification of misspeculation only on iteration boundaries. Thus, in the worst case, a wait of an entire iteration length may be necessary.
- RW [Recover Worker]: Time taken to mark the committed memory pages as COW, remap the page tables of the workers, and flush the workers' TLBs.

As the table shows, the cost of recovery is noticeable. To reduce this cost further, the recovery iteration may itself be executed in a multi-threaded fashion respecting all dependences. Also, asynchronous resteer of workers into the recovery mode can completely eliminate the *Enter Recovery Mode* overhead. Investigating these possibilities is left to future work. Finally, the last column shows that there is often a non-trivial memory overhead that is incurred for the speedups obtained.

5.3.5 SMTXs vs Single-threaded Transactions

Here, we quantitatively measure the utility of multi-threaded transactional semantics over single-threaded transactional semantics. The outermost loop in `batch_process` in `197.parser` is parallelized with Spec-DOALL using single-threaded transactions, and also separately with Spec-PS-DSWP using SMTXs. The system proposed in this paper is used to support both parallelizations. The results of the parallelizations are shown in Figure 11. The minimum number of threads is 2 and 3 respectively for Spec-DOALL and Spec-PS-DSWP because of the commit thread and an additional stage in case of pipelining.

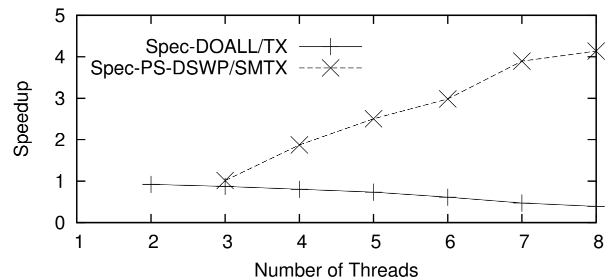

Figure 11: Spec-DOALL using single-threaded transactions vs Spec-PS-DSWP using SMTXs on M1

In `197.parser`, each transaction updates a global pointer that is used to scan the input for sentence delimiters. This loop-carried dependence causes frequent misspeculation in Spec-DOALL thereby serializing the iterations. In contrast, Spec-PS-DSWP creates SMTXs and synchronizes the scanning by executing it in a sequential stage and executing the remainder of each iteration in a different thread, in a schedule similar to that in Figure 6. The Spec-DOALL implementation may be improved by employing techniques such as speculative fission [33]. However, this results in sequential execution of the dependence recurrences outside the Spec-DOALL region; Spec-PS-DSWP with SMTX executes them in parallel using pipelining. Also, the speculative fission technique can incur excessive buffering overheads, especially in the case of outermost loops. Memoization [23] cannot help either because the loop is invoked only once. A detailed comparison with Spec-DOACROSS using single-threaded transactions can be found in [22].

6. Related Work

Traditional loop parallelization techniques such as DOALL and DOACROSS rely on regular structure in the program [3]. These techniques perform well for scientific programs but are less profitable for general purpose applications, where irregular control flow and data access patterns are the norm. To mitigate the inherent irregularity, some speculative techniques, loosely classified as Thread-Level Speculation (TLS), have been developed in recent years [5, 24, 28, 34]. As shown in Section 2, speculative pipelining schemes have many benefits over TLS. The execution model presented in this paper is inspired by the success of similar models described in [7, 29, 32]. Speculation is often needed to create the

pipeline structure and to extract DOALL-style parallelism in some stages of the pipeline. However, since transactions (loop iterations) are split across multiple threads, most existing TLS and transactional memory systems cannot be used. Vachharajani [31] introduced the notion of Multi-threaded Transactions (MTXs), along with a *hardware versioned memory* system that can support MTXs. The SMTX system, presented in this paper, is a *software* system that can support MTXs.

Hardware TLS proposals depend on special hardware to buffer speculative state, detect misspeculation and recover from it. In contrast, SMTX is a software-only system that enables speculative parallelization on commodity multicore hardware. Software TLS memory systems and software transactional memory systems have been extensively studied [18, 19, 25, 26]. As previously mentioned, these support only Speculative DOALL and DOACROSS execution models. In addition to supporting these models, the proposed SMTX system can support the speculative pipelining with stage replication model (Spec-PS-DSWP [8]).

Abadi et al. use page-level memory protection to provide strong atomicity in an STM [1]. By using page level metadata, their system discovers conflicts between, and synchronizes, transactional and non-transactional updates. While their system has multiple (two) versions of the virtual address space and one version of the physical pages, the SMTX system has a single version of the virtual address space and multiple versions of the physical pages.

Oancea et al. [19] present a family of speculation mechanisms that trade off dependence-tracking precision for improved latency and memory overheads of speculation. Such lightweight speculative models could be combined with the SMTX system to match an application's execution patterns.

Among the more recent *software-only* speculative loop parallelization strategies, there are three techniques that have all demonstrated good performance. They are discussed in detail below. The first two systems are thread-based; the discussion highlights the advantages (over and above those discussed in [10]) of the process-based SMTX system over thread-based solutions.

Tian et al. [30] proposed a Copy-Or-Discard (CorD) execution model for speculative parallelization. They achieve excellent speedup (3.7x to 7.8x on 8 cores) on six benchmarks. CorD does not support MTXs. CorD's execution model is such that each worker thread and the main thread are synchronized on every iteration, putting the cross-thread communication latency on the critical path. Combined with the sequential execution of the prologue and epilogue, it is crucial to keep these sequential, non-speculative segments small in size (execution time). This may necessitate excessive low-confidence speculation that will end up negating any benefits of parallelization. To overcome this, pipelining may be used to effectively parallelize such loops, with earlier pipeline stages working on later iterations. By supporting a more generally effective parallelization strategy in Spec-PS-DSWP, SMTX can claim wider applicability than CorD. CorD uses a multi-threaded approach which partitions the virtual address space. This penalizes every load and store operation by necessitating a table lookup to determine whether the object being accessed exists in the speculative worker's memory partition. By maintaining the same virtual address space across all workers, SMTX is able to guarantee that the load/store addresses will be valid across all of them.

Mehrara et al. [18] proposed a light-weight transactional memory system called STMlite that is optimized for loop parallelization. STMlite does not support MTXs. The proposed SMTX system architecture shares some features with STMlite: a centralized transaction commit manager and conflict detection that is decoupled from the main execution. However, note that the actual conflict detection mechanism is different: in STMlite, each transaction computes read and write signatures that are compared at commit time. STMlite allows transactions to compete for commit ordering; in contrast, due to its focus on loop parallelization, the SMTX system currently requires transactions to have a total ordering. The system can be easily modified in the future to loosen this restriction. As in other thread-based systems that implement *lazy versioning*, loads in STMlite will potentially have to scan the *store queue* to get the last write to that location. This significantly penalizes loads (STMlite uses caching of recent stores to improve the situation). By virtue of using Memory Versioning, loads are not penalized in the SMTX system: stores incur a page-fault penalty that is amortized over all the writes to that page.

Ding et al. [10] proposed a software system to support Behavior Oriented Parallelization (BOP). The BOP system does not support MTXs. The BOP system (and related systems such as those in [4, 14]), like the SMTX system, uses processes for data protection during speculative execution. However, many design choices differentiate the SMTX system from these other systems. To support MTXs, SMTX uses store-forwarding so that subTXs of an MTX enter the same memory version. To protect shared data and detect violations, the BOP system restricts page permissions and installs custom page-fault handlers to intercept reads and writes and records the type of access of a page in each process. This *access map* is used at commit time to detect any dependence violation, and is used to update committed memory in a rolling fashion: this puts inter-core communication latency on the critical path. In the SMTX system, a separate commit thread owns committed memory allowing an acyclic communication pattern. Note that in spite of this centralization, the commit and try-commit algorithms are parallelizable. In BOP, access violations are tracked at the page granularity resulting in the false-sharing problem. To alleviate this, each global variable is placed on a separate page in the BOP system resulting in memory overhead. The SMTX system does not suffer from the false sharing problem. In BOP, some variables are checked using a value-based checking algorithm: the values of the variables at the beginning and end of each task are compared to see whether they are the same. Note that the value must be reset on at least a few *consecutive* PPR (Possibly Parallel Region) instances. This is a restricted form of the general flow-dependence detection algorithm used in the SMTX system; for speculation to succeed, the value needs to be reset only before the speculative load (which need not be in the next transaction) reads it. Also, the checking is done off the critical path by a separate thread using an algorithm that is parallelizable.

7. Summary

This paper introduced the SMTX system: a software runtime system that supports Multi-Threaded Transactions to enable speculative parallelizations of general-purpose applications. By maintaining the same virtual address space with different mappings to physical memory, the workers are able to operate on their private memory without much performance penalty, with physical pages being privatized *on-demand* using the Copy-On-Write mechanism. A novel recovery scheme that remaps the virtual address space of the workers to the non-speculative physical pages, and a value-based conflict detection scheme are presented. This work extracts up to 21.71x speedup on today's 24-core multicore machine.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We thank Tim Harris for his insightful suggestions. We also thank the anonymous reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. (CCF-0811580).

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196. ACM, 2009.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37. ACM, June 2006.

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

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 81–96. ACM, 2009.

[5] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108, August 2002.

[6] 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*, pages 72–81. ACM, 2008.

[7] 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*, pages 69–84. IEEE Computer Society, 2007.

[8] 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.

[9] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[10] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234. ACM, 2007.

[11] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture Code Optimization*, 2(3):247–279, 2005.

[12] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 43–52. ACM, February 2008.

[13] GNU Image Manipulation Program. http://www.gimp.org.

[14] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A software system for speculative program optimization. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 157–168. IEEE Computer Society, May 2009.

[15] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222. ACM, 2007.

[16] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[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*, page 75. IEEE Computer Society, 2004.

[18] 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*, pages 166–176. ACM, 2009.

[19] 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*, pages 23–32. ACM, 2008.

[20] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375. IEEE Computer Society, 2007.

[21] 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*, pages 105–118. IEEE Computer Society, 2005.

[22] E. Raman. *Parallelization Techniques with Improved Dependence Handling*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, June 2009.

[23] E. Raman, N. Vachharajani, R. Rangan, and D. I. August. Spice: speculative parallel iteration chunk execution. In *CGO '08: Proceedings of the 2008 International Symposium on Code Generation and Optimization*, pages 175–184. ACM, 2008.

[24] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[25] P. Rundberg and P. Stenstrom. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3, October 2001.

[26] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

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

[28] 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.

[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*, pages 356–369. IEEE Computer Society, 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*, pages 330–341. IEEE Computer Society, 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*, pages 49–59. IEEE Computer Society, 2007.

[33] 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.

[34] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *MICRO '02: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 85–96. IEEE Computer Society Press, 2002.