A Lightweight In-Place Implementation for Software Thread-Level Speculation

Cosmin E. Oancea Alan Mycroft
Computer Laboratory, University of Cambridge
Cambridge CB3 0FD, United Kingdom
{Cosmin.Oancea,Alan.Mycroft}@cl.cam.ac.uk

Tim Harris
Microsoft Research
Cambridge CB3 0FB, United Kingdom
tharris@microsoft.com

ABSTRACT

Thread-level speculation (TLS) is a technique that allows parts of a sequential program to be executed in parallel. TLS ensures the parallel program's behaviour remains true to the language's original sequential semantics; for example, allowing multiple iterations of a loop to run in parallel if there are no conflicts between them.

Conventional software-TLS algorithms detect conflicts dynamically. They suffer from a number of problems. TLS implementations can impose large storage overheads caused by buffering speculative work. TLS implementations can offer disappointing scalability, if threads can only commit speculative work back to the "real" heap sequentially. TLS implementations can be slow because speculative reads must consult look-aside tables to see earlier speculative writes, or because speculative operations replace normal reads and writes with expensive synchronisation primitives (e.g. CAS or memory fences).

We present a streamlined software-TLS algorithm for mostly-parallel loops that aims to avoid these problems. We allow speculative work to be performed *in place*, so we avoid buffering, and so that reads naturally see earlier writes. We avoid needing a serial-commit protocol. We avoid the need for CAS or memory fences in common operations. We strive to reduce the size of TLS-related conflict-detection state, and to interact well with typical data-cache implementations. We evaluate our implementation on off-the-shelf hardware using seven applications from SciMark2, BYTEmark and JOlden. We achieve an average 77% of the speed-up of manually-parallelized versions of the benchmarks for fully parallel loops. We achieve a maximum of a 5.8x speed-up on an 8-core machine.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Design, Performance

Keywords

Thread-level speculation (TLS); Roll-back; Read-After-Write (RAW), Write-After-Read (WAR), Write-After-Write (WAW) dependencies.

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.

1. INTRODUCTION

Scientific applications are often amenable to *dependency-analysis* based static loop parallelisation [1] (predictable control flow and loop bounds, limited object aliasing, etc.); however, this does not extend to typical non-scientific code. Many applications nonetheless do have significant loop parallelism. Even common scientific programs, for example the FFT code below, when analysed give an iteration space falling outside Presburger arithmetic, thus precluding standard parallelisation techniques (in the example `dual` must be treated as an unknown for the inner loops thereby giving a non-linear system – this is more clearly seen after normalising the loop on `b`). Furthermore, even the abstract interpretation framework of Masdupuy [12], based on trapezoid congruences, only statically finds two threads.

```
for(dual=1; dual<N; dual*=2) {
  /* ... */
  for(a=1; a<dual; a++)         { /* do-all loop*/
    for(b=0; b<n; b+=2*dual)     {
      int i=2*(b+a), j=2*(b+a+dual);
      READ  (x[i], x[i+1], x[j], x[j+1]);
      ASSIGN(x[i], x[i+1], x[j], x[j+1]);
} } }
```

When static analysis fails to prove the absence of dependencies, run-time techniques may still extract parallelism. For example Rus *et al*. use an interval-based algebra to statically extract symbolic invariants, via hybrid analysis [20], and to translate them into a hierarchy of predicates, via sensitivity analysis [19]. Each level of the hierarchy gives a sufficient condition for do-all parallelism to exist; they are tested at run-time, in the order of their time complexities. The extraction of these predicates however may still be hindered by the same factors that may hinder static analysis, e.g. aliasing. If either the predicates evaluation fails or it is computationally inefficient, exact but potentially expensive inspector or software thread level speculation (TLS) algorithms are used.

Software TLS solutions usually extract parallelism from mostly-parallel loops of a single-threaded application: (i) a window of consecutive loop iterations is executed in parallel by a set of threads, and (ii) the original reads and writes to shared data are expanded with a scheme to detect cross-iteration dependency violations at run-time, and (iii) a safety-net mechanism is used to recover from violations – e.g., rolling back to a safe point. Since rollbacks are expensive, software-TLS is usually effective when less than 1% of code needs to be re-executed.

Existing software TLS solutions buffer their speculative updates, and commit them to non-speculative memory only when it is guaranteed that speculation succeed. This leads to so-called *serial commit* implementations, in which only the thread running the oldest iteration is allowed to write back its updates. As we show in Sec-

tion 2, all serial-commit algorithms exhibit weaknesses: (i) limited scalability in the number of processors contributing to speed-up, (ii) non-constant (and potentially high) instruction overhead [18, 26] – e.g. to satisfy same-iteration RAW dependencies, a read from address `a1` needs to check whether the current iteration has updated `a1`, (iii) high memory overhead [5, 6], (iv) require locking [21] or expensive memory fences [5, 18]. (The only parallel-commit implementation [18] exhibits huge memory overhead.)

We observe that the distance, in terms of efficiency, between evaluating at run-time the simple and sufficient predicates for parallelisation and the application of exact inspector/TLS algorithms is big enough to warrant an intermediary refinement. In our perspective, the comparative advantage of software, as opposed to hardware TLS, is the possibility to combine different instances of TLS solutions on disjoint memory partitions. Dynamic analysis techniques can determine which model is best suited for the type of dependencies that occur on each memory partition, and furthermore can adapt the TLS instance to exploit regular access patterns. We call these adaptive TLS instances *lightweight*, since their design trades off the potential for discovering false-positive violations for an efficient implementation of the (hopefully) common case. Companion papers [13, 14] add detail on this perspective: how TLS instances are combined and how patterns are identified via dynamic analysis.

The contribution of this paper is a lightweight, *in-place* TLS implementation: write operations directly update the non-speculative memory. We make five main contributions: *first*, the absence of a serial commit phase improves *scalability*, as shown in Section 4.2.

Second, our dependency tracking scheme has *constant* instruction overhead, comparable to the ideal case of current solutions.

Third, our speculative read/write operations avoid the use of atomic compare-and-swap (CAS) operations, or memory fences. The designs are intricate, and so we present a proof sketch in Sections 3.4 and 3.5. In particular, the speculative read operation modifies TLS metadata, and hence read-races are problematic. Our solution tracks dependencies in general at operation level; when a read-race is detected it conservatively switches to a coarser granularity level, but only for the racy access. This is effective in practice because it allows read contention at negligible overhead; such partially parallel loops – i.e. frequent contention of both reads and writes – are usually exploited via inspector-based techniques [10, 16]. Under mis-speculation, we perform *rollback recovery* to a safe point: the end of the iteration preceding the iteration detecting the violation.

Fourth, we show that our dependency tracking is sound under a "safe" language, such as Java. We identify subtle safety issues for languages using pointer arithmetic and indirect function calls, such as C++ and suggest possible solutions.

Finally, our implementation permits run-time access pattern adaptation. The TLS memory overhead is reduced by using a function `hash` computed via dynamic analysis [13], where `hash`'s cardinality gives the size of the data structure used in tracking dependencies. Under *irregular* access patterns, `hash` often approximates the perfect-hash signature function [2] that disambiguates the addresses accessed in any concurrent iteration window. We show in Section 4.2 that trading-off a few false-positive violations for small(er) speculative storage is beneficial for the workloads we studied: *regular* access patterns usually correspond to a `hash` of tiny cardinality that gives a cache-friendly layout to the TLS metadata. Running the FFT loop above on 4 cores, for `dual` ≥ 4, where thread $0 \leq i < 4$ executes iterations $i + 4\mathbb{Z}$ and `x` is the start address of array `x`, the function `hash(y)= ((y-x) / (2*sizeof(x[0]))) % 4` exhibits the nice property that thread i accesses only addresses `a1` with `hash(a1)=i`. This implies that a dependency-tracking structure of only 4 elements does not yield any false-positive violations. More importantly, the memory-friendly layout of TLS metadata – no cache conflicts between different pieces of meta-data – allows TLS' time overhead to decrease significantly against the periodic cache miss of the original code; we achieve 90% of the speed-up of the hand-parallelised, non-speculative version of the FFT loop.

The *penalty* is to allow cross-iteration WAW and WAR, to generate the occasional rollback, in addition to RAW dependencies. Furthermore, due to its in-place nature, our rollback recovery procedure is more expensive than the one of serial-commit implementations; while we expect to outperform the serial-commit implementations on fully parallel loops, the efficiency will decay faster with the number of violations than with a serial-commit model.

In terms of *performance*, experiments on both regular and graph-based tests show effective speed-ups (up to 5.8x on a 8-core machine) even when treating most data accesses as speculative. On fully parallel loops we achieve on average 77% of the speed-up of hand-parallelised code, with memory overhead at under 1% of program data for regular applications and between 10–40% for the graph-based ones. The tests also show that our implementation can tolerate a 0.5% (iteration-based) rollback ratio.

The rest of the paper is organised as follows. Section 2 introduces TLS at a high-level, briefly describes several software-only TLS solutions, identifies the main trade-off axes, and places our solution within these coordinates. Section 3 introduces the main components of our solution, and shows the simplified pseudo-code of the speculative load and store operations, which assume "atomic" execution. Sections 3.4 and 3.5 fill in the details by presenting the algorithm that implements the atomic behaviour and that preserves memory sequential consistency. Soundness proofs are sketched. Section 4 shows performance results and compares our implementation against another one that employs a serial commit phase, while Section 5 concludes.

2. TLS SOLUTIONS COMPARISON

In this section we survey related work on software TLS implementations in Section 2.1, we identify design axes on which to compare them in Section 2.2, we place our *in-place* solution within this space in Section 2.3, and finally we recount more remotely related work in Section 2.4.

For simplicity, throughout the paper we consider speculation on a single loop. We use: `M` for the number of loop iterations, `P` for the number of processors, `N` for the size of the data-structure requiring speculative support, `C` for the maximal number of concurrently-executing iterations, and `W` for the maximal number of per-iteration writes. Finally, thread identifiers (*id*) are numbered so that iteration i will be executed by thread i. The thread executing the lowest numbered iteration is referred to as the *master thread*.

2.1 Current Software-TLS Solutions

Rundberg and Stenström's solution [18] (S-TLS) directly simulates the operation of a hardware-based TLS cache protocol. In our opinion, this solution's *main strengths* are that it features a parallel commit phase and it effectively minimises the potential for false-positive violations, as dependencies are tracked at the speculative load/store operation level. *One downfall* is that it exhibits a huge $O(\mathtt{M} * \mathtt{N})$ memory overhead. In the presence of aliasing, it is often the case that a significant part of the original-data requires speculative support, which makes this solution impractical for these cases. Furthermore, (i) the speculative instruction overhead is non-constant and can be high for certain access patterns, (ii) the parallel commit checks all locations, not only the updated ones, and (iii) it requires expensive memory fences (`mfence` for X86).

Solution	Memory Overhead	Precision	Scalability
S-TLS	High	High	Medium-High
OS-TLS	Medium	High	Medium
SpFSC	Small	Medium-High	Medium
SpRO	0	Low	High
SpLIP	$\epsilon-$Small	Medium	High

Table 1: TLS Solutions Classification. (SpLIP – ours.)

Cintra and Llanos's [5] and **Rauchwerger et al.'s** [6, 17] software TLS solutions (OS-TLS) decrease TLS's memory overhead (of S-TLS) to roughly $O(\mathtt{N*C})$ via a sliding window mechanism in which a *serial* commit phase updates the non-speculative memory with the writes performed in the current sliding window. This limits scalability since now only a fixed number of processors may contribute to speed-up. As with S-TLS, dependencies are tracked at the operation level and only RAW may cause rollbacks. TLS support is applied at data-structure level, and `mfences` are needed.

Welc et al. propose an approach [26] (SpFSC) for integrating *safe futures* in Java. Although not discussed in their paper, we notice that their solution may effectively reduce the TLS memory overhead. *First*, each thread buffers the written values (`W*C` overhead). *Second*, the size of the dependency-tracking data structure does not depend on `N` or `M` but, roughly, on the range of memory locations accessed by `C` (concurrent and contiguous) iterations, under the penalty of introducing potential *false-positive* violations. Our understanding is that a perfect hash-like mapping [2], `hash`, is employed between memory locations and indexes in the speculative tracking structure. This is achieved by *compromising scalability*, as the solution employs a serial commit phase, and *rollback detection precision*, as the dependency-tracking mechanism is applied by the master iteration. In our experiments, the overhead associated to satisfying RAW same-iteration dependencies for this type of solution can be significant. Also, the serial commit phase exhibits higher overhead than OS-TLS.

Pickett and Verbrugge's framework [15] is similar, at a high-level, to that of Welc et al. in that it speculates at method level (Java) and their technique resembles that of STM. While we believe the same trade-offs hold, the main difference is the use of predictors, and hence of a more precise, value-based solution to detect violations. The framework is fully-automated and attempts to maximise parallelism by allowing speculative threads to create threads of their own. However, the high TLS overhead and the non-adaptive spawning strategy restricts gains: speed-ups were rarely observed.

At this point we introduce a simple TLS solution: **Read-Only** TLS (SpRO). This is the most effective, scalable, and imprecise of all, and can be seen as the intersection between our in-place implementation and the serial commit ones. A speculative load simply returns the contents of the to-be-read address, while a speculative store causes a rollback. The *rollback* procedure executes one iteration *sequentially* to ensure system progress. When composed with other solutions (for disjoint address ranges), SpRO yields important gains when employed on memory partitions of variables that are rarely written, but are not statically provable read-only (the overhead-free read compensates for the occasional rollback).

2.2 TLS Solutions Classification

Our brief literature survey hints that several trade-offs exist:

Memory Overhead ranges from S-TLS's $O(\mathtt{N*M})$ to OS-TLS's $O(\mathtt{N*C})$ to roughly $O(\mathtt{C*W})$ for SpFSC, to $O(1)$ for SpRO.

Scalability – we identify two sub-directions:

(i) "Will adding more processors yield a further speed up, assuming that there are no further conflicts?" The answer is *no* for the implementations exhibiting a serial commit phase: OS-TLS, SpFSC. This is because, if the per-iteration writes account for `1/k` of the iteration time cost, then, intuitively, the parallel execution resembles a `k`-stage pipeline, which is best exploitable by `k` processors.

(ii) Instruction overhead is not constant: in the worst case it is $O(\mathtt{M})$ for S-TLS, $O(\mathtt{C})$ for OS-TLS, and $O(\mathtt{W})$ for SpFSC.

Precision – violation detection accuracy:

(i) Dependencies are tracked at speculative operation level – S-TLS, OS-TLS, or iteration-window level – SpFSC (write operations appear to occur at the time of the serial-commit phase).

(ii) "What dependencies may cause violations?" SpLIP, our solution, exploits this trade-off axis: all run-time cross-iteration dependency violations – RAW, WAW, WAR – trigger rollbacks, in addition to potential false positives due to collisions under the hash function.

The first four rows in Figure 1 classify the TLS solutions that we have studied. SpRO gets high marks at memory-overhead and scalability but there is certainly enough room for improvement on the precision axis. The first three solutions are exact – only RAW dependencies may cause rollbacks. Their main strategy for reducing the memory overhead is to compromise the system scalability by employing a serial commit phase. This would be unsuitable for applications in which speculative writes represent a significant fraction of the total iteration execution time.

2.3 SpLIP: Our In-Place Solution

Our new TLS implementation, SpLIP, aims to: (i) accelerate the hopefully-common case of loop iterations that execute without dependencies between them by providing speculative operations of constant-instruction overhead, and (ii) to retain scalability in such workloads by avoiding a serial commit phase. Our key technique, inspired by recent STM implementations, is to combine in-place updates with TLS. As with in-place STMs, we must log values that get overwritten, and retain enough meta-data to be able to roll back to a consistent state. However, as we show in Section 3.4, unlike typical STM algorithms, we avoid *any* atomic read-modify-write operations or memory fences on speculative read/write operations. The downside is that cross-iteration WAW, WAR, RAW and false-positive dependencies may cause rollbacks. As we show in Section 4.2, for the studied workloads, the performance and scalability that we gain seem to offset the costs that this incurs.

Table 1 makes it clear that there is no "universal panacea" for software-TLS. Perhaps this is to be expected as the latter's main strength resides in flexibility. While hardware-TLS is restricted to use one scheme for all programs, software TLS has the comparative advantage that it may compose instances of various TLS solutions, to parallelise an application (e.g. for different data-structures; see [14]). Our perspective on software TLS is a tree of solutions, in which the nodes closer to the root score higher on the combined precision-scalability axes. With the available solutions, the root is S-TLS, leaves are SpRO, SpLIP and SpFSC, while OS-TLS is either a node or a leaf. Lightweight solutions are leaves, as they are more effective in both space and time, but less precise. If dynamic analysis indicates no suitable leaf, one of the nodes or the root is chosen.

2.4 Other Related Work

Hardware-centric TLS proposals [4, 7, 23, 24, 25] generally give better speed-ups than software ones, especially on iterations of small granularity. However they involve non-trivial and expensive changes to the basic cache-coherence infrastructure and avoiding overflow of the limited speculative storage can restrict gains [11].

Among other software TLS approaches from the literature we enumerate Kazi and Lilja's multi-thread pipelining architecture [10], Zilles and Sothi's master-slave model [28], and Chen and Olukotun's framework [3] that speculatively exploits method-level parallelism for Java applications. These scheme rely heavily on compiler support, hence comparing overheads is difficult.

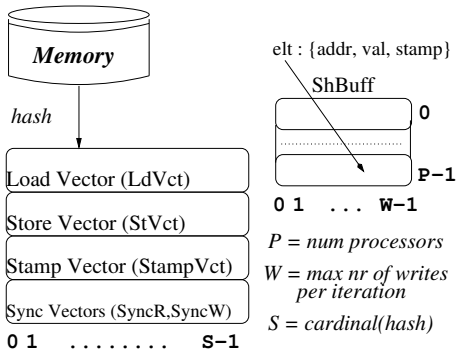

Figure 1: Speculative Storage Structure

```
atomic WORD specLD(
  volatile WORD* addr,
  int itNr ) {
1   int i = hash(addr);
2   if(LdVct[i]<thNr)
3     LdVct[i]=itNr;
4   WORD val = *addr;
5   if(StVct[i]<=itNr)
6     return val;
7   else throw
8     Dep_Exc(itNr-1);
}
```

```
atomic void specST(
  volatile WORD* addr,
  WORD new_val, int itNr)   {
1   int i    = hash(addr);
2   int prev = StVct[i];
3   if(prev > itNr)
4     throw Dep_Exc(itNr-1);
5   StVct[i] = itNr;
6   save( addr, *addr,
          StampVct[i]++ );
7   *addr     = new_val;
8   WORD load = LdVct[i];
9   if(load > itNr)
10    throw Dep_Ex(itNr-1); }
```

Figure 2: SpLIP: Speculative Load/Store Operations

Work on TLS-related compiler optimisations includes data-flow algorithms for identifying idempotent references [11], aggressive instruction scheduling techniques aiming at reducing the stalls associated with scalar values [27], and other optimisations related to loop inductors, light thread synchronisation locks, reduction operators and loop invariant register allocation [4]. Our implementation would also benefit from all such optimisations.

For completeness we conclude this section with a comparison with work on software transactional memory (STM). STM provides a mechanism for making a set of memory accesses appear to happen as a single atomic step. There are two main differences between STM and TLS. First, TLS requires that different threads' speculative work be committed in loop-iteration order. With STM transactions any serial order is typically acceptable. Second, TLS deals with a simple programming model in which execution is either sequential (outside a loop), or parallelised via TLS (inside a loop). Problems with concurrent speculative / non-speculative access to a given location do not occur, since TLS is usually applied on single-threaded applications. Saha et al. [21] and Harris et al. [8] designed STM algorithms using in-place updates. In addition to constraining commit order, our TLS implementation avoids the use of atomic-CAS operations on updates.

3. SPLIP IMPLEMENTATION

Section 3.1 presents the structure of the speculative memory and high-level ideas. Section 3.2 shows the atomic-based implementation of the speculative load/store operations. Section 3.3 proves the implementation's safety. Finally, Sections 3.4 and 3.5 show how to implement the speculative operations' atomicity effectively (without CAS or `mfence` instructions).

3.1 Speculative Storage Structure

Figure 1 depicts how the speculative memory is organised. For the moment assume that the map `hash` between memory locations and entries in the dependency-tracking vectors is one-to-one. `LdVct`/`StVct[i]` hold the maximal iteration number ($\equiv$ thread *id*) that, at a time, has read/written an associated memory location (`a1`, with `hash(a1)=i`). `SyncR`/`SyncW` are used in Section 3.4 for synchronisation between readers and writers without needing atomic CAS operations.

We assume a thread-pool matching the available processors, `P`. For simplicity we take `C==P`, where `C` is the size of consecutive, concurrent iteration window. (We can choose however `C` a multiple of `P`.) Threads are re-used under the policy of assigning the next iteration to the first available thread. With `C==P`, we have that always P consecutive iterations are executed concurrently. As with in-place STM implementations, prior to an update, the address, previous value and a time-stamp are saved in a *shadow* buffer – `ShBuff`, where `StampVct[i]` holds the "current time" for the class of addresses `a1` with `hash(a1)==i`. `ShBuff` is a two-dimensional array, of first dimension `P`. While executing iteration `j` a thread "owns" (has exclusive access to) `ShBuff` row `j`%`P`, and its writes access consecutive entries; this gives good cache behaviour. Of course, `ShBuff` is also accessed during rollback.

3.2 Speculative Load/Store

Figure 2 shows the simplified implementation of the speculative load/store operations. We initially assume these operations execute atomically; we return to the synchronisation protocol we use to ensure this in Section 3.4. Both TLS operations access the original memory *in place*, and they may discover dependency violations. These are signalled via exceptions (`Dep_Exc`) that register the *safe recovery point* – the highest iteration up to which the program is guaranteed to have executed correctly. The variable `i` denotes the index in the `LdVct`/`StVct` corresponding to `addr`. The values stored in each entry of `LdVct`/`StVct` are monotonically increasing in time (see lines 2/3 in `specLD`/`specST`) and give the number of the highest iteration that has read/written the content of any address `a1` with `hash(a1)==i`. `SpecLD` is the simpler of the two operations. A speculative load is successful as long as there have been no speculative writes to the same location by later iterations – i.e. it is successful if `StVct[i]` $\leq$ `itNr`. Otherwise, it detects a WAR violation at line 10; *the safe point* is the end of iteration `itNr-1`, since the current one (`itNr`) cannot complete.

Conversely, a speculative store is successful so long as there have been no speculative accesses at all (either loads or stores) to the same location by later iterations – i.e. it is successful if `StVct[i]` $\leq$ `itNr` and `LdVct[i]` $\leq$ `itNr`. The `specST` operation may fail due to a RAW violation (line 10), when it determines that a successor iteration has read a value that should have been provided by the current iteration. Note that the safe-point is `itNr-1` and not `load` (from Figure 2) as one may naively guess – any successor of the current thread and predecessor of `load` may have also misread, and the framework cannot detect it. Furthermore, `specST` may fail due to a WAW violation (line 4).

Skipping the update without signalling a violation is incorrect because: First, the previous store may have referred to a different address in $\texttt{hash}^{-1}\texttt{(i)}$ and hence the sequential semantics requires an update. Second, it may fail to detect a RAW violation. (E.g. consider the time-ordered sequence of accesses to address `a1`: thread $T3$ loads, $T4$ stores, $T5$ loads, $T2$ stores. Thread $T2$ will not identify the RAW violation corresponding to thread $T3$.) If successful, the *former* content of address `addr` is saved into the thread's `ShBuff` row (line 6) prior to the update in line 7.

Note that dependency detection is still sound even if hash is a many-to-one function. Hash collisions may cause false dependencies to be detected, but never miss real dependencies. (This refinement corresponds to the conservative, hence safe, approach that treats a memory access to address a1 as an access to all locations in $\texttt{hash}^{-1}(\texttt{hash(a1)})$.)

Note that, as the execution window progresses, the shadow buffer and the LdVct/StVct vector values are re-used, and if no violation occurs, there is no need to clear them at the iteration's end (unlike other solutions). Furthermore, the instruction overhead is constant, violations are tracked at instruction level and forwarding occurs (only) implicitly via updates to non-speculative data. The speculative memory size depends on the number of per-iteration writes, and on hash's cardinality. The latter is small for regular access patterns.

3.3 Correctness. Roll-back Recovery.

This sections shows *first* that, when specLD/specST operations are atomic as in Figure 2, the oldest source iteration involved in a dependency violation will discover it, and *second*, that our rollback technique is sound. For the purpose of demonstrating the correctness of the dependency tracking mechanism, it is useful to consider all dependency violations that share the source iteration as belonging to the same class:

DEFINITION 3.1 (DEPENDENCY EQUIV. CLASS). *Let δ_1 and δ_2 be two dependency violations with $\delta_i = (s_i, t_i)$ being the iteration numbers (ids of thread executing these iterations) of the source and target of the violations. Now define $\delta_1 \sim \delta_2 \Leftrightarrow s_1 = s_2$ and note that it is an equivalence relation. (Note that the source id is always smaller than the sink id).*

THEOREM 3.1 (DEPENDENCY-TRACKING). *Assume that under* TLS *speculative execution the read/write access to all variables used in the code of a loop is accomplished via the specLD/specST functions in Figure 2. Then, from all the run-time violations, precisely one per equivalence class is detected, and the thread that detects it is the one executing the source of the dependency. (It follows directly that the lowest iteration involved in a violation will detect it prior to consuming any dirty values.)*

PROOF. Consider a *true-dependency* violation (RAW): thread id2 has read (via specLD) the content of the address a1 before thread id1 has updated (via specST) a1, and id1 < id2. Then, at lines 8–9 of specST in Figure 2, thread id1 will find that id3=LdVct[hash(a1)]≥id2>id1 due to the monotonic behaviour of the elements of LdVct. Note that the detecting thread (id1) is the source of the dependence and that no other violations with source id1 will be discovered, since thread id1 throws an exception. The anti- and output-dependency violations (WAR/WAW) are detected in line 8 of specLD and line 4 of specST and the proofs are similar. □

The remainder of this section shows that program execution under our TLS model preserves the semantics of the sequential execution. Figure 3 shows the layout of the speculative thread. The thread receives a new iteration to execute (id), and serially executes the code that would otherwise yield too many dependency violations or that contains un-rollable operations (IO) together with the dependent instructions (line 4). (We assume that the compiler has moved all IO-independent instructions before the first IO and has peeled-off the IO-independent part of the first iteration.) Then the induction variables that can be determined solely from the iteration number are updated (line 5). The TLS-preferred granularity is achieved by unrolling the loop U times (line 6).

```
class SpecThread {  //...
1 void run() {
2  while( true ) {
3   try {                              10 }catch(AnyError e) {
4    id = AcqAndSerialExec();          11  waitBecomeMaster();
5    set_IndVars();                    12  killSpecThreads ();
6    for(int i=0; i<U; i++){           13  rollbackProc(id-1);
7     if(end_of_loop_cond())           14  if(!SpecEnded())
8      throw EndOfLoopExc();           15    respawnThreads();
9     iteration_body();    }           16  else return;} } }};
```

Figure 3: Speculative Thread Layout – Pseudocode. For C, all catchable errors are treated similarly to regular exceptions.

Upon detecting a dependency-violation, or any other exception or error or the end of the loop, the thread waits to become master (line 11). If it is master, then it *kills* all its successors (line 12 – hence it is the only live thread) and initiates the rollback procedure (13). Theorem 3.1 guarantees that one of the violator-threads will eventually become master since its "correct" predecessors will eventually finish executing their iterations.

The rollback procedure clears the load/store vectors and restores the original memory to the safe-point configuration, by using the information in the ShBuff entries corresponding to threads that are successors of the safe-point – this is enough, since specST preserves the sequential update order. The algorithm complexity is $O(\texttt{W}*\texttt{C}*log(\texttt{W}*\texttt{C}))$, with C and W defined in Section 2. The algorithm aggregates the considered entries, but only the lowest timestamp tuple is retained from a sequence of identical-address tuples. The address of each tuple is then updated with the value member. The latter procedure lacks with serial-commit models since the non-speculative state is always updated only after the speculation has proved successful; hence SpLIP rollback recovery is more expensive. If the number of rollbacks exceeds 1% of the number of executed iteration, the framework reverts to sequential execution. Furthermore, since the master thread may cause rollbacks, *one iteration is executed sequentially to ensure that the system "makes progress"*. (Otherwise, the same violation is discovered multiple times, hence speculation may be abandoned in spite of the fact that the code exhibited only one run-time violation.) Finally, threads are re-spawned and assigned new iterations to execute.

THEOREM 3.2 (ROLLBACK INVARIANT). *Under* SpLIP, *assume the thread executing iteration* rollId *is* master *and it has detected a violation. Rolling back the updates of* rollId *and all its successor iterations, in the manner presented above, constructs the same program state as the sequential execution of the iterations up to and excluding* rollId.

PROOF. Let $update_{i,l} = \{it, val, old\_val\}$ be the update performed by iteration it on the memory location l (old_val= *l; *l= val;), where a fresh i is chosen in accordance to the time-ordering of the updates. Let $Wr(l) = (update_{i,l})_{i\geq 0}$, be the whole update sequence for l. Fix l. Let γ be the smallest integer such that $update_{\gamma,l}.it \geq rollId$. The updates to l prior to γ respect the sequential program order and were performed by threads with ids less than $rollId$ (otherwise, the *rollId* thread cannot become master since a lower id thread would detect a WAW violation – by Theorem 3.1). The theorem says that the correct value of l after sequentially executing $rollId - 1$ iterations is $update_{\gamma,l}.old\_val$. Assume this does not hold for some l. It necessarily follows that $\exists k > \gamma$ such that $update_{k,l}.it < rollId$ (the update of an iteration that is not rolled back is missed). But then, the iteration $\alpha = update_{k,l}.it$ is the source of a WAW dependence, which, by

virtue of Theorem 3.1 should be detected by iteration α. This contradicts the hypothesis assumption that thread *rollId* is master. □

Theorems 3.1, 3.2 and the following discussion about other mis-speculation effects give the safety result for a language (without pointer arithmetic and without user-level indirect function calls) such as Java:

Infinite Loop – Theorem 3.1 guarantees that: First, the master thread cannot consume an incorrect value. Second, there must be a thread that discovers a dependency-violation and eventually becomes master. Therefore the thread executing the infinite loop will be eventually killed.

Error/Exception/End of Loop – If the thread becomes master, then these effects are not due to mis-speculations: the updates of successor threads are rolled-back and the speculations ends either by throwing an exception or normally. Otherwise, a mis-speculation will be detected and the thread will be killed.

Garbage Collection (GC) – in a language like Java, we could still implement our dependency tracking, but instead of tracking (real) addresses, we use each `Object`'s `hash` field for as pseudo-addresses in addition to fixing offsets inside the containing object/array for basic-type fields. These are guaranteed to be constant, and hence GC's memory reordering does not affect TLS.

A language allowing pointer arithmetic and indirect function calls such as C++ exhibits additional safety issues that we briefly outline in the remaining of this section. We use the term *dirty* to generically denote any mis-speculated value that is propagated before the violation that created it is detected. We draw attention that the discussed issues below appear also with serial-commit implementations, since a speculative thread may consume an uninitialised value that should have been updated by a predecessor thread.

First, we observe that a mis-speculation may cause, for example, a `NULL` pointer to be dereferenced, which leads to a segmentation fault on Linux. It follows that these relevant hardware errors should be trapped and treated in the same way as exceptions in the above discussion. For example, under POSIX standard, the only uncatchable signals are SIGKILL and SIGSTOP, since a user should be able to kill/stop its program. The SIGSEGV signal, which denotes an invalid memory reference is catchable.

Second, dirty references may corrupt TLS metadata either directly, or indirectly, for example by corrupting `malloc`'s metadata. We address this issue by protecting the metadata corresponding memory segments with SpRO: now only `specLD`/`specST` may update TLS metadata but a dirty update to TLS metadata will fail and will generate a rollback.

Third, accesses to variables on stack frames following the one corresponding to the speculative loop are not protected via TLS since they are thread private. However, each assignment to such a variable is guarded by a check that guarantees that the address of the variable is inside that stack-frame, but different than the return pointer of that frame. Otherwise, the update fails and a rollback is started. This solves the problem of a dirty reference causing an unpredictable jump in the program by overwriting the return pointer.

Finally, the difficult problem is that of a "wild" branch being taken because of a dirty indirect function call: either via a dirty function pointer or a dirty update to an object *v-table*. The simple solution is to conservatively wait for the current thread to become master before calling an indirect function. This solution is not practical under frequent indirect calls. A more refined solution, which is beyond the scope of this paper, would be to check at run-time that the function pointer to be called is valid: it belongs to a speculative version of a function of the same signature to the one that is called. This refinement would still give correct rollback recovery.

3.4 Non-locking Implementation of specLD/ST

This section shows how to implement `specLD` and `specST` without assuming atomic execution and without (expensive) CAS instructions. We initially assume sequentially consistent hardware. Section 3.5 discusses our practical implementation for the Intel IA-32 memory model. We need to ensure that (i) from a group of concurrent violations, the lowest numbered iteration that may have been involved in a dependency is detected – the minimal *safe point* property, and (ii) even if concurrent writes to the same memory location are associated with the same time stamp, the *rollback* mechanism is still correct.

For presentation reasons, we break our solution and proof sketch into four parts: *first* we have already proved soundness in Section 3.2 for the case when `specLD/ST` execute atomically.

Second, we show that the non-atomic interaction between concurrent `specLD` and `specST` calls is still sound, as long as same-type operation are assumed to execute atomically. This is discussed in section 3.4.1.

The *third* and *fourth* steps provide the modifications that allow multiple `specLD` and `specST` operations to execute safely without assuming atomicity, and are presented in sections 3.4.2 and 3.4.3.

3.4.1 Interaction between specLD and specST

This section shows that, with the *non-atomic* implementation in Figure 2, if a load and a store are in a race condition, i.e. accessing same `LdVct/StVct` indexes, then either a RAW or a WAR violation is detected, and the minimal safe point is preserved.

There are two inconsistent cases that may lead to a violation of the sequential semantics. First, a store, reaching line 8, may find a load operation accessing the same index `i` in an inconsistent state: `LdVct[i]` has been updated but the memory has not been read yet (`specLD` execution is somewhere between lines 2 and 4). In this case, if the condition in line 9 holds, the current store conservatively assumes that the load has read a value produced by an earlier write and signals a violation. The safe point is the store's `itNr-1`, and is minimal (the load's `itNr` necessarily greater).

Second, a load (`ldTh`) may find a store operation accessing the same index `i` in an inconsistent state: the current thread (`currTh`) has updated `StVct[i]`, but the value "consumed" by `ldTh` belongs to an earlier store (`wrTh`).

- If `currTh<wrTh` then the current store yields a violation since the sequential order of writes was not respected. If `ldTh<currTh` then the load also yields a violation since it conservatively assumes that the needed value was overwritten. Either way the minimality of the safe point is ensured.
- If `currTh==wrTh` then the state is consistent from the `ldTh` point of view. (However `currTh` may conservatively yield a violation later at line 10.)
- Otherwise: `currTh==StVct[ind]>wrTh`. All the following cases detect the lowest violator:
  (i) if `currTh>ldTh` then `specLD` detects a WAR violation at line 8,
  (ii) if `currTh<ldTh` then by virtue of the load vector monotonicity we have `currTh<LdVct[ind]` and `specST` detects a RAW violation at line 10,
  (iii) the case `currTh==ldTh` cannot happen: the same thread cannot execute at a time both a load and a store.

3.4.2 Interaction between non-atomic specLDs

The third stage implements atomicity semantics for `specLD` – see pseudocode in the left-hand side of Figure 4 (`specST` is shown

```
WORD specLD( volatile WORD* addr,
             int itNr, int TH_ID  )       {
1    int i    = hash(addr);
2    SyncW[i] = TH_ID;
3
4    if( LdVct[i]  < itNr )
5        LdVct[i] = itNr;
6
7    if( SyncW[i] != TH_ID )
8        SyncR[i] = itNr + P;
9
10   WORD val = *addr;
11
12   if( StVct[i] <= itNr ) return val;
13   else        throw Dep_Exc(itNr-1);
                                        }
```

```
void specST( volatile WORD* addr,
             WORD new_val, int itNr, int TH_ID )   {
1    int i    = hash(addr);
2    SyncW[i] = TH_ID;
3
4    if(StVct[i] > itNr)  throw Dep_Exc( itNr-1 );
5    StVct[i] = itNr;
6
7    if(SyncW[i] != TH_ID) throw Dep_Exc( itNr-1 );
8    WORD old_val = *addr; int stamp = StampVct[i]++;
9    ShBuff[itNr%P] = { addr, old_val, stamp }; //save( ... );
10   *addr    = new_val;
11
12   WORD ld = max( LdVct[i], SyncR[i] );
13   if( (ld > itNr) || (StVct[i] != itNr) )
14                       throw Dep_Exc( itNr-1 ); }
```

Figure 4: Implementing Speculative *Load* (left side) and *Store* (right side) Operations using Normal Memory Accesses. The `volatile` keyword assumes the weaker, C++ semantics – it does *not* introduce WRITE-READ barriers.

on the right-hand side). Consider two threads $\texttt{id}_0 < \texttt{id}_2$ in a read-race case – i.e. executing concurrently lines 2–7 for the same `addr`. The bad case is when $\texttt{id}_0$ is the last to be recorded in `LdVct[i]` since this breaks `LdVct[i]`'s monotonicity to the result that a later write performed by thread $\texttt{id}_1$, with $\texttt{id}_0 < \texttt{id}_1 < \texttt{id}_2$ may fail to detect a RAW violation involving $\texttt{id}_2$.

Our approach is to detect the read race via the use of `SyncW` in lines 2 and 7, where `TH_ID` is a per-thread unique value. Note that *all but one* threads involved in the read race *detect the race*. If a race is detected, a conservative-high value, `itNr+P`, is inscribed in `SyncR[i]`, where `P` denotes the number of processors. No matter of the update order at line 8 – threads may be arbitrarily pre-empted – the system's design guarantees that iterations greater or equal to `itNr+P` cannot be running while iteration `itNr` is running (see Section 3.1). Hence at the time of any update at line 8, any iteration involved, but not aware of the race is lower than `itNr+P`. The monotonicity of `LdVct[i]` is fixed now since `specST` uses in the RAW test `max(LdVct[i],SyncR[i])`, at line 13.

The fact that a concurrent write $\texttt{id}_w$ has happened within the interval in which the monotonicity was broken, luckily, does not give any more problems. We sketch the proof here. Assume $\texttt{id}_M$ is the read that was involved but not aware of the race, and that $\texttt{id}_m$ is the value the problematic write finds in `LdVct[i]`. (Note that the broken monotonicity case necessarily requires that $\texttt{id}_w$ was inscribed in `StVct[i]` before $\texttt{id}_m$ reads it at line 12.) If $\texttt{id}_m > \texttt{id}_M$ then `LdVct[i]`'s monotonicity was preserved (safe). Otherwise, if:
(i) $\texttt{id}_w < \texttt{id}_m < \texttt{id}_M$ then the write detects at line 14 a RAW violation of safe point $\texttt{id}_w\texttt{-1}$, or if
(ii) $\texttt{id}_m < \texttt{id}_w < \texttt{id}_M$ or $\texttt{id}_m < \texttt{id}_M < \texttt{id}_w$ then the read $\texttt{id}_m$ detects at line 13 a WAR violation of safe point $\texttt{id}_m\texttt{-1}$. In both cases the minimality of the safe point is preserved.

Note that our proof sketch allows for an arbitrary number of concurrent threads. Also our solution does not introduce any rollbacks on read-only accesses. The conservative monotonicity fix above may introduce false-positive violations under *both* read and write races; however, if such races are frequent then frequent violations will happen anyway, and software TLS would likely be ineffective.

3.4.3 Interaction between non-atomic specSTs

The fourth stage implements atomicity semantics for `specST` – see the right-hand side pseudo code of Figure 4. The technique is similar: `StVct` is used to decide whether or not the current store was involved in a race-condition (it is set in line 5 and tested at the end in line 13). If so, a rollback is generated – this technique increases the number of found WAW violations by a factor of two, since it is anyway a 50% chance that a WAW dependency is violated. Without the use of `SyncW` on lines 2 and 7 it may happen that the lowest iteration involved in a race, is not aware of the race, succeeds executing the store, and starts a new iteration, before one of the other conflicting stores initiates the rollback. In this unlucky case one of the `ShBuff` rows needed by the recovery procedure may have been over-written. Instructions on lines 2 and 7 fix this issue by guaranteeing that the lowest iteration involved in a race is either aware of the race and causes a rollback or it executes lines 5–13 in mutual exclusion. The proof sketch assumes that threads id_1 and id_2, $id_1 < id_2$, concurrently write an address of index `i`:

- Thread id_2 executes line 2 after id_1 ($\texttt{SyncW[i]} == id_2$).
  If this happens before thread id_1 reads `SyncW[i]` at line 7, then thread id_1 detects a rollback at lines 4 or 7.
  Otherwise, thread id_1 initiates a rollback at line 14 or executes the (code) lines 5-to-13 in mutual exclusion. Note that a thread with an *id* higher than id_1 cannot already be inside the 5-13 region since then thread id_1 would have thrown an exception at line 4. If a lower number thread is already inside region 5-13, it will discover the violation at line 13 and the minimality of the safe point is preserved.

- Thread id_1 executes line 2 after id_2 ($\texttt{SyncW[i]} == id_1$).
  If this happens before thread id_2 reads `SyncW[i]` in line 7, then thread id_2 initiates a rollback at line 7, and allows id_1 to execute atomically region 5–13.
  Otherwise, thread id_2 is already inscribed in `StVct[i]`, hence thread id_1 initiates a rollback at line 4.

Finally, for the `specST` implementation in Figure 4, we note that several writes to the same location may be stamped with the same time. However, in this case the saved values are the same too, since `addr` *is read before the stamp counter is incremented*, hence the rollback procedure may validly use any of these. If the number of per-iteration writes exceeds the predicted dimension of the `ShBuff` row, that row is dynamically re-allocated. This does not raises concurrency issues since the row is "owned" by one thread.

3.5 Memory Ordering on Intel IA-32

This section discusses what is required to make `specLD`/ `specST` sound on the Intel IA-32 architecture, even though this hardware does not provide sequential consistency. Our solution exploits the details of Intel's specification [9] (relevant Chapters 7.1 and 7.2).

The Intel IA-32 memory model is fairly strong, when compared with those of the ALPHA, or the POWER PC. Given two distinct

addresses "x" and "y", the only re-ordering allowed by the IA-32 model is for a later read to appear to be performed before an earlier store. For example, consider the following 4 patterns:

```
 PATTERN 1   |  PATTERN 2    |  PATTERN 3   |    PATTERN 4
  x=y=0;     |   x=y=0;      |              |  TH 0   TH 1
TH 0  TH 1   | TH 0   TH 1   | TH 0   TH 1  | u=..;  v=..;
x=1;  ..=y;  | x= 1;  y= 1;  | t=..;  u=..; | z=..;
y=1;  ..=x;  | mfen;  mfen;  | ..=W;  ..=W; | ..=W;  x=..;
             | ..=y;  ..=x;  |              | ..=x;  t=..;
             |               |              | ..=W;  ..=W;
```

According to the IA-32 documentation, Pattern 1 is always sequentially consistent for any locations x and y (i.e. thread 1 cannot read y==1 and x==0). When x and y are different locations, Pattern 2, without mfences, is the only one that allows (memory) re-ordering (i.e. **not** sequentially consistent) – for example both threads 0 and 1 may read at the end x==y==0. One fix, for examples like Pattern 2, is to use the mfence instruction. Unfortunately this is costly in typical implementations. Furthermore, Pattern 2 occurs in three places in the implementation of SpLIP's speculative operations, which would necessitate numerous fence operations.

To avoid fences, we can exploit other details of the IA-32 memory model and, in particular, its behaviour when different threads access parts of the same word. Only for the purpose of this section, let W be a 64-bit aligned word, and let t, u, v and z be 16-bit aligned subwords of W, with t and W starting at the same address. The IA-32 model guarantees that read/write accesses to aligned subwords/words are guaranteed to be atomic. We denote by $\mathtt{x} = \mathtt{y}$ or $\mathtt{x} \neq \mathtt{y}$ whether x and y are considered the same memory location or not. We claim that Pattern 3 is sequentially consistent since any interpretation of "different locations" does not exhibit the problematic Pattern 2: (i) $\mathtt{t} \neq \mathtt{W}$ and $\mathtt{u} \neq \mathtt{W}$ or (ii) $\mathtt{t} = \mathtt{W}$ and $\mathtt{u} \neq \mathtt{W}$ or (iii) $\mathtt{t} = \mathtt{W}$ and $\mathtt{u} = \mathtt{W}$.

This pattern has been tested on ***one*** hardware configuration and the non-consistent case has ***not*** been observed[1]. There are concerns that Intel's specification is weak – see Sarkar et al. [22], and hence ambiguous. The sequentially consistent behaviour of Pattern 3 is likely due to flushing the cache line when cache coherency detects concurrent accesses to a word and one of its contained subwords. This is a significantly cheaper and more scalable mechanism than mfence. If this behaviour was unintended, the scalable results of Section 4.2 argue in favour of standardising it.

We represent LdVct[i], StVct[i], SyncR[i] and SyncW[i] in Figure 4 as interleaved, aligned 16-bit subwords of a 64-bit word; reading any of these is replaced by reading the full word and computing the required value. We insert an inter-thread synchronisation barrier every 2^{16} iterations to clear TLS metadata. Pattern 3 proves the sequential consistency of (i) the accesses to LdVct[i] and SyncW[i] for concurrent specLD executions and (ii) the accesses to StVct[i] and SyncW[i] for concurrent specSTs.

We discuss now the concurrency between specLD and specST, which is depicted in Pattern 4: t, u, v, z and x stand for SyncW[i], LdVct[i], StVct[i], SyncR[i] and addr, respectively. Pattern 3 showed that the accesses to TLS metadata t, u, v, z and W are (always) *not* re-ordered. What about x?

The delicate case is when, in memory ordering, the accesses to TLS metadata of thread 0 (the load) come after the accesses of thread 1 (the store). This is because the value read by thread 0 from address x is *not* guaranteed to be the one thread 1 updated. The write to t just after writing x in thread 1 and the read from W just before reading x in thread 0 provide the fix-up. In the interesting case when t=W, Pattern 1 prevents now re-ordering thread's 1 updates to x and t: thread 0 necessarily reads the good value of W (by hypothesis), which implies also that the updated value of x is read. The explanation is similar to that of Pattern 3 – all possible interpretation for "different locations" are considered. Note that now any SyncW[i] is broken into two bytes: each is accessed by either specST or specLD. This still allows up to 256 processors.

The other case is still sound: when the accesses to TLS metadata are interleaved in memory ordering, the framework detects either a RAW or a WAR violation (we have shown this already). Thus, it is of no consequence that x's value was inconsistent.

[1] In contrast, Pattern 2 with x and y subwords of the same word has been observed to be inconsistent; $\mathtt{t} = \mathtt{W}$ and $\mathtt{u} = \mathtt{W}$ does not imply $\mathtt{t} = \mathtt{u}$ (not transitive).

4. PERFORMANCE RESULTS

This section aims at demonstrating the following: *First*, our technique works well on code with regular patterns, even when most accesses use speculative support. We obtain on average 74% of the speed-up of the hand-parallelised code: 65% when SpLIP is used alone and 82% when SpLIP is used in combination with SpRO, on disjoint memory partitions. Other work has shown ratios up to 71%; however, this has only been for workloads where static analyses can identify a significant number of accesses as thread private.

Second, SpLIP performs reasonably well even in the absence of regular access. The speed-ups on three graph-based applications is on average within 66% of that of hand-parallelised code.

Third, under rare WAW/WAR dependencies, SpLIP performs in general better than a serial-commit implementation, because the latter may exhibit (i) significant overhead in satisfying iteration-independent RAW dependencies and (ii) limited scalability in numbers of processors.

We first describe the testing methodology and then present and comment on the speed-up results. All the tests were performed on a two quad-core AMD Opteron processors machine – model 2347 HE, 8 cores running at 1.9 GHz – with 16Gb of RAM memory, running Linux (Fedora Core 8). We used the *gcc-4.3.2* compiler at -O3 optimisation level, and the *pthread* library. We have checked that *gcc* is generating the correct movw and movq instructions in the order required by the implementation of specLD/ST.

4.1 Testing Methodology

We tested seven applications from three benchmarks: SciMark2, BYTEmark and JOlden. We chose to select only applications whose *loop-kernels* exhibit significant amount of parallelism; thus parallelising the kernel significantly improves the total application run time. Where needed, we break down the kernel into structurally different loops that we test separately. We did not consider the initialisation phase, since we found it either trivial or not parallel enough. JOlden is written in Java; we have re-written in C++. The tests did not exhibit indirect function calls, hence applying speculation was safe – see the concluding discussion of Section 3.3.

The tests exhibiting regular patterns are: (i) Idea, where Cipher and DeKey stand for the main loops involved in the encryption and decryption algorithms, (ii) SparMult that implements a multiplication algorithm over sparse matrices, (iii) FFT that implements the Fast Fourier Transform algorithm, and (iv) NeuralNet that simulates a back-propagation neural network, whose loops structurally fall in one of two classes: NeuralNetFW that processes the forward pass through the output layer, and NeuralNetBW that adjusts the weights of the output layer.

We test three graph-based applications with no regular patterns: (i) TSP – Karp's travelling-salesman problem algorithm, (ii) EM3D – the propagation of electromagnetic waves through 3D objects, and (iii) BH – Barnes-Hut's hierarchical force-calculation algorithm.

Col1	Col2	Col3	Col4	Col5	Col6	Col7	Col8	Col9	Col10
Test	P=1	P=2	P=4	P=6	P=8	R	F/L/O	$8R^{+}$	R
IdeaDeKy^{RO}	1:.57:.37	1.93:1.12:0.75	3.80:2.24:**1.36**	5.19:3.33:1.36	**6.36:4.38**:1.35	0	3.53:4.01:5.19	3.53	16
IdeaDeKy	1:.53:.28	1.93:1.05:0.62	3.80:2.05:1.19	5.19:3.05:**1.35**	**6.36:3.80**:1.35	0	2.77:3.23:4.87	3.65	16
IdeaCiph^{RO}	1:.79:.80	1.98:1.57:1.59	3.91:3.07:3.13	5.69:4.61:4.66	**6.79:5.88:5.86**	0	5.69:5.95:6.01	5.42	3
IdeaCiph	1:.36:.30	1.98:0.69:0.60	3.91:1.38:1.11	5.69:2.06:1.73	**6.79:2.76:2.28**	0	0.87:0.75:3.33	2.58	3
SparMult^{RO}	1:.78:.75	1.75:1.46:1.43	2.72:2.51:2.40	**2.81:2.79:2.72**	2.64:2.60:2.58	0	2.59:2.60:2.80	2.58	16
SparMult	1:.28:.24	1.75:0.55:0.46	2.72:1.10:0.90	**2.81**:1.62:1.35	2.64:**2.13:1.93**	0	0.19:0.60:2.19	2.04	16
NeuNetFW^{RO}	1:.82:.83	1.80:1.59:1.52	**3.14:2.87:2.80**	2.46:2.38:2.41	2.31:2.29:2.28	0	2.27:2.28:3.19	2.16	2
NeuNetFW	1:.40:.34	1.80:0.77:0.71	**3.14**:1.43:1.30	2.46:1.77:1.40	2.31:**2.09:1.93**	0	0.78:0.13:2.51	1.88	2
NeuNetBW	1:.26:.23	1.90:0.51:0.44	**3.09**:0.97:0.74	2.61:**1.68:1.10**	2.48:1.56:1.02	0	0.89:0.96:2.25	1.49	39
FFT	.8:.66:.54	1.45:1.25:**1.03**	2.84:2.47:1.03	3.28:2.95:1.02	**3.91:3.51**:1.01	0	2.92:2.83:3.72	3.21	3
$TSP_{Mem:2^{16}}$	1:.57:.02	1.98:1.23:0.03	3.85:2.55:0.06	5.27:3.48:0.07	**6.76:4.80:0.09**	0	2.08:2.79:5.80	4.12	22
$EM3D_{Mem:2^{15}}$	.8:.25:.01	1.41:0.49:0.03	**2.51**:0.94:0.06	2.30:1.28:0.09	2.16:**1.44:0.13**	32	0.77:0.81:2.17	0.78	280
$BH_{Mem:2^{14}}$	1:.68:.65	1.90:1.53:1.47	3.62:2.79:2.69	5.61:4.08:3.86	**7.38:5.29:4.82**	1	3.53:2.12:5.30	4.68	5
$BH_{Mem:2^{21}}$	1:.40:.27	1.90:0.78:0.58	3.62:1.53:1.18	5.61:2.25:1.75	**7.38:2.84:2.28**	0	2.12:1.75:3.08	2.22	4

Table 2: Speed-ups on up to 8 Processors. Col1 shows the tested loops. X^{RO} means that we use both SpRO and SpLIP for test X. $X_{SpMem:2^y}$ means that dependency-tracking vectors have 2^y entries. Entries in Col2–6 are of form x:y:z where x is the non-speculative speed-up – sequential time divided by parallel time, while y and z are the speed-up of SpLIP and SpLSC [14] (serial-commit), respectively. P denotes the number of processors used. Col7 shows the number of rollbacks on 8 processors. Col8 shows SpLIP's speed-up when mfence or CAS instructions are used. The last number is the speed-up under a 0-cost mfence. Col9 and Col10 show SpLIP's speed-ups and number of rollbacks when 0.5% of the iterations yield dependency violations.

To apply TLS, we use the PolyLibTLS library [14] that encapsulates SpLIP, SpRO and SpLSC – a serial-commit solution. We also use dynamic analysis [13] to compute the iteration/memory space partitioning, to create suitable iteration granularity (thousands of instruction range), and to compute the hash function of SpLIP. The latter has the form (i) hash(x)=x%Q with large Q for graph-based applications, and (ii) hash(x)=((x-a)/q)%Q with small Q for regular applications. With the selected loops, the rest was done manually. Array/pointer accesses are assumed to alias everywhere and require TLS support: x=y becomes specST(&x,specLD(&y,i),i); here i is the iteration number. Privatisable scalars are not protected via TLS. Accesses that would generate many violations (e.g. iterators) are computed serially, but are protected via TLS. Threads, at most eight, are re-used via a thread-pool. The non-speculative parallel version corresponds to a straightforward (sometimes unsafe) parallelisation of the sequential code.

We diverged from the public benchmarks by increasing the array sizes to make the asymptotic behaviour more easily measurable. For EM3D we changed a random distribution of nodes that generated too many dependency violations into a more structured one, which still features the occasional rollback. For TSP, we changed the recursive tree-traversal implementation into one based on iterators, and hence exploitable via TLS. (The recursive implementation is exploitable via method-level speculation, but only on 2 processors.)

4.2 Speed-up Results

Entries in columns **Col2–6** in Table 2 have the shape x:y:z. x is the hand-parallelised, *non-speculative* speed-up – computed as the ratio between the sequential and parallel execution timings. y is SpLIP's speed-up (this paper's implementation). z denotes the speed-up obtained by SpLSC [14] – a lightweight, serial-commit implementation that can exploit regular patterns in the same manner as SpLIP. Since it exhibits similar memory overhead we use it to compare against serial-commit implementations. P denotes the number of processors used. We specify that the performance results reported in companion papers [13, 14] were assuming a sequentially-consistent hardware – i.e. 0-cost mfence – and thus give an upper bound for efficiency. The results reported here take into account the overhead corresponding to the changes that make the algorithms sound under the Intel IA-32 memory model. The most significant difference is for SpLSC on test $IdeaDeKey^{RO}$, where these changes make the serial commit phase more expensive, which in turn impacts on the scalability of the algorithm. The performance difference between the hand-parallelised code and the speculative one is dominated by the overhead introduced by the specLD/ST functions.

Comparing against SpLSC we observe that when the number of writes per iteration is significant then: *First*, SpLSC's *read* operation *increases* in cost since, if a lightweight test fails to guarantee that the current iteration has not updated that location, the write-buffer is searched; if no update is found, then the non-speculative value is returned. We referred to this as the overhead associated with same-iteration RAW. This overhead is significant for IdeaDeKey and NeuralNetBW, and huge for TSP and EM3D.

Second, SpLSC's serial commit phase *prevents scalable speed-up* on IdeaDeKey and FFT. We believe all serial-commit implementations will behave similarly.

The use of SpRO significantly improves performance on the tests IdeaCipher and SparMult, because they exhibit read-access races – even though they do not generate false-positive violations, the cache-conflicts at LdVct/StVct level are expensive. The TLS memory overhead is within $< 1\%$ of the original storage for regular applications and between 10–40% for the graph-based ones. The last two BH tests shows the benefits of using a reduced memory-overhead via a perfect-hash-like mapping between memory locations and indexes in dependency-tracking vectors. This benefit is even more pronounced under regular patterns.

We also observe that the gap between the optimal and the SpLIP speed-ups narrows with the number of processors. This is to be expected since (i) writing Ov for the TLS time-overhead on 1 processor, we expect the overhead on P processors to be Ov/P and (ii) due to memory bandwidth issues, the optimal code may reach its upper limits for fewer processors (4 for NeuralNetBW and EM3D) than SpLIP, which continues to improve to 8 processors.

Column **Col7** in Table 2 shows the number of rollbacks on 8 processors for SpLIP.

Column **Col8** shows the speed-up obtained when (i) mfence instructions are used for memory ordering; (ii) CAS instructions are used to ensure LdVct/StVct[i] monotonicity; and (iii) a sequentially consistent memory is unsafely assumed – i.e. a 0-cost mfence. This is an estimate of the maximal TLS speed-up. We

observe that the cost of both locking, and surprisingly, `mfences` is high, hence neither is suitable for TLS.

Columns **Col9** and **Col10** show SpLIP's speed-ups and number of rollbacks when 2% of iterations generate violations. This translates into an additional 0.2–1% of rollbacks.

5. CONCLUSIONS

This paper has presented an in-place TLS implementation that is more scalable than the serial-commit ones. The implementation features a constant speculative operation instruction overhead, which is comparable with the ideal case of previous solutions, and can exploit applications' regular patterns resulting in a small speculative storage with cache-friendly layout. We have shown how to safely eliminate both locking and memory ordering instructions from the implementation of the speculative operation. We found these mechanisms too expensive for TLS. The penalty resides in relaxing the system's precision: RAW, WAW, WAR and false-positive dependencies may cause rollbacks.

While other software-based solutions yield good speed-ups only when there is enough independent computation that does not require speculative support, we have shown that our solution achieves significant speed-ups, as high as 588%, even when nearly all accesses are protected with speculative support. Tested on nine loops that implement the loop kernels of seven, both regular and graph-based, applications, our implementation realises on average 77% of the speed-up of the hand-parallelised, non-speculative program.

6. ACKNOWLEDGMENTS

The authors thank Susmit Sarkar and Peter Sewell for their help with testing the memory ordering patterns discussed in Section 3.5 on one hardware configuration.

7. REFERENCES

[1] V. S. Adve. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Int. Conf. High Perf. Comp. (SC)*, Nov 1995.

[2] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Int. Symp. Comp. Arch. (ISCA)*, Jun 2006.

[3] M. K. Chen and K. Olukotun. Exploiting Method Level Parallelism in Single Threaded Java Programs. In *Int. Conf. Par. Arch. and Comp. Tech. (PACT)*. Oct 1998.

[4] M. K. Chen and K. Olukotun. The JRPM System for Dynamically Parallelizing Java Programs. In *Int. Symp. Comp. Arch. (ISCA)*, Jun 2003.

[5] M. Cintra and D. R. Llanos. Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In *Int. Symp. Princ. Pract. of Par. Prg. (PPoPP)*, Jun 2003.

[6] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Int. Par. Distr. Proc. Symp. (IPDPS)*, Apr 2002.

[7] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Int. Conf. Arch. Sup. Prg. Lang. Op. Sys. (ASPLOS)*, Oct 1998.

[8] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Int. Conf. Prg. Lang. Design Impl. (PLDI)*, Jun 2006.

[9] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, chapter 7. In `http://download.intel.com/design/processor/manuals/253668.pdf`, Sep 2008.

[10] I. H. Kazi and D. J. Lilja. Coarsed-Grained Thread Pipelining: A Speculative Parallel Execution Model for Shared-Memory Multiprocessors. *IEEE Tran. Par. Distr. Sys.*, 12(9), Sep 2001.

[11] S. W. Kim, Chong-Liang Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution. In *Int. Symp. Princ. Pract. of Par. Prg. (PPoPP)*, Jun 2001.

[12] F. Masdupuy. Array Operations Abstraction Using Semantic Analysis of Trapezoid Congruences. In *Int. Conf. Supercomputing (ICS)*, Jul 1992.

[13] C. E. Oancea and A. Mycroft. Set-Congruence Dynamic Analysis for Software TLS. In *Lang. Comp. Par. Comp. (LCPC)*, Aug 2008.

[14] C. E. Oancea and A. Mycroft. Software Thread-Level Speculation – An Optimistic Library Implementation. In *Int. Worksh. Multi-Core Soft. Eng. (IWMSE)*, Jan 2008.

[15] C. J. F. Pickett and C. Verbrugge. Software Thread Level Speculation for the Java Language and Virtual Machine Environment. In *Lang. Comp. Par. Comp. (LCPC)*, Oct 2005.

[16] L. Rauchwerger, and N. M. Amato, and D. A. Padua. A Scalable Method for Run-Time Loop Parallelization. In *Int. Conf. Supercomputing (ICS)*, Jul 1995.

[17] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Parallel and Distributed System*, 10 No 2(2):160–199, Feb 1999.

[18] P. Rundberg and P. Stenström. An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. *The Journal of Instr.-Level Par.*, 1999.

[19] S. Rus, M. Pennings, and L. Rauchwerger Sensitivity Analysis for Automatic Parallelization on Multi-Cores. In *Int. Conf. Supercomputing (ICS)*, Jun 2007.

[20] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. In *Int. Journal of Par. Prg., 31(4)*, pages 251–283, Aug 2003.

[21] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Int. Symp. Princ. Pract. of Par. Prg. (PPoPP)*, Mar 2006.

[22] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The Semantics of X86-CC Multiprocessor Machine Code. In *Int. Symp. Princ. of Prg. Lang. (POPL)*, Jan 2009.

[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Int. Symp. Comp. Arch. (ISCA)*, Jun 1995.

[24] J. G. Steffan, C. G. Colohan, A. Zhai, and T. Mowry. A Scalable Approach for Thread Level Speculation. In *Int. Symp. Comp. Arch. (ISCA)*, Jun 2000.

[25] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse The MAJC Architecture: A Synthesis of Parallelism and Scalability. In *Symp. Microarch. (MICRO)*, Dec 2000.

[26] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *Int. Conf. Obj.-Orien. Prg. Sys. Lang. Appl. (OOPSLA)*, Oct 2006.

[27] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *Int. Conf. Arch. Sup. Prg. Lang. Op. Sys. (ASPLOS)*, Oct 2002.

[28] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Symp. Microarch. (MICRO)*, Nov 2002.