Compiler-Driven Software Speculation for Thread-Level Parallelism

PARASKEVAS YIAPANIS, GAVIN BROWN, and MIKEL LUJÁN, University of Manchester

Current parallelizing compilers can tackle applications exercising regular access patterns on arrays or affine indices, where data dependencies can be expressed in a linear form. Unfortunately, there are cases that independence between statements of code cannot be guaranteed and thus the compiler conservatively produces sequential code. Programs that involve extensive pointer use, irregular access patterns, and loops with unknown number of iterations are examples of such cases. This limits the extraction of parallelism in cases where dependencies are rarely or never triggered at runtime. Speculative parallelism refers to methods employed during program execution that aim to produce a valid parallel execution schedule for programs immune to static parallelization. The motivation for this article is to review recent developments in the area of compiler-driven software speculation for thread-level parallelism and how they came about. The article is divided into two parts. In the first part the fundamentals of speculative parallelization for thread-level parallelism are explained along with a design choice categorization for implementing such systems. Design choices include the ways speculative data is handled, how data dependence violations are detected and resolved, how the correct data are made visible to other threads, or how speculative threads are scheduled. The second part is structured around those design choices providing the advances and trends in the literature with reference to key developments in the area. Although the focus of the article is in software speculative parallelization, a section is dedicated for providing the interested reader with pointers and references for exploring similar topics such as hardware thread-level speculation, transactional memory, and automatic parallelization.

Categories and Subject Descriptors: D.1.3 [**Concurrent Programming**]: Parallel programming, Distributed programming; D.2.2 [**Design Tools and Techniques**]: Software libraries; D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages; D.3.5 [**Processors**]: Compilers, Optimization, Runtime environments

General Terms: Languages, Design, Algorithms, Performance

Additional Key Words and Phrases: Thread-level speculation, speculative parallelization, runtime parallelization, automatic parallelization, multicore processors

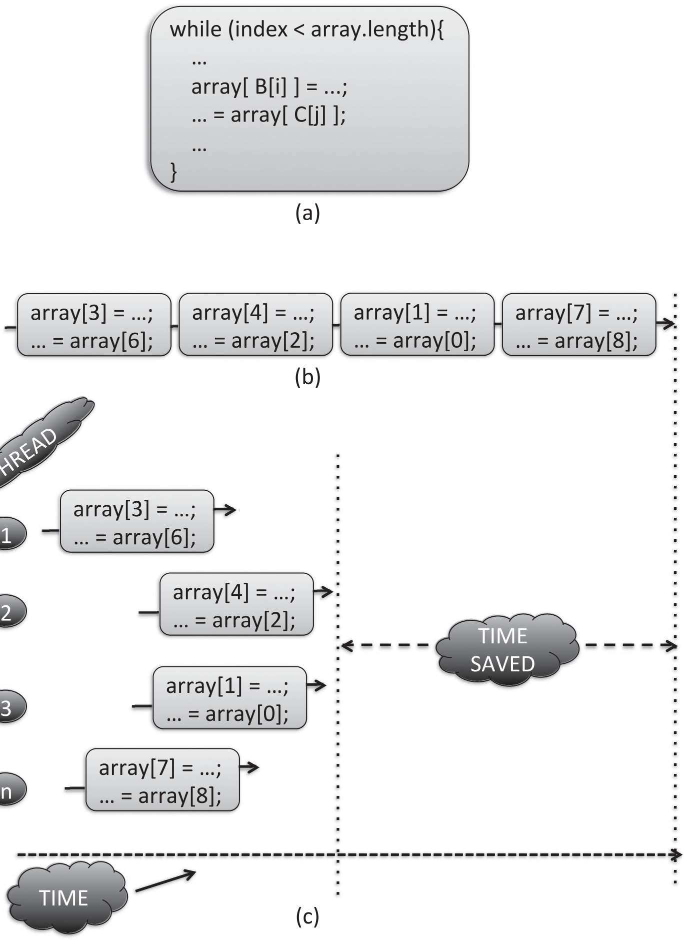
ACM Reference Format:
Paraskevas Yiapanis, Gavin Brown, and Mikel Luján. 2015. Compiler-driven software speculation for thread-level parallelism. ACM Trans. Program. Lang. Syst. 38, 2, Article 5 (December 2015), 45 pages.
DOI: http://dx.doi.org/10.1145/2821505

1. INTRODUCTION

In recent years, multicore chips became the standard configuration in commercial computing. In order to harness the computing power they have to offer, applications need to be structured in such a way that will yield efficient utilization of the available resources. Parallel programming accomplishes that by dividing the computation across the available processors (or threads), yet this process involves experienced

This research was conducted with support from the UK Engineering and Physical Sciences Research Council (EPSRC), on grants PAMELA EP/K008730/1, AnyScale Apps EP/L000725/1, and iTLS EP/G000662/1. Dr. Luján is supported by a Royal Society University Research Fellowship.
Authors' addresses: P. Yiapanis, G. Brown, and M. Luján, School of Computer Science, University of Manchester, Oxford Road, M13 9PL; emails: pyiapa@gmail.com, {gavin.brown, mikel.lujan}@manchester.ac.uk.

ACM 0164-0925/2015/12-ART5 $15.00
DOI: http://dx.doi.org/10.1145/2821505



while (index < array.length){
...
array[B[i]] = ...;
... = array[C[j]];
...
}
(a)
array[3] = ...; ... = array[6];
array[4] = ...; ... = array[2];
array[1] = ...; ... = array[0];
array[7] = ...; ... = array[8];
(b)
THREAD
1
2
3
n
TIME SAVED
TIME
(c)

Fig. 1. (a) Code fragment of loop to be parallelized. (b) Sequential execution. (c) Sample parallel execution.

software engineers. An ideal scenario is when the compiler automatically can restructure a sequential program into a faster parallel version offline. Although promising and inexpensive, in terms of application performance, offline automatic parallelization is sometimes limited mainly due to insufficient runtime information or the inability of the parallelizing compiler to perform the transformation due to complex interprocedural relationships. For example, it is difficult to perform static dependence analysis on code that makes extensive use of pointers, which is typical for modern languages such as C++ or Java. More sophisticated memory dependence analysis (such as *points-to* analysis [Nystrom et al. 2004]) can help, but parallelization often fails due to unresolved memory accesses. Furthermore, loops with unknown number of iterations make it hard to parallelize since there is no information on how to schedule the loop (e.g., the loop might terminate abruptly due to a runtime condition). Also, when subscripted subscripts are used to access array elements, the actual memory locations may not be available until runtime. This can be illustrated with the following example: assume that one wishes to parallelize the loop shown in Figure 1(a). Similarly, assume that integer arrays B and C are populated at runtime. That is, there is no feasible means of performing any static analysis (manual or automatic) to prove correct parallel executions by eliminating the possibility of data dependencies across threads. Therefore, conventional parallelizing compilers must conservatively produce sequential code for the loop in order to guarantee correct execution. Consider now the instance of sequential execution shown in Figure 1(b). Clearly, the values populated for the array indices did not yield any data dependencies amongst them, and thus, the compiler could have generated a parallel code such as the one in Figure 1(c), and allow the application to run in parallel (assume for simplicity one loop iteration per thread). However, this was not possible since the indices were populated at runtime.

One solution to the preceding problem is to proceed with parallel execution speculatively until sufficient information is collected, providing mechanisms to maintain sequential program correctness. Such a solution is known in the research literature as *Speculative Parallelization*, *Thread-Level Speculation*, or *TLS* in short. All three terms will be used interchangeably throughout the rest of the article. Speculative parallelism is also possible at the instruction level but this article is focused on the thread level. For the rest of the article thread-level parallelism is assumed unless otherwise specified.

The first part of this article, "Fundamentals of Speculative Parallelization," explains speculative parallelization and its mechanics as well as the major design requirements for such an execution model. Furthermore, an overview of an earlier model for nonspeculative runtime parallelization, identified as *Inspector/Executor*, is briefly discussed.

PART I: FUNDAMENTALS OF SPECULATIVE PARALLELIZATION

2. SPECULATIVE PARALLELIZATION

2.1. Brief Description

Speculative Parallelization (Thread-Level Speculation—TLS) is a technique that facilitates automatic parallelization via optimistic execution of potentially independent threads. It is particularly useful where there exist ambiguous dependencies that cannot be resolved statically by the compiler. In the previous example of Figure 1, a TLS system circumvents the conservatism of a static compiler by executing the threads (which are formed by loop iterations in this case) in parallel assuming that the run-time values of $B[i]$ and $C[j]$ will not trigger any cross-thread conflicts. For instance, in this case TLS would execute the loop iterations in parallel, while at the same time underlying mechanisms would monitor every speculative access to ensure that the parallel execution will produce the same results as if the program was executed sequentially. In one design, any speculative memory updates are stored locally to the thread and eventually written back to main memory given correct execution. Figure 2(a) shows the case where all speculative threads executed successfully and thus are allowed to *retire* or *commit* by propagating the buffered updates back to main memory. Sometimes we have the case of a memory dependency like the one shown in Figure 2(b). In this case, Thread 3 has loaded a value that was not produced by the correct store.[1] This action causes what is known as a *Read-After-Write (RAW)* data dependence violation. As a result, the offending threads need to *squash* by initiating the *rollback* procedure (in this case discard any buffered updates) and reexecute in the correct order, that is, Thread 3 before Thread n (see Figure 2(c)).

The majority of the TLS systems this article describes address only data dependence speculation. Data dependence speculation breaks dataflow dependencies between memory operations by guessing that they will access different locations, thereby

[1]Assuming that iterations are assigned to threads in order, Thread 3 should have read a value produced by itself or a previous speculative thread. Instead, Thread 3 consumed a value produced by a future thread and that violates sequential program semantics.

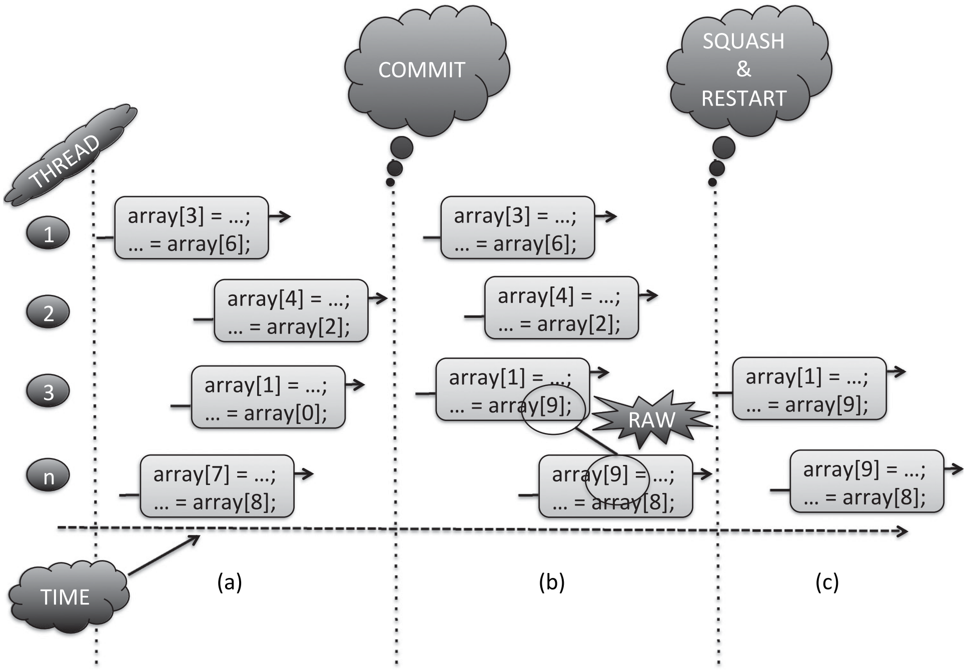


Fig. 2. (a) Speculative loop execution without dependencies. (b) Speculative loop execution with dependency. (c) Reexecution of offending threads.

making the operations independent of one another. This is also known as *alias speculation* (or memory address disambiguation). There are also two other important types of speculation employed by parallelizing compilers: *value speculation* and *control speculation*.

Control speculation [August et al. 1998] is used to execute unresolved edges in the control flow graph. It breaks control dependencies that exist between branches and other operations by predicting that a branch will go in a particular direction, thereby making an operation's execution independent of the branch. A misspeculation is detected if the predicted value of a branch condition does not match the actual value at runtime.

Value speculation [Prabhu and Olukotun 2005] speculates on dependencies using a predicted value instead of the actual value to satisfy the dependence. Whenever the dependence executes and the actual value is not the predicted value, misspeculation is flagged [Raman et al. 2010].

For the rest of the article, alias speculation is assumed unless otherwise explicitly stated for a particular TLS system.

2.2. Design Specification

Implementing the underlying mechanisms that will guarantee correct execution in TLS requires certain design decisions. From the brief description given previously in Section 2.1 the main requirements for supporting speculative parallelization can be categorized as follows (also identified by Cintra and Llanos [2003, 2005]):

—**Metadata management**. A way to know which memory locations are accessed and by which threads. This will facilitate identifying whether or not any threads have accessed memory locations in a way that invalidates speculation.

—**Version management**. A way to manage speculative data. When threads execute speculatively, different versions of the data are produced. A mechanism is required to manage temporary (speculative) data and maintain consistency among operations.
—**Detect data dependence violations**. A way to identify potential data dependence violations.
—**Commit and rollback operations**. A way to maintain main memory at a correct state. That is, to be able to commit the correct values and rollback execution to a correct state when necessary.
—**Scheduling speculative threads**. An efficient way to schedule speculative work and threads.

The following sections provide an overview of each requirement and discuss the main implementation options available.

2.2.1. Metadata. Speculative execution proceeds in parallel optimistically with the hope that the code executes correctly without violating the sequential program semantics. If the program happens to be fully parallel (i.e., without any data dependencies across parallel threads), then it will execute successfully and values in main memory will be correct. But if a data dependency did exist across threads, then the values in main memory are likely to be incorrect if the threads were executed out of order. Thus, the system must take precautions in the event of potential failures so that memory is always left in a correct and consistent state. As we will discuss in more detail in Section 2.2.2, there are two main implementations to accomplish that. One way is to have speculative threads putting aside any tentative stores and only placing them to main memory if execution is proven to be correct (deferred updates). Another way is to put aside the current memory values and allow speculative threads to store directly any updates to main memory (in-place updates). This allows the TLS underlying system to monitor memory accesses and revert memory back to a correct state when required. Furthermore, a speculative thread needs a way to know when it is accessing the same memory location as another speculative thread and whether or not the other thread has performed an update there, effectively allowing the detection of memory conflicts.

To enable a the system to monitor memory accesses, some auxiliary data (referred to in this work as *metadata*) are required to be associated with the user data structures as well as the speculative threads. Figure 3 shows one way in which speculative memory access information can be kept in the system. This scheme is valid for systems implementing both in-place and deferred updates. The various ways metadata can be exploited are explored later under the “Version Management” section.

Metadata to Reflect User Data Structures. Figure 3(a) illustrates how metadata are arranged to reflect the user data that can potentially be accessed in a speculative way. Every user datum that can be potentially accessed speculatively (i.e., a datum that is shared among speculative threads) is associated with a record. This record represents actions on user data by a given speculative thread. For instance, a general TLS system could maintain information about which particular thread is operating, reading, or writing at a given moment in time on a given user memory location. The thread actions are denoted with the labels “Lock Tid,” “Load,” or “Store” in Figure 3(a). “Lock Tid” is used by a thread in order to acquire *exclusive ownership* of a particular memory location. That is, by using a locking primitive a thread stores a unique thread identifier (Tid) in the “Lock Tid” column and proclaims ownership of the location associated with that record. This action prevents another speculative thread from performing an operation at the same memory location simultaneously with the current owner thread allowing predictable results. The owner thread is allowed to perform a speculative operation, either a load or a store, by setting the corresponding record with its unique thread

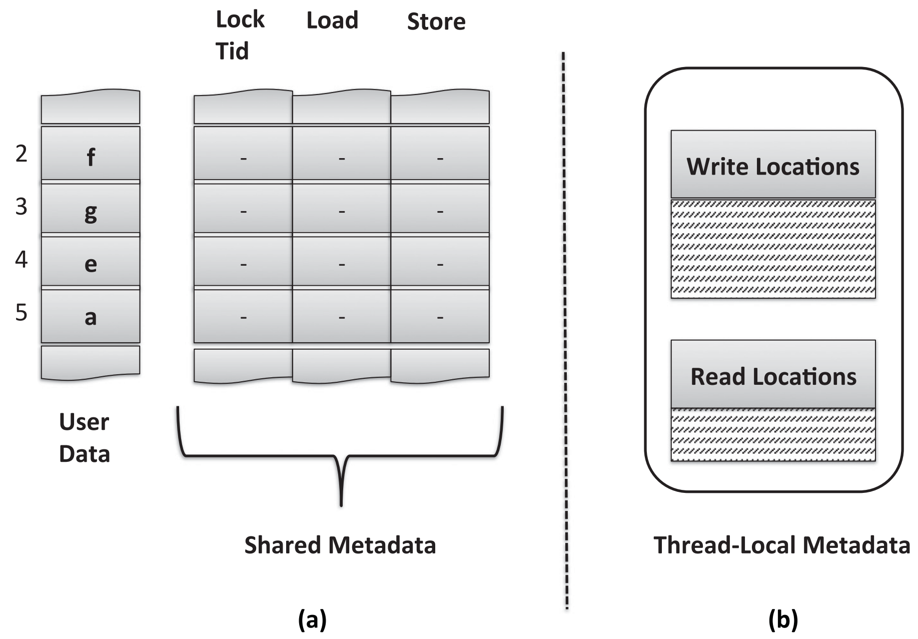


Fig. 3. (a) Metadata associated with the user data structure. This allows tracking of speculative threads' memory accesses to ensure correct execution. Since threads speculate that there will be no memory conflicts between them, a mechanism is required to identify misspeculations in order to ensure that main memory is left at a valid state. A speculative thread wishing to access memory location x can check the "Lock Tid" field for x to examine whether another thread is currently accessing the same location or not. Once the lock is acquired a thread can indicate whether it has performed a Load or a Store by setting the corresponding field in the locked record. (b) Metadata associated with every speculative thread. This allows a speculative thread to keep track of memory locations accessed by it. Also, any tentative writes (or current memory values—depending on the implementation) can be saved there.

identifier. The lock is released immediately after the speculative operation takes place in order to allow other threads to examine the information on that location.

Metadata for Speculative Threads. Apart from the metadata required to reflect any user memory location, speculative threads also require some extra information regarding the locations they access during their lifetime. For example, Figure 3(b) implies that information is kept regarding which memory locations have been accessed for reading or writing. When a speculative thread performs a read operation on a memory location, the address of that location is saved in a thread-local set that contains all locations read by that thread while executing a particular speculative region. This set is known as the *read-set*. Similarly, when a thread performs a speculative write on a memory location, that location is recorded on the thread-local *write-set*. In the case of deferred-updates, the write-set usually contains also the speculative value produced by the thread for the address accessed for writing. For in-place updates, the original value in memory is recorded before memory is updated with the speculative value. The reason for that is discussed later in the next section ("Version Management").

2.2.2. Version Management. While threads execute speculative code, different versions of data are being produced. *Version Management* refers to the way those different versions are maintained by the TLS system. Typically, there are two major approaches for that: *Lazy Version Management (LVM)*, also known as *deferred updates* and *Eager Version Management (EVM)*, also known as *in-place updates*.

When LVM is used as a choice, speculative threads require at least a write buffer per thread in order to keep any tentative stores. Speculative loads search first the thread's local buffer in case they find an associated value for that location there. If not, the value needs to be loaded from memory. Speculative stores just need to add or update the corresponding value in the local buffer. At the end of a thread's speculative execution (and provided that there was no conflict for this thread) the results from the local buffer are propagated to main memory to make visible the updates to other threads. In the case of a conflict, the speculative thread only needs to discard its local write buffer since there was no modification of the actual data in memory.

EVM systems, on the other hand, update memory locations directly when the speculative store occurs rather than delaying the action. This involves having a buffer that preserves the original value of the memory location just before the update. This buffer is known in the literature as the *undo log* since in the case of a conflict the log is used to restore the memory back to a correct state. Speculative loads can use the values from memory, since the new values are already there. Upon successful commit, the thread simply discards the undo log without requiring any value propagation as in LVM.

2.2.3. Conflict Detection. A conflict can occur when two or more speculative threads access the same memory location in a way that causes a data dependency violation. Depending on the version management system used, different actions may or may not cause violations. There are three types of data dependencies: *flow*, *anti*, and *output dependencies*, which give rise to RAW, *Write-After-Read (WAR)*, and *Write-After-Write (WAW)* hazards,[2] respectively. A RAW violation is caused when a thread loads a value that was not produced by itself. WAR and WAW violations arise due to reuse of memory locations. A system that uses LVM does not need to worry about WAR and WAW dependence violations since the updates are buffered and speculative loads use those instead. This is somewhat similar to the *Register Renaming* action taken at the hardware level to prevent those kind of hazards. In contrast, EVM has to take precautions for WAR and WAW dependence violations since the values in memory are always up to date. Nevertheless, both EVM as well as LVM systems need to be observant for RAW violations.

There are two types of conflict detection: *Lazy Conflict Detection (LCD)* and *Eager Conflict Detection (ECD)*. LCD implies that threads may be allowed to run through their respective speculative code without checking for conflicts on every access. Conflict detection can occur at a later stage as long as that happens before thread commit. In this way, eager checks during execution are eliminated. ECD checks for conflicts usually on every speculative access in order to catch any violations as soon as they arise. The idea here is to prevent any wasted work after a conflict has happened.

Another aspect is the granularity of conflict detection. Conflicts may be detected at a fine-grained level (e.g., word level) or at a coarse-grain level (e.g., object level). This introduces a notion of false sharing in which a TLS implementation treads two speculative threads as conflicting even though they have accessed distinct locations. Techniques using value-based (see Section 7.1.7) checking provide one way to avoid false conflicts.

2.2.4. Commit and Rollback. If the TLS system confirms that a group of speculative threads had correct execution the commit phase allows those threads to provide memory with the correct values. A system using LVM typically commits those values sequentially. That is, the threads propagate their speculative (currently buffered) values

[2]RAW, WAR, and WAW hazards are with respect to sequential program order.

to main memory one by one in order[3] of speculation (i.e., starting from the least and moving toward the most speculative thread). The reason is that, during speculation some threads might have triggered a conflict due to a RAW dependence violation and thus had to wait for the correct value to be produced. A less speculative thread is always more correct than a more speculative thread. If the threads are allowed to commit out of order, then this assumption is lost and the wrong values may be populated. On the other hand, since on an eager management system updates are performed in place, at the end of speculative execution the committed values are already in the correct place in memory. Thus, commit across multiple speculative threads occurs in parallel.

When a conflict arises, main memory must be restored to the last known correct state. This involves squashing the offending threads and rollbacking speculative state. With LVM this procedure is very simple, effective, and threads can operate in parallel. Since LVM systems buffer speculative updates, each thread is allowed to proceed in parallel with each other and discard their speculative values. For an EVM system the rollback process is more time consuming. Since speculative stores are written in place, main memory is already "polluted" with incorrect values by the time a conflict is detected. Thus, memory must be restored to the latest known correct state.

In cases where there is a frequently recurring RAW dependence between speculative threads, it is possible for the compiler to insert synchronization and enforce the dependency by having the reader explicitly block on the writer. However, this creates a dependence chain across the threads that may possibly limit the parallel speedup [Steffan 2003].

2.2.5. Scheduling. The way iterations are scheduled to run across the available threads can have significant impact in the final performance. Traditional scheduling possibilities include *static* and *dynamic scheduling*. Static scheduling partitions the loop into equal chunks[4] of iterations based on the number of available threads. The thread that will execute a particular chunk is decided statically. In contrast, dynamic scheduling allows those chunks to be assigned to threads at runtime. The simplest dynamic scheduling technique is *self-scheduling* [Tang and Yew 1986], where all the chunks are unit size.

While the static and dynamic scheduling techniques are effective for parallel loops, in Cintra and Llanos [2003, 2005] the authors argue that they are not very well suited for TLS. Static scheduling will perform poorly when there is load imbalance and frequent data dependence violations. Dynamic scheduling is not practical when the number of iterations is very large as the memory overhead of the speculative structures is effectively proportional to the number of iterations. This is because each thread requires its own set of data structures for speculative execution.

A different scheduling technique known as *Sliding Window* (originally introduced by Dang et al. [2002]) was evaluated amongst different scheduling techniques by Cintra and Llanos [2003, 2005] and found to be a good alternative for TLS. Under sliding window (see Figure 4), chunks of iterations are assigned into windows of size W. At any time, there are only W active threads (thus no overlapping of windows execution) and the memory overhead is proportional to W, regardless of the total number of chunks. The window moves forward (slides) when all iterations in the window have finished. This allows better load balancing, decrease in likelihood of dependence violation, and better decoupling of memory overhead [Cintra and Llanos 2003, 2005].

[3]Assume n iterations are mapped on n threads in order (thread 0 has iteration 0, thread 1 has iteration 1, and so on). This specifies a speculation order in which thread k is less speculative than thread $k+1$, where $0 < k < n$.
[4]Chunks of iterations form units of commit in this case.

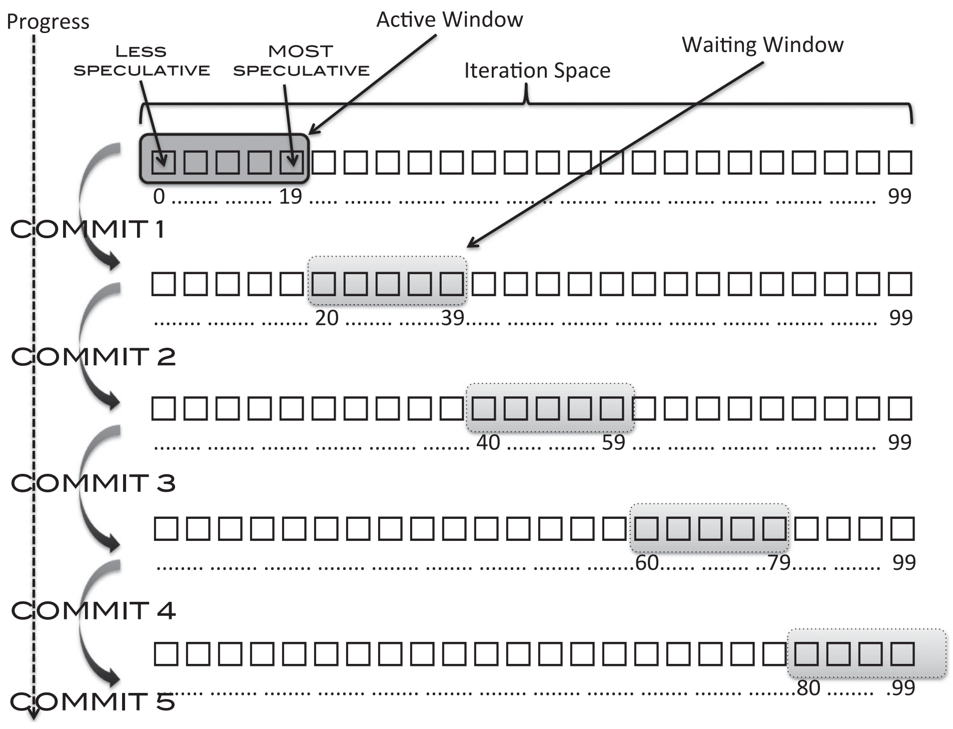


Fig. 4. Sliding window scheduling.

Figure 4 shows a speculative execution instance that uses the sliding window scheduling policy. Iterations are mapped to windows of size W. The slots in the window dictate the speculation order. That is, slot 0 is the least speculative one and slot $W - 1$ the most speculative (e.g., iterations assigned to slot 0 are less speculative than iterations assigned to slot 1). Only one window is active at any given time. The next window will be activated only when all iterations from the previous window have committed.

3. INSPECTOR/EXECUTOR: A NONSPECULATIVE APPROACH TO RUNTIME PARALLELIZATION

Early work on runtime parallelization involved a nonspeculative technique known in the literature as *Inspector/Executor* [Zhu and Yew 1987; Salz et al. 1989; Salz and Mirchandaney 1991; Salz et al. 1991; Chen et al. 1994; Rauchwerger and Padua 1994b]. As the name implies, this method involves the generation of two versions of the loop to be parallelized during compilation. The first version, called *Inspector*, would simply execute a stripped version of the original loop that contains only accesses to shared mutable data (although implementations are free to incorporate as much of the backward slice as is necessary to materialize those memory addresses), investigating whether the loop contains any data dependencies that prevent it from being parallelized. The *Inspector* is not required to replicate all of the computation done by the original program (only memory accesses or a fraction of it), and thus could be executed quicker.

3.1. Work on Inspector/Executor

One of the most notable *Inspector/Executor* models in the runtime parallelization literature was proposed by Rauchwerger and Padua [1994b]. Their approach used the inspector to simply detect whether or not a loop is fully parallel. Such loops are known as *DOALL* loops. In Rauchwerger and Padua [1994b] each inspector thread would be allocated its own portion (chunk of iterations) of the stripped version of the original loop. Each thread will also make use of some thread-local auxiliary metadata (similar to the one in Figure 3(a) but without a "Lock Tid") reflecting each memory location of the shared mutable data that can be accessed during parallelization. In this simplified version, each inspector thread will mark any loads or stores performed during the execution of its portion of iterations. A location in the thread-local auxiliary metadata is marked to indicate the corresponding action (e.g., Load or Store) by a particular thread. At the end of the inspection phase, all inspector threads will check each others findings to ensure that the same memory location was not accessed by different threads in a way that violates the sequential semantics of the loop. For instance, if two threads perform a store at the same memory location but in the wrong order the resulting value would be unpredictable. The only valid way multiple threads can access the same memory location is if they just read from that location. Since reading from a memory location does not affect its value, then threads are safe to read in any order they wish from there. The second version, called *Executor*, will execute the loop in parallel across multiple threads, given that the inspector version indicated so. Otherwise the loop will be executed sequentially.

The focus of earlier attempts of the *Inspector/Executor*, such as the one proposed by Zhu and Yew [1987], was mainly on partially parallel loops and they were using the inspector phase to order the iterations in groups that contained parallel iterations. Within those groups, iterations could execute in parallel between them, but the groups themselves have to be separated using synchronization. Their scheme was divided in multiple stages with each stage including both an inspector and an executor. Every stage would gather information (inspection phase) of which iterations are allowed to execute in parallel without any conflicts and record this information into an auxiliary data structure. Then, the executor would execute those iterations in parallel with the aid of the auxiliary data structure. The next stage would do the same and the process would continue in a repetitive fashion until all iterations of the loop finish. Later work, such as Salz et al. [1991], provide more optimized versions of the *Inspector/Executor* technique by statically partitioning the iterations of the loop among processors and then reordering the iterations within each partition at runtime.

3.2. Weakness

As noted earlier, *Inspector/Executor* relies on extracting a smaller version of the loop to execute faster. The model was successful mainly in cases where an efficient inspector loop could be obtained [Rauchwerger 1998]. That is, if the inspector loop is not able to be decoupled sufficiently from the loop (i.e., small code replication), then the inspector becomes computationally expensive making it nearly equivalent to the loop itself. Mainly for this reason, research in runtime parallelization turned towards solutions employing speculation. Even though the *Inspector/Executor* model was not very successful it did form the basis of first research ideas in speculative parallelization [Rauchwerger and Padua 1994a, 1995].

PART II: ADVANCED TOPICS IN SPECULATIVE PARALLELIZATION

4. INTRODUCTION

The first part of this article introduced *speculative parallelization*, a way to parallelize an application at runtime. There has been numerous work in speculative parallelization over the last two decades. The second part, "Advanced Topics in Speculative Parallelization," identifies and delivers the most important ones while discussing various

Table I. The Design Points that Make Up a Speculative Parallelization System

TLS Design Choices
Version Management
Conflict Detection
Commit
Rollback
Execution Model
Work Scheduling
Metadata

Table II. Advances in the Literature of Speculative Parallelization

Year	Work
1995	`LRPD` test [Rauchwerger and Padua 1995]
1998	Gupta and Nim [Gupta and Nim 1998]
2000	`Softspec` [Bruening et al. 2000]
2001	Rundberg and Stenström [Rundberg and Stenström 2001]
2002	`R-LRPD` [Dang et al. 2002]
2003/2005	Cintra and Llanos [Cintra and Llanos 2003, 2005]
2008/2010	`CorD`-based systems [Tian et al. 2008, 2010]
2009	`SpLIP` [Oancea et al. 2009]
2009	`STMLite` [Mehrara et al. 2009]
2010	`SMTX` [Raman et al. 2010]
2013	`MiniTLS`, `Lector` [Yiapanis et al. 2013]

advances in the area of speculation. The work discussed in the rest of this article concerns software implementations of speculative parallelization systems unless otherwise stated. A significant portion of Part I explored the various dimensions to implement a speculative parallelization system: metadata, version management, conflict detection, commit/rollback, and scheduling speculative threads. This information is summarized in Table I.

Those categories will lead the structure of Part II while explaining the main work in software speculative parallelization (summarized in Table II). As in Part I the terms *Speculative Parallelization*, *Thread-Level Speculation*, and *TLS* will be used interchangeably throughout the rest of the article. To aid the information flow of this part, some categories will be merged together and one more category will be added in regard to the runtime execution model each system utilizes.

5. EXECUTION MODEL

5.1. Speculative Parallelization

The first trace of speculative parallelization in the literature was the `DOALL` test proposed by Rauchwerger and Padua [1994a]. The `DOALL` test is a nonspeculative runtime technique for *DOALL* loop identification (i.e., loops without cross-iteration data dependencies) initially practiced using the *inspector/executor* model. The test later became the core of one of the earliest and most influential work on speculative parallelization known as the `LRPD` test (Lazy Reduction Privatization `DOALL` test) [Rauchwerger and Padua 1995], proposed also by Rauchwerger and Padua.

`LRPD` [Rauchwerger and Padua 1995] eliminates the overheads of a possibly poor inspector (one that contains a large portion of the original loop) by combining the *inspection* and *execution* face in a single step. Furthermore, certain types of anti and output dependencies are removed by using a transformation known as *privatization*. Privatization transforms certain shared variables into private copies for each thread cooperating on the execution of the loop. For instance, variables that are first *defined* (written) every time before they are *used* (read) inside the same loop iteration, can be safely privatized. Sometimes, variables initialized outside of the loop could be privatized if a *copy-in* mechanism is provided (i.e., provide a copy of the external value for the first use of that variable in each iteration). Similarly, if a privatized variable is required after the loop execution, a *copy-out* mechanism needs to be provided to ensure that the correct value is copied out to the original (nonprivatized) version of that variable.

While inspection of memory accesses takes place by a thread, the actual memory values of the shared user data structure are computed as well, only instead of being placed immediately to main memory, they reside in thread-local storage until inspection completes. This introduces a notion of *speculation* on the values being produced during the loop execution. Yet another novel feature of LRPD is the ability to identify and handle code that fits the *reduction*[5] pattern, thus allowing for more loops to be parallelized.

The majority of subsequent speculative system implementations in the literature [Gupta and Nim 1998; Bruening et al. 2000; Rundberg and Stenström 2001; Dang et al. 2002; Cintra and Llanos 2003, 2005; Tian et al. 2008; Oancea et al. 2009; Mehrara et al. 2009; Tian et al. 2010; Yiapanis et al. 2013] are considered as an extension to the LRPD test.

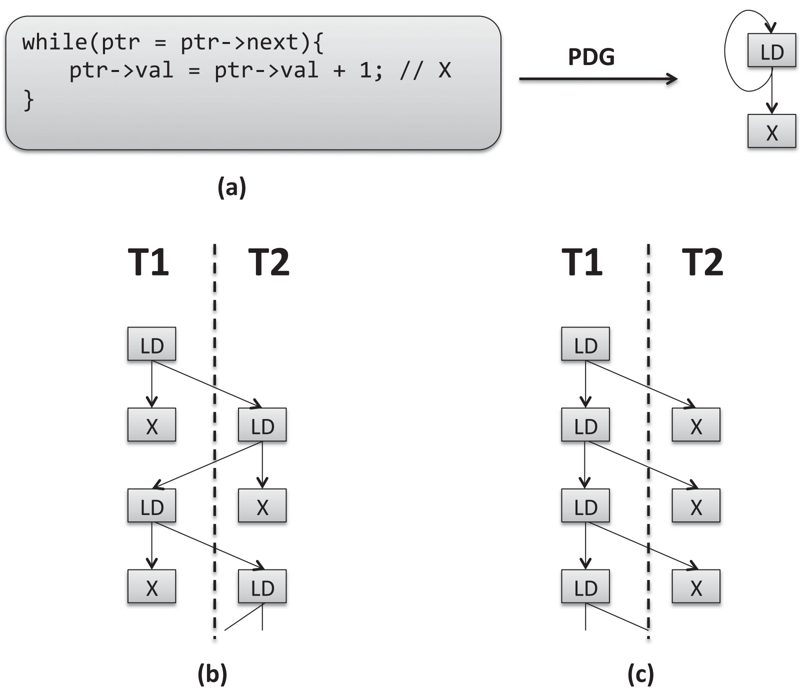
5.2. Speculative Parallelization with Inspection Support

Luján et al. [2007] investigated an idea inspired from the inspector/executor model in which sequential program execution is preserved together with speculative execution in order to minimize the overheads caused from potential misspeculations. More specifically, the least speculative thread executes its assigned portion of iterations nonspeculatively while writing its results directly to memory. This is achieved by extracting a lightweight inspector thread—coined as the *co-inspector thread*—(i.e., one that consists only of the memory accesses) from the least speculative thread, off-loading this way the least speculative thread from any access tracking. The co-inspector thread allows the least speculative thread (which is now nonspeculative) to continue executing the original loop unaware of the speculative threads, of tracking memory accesses, or of applying tests to establish the sequential semantic equivalence.

Yiapanis et al. [2013] propose a TLS system, called Lector, where the techniques of speculative parallelization and inspector/executor are combined together. For simplicity, assume that speculative parallelization is implemented in a similar manner to the LRPD [Rauchwerger and Padua 1995]. From the inspector/executor model, only the inspection loop is manipulated. Lightweight threads, coined in Lector as *inspector threads*, are extracted and applied in a similar fashion as the DOALL [Rauchwerger and Padua 1994a] runtime test. While inspector threads are running, speculation continues as usual. Inspector threads do not replicate the entire code from the loop and thus are expected to be faster than typical TLS threads. If the inspector threads determine that the loop is fully parallel, speculation is withdrawn and parallel execution continues without the speculative overheads. If inspector threads fail due to data dependencies, speculation continues without any changes.

The traditional inspector/executor model would suffer performance losses in two cases: (a) when the inspector replicates a large portion of the loop, and (b) when the inspector identifies data dependencies since the inspection time is completely lost. Combining the model with speculation, like in Lector, these two drawbacks are addressed as follows: (a) Even if inspection completes at the same time as speculation, at least the loop was executed speculatively rather than having to be computed serially. (b) If the inspector identifies data dependencies, the inspection time is amortized by having speculation executing the loop simultaneously.

[5]Reduction operation of the form: $x = x \otimes exp$, where $\otimes$ is an associative operation and variable x does not occur in exp or anywhere else in the loop.



```
while(ptr = ptr->next){
    ptr->val = ptr->val + 1; // X
}
```

Fig. 5. (a) Linked-list traversal. (b) DOACROSS scheduling. (b) DSWP scheduling. This example appears in Ottoni et al. [2005a].

5.3. Decoupled Software Pipelining with Speculation Support

Raman et al. [2010] demonstrated a different approach to runtime parallelization by enhancing a technique known as *Decoupled Software Pipelining (DSWP)* [Rangan et al. 2004] with speculation support.

DSWP is somewhat similar to another earlier form of offline-based parallelism transformation known as *DOACROSS* [Cytron 1986]. DOACROSS targets loops with cross-iteration data dependencies and enables parallelism by scheduling parts of each loop iteration across multiple threads. In DSWP each thread executes part of the loop for all iterations and threads are scheduled is such a way to form a pipeline. To better understand the difference, consider an example[6] code traversing a linked list as in Figure 5(a) and its dependence graph (the graph showing the program's data dependence relationships). The pointer chasing load is labeled "LD" and the loop body is labeled "X."

Figure 5(b) illustrates how the DOACROSS technique would schedule the loop iterations among two threads ($T1$ and $T2$) in an alternate fashion. This way the body of the loop in one thread can be executed in parallel with the next field traversal load of the other thread (in a pipelined fashion). In contrast, the DSWP technique (see Figure 5(c)) schedules the iteration of the loop in a way that half of the iteration (the pointer chasing load) is in one thread and the other half (the body of the loop) in another thread. What DSWP aims to optimize over the DOACROOS technique is to keep the loop's critical path (the longest path in the dependence graph) dependence chain in the same thread. If the critical path has to be routed across threads as in the DOACROSS

[6]This example was taken from Ottoni et al. [2005a].

example, the total execution time of the loop may increase due to communication costs. Placing the critical path in the same thread allows decoupling from that communication latency among multiple threads. One limitation of DSWP is the requirement that the loop must be able to be broken up in such a way that all instructions from the same recurrence (strongly connected component) in the flow graph can be placed on the same thread. This limits the scalability of DSWP to the number of strongly connected components in each loop, which is typically much smaller than the number of loop iterations [Rangan et al. 2008]. This issue is addressed in PS-DSWP [Raman et al. 2008], by enhancing DSWP with DOALL parallelization capabilities. After each recurrence in the loop is isolated in an individual pipeline stage using DSWP, stages that are free of interiteration (loop-carried) dependencies are parallelized in a DOALL fashion to exploit iteration-level parallelism.

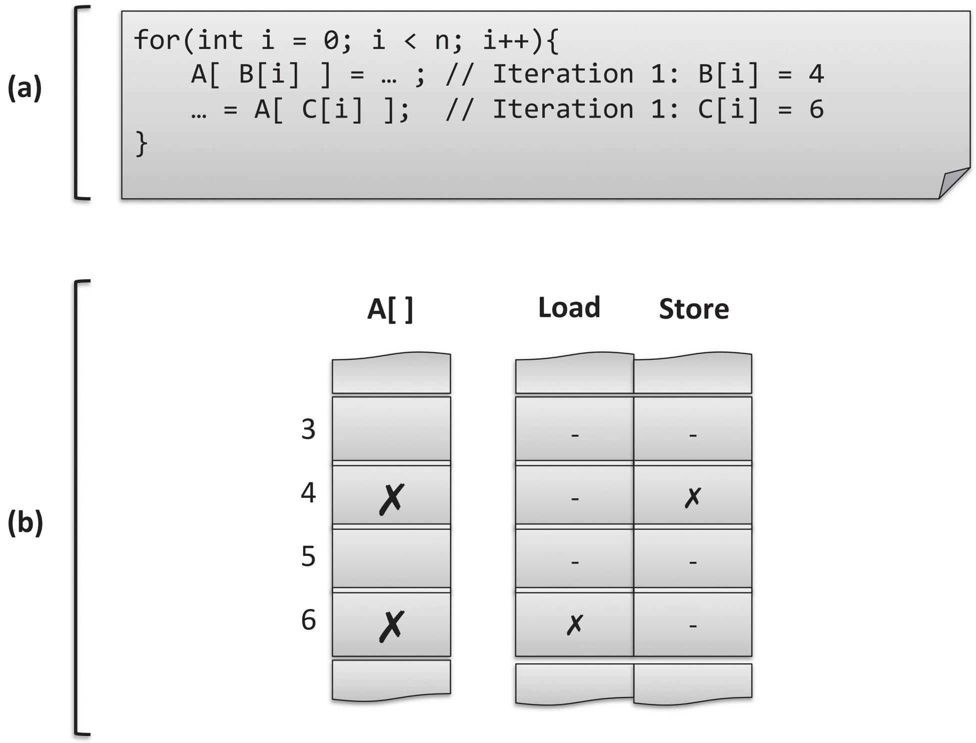
By default, DSWP is a nonspeculative technique and therefore has to respect all dependencies in the loop. In an attempt to allow more loops to be parallelized, Vachharajani et al. [2007] proposed the first system toward speculative DSWP, although, requiring specialized hardware support. Raman et al. [2010] present a software approach for DSWP by providing a software TLS back-end to the initial idea in Vachharajani et al. [2007]. In their work, they coin this TLS support as *Software Multithreaded Transactions (SMTX)*.

SMTX builds on top of Speculative Parallel-Stage DSWP (Spec-PS-DSWP) [Bridges 2008], which extends PS-DSWP by using speculation to break interiteration dependencies to expose data-level parallelism, allowing the replication of pipeline stages with no loop-carried dependencies. In SMTX a loop iteration is executed in a staged manner by multiple threads, making the atomic unit multithreaded (in contrast to TLS that the unit of atomicity is a single thread). These atomic units are supported by atomic sets of memory accesses called Multithreaded Transactions (MTXs) [Vachharajani et al. 2007] allowing speculative work done in different pipeline states (by different threads) to be committed together. MTXs can themselves contain multiple sub-Transactions (subTXs), each of which is executed by only one thread and ordered by the program order of the sequential loop. Typically, a subTX corresponds to the execution of a pipeline stage on each iteration. SMTX is a process-based system. Speculative threads are executed on different UNIX processes and a separate *commit-unit* is used to handle the nonspeculative state.

Apart from alias speculation, SMTX also supports control and value speculation.

6. METADATA AND VERSION MANAGEMENT

Typically, work on speculative parallelization manipulates additional data structures to indicate an action on a particular memory location (this idea was explained in Part I, Section 2.2). Usually these data structures are implemented using arrays or hash tables. Since each of those array elements reflects a memory location, they are referred to as *shadow arrays*. There are three ways that have been mainly used in literature to maintain metadata: (a) keep the shadow arrays private to threads and check for correctness only at the end of speculation, (b) keep shadow arrays shared among threads so that each thread could see what other data threads are accessing during speculative execution, and (c) keep shadow arrays private to threads but provide means to enable early detection in case of a conflict. In other words, allow threads to commit partial results during execution so that conflicts are not delayed until the end. The next sections elaborate more on those three types of maintaining metadata.



(a)

```
for(int i = 0; i < n; i++){
   A[ B[i] ] = … ; // Iteration 1: B[i] = 4
   … = A[ C[i] ];  // Iteration 1: C[i] = 6
}
```

Fig. 6. DOALL test [Rauchwerger and Padua 1994a] basic data structures.

6.1. Speculation with Private Shadow Data

Early work on speculative parallelization was fully optimistic in the sense that it was more effective when speculation triggered no conflicts. Speculative threads did not attempt to communicate between each other until only after the end of their corresponding speculative execution. Thus, a misspeculation was only detected after the final commit.

6.1.1. `DOALL`*.* Even though the DOALL test [Rauchwerger and Padua 1994a] is not a speculative technique *per se*, it is described here as it forms the baseline for schemes proposed later in literature. The DOALL test involves the manipulation of helper data structures in order to track any memory accesses performed on the user shared data structures. Loads and stores are marked during program execution based on memory accesses using those helper or *shadow* data structures. In its simplest form, the compiler analyzes the loop to be parallelized and generates shadow data structures for each user data structure under question. Figure 6(a) shows an example of a candidate loop for *DOALL* parallelization. In order for the loop to be classified as *DOALL*, no data dependencies must exist among different iterations (or chunks of iterations). The shared data structure in question for this example is an array $A[n]$, where n indicates the size of the array. Assuming that integer arrays B and C are populated at runtime, there is no way to analyze A for data dependencies at compile time as its indices are unknown by that time. During compilation two versions of the loop are generated. The first loop will be used to compute all the indices and perform the DOALL test, without actually modifying any shared data. The second loop, provided that the DOALL test was a success, will use those indices to access the actual storage and perform the computation. Two additional arrays, *Load*[ ] and *Store*[ ], are introduced to record the indices of loads and stores, respectively, for array *A* (see Figure 6(b)). The two arrays have the same length as *A* and are used only to mark index locations. Loop iterations are distributed among multiple threads and the shadow structures are replicated for each thread to perform the marking concurrently (without the need of synchronization). At the end of the inspection/marking phase, the different copies of those shadow structures are examined. For a given index *i*, if *Load*[*i*] and *Store*[*i*] are both marked in different threads (i.e., chunks of iterations), then the loop is NOT a *DOALL*. If no load from one thread intersects with a store from a different thread in the same memory location, then the loop is classified as fully parallel. Several extensions to this simplified version have been proposed over time and will be discussed next. Given the test yields a success, the second loop that will perform the actual computation can be executed as a *DOALL* loop across multiple threads.

6.1.2. The `LRPD`*.* The `LRPD` test [Rauchwerger and Padua 1995] takes the `DOALL` test [Rauchwerger and Padua 1994a] a step forward and actually computes the loop in a speculative fashion while inspecting. `LRPD` allocates five data structures in total: four boolean-valued shadow arrays to indicate actions on memory locations and another array as temporary space for speculative values to be stored during execution. The four shadow arrays are defined as follows:

—**Load:** An element *Load*[*i*] in this array is set to "true" to indicate a speculative load performed on a memory location *i* in the user space.
—**Store:** Similar to the load array, an element in this array is set to indicate a speculative store operation on a memory location.
—**NotPrivatizable:** A thread sets an element in this shadow array if a load was performed in a location without a preceding write by the same thread. This is done to prevent the possibility of the loading thread reading a value produced by a different thread in the wrong order. Thus, the array is used to indicate an element that cannot be privatized.
—**NotReduction:** Indicates that memory location cannot participate in a reduction operation.

When the loop is parallelized speculatively, every thread is assigned a chunk of iterations to execute. Each thread has its own copy of the preceding data structures that reflect only the memory locations accessed by that thread. A local copy of those data structures can be accessed only by the thread that owns it. Therefore, during execution there is no need for synchronization to mark those arrays. Nevertheless, at the end of the speculative execution, all these local copies must be merged in order to check for cross-thread memory accesses.

`LRPD` is an example of a lazy version management system, since tentative stores reside in private storage and only become visible to memory at a thread's commit time.

6.1.3. `Softspec`*.* `Softspec` [Bruening et al. 2000] is a TLS scheme that attempts to parallelize *DOALL* loops with stride-predictable memory accesses. A memory access is stride predictable if the address it accesses is incremented by a constant stride for each successive dynamic instance. An example would be a loop accessing each element in an array in a sequence. In `Softspec`, a potentially *DOALL* loop is transformed into four loops: a *profile* loop, a *detection* loop, a *speculation* loop, and a *recovery* loop.

The profile loop executes only the first three iterations to determine whether the loop has a constant stride. This requires a temporary data structure to record any memory addresses from loads or stores performed during these iterations. If the strides are not consistent, then speculation is not performed and the loops are executed sequentially.

The detection loop determines how many iterations can be executed in parallel before a memory dependence occurs. That is, for each pair of memory accesses it determines how many iterations can be executed before the addresses of those memory accesses become equal (in different iterations). If sufficient iterations cannot be executed in parallel, then speculation is abandoned.

As the name implies, the speculation loop is the one that actually does the speculative work. Memory is updated in-place (i.e., eager version management) and original values are stored in thread-private data structures in case of a rollback. This step requires a data structure per thread to record the original memory values before speculative updates. A per-thread shadow data structure of boolean values (used for flags) is also required for every memory access (loads and stores). These are used as flags to indicate a mispredicted address.

Finally, the recovery loop is used to restore memory in case speculation turned out to be incorrect. This is equivalent to the original sequential loop but it is used only to execute the misspeculated iterations.

6.2. Speculation with Shared Shadow Data

In the work discussed previously [Rauchwerger and Padua 1994a, 1995], threads avoid communication between them until the end of speculative execution. Other proposals, such as the ones that will be discussed next, expose the metadata to multiple threads during execution but prevent data races between threads through the use of locks or Compare-and-Swap (CAS) operations. One important reason that a TLS system designer might choose to do that is to allow threads to detect early misspeculations.

6.2.1. Rundberg and Stenström. Rundberg and Stenström [2001] proposed a speculative parallelization scheme in which a thread must first secure *exclusive ownership* of a particular location before any speculative access. This is enabled by requiring a thread to acquire a lock associated with each memory location that may be accessed in a speculative manner.

Every user shared data structure is shadowed by three helper arrays. Every location of these shadow arrays is associated with an individual memory location in the user shared data structure, the same way as in the `LRPD` test [Rauchwerger and Padua 1995]. The first shadow structure is an array of locks used to indicate ownership of a particular location by a thread. The remaining two are used to keep an identifier for a thread that has performed a speculative load or store on a particular location, respectively. Also, a private storage is maintained for each thread to buffer any speculatively produced values.

6.2.2. Cintra and Llanos. Cintra and Llanos [2003, 2005] use a slightly different layout for the metadata than the work from Rundberg and Stenström [2001]. Apart from the private storage for speculative values, three other shadow arrays are used as follows (and illustrated in Figure 7):

—**AT:** The AT array, short for Access Type, shadows the user memory locations. It is used to contain information about the access type upon a memory location by a thread. Access types can be in *Read* or *Mod* state (corresponding to Load or Store actions, respectively), in *NotAcc* (to indicate a memory location never accessed before by different thread) state, or in *ExpLd* state (to indicate a memory location could have potentially been written by a different thread and consumed by the current thread—this action is termed as *exposed load* in this work).

—**IA:** Indirection Array. This array is just a summary of memory locations in a state other than *NotAcc* (not accessed) for each thread. It is used to speed up checking

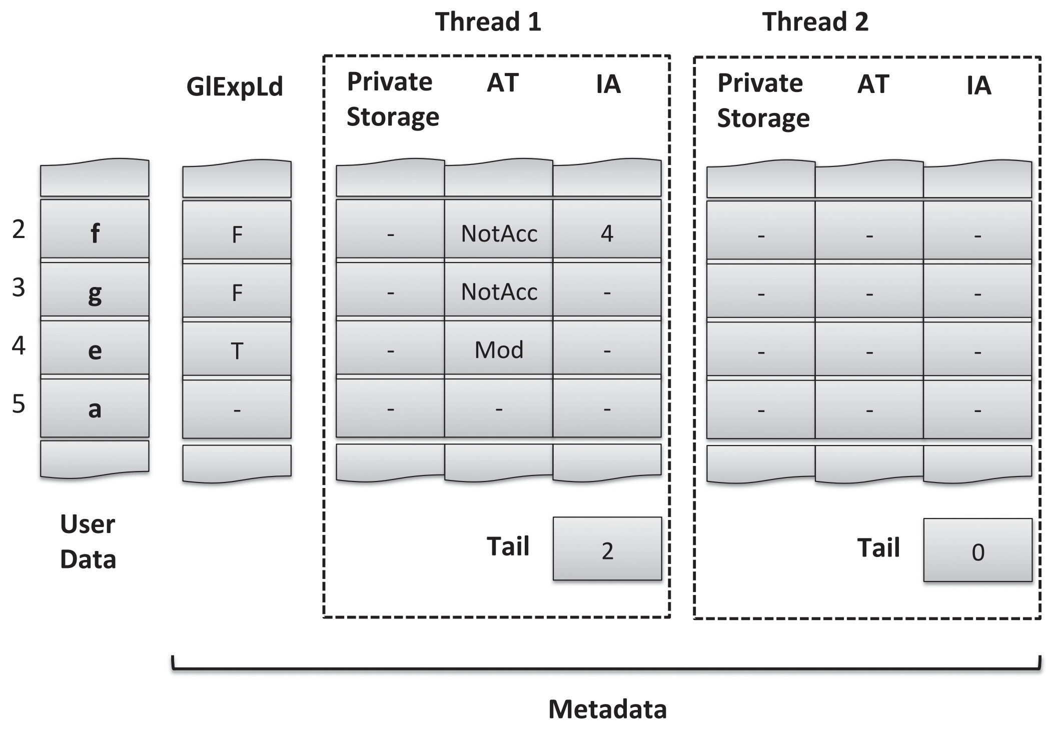


Fig. 7. The data structures used by Cintra and Llanos [Cintra and Llanos 2003, 2005]. "AT" stands for Access Type,"IA" stands for Indirection Array, and "GlExpLd" stands for Global Exposed Load. The values inside the "GlExpLd" can be either true (T) or false (F).

which data a given thread has accessed. An integer variable "Tail" is used to indicate the last element of this array.

—**GlExpLd:** An element of this array (Global Exposed Loads) reflects a memory location across multiple threads. If a given memory location i shadowed by GlExpLd is consumed by a speculative thread but not written first (i.e., exposed load), then $GlExpLd[i]$ is set to indicate a potential violation. This is another attempt to speed up the checking process. If a memory located has never been exposed loaded, then there is no need to be included in any checking for violation.

The work of Cintra and Llanos [2003, 2005] is also an example of lazy version management. In their work, to prevent certain memory access violations, threads are allowed to communicate values between them using a mechanism known as *value forwarding*. This will be revisited in the discussion regarding conflict detection.

6.2.3. SpLIP. SpLIP [Oancea et al. 2009] is a TLS system supporting *in-place* updates (eager version management). A thread-local buffer is still required, only in this case it is used to record the original value of a memory location just before the speculative memory update is performed. Two supporting data structures are required in order to record the thread ID that is currently performing a load or a store for a given location, respectively (illustrated in Figure 8). Up to this point the data structures used are very similar to the ones used in the work by Rundberg and Stenström [2001]. A major difference from Rundberg and Stenström [2001] is the interesting way exclusive ownership is defined. SpLIP takes advantage of certain properties of the Intel ×*86* architecture in order to abolish locks or *CAS* operations used by conventional TLS

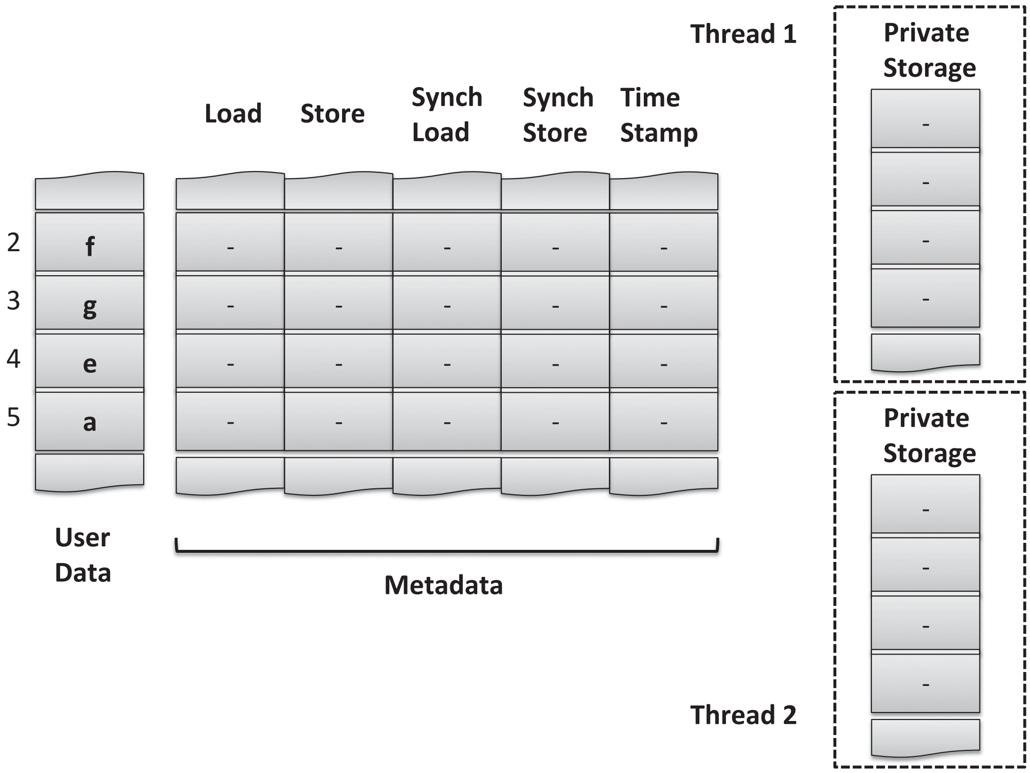


Fig. 8. Metadata organization for SpLIP [Oancea et al. 2009].

systems to protect a speculative location from multiple thread accesses. This comes at the cost of increasing memory requirements by requiring two additional shadow data structures (*SynchLoad* and *SynchStore* in Figure 8) to ensure proper synchronization between multiple thread accesses. Intel ×*86* guarantees that read/write access to a 64-bit word occurs atomically as well as access to any subwords of the corresponding word. Exploiting this information the four shadow structures (two for read/write marking and two for read/write synchronization) are implemented 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. The authors of SpLIP suggest that this sequentially consistent behavior is likely due to flushing the cache line when cache coherency detects concurrent accesses to a word and one of its contained subwords.

The *TimeStamp* data structure is used to indicate the relative time that a thread performed a store operation on a memory location. This feature facilitates recovery from conflicts and will be discussed in more detail in Section 7.2.

6.2.4. MiniTLS. MiniTLS [Yiapanis et al. 2013] is another system that implements *in-place* updates; however, unlike SpLIP, a bit-map data structure is utilized to record read/write information, minimizing this way the memory footprint for speculative information. The layout is illustrated in Figure 9.

Every speculatively accessed datum is shadowed by a sequence of bits divided into buckets relative to the number of running threads. The first two buckets (starting from left to right) are used to acquire ownership of a particular location by setting the bit of the appropriate thread ID. Multiple concurrent readers and only a single writer (and

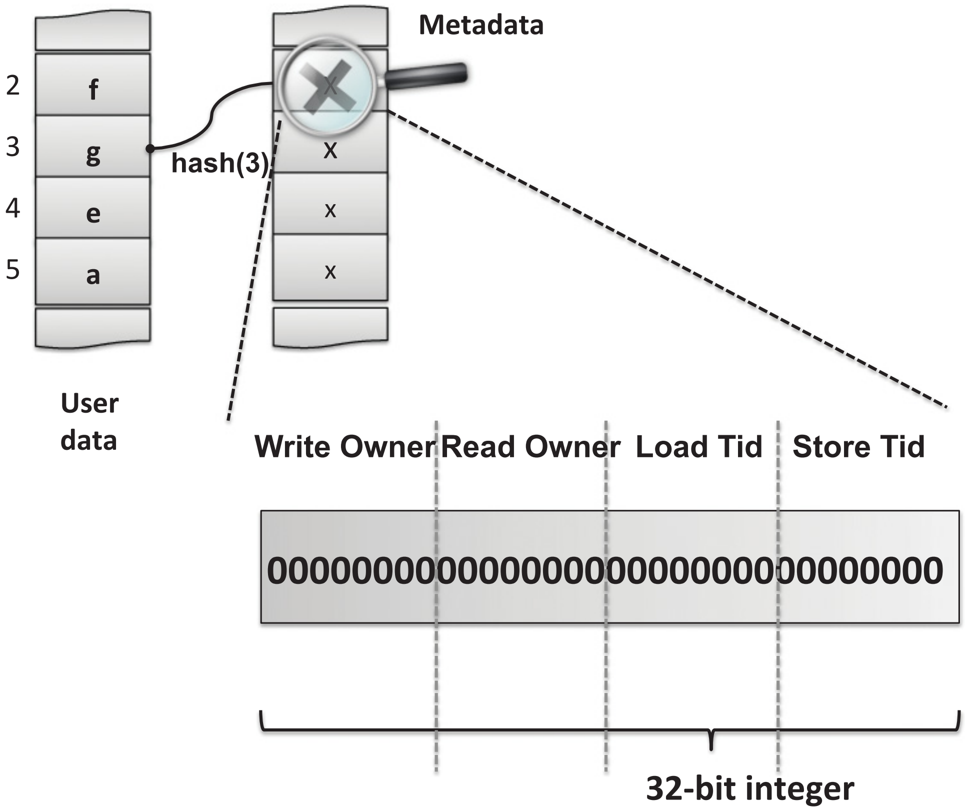


Fig. 9. Metadata organization for `MiniTLS` [Yiapanis et al. 2013].

no readers) are allowed at any given point in time. The ownership bits are *bounded spinlocks*[7] set by using CAS operations. The rest of the buckets are used to capture reader and writer thread ID's during speculative execution. Apart from minimizing the memory footprint required for speculative metadata, this configuration allows for an interesting optimization to parallelize the rollback operation. This optimization is discussed in Section 7.2.

6.3. Speculation Using a Centralized Manager Thread

The work discussed previously concerned either systems that communicate their shadow arrays at the end of speculation or work that the shadow arrays are shared between multiple threads. The following paragraphs discuss a different approach in which although shadow data are distributed among threads, there exists a centralized manager unit to ensure misspeculations are detected before the end of speculation. Also, if the manager is executed in a dedicated thread and it is the only thread allowed to commit values to memory, then commit-time locks can be avoided. Nevertheless, locks may still be required to coordinate multiple threads from accessing the manager thread and *vice versa*.

[7]Bounded spinlocks allow a thread to check a bounded number of times on a lock held by another thread. This allows the thread, when the counter expires, to perform a more useful operation if desired rather than blocking on a particular lock.

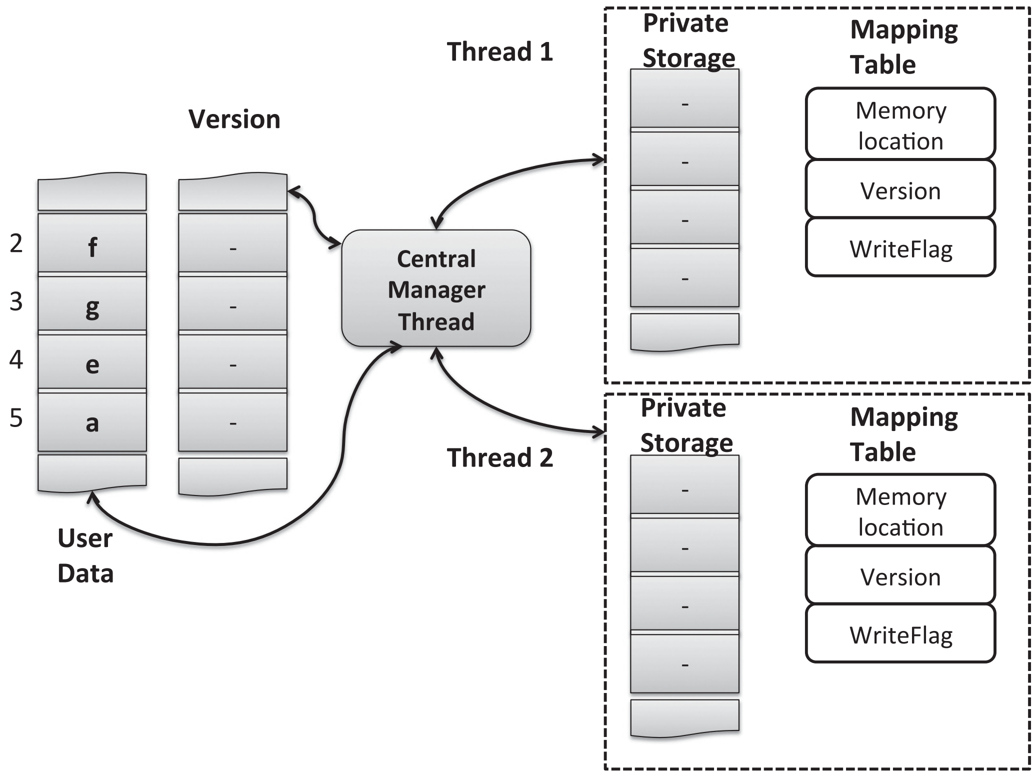


Fig. 10. Metadata used for `CorD` [Tian et al. 2008].

6.3.1. *`CorD`.* Tian et al. describe `CorD` [Tian et al. 2008], a TLS system that maintains a central manager unit that is dedicated to only one thread. The manager is the only thread that has access to user data and it is not speculative. Each speculative worker thread maintains its own private space for marking and execution. When a speculative thread is created, any value that is needed, is copied-in from the nonspeculative state (the user data) to the speculative one (the worker's thread private space) and later the results are copied back if speculation was successful. This transfer of data between the user space and the speculative space is performed only through the central manager. A mapping table is maintained for each speculative thread that has entries associated with each variable's copy. *Version* numbers are used between the two states (user space and speculative space) in order to detect misspeculations. When a memory location is needed, the main thread provides a copy of the associate value to a speculative thread and stores an integer value in the "version shadow array" (shown in Figure 10) for that location. The version number is also provided to the speculative thread's mapping table. If that user location has changed during execution the version number will be updated. When a speculative thread finishes execution, its mapping table will be sent to the manager thread. The mapping table will be traversed by the manager thread to identify any expired versions. The "WriteFlag" is set to *true* by a speculative thread to indicate the memory locations need to be copied back to the user space. Work from speculative threads is committed in-order by the main thread.

Furthermore, in Tian et al. [2010] they address certain challenges specific to applications manipulating dynamic data structures. For example, the copying of objects is limited to only those that are modified by a speculative thread. Also, when an object is moved to the speculative space, all other references that may point to that object must be changed to point to the speculative copy rather than the original. This causes the address translation problem. To overcome copying all those references, the authors introduce *double pointers*. Under this idea, every pointer variable in the program is modified by the compiler and represented by two pointers. One for the nonspeculative state and one for the speculative state. In this way, extra copying is avoided by using the appropriate pointer for corresponding space.

One requirement in `CorD` is that the user space and speculative space remain completely separated. The manager thread supplies any value needed to a newly created speculative thread, even values to be loaded. This could cause unnecessary copying overhead since, anyway, if a value needs to be loaded it could be retrieved on demand from main memory. In any case, `CorD` would invalidate a copy that was updated during speculative execution. Also, in many scenarios pointer values may be undefined by the time they are required to be copied (i.e., when the speculative thread is created). `CorD` addresses this issue by delaying the copying of the pointer address up until it is actually read for the first time.

6.3.2. `STMLite`. Mehrara et al. [2009] present `STMLite`, a Software Transactional Memory (STM) model modified to support speculative parallelization. `STMLite` aims to reduce the overhead associated with validating the read-set by decoupling the conflict detection and commit process from the main transactional execution using a centralized Transaction Commit Manager (TCM). Also, individual locks for copying out the write-set are avoided. The TCM, runs on a dedicated thread and allows only one transaction (or in this case, a chunk of iterations) at a time to write to a particular location (note that, multiple transactions can update different locations concurrently). During execution, each transaction computes their read and write accesses in the form of software *signatures* (inspired by hardware signatures [Ceze et al. 2006]), while buffering their speculative store values. At the end of their speculative execution, signatures are sent for validation to the TCM. If granted by TCM, a transaction is safe to *commit* its buffered values. The relative start and commit times of transactions are tracked using a *global clock* mechanism (a shared counter used to maintain consistency between transactions). The clock is incremented every time a transaction *commits* and it is used to invalidate other "live" transactions with an out-of-date value. This is possible since every memory location is associated with a version number. Before the transaction terminates, as part of its commit procedure all write-sets are revalidated to ensure nonoverlapping write-back into main memory. If successful, the global clock is incremented atomically, all speculative updates are propagated to memory, and all writes are amended to hold the new value of the global clock. This way the next committing transaction can ensure consistency by comparing their previously recorded local value of the global clock against the current up-to-date clock of any values read. The global clock idea for STM was first used in Spear et al. [2006].

To ensure correct parallel execution, `STMLite` forces in-order chunk commit. Since STM does not require the execution to obey the sequential program's order, some extra metadata are required. More specifically, the loop ID of the last committing parallel loop and the chunk ID of the last committed chunk in that loop. The authors refer to this information as *Loop Chunk Commit Log* (LCCL). The loop ID is assigned to each loop at compile time. `STMLite` allows only one in-flight parallel loop at a time by introducing a barrier at the end of each chunk. This prevents multiple instances of the same loop running together and distorting each other's execution in case a parallel loop is invoked multiple times.

6.3.3. `SMTX`. `SMTX` [Raman et al. 2010] also uses a centralized commit unit in a similar manner as in `STMLite` [Mehrara et al. 2009]. While prior lazy version management systems had to check write sets, SMTX takes advantage of Operating System (OS) process isolation to avoid doing so and making speculative loads cheap. The speculative regions are executed inside UNIX processes. This allows the underlying virtual memory to transparently create private physical copies of locations being updated (at the page granularity). As a result, any subsequent load operation to a memory location being updated in the same speculative thread will have the up-to-date value (with respect to that thread) available without the need to scan the write-set.

Each thread maintains a write-buffer that is part of the transaction's private memory. This is used to keep any potential memory updates (if speculation is correct). Special data structures serve as communication channels between each speculative thread and the commit unit. These are single-producer/single-consumer lock-free queues that allow transfer of speculative writes as well as any other message.

The CorD-based systems [Tian et al. 2008, 2010], STMLite [Mehrara et al. 2009], and SMTX [Raman et al. 2010] employ lazy version management.

7. CONFLICT DETECTION, ROLLBACK, AND COMMIT

Earlier, Section 6 described the methods that different TLS systems devised to enable a way of communication between the accesses from speculative threads and the rest of the system. The following paragraphs will provide the critical piece that is missing from the puzzle, which is how each of those approaches maintains correctness during speculative execution. This section is structured based on the two popular ways that a TLS system can use to detect a conflict: *Lazy* and *Eager* conflict detection. While discussing conflict detection, information about different types of commit and rollback will be explained.

7.1. Lazy Conflict Detection

A thread that delays the validation for correctness to the end of its speculative execution is said to employ a *lazy* conflict detection mechanism. The benefit of acting as such is to avoid the overhead of checking for correctness upon every speculative access. The downside is that if a misspeculation is detected, any work performed during speculative execution is wasted. So far in the literature of software speculative parallelization, lazy conflict detection has been used only by lazy version management systems.

7.1.1. LRPD. LRPD test [Rauchwerger and Padua 1995] is a very simple and effective technique for applications that have no data dependencies. The user data structure is shadowed by helper arrays to mark the loads and stores from each thread. At the end of speculative execution the arrays are examined.

Rollback and Commit. In the simplest case, a conflict can arise *if two threads have marked the same location, one thread as a store and the other one as a load*. That is, the same memory location has been read and written in different iterations (threads in this case) indicating a RAW or WAR violation. Nevertheless, if the load is preceded by a store from the same thread, then there is no conflict since speculative values are committed lazily. Furthermore, the LRPD keeps track of the total number of store operations performed by all threads. If that number is different from the total number of store operation markings in the shadow arrays, then some locations have been overwritten causing a WAW violation. Also, violation is caused if a variable was not able to be privatized but written during speculation. When a conflict is detected all speculative work is discarded and the loop reexecutes sequentially. If the loop is found validly parallelized, then all threads can commit their speculative results in parallel to main memory (in parallel only if no two or more threads have performed an update to the same memory location, otherwise sequentially and in-order to avoid WAW conflicts).

The LRPD test [Rauchwerger and Padua 1995] is probably one of the laziest[8] TLS systems. The test for correctness is performed at the end of the entire speculative execution. While this scheme is ideal for an application that contains no data dependencies, many opportunities for parallelization remain unexploited.

7.1.2. Softspec. Softspec [Bruening et al. 2000] employs lazy conflict detection. The system tries to predict the memory accesses of the loop before speculation begins (using constant strides). During speculative execution, memory updates are performed in-place while original memory values are preserved in thread-private data structures. At the same time, the predicted addresses for every speculative access (both loads and stores) are compared with the actual addresses and the result is also stored in thread-private data structures. The flags are checked at the end of each iteration and if a flag indicates a misprediction, then speculation fails.

Rollback. During rollback all stores performed during and after the misprediction are undone and those iterations are reexecuted sequentially by the recovery loop. The recovery loop is used to execute the remaining iterations and it is essentially identical to the original sequential loop. Memory updates during speculation are using the predicted addresses rather than the actual addresses. This allows for recovery in parallel in case of misspeculation.

7.1.3. R-LRPD. As mentioned earlier, the main disadvantage of the LRPD [Rauchwerger and Padua 1995] is having to waste all speculative work performed during execution, by discarding all computed values, when the test was unsuccessful. The R-LRPD (Recursive-LRPD) test [Dang et al. 2002] aims to address this shortcoming. To a great extend R-LRPD is the same as its predecessor. The loop is executed speculative using the same technique as in LRPD and performing the correctness test at the end. If the test is "passed," then the process is equivalent to the LRPD test. However, in the unfortunate case where the test fails, a complete *rollback* is not necessary. Instead, all iterations of the loop under question that computed any results before the misspeculation is triggered, are allowed to *commit*, whereas the rest of the iterations are reexecuted in the same fashion. This process continues in a recursive fashion until all the executions of the loop complete.

This optimized *rollback* procedure of R-LRPD is made possible by using a *sliding window* scheduling mechanism. Only a set of iterations are executed in the window at a time. Thus, successful execution of iterations inside the same window phase appears the same as LRPD. When a conflict arises, it is isolated within the current window of execution.

7.1.4. Gupta and Nim. Gupta and Nim [1998] also aim to address the shortcomings of LRPD by proposing some simple extensions. The main extension concerns elements not able to be privatized. In LRPD a thread that loads a value initialized by a different thread causes a violation. In the scheme proposed by Gupta and Nim [1998] a thread that attempts to load a value not initialized by itself can decide to stall until the previous threads have finished execution. This is done by inserting synchronizations for the threads that may require a value not yet produced. This scheme assumes that threads are totally ordered and iterations are assigned to threads in increasing order. This implies that a lower thread in the ordering will always be less speculative than a higher thread. The stalled thread can proceed only when its less speculative threads have finished and thus commit the value needed in main memory.

[8]Laziest in terms of conflict detection.

7.1.5. `CorD`*.* A system, namely, `CorD` [Tian et al. 2008], with a centralized manager thread was introduced earlier in Section 6. Figure 10 shows the metadata organization for that system. The manager thread is the only one allowed to interact with the user data. When execution begins the central manager creates a speculative thread copying-in to that thread's private storage any values needed from the user space. For every user memory location the manager associates a version number. When a value is required by a thread, the current version number is also given by the manager. During execution a speculative thread loads and updates values only in its local storage. The speculative thread also maintains a mapping table that contains the address of a given location, its original version number when copied-in, and a flag. The flag is set only to indicate a location that has been updated locally.

When the manager consults the mapping table of a speculative thread the version numbers must be examined. If the thread contains a value that has an expired version, then a violation is raised.

Commit. When a speculative thread completes, the manager is notified. The manager will consult the mapping table of that thread to identify which locations have the flag set and therefore be copied-out to the user space. When a value is updated in the user space, its corresponding version number also changes.

Rollback. When a violation is raised, the manager notifies the speculative thread to discard any work done and the values are copied-in again to restart execution.

7.1.6. `STMLite`*.* The process for conflict detection in `STMLite` [Mehrara et al. 2009] is similar to the one implemented in `CorD` [Tian et al. 2008]. Once a speculative thread completes execution, the addresses of the values read and written are send to the central thread. The central thread also maintains a log containing the values committed by previously successful speculative threads. The read and write addresses are compared against the log of committed values. `STMLite` uses read and write signatures for the granularity of conflict detection, which are essentially hash-based representations of all reads and writes performed during execution.

Rollback. If a conflict is identified between a value consumed by the current speculative thread and a value residing inside the log of committed values, the central manager notifies the thread to discard its local buffer and restart execution.

Commit. If the values used by the speculative thread do not intersect with any of the values in the committed log the central manager notifies the thread to start copying any updates to main memory. Since the central manager in this case (and unlike in `CorD` [Tian et al. 2008]) allows speculative threads to begin committing their values independently and in parallel, a mechanism is required to prevent concurrent memory updates by multiple committing threads. This is implemented using an additional data structure that shadows the number of active threads in the system. This data structure is used as a synchronization point, which indicates when a particular thread has finished committing its values. The manager thread contains the addresses to be committed by all threads. Before the manager allows a thread to begin the commit process, it consults the write signatures from the currently committing threads. If any of the write signatures from the speculative thread waiting for permission to commit conflicts with a thread currently committing, then the manager will postpone the commit for the waiting thread until the others finish.

7.1.7. `SMTX`*.* `SMTX`, the speculative parallelization system proposed by Raman et al. [2010], also uses a lazy conflict detection mechanism. Like `CorD` [Tian et al. 2008] the central manager is notified when a thread completes executing its corresponding speculative region. The central manager is then responsible to identify any conflicts among speculative threads. There are a few important differences from CorD. One is the extra complication from supporting the subtransactions (subTXs). The subTXs must commit within the context of their parent transactions. This is required to maintain consistency among speculative threads. The authors address this issue by having the subTXs copying on demand the values they require from previous sub-TXs (or the parent transaction). Consequently, the parent transaction as well as its children all have the same version of memory. Second, SMTX supports value-based checking. In value-based checking a conflict is based on a log of actual values read from memory rather than memory locations themselves. During execution a thread buffers any memory addresses read from main memory along with their actual values. A conflict is detected when a previously buffered value and the value for a given memory location in main memory is different from what was logged earlier (i.e., another thread has updated the value in that memory address). Value-based checking was also previously used in Ding et al. [2007].

Commit and Rollback. A worker thread that completes speculative execution will forward its read-set (address and value pair of memory locations that it has read) to the central manager, known as *commit unit* in SMTX, along with its write-set. To take the overhead of speculation management off the critical path, SMTX uses a separate *try-commit* process that is responsible for validating the speculative threads in addition to the commit unit (also running on a different process) that is responsible for committing the speculative writes. If dependence violation is encountered, the commit unit is notified to initiate the recovery. To recover from misspeculation, the SMTX system remaps the virtual address space of the speculative threads to the committed memory state of the commit unit. The entries in the offending threads' write-buffers are flushed and the threads are restarted. If the speculative thread is proved successful, the commit unit propagates the speculative thread's writes to main memory.

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

7.2. Eager Conflict Detection

7.2.1. Rundberg and Stenström. In the software TLS system proposed by Rundberg and Stenström [2001], a given user memory location is protected by a lock as well as keeps information about its latest reader and writer threads. This feature allows a speculative thread to check in isolation (from other threads) whether reading or writing the value in question can cause a violation. Their implementation prevents a speculative load to cause a data dependency violation by supporting a technique known as *value forwarding*. Using this technique, a thread performing an *exposed load* (i.e., a load on a value not produced by the same thread) is allowed to search backwards (in terms of speculation order) and find the latest value produced by a less speculative thread to serve that particular load. After the latest value is found, it can be forwarded from the less speculative thread's buffer to the more speculative thread's buffer bypassing any rollback related costs.

Rollback. A conflict is caused only as a result of a thread reading a value "too early." For instance, imagine a thread that is about to perform a speculative store on location x. The thread first locks x and then checks if a different thread has already loaded that location. If indeed another thread has loaded that value and that thread is more speculative, then a misspeculation is detected by the system as this causes a RAW conflict. As a result, the more speculative thread that caused the conflict and all its successor threads must be squashed and discard their local buffers.

Commit. A thread that carried out its speculative execution without any conflict arising is allowed to commit its results to main memory. Typically, in a system that employs a lazy version management, such as this one by Rundberg and Stenström [2001], speculative threads commit their buffered values in order one by one. Their scheme allows this process to be optimized by offering a *parallel commit* phase. Recall that the shadow arrays keep track of the latest thread that has written on a particular location. Even if multiple threads have performed a write on a location during speculation, the only one that must provide memory with the correct value for that location is the latest thread. Before commit, the "Store" shadow array that corresponds to the memory locations to be updated is inspected to find the latest threads recorded there. Threads can proceed committing in parallel by having each memory locations updated by the latest thread written on them.

Value Forwarding Synchronization. The authors also present a more efficient implementation where Load and Store shadow array locations are represented at the byte size. Since the architecture they tested supports atomic byte operations, this enables them to perform speculative loads by issuing low level hardware atomic loads without using explicit locks. Such an optimization needs to be handled with care when the system allows value forwarding because the future thread must be able to "see" the correct value produced by the past thread. In other words the thread that owns the value to be forwarded must first write that local copy before the forwarding thread loads it. Rundberg and Stenström [2001] handle this case in the following way: the thread that owns the value to be forwarded is allowed first to write a $0 \times 0F$ to its byte in the Store shadow array, then perform the update of the local copy, and after this must write $0 \times FF$ to that byte. A load, from the thread that requires that value, discovering a $0 \times 0F$ to the byte of the store shadow array simply needs to wait for it to change to $0 \times FF$ before it performs the forwarding load operation.

7.2.2. Cintra and Llanos. Cintra and Llanos in Cintra and Llanos [2003, 2005] experiment with both lazy and eager conflict detection. The idea of implementing lazy conflict detection is so that the cost of checking every memory location (which usually requires synchronization) is avoided. However, delaying the conflict detection will cause wasted work if a conflict is triggered. Cintra and Llanos in Cintra and Llanos [2003, 2005] indicate that the cost of checking for violations on every speculative access is negligible compared to the cost over checking only at the end of speculative execution where a significant amount of work might be wasted. The conflict detection idea is the same as in the TLS system by Rundberg and Stenström [2001]. Rollback is only triggered by stores as loads do not cause conflicts when value forwarding is enabled. Their system differs, in terms of commits, from the one by Rundberg and Stenström [2001] in two ways: (a) Cintra and Llanos further optimize their systems for cases that there were no *exposed loads* present on a given window execution. This is accomplished by introducing an extra data structure that raises a flag whenever any one thread performs the first *exposed load* for a given location (see Figure 7). (b) Cintra and Llanos employ a serial commit phase.

7.2.3. `SpLIP`. Recall from Section 6.2 that `SpLIP` [Oancea et al. 2009] and `MiniTLS` [Yiapanis et al. 2013] differ from all other TLS systems in that they are the only software TLS systems implementing an eager version management system. Eager version management systems perform speculative updates to main memory but buffer the original values before they do so. Threads directly update the main memory with speculative values and as a consequence all three types of violations are possible (RAW, WAR, WAW).

SpLIP [Oancea et al. 2009] uses five shadow arrays to facilitate conflict detection (recall Figure 8):

—**Load:** To mark the latest thread that performed a load on a memory location.
—**Store:** To mark the latest thread that performed a store on a memory location.
—**TimeStamp:** To mark the relative time that a thread performed a store on a memory location.
—**SynchLoad:** Facilitates synchronization by tracking the current thread performing a load operation.
—**SynchStore:** Facilitates synchronization by tracking the current thread performing a store operation.

A conflict can be detected as follows:

—**Read-After-Write:** A speculative thread attempts to perform a store to a memory location but discovers that a more speculative thread has already consumed the value from there.
—**Write-After-Read:** A speculative thread attempts to perform a load from a memory location but discovers that a more speculative thread has already stored a value there.
—**Write-After-Write:** A speculative thread attempts to perform a store to a memory location but discovers that a more speculative thread has already stored a value there.

Rollback. When a conflict is detected, the offending threads must go through their local buffers, which include the original values, and restore memory back to the latest known correct state. An issue arises when more than one thread involved in the violation has written to the same memory location. That is because only one of them must restore the correct value back to main memory—the one that has the earliest copy in terms of speculation order. SpLIP [Oancea et al. 2009] uses a "TimeStamp" shadow array to record the relative time a thread has stored to a location. Using this shadow array the system is able to recover the earliest value that needs to be rolled back. SpLIP aggregates the write-sets of all speculative threads involved in the violation in a sequential manner by comparing timestamps in case multiple threads access the same location.

Commit. Since threads directly update the speculative values in memory, if no conflict is detected, then the final results are already there. Thus, commit implicitly happens *in parallel*.

Synchronization Arrays. SpLIP does not use any locks or CAS operations but instead uses two additional data structures to facilitate synchronization (recall Figure 8). When a speculative load is performed, all the operations are surrounded by actions on the *SynchStore* array. When speculative load starts, the *SynchStore* array is initialized with the ID of the thread that performs the load (i.e., $SynchStore[x] = T$). At the end of the speculative load, the condition $SynchStore[x] == T$ must hold for the operation to be valid. $SynchStore[x]$ may change while the load is performed only if a speculative store executed concurrently (the speculative store also sets $SynchStore[x] = T$ to indicate the action). In a similar way, *SynchLoad* is initialized with the thread ID that performs a load and checked within the speculative store operation.

The checks to the synchronization arrays are carefully placed before and after certain instructions and rely on the memory ordering of the architecture to be correct.

7.2.4. `MiniTLS`. Although `MiniTLS` [Yiapanis et al. 2013] and `SpLIP` [Oancea et al. 2009] use an eager version management technique, their implementations are fundamentally different. `SpLIP` employs two data structures for handling the iterations that load and store for each speculative access. Two additional data structures are used for imposing order to accesses to each location between reading and writing threads. Yet another data structure is required for storing a timestamp for a particular location in case of a rollback. `MiniTLS` compresses all the information in one BitMap data structure (recall Figure 9). The rollback procedure also differs from `MiniTLS`. `SpLIP` aggregates the write-sets of all speculative threads involved in the violation by comparing timestamps in case multiple threads access the same location. In contrast, `MiniTLS` requires no aggregation, and each thread involved in the violation proceeds in parallel with each other for rollback, without using timestamp comparisons.

Rollback. `MiniTLS` implements a parallel rollback operation, which can help in reducing the overheads when misspeculation occurs. When the rollback phase initiates the system first must identify which is the least speculative thread that modified each location. This is done in parallel by allowing each participating speculative thread to visit its write-set data structure and for each element in the write-set to check in the shadow data structure whether any other thread has modified it. To access the shadow data structure threads use CAS operations. Should more than one thread have written a given location, the least speculative thread to have modified it is identified in the following way: If the speculative thread is the first one to have modified it, the thread can go ahead and restore the value for that memory location. If the speculative thread is not the first one to modify it and aliasing on that location ($hash(x)$) is possible, the speculative thread has to check whether its memory location x is actually contained within the write-set of the less speculative threads denoted in the shadow data structure for $hash(x)$. If it is not found in these less speculative write-sets, that speculative thread will restore the value for memory location x. Once the memory state has been rolled back, the participating speculative thread can reset the pertinent memory locations in the shadow data structure in parallel.

8. WORK SCHEDULING

8.1. Static and Dynamic Work Scheduling

`LRPD` test [Rauchwerger and Padua 1995] and `Softspec` [Bruening et al. 2000] can be applied either with *static* or *dynamic* scheduling. With static scheduling the iteration space is divided evenly among the number of threads. Using dynamic scheduling, large chunks of iterations are assigned at runtime to the number of threads (usually the number of threads is significantly smaller than the number of chunks).

The work by Gupta and Nim [1998] can also use both static or dynamic scheduling. The only requirement is that threads are assigned contiguous chunks of iterations and threads are totally ordered by their speculation level. This allows the notion of less and more speculative threads. Remember that work performed by a less speculative thread can potentially invalidate a more speculative thread. The same scheduling policy is also used by many others [Rundberg and Stenström 2001; Tian et al. 2008, 2010; Mehrara et al. 2009; Oancea et al. 2009; Raman et al. 2010].

8.2. Sliding Window Work Scheduling

Three work scheduling strategies have been investigated using `R-LRPD` [Dang et al. 2002] to recover from misspeculation. First, the *Nonredistribution (NRD)* strategy is examined, where failed iterations must reexecute their work in the threads that they were originally assigned to. A major issue with NRD is the load imbalance that can be potentially introduced when some threads finish earlier than others and have to wait sitting idle. This problem is addressed in the other two work scheduling techniques. The *Redistribution (RD)* strategy allows iterations, involved in *rollback*, to subdivide themselves and be distributed among different threads. Finally, a *Sliding Window* scheduling strategy is applied where contiguous chunks of iterations are assigned to a group (or window) of threads in a way that satisfies a speculation order. Iterations in lower indices of the window are less speculative than iterations in higher indices and a less speculative chunk of iterations has priority over a more speculative chunk. This ordering is important when different chunks of iterations are involved in a data dependency violation or they are ready to commit.

Two types of sliding window are evaluated by Cintra and Llanos [2003, 2005]: the so-called *aggressive* sliding window that proceeds to retrieve a new chunk of iterations whenever the least speculative thread commits their results and the conservative sliding window that reloads only when all speculative threads on the window commit their results. The *aggressive* sliding window was found to be superior to the *conservative* one, however, at the cost of higher implementation complexity. `MiniTLS` and `Lector` employ a *conservative* sliding window for work scheduling [Yiapanis 2013].

9. DISCUSSION

This section provides critical discussion regarding the applicability and performance of the TLS systems descried in this article. Furthermore, discussion is provided regarding trade-offs in designing TLS systems, other sources of speculation, and application of TLS on different areas.

9.1. Applicability

Many of the early TLS systems were mainly targeting scientific and numeric applications using fixed-size data structures on counted loops. This is not the norm for general-purpose applications that may contain loops with arbitrary control flow. Also, applications written in recent languages would require TLS systems that can handle complex heap-based data structures and dynamic allocation.

`LRPD` test [Rauchwerger and Padua 1995], Gupta and Nim [1998], `Softspec` [Bruening et al. 2000], Rundberg and Stenström [2001], `R-LRPD` [Dang et al. 2002], and Cintra and Llanos [2003, 2005] target programs that use fixed-size arrays to store shared data. As a consequence of this, the data structures used in the approaches to record memory access information are constructed out of fixed-size arrays, mirroring the layout of the shared data. Such memory access tracking schemes are not suitable for parallelizing programs that use dynamic data structures, since those structures may grow (or change structurally in other ways) during speculative execution. In `Softspec`, the authors propose that loops over nonarray data structures may be parallelizable if memory allocation collaborates by ensuring an appropriate layout (e.g., linked lists having elements allocated contiguously in memory). An important limitation of `Softspec` is that it is only applicable to loops performing easily predictable memory accesses.

Another reason why these approaches could struggle to parallelize dynamic programs is that they target loops with simple iterators (i.e., either integer-indexed *for*-style loops, or *while* loops without an explicit index). This does not address programs where the parallelizable computation is in the form of function calls, recursive or not, or even loops over complex data structures. The rest of the systems described provide support for dynamic data structures.

`SpLIP` [Oancea et al. 2009] provides a lightweight solution to speculation with wider applicability than previous work. Due to its reliance on in-place updates, `SpLIP` requires a mechanism to ensure an appropriate ordering of memory operations. The authors point out that common implementations of memory barriers can be costly, and describe a way to exploit certain properties of the $\times 86$ architecture to achieve the desired effect more efficiently. However, this limits the applicability of SpLIP due to its reliance on a single instruction architecture.

Even though CorD [Tian et al. 2008, 2010] provides support for dynamic data structures, as the authors in Johnson et al. [2012] point out, the transformation to convert pointers to double pointers assumes that all accesses conform to the objects declared type, but in reality may fail due to reinterpretation casts. Static analysis cannot always determine whether an object is ever reinterpreted. The transformation also assumes that all pointer values are visible in the internal representation, but Cs weak types allow "disguised" pointers [Boehm 1996].

STMLite [Mehrara et al. 2009], SMTX [Raman et al. 2010], MiniTLS [Yiapanis et al. 2013], and Lector [Yiapanis et al. 2013] support pointers and dynamic allocation.

All the other TLS systems described in this article provide a more coarse grain loop decomposition in contrast to SMTX that partitions the loop program dependence graph into fine-grained threads organized into a pipeline. In codes with recurrences, a coarse-grained speculation will force the dependencies that participate in the recurrences to be communicated between threads. This places communication latency on the critical path. This, along with support for control and value speculation make SMTX more suitable for such codes. Furthermore, SMTX can be used in conjunction with different conflict detection mechanisms to match the right granularity/precision of tracking according to the application being parallelized.

9.2. Performance Considerations

Since there is no attempt for synchronization between threads during the speculative execution phase (and hence no limits to scalability in that respect), the LRPD test is amenable to using a large number of speculative threads. However, the lack of synchronization also means that a dependency between threads will lead to misspeculation. Moreover, the misspeculation will only be detected when all threads have finished, leading to work done by threads being wasted. This implies that the LRPD test is only useful if the iterations of loop being parallelized are independent from one another.[9] Furthermore, the memory overhead is high since it depends on the size of the shadow data structure and the number of iterations.

Unless there is a high confidence that speculative execution will be successful, it is probably more beneficial that communication exists between speculative threads (or through a shared unit) to avoid the high cost of unsuccessful speculation.

Rundberg and Stenström's system requires synchronization between threads, by using mutual-exclusion locks at speculative memory accesses; the Cintra and Llanos approach does not rely on explicit locks in such situations, but does make use of memory *fences*, also known as *barriers*, which also carry a significant cost (though typically less than that of a lock). MiniTLS and Lector rely on *CAS* operations, which are also about as expensive as memory barriers. This implies that these approaches may not be suitable for parallelizing program regions that spend a significant portion of their execution time doing memory accesses that are not statically analyzable.

Another downfall with Rundberg and Stenström's approach is that it exhibits a huge memory overhead that depends on the number of loop iterations and the size of their auxiliary data structures for speculation support. Techniques employing a sliding window (Cintra and Llanos, R-LRPD, MiniTLS, and Lector) often have smaller memory requirements since the memory overhead is proportional to the window size. However, this comes at the cost of scalability since now only a fixed number of threads may contribute to speedup. Nevertheless, all these solutions can also use dynamic scheduling

[9]With the exception of dependencies between iterations executed by the same thread, which do not have any effect on the operation of the LRPD test.

if needed (at the expense of higher memory overhead). In addition, a conservative window may hinder performance since all threads in the window must complete execution before the window moves. As a result, threads completed earlier than others will have to stall until all threads in the current window complete. An aggressive window implementation will at least allow completed threads to move to the next piece of computation while others are still working.

SpLIP provides a lightweight approach with no locks and a low memory overhead that depends roughly on the number of per-iteration writes. SpLIP shows scalability as the number of threads increases and the shared data structures do not become a bottleneck due to lock elimination. The shared shadow structure is fixed during speculative execution but hashing conflicts may cause false dependencies (as with MiniTLS, which also has low memory overhead but uses CAS operations). As described earlier, SpLIP relies on certain properties of $\times 86$, which in effect limits the system to a single architecture.

The TLS systems that use a shared shadow data structure allow for early conflict detection in contrast to the systems that use private shadow data structures, which are more scalable but are costly in the case of misspeculation. In order to increase parallelism, these systems allow multiple threads to access the shared data structures as well as the nonspeculative state concurrently by using locks, memory barriers, or CAS operations. These operations are expensive and may be costly when frequent. A possibly better approach is to have a separate thread to manage the nonspeculative state while multiple threads can perform speculation concurrently. This approach is taken by CorD, STMLite, and SMTX. The obvious downside of centralized control is the lack of scalability, but for a modest number of threads it should not be a problem. The authors of these systems report excellent speedups for the amount of threads they experimented with. Furthermore, Mehrara et al. [2009] discuss the possibility to add several coordinating central points to improve scalability in the presence of a large number of threads, while Raman et al. [2010] explain that the commit and try-commit phases in SMTX are parallelizable. The memory overheads for CorD and SMTX depend on the maximum number of concurrently executing iterations. STMLite has lower memory overheads due to the use of signatures at the cost of precision.

One limitation with CorD is that each speculative thread is required to synchronize with the central manager (the nonspeculative thread) on every iteration. This puts the cross-thread communication latency on the critical path and limits performance.

Both STMLite and SMTX support a centralized transaction commit manager and conflict detection that is decoupled from the main execution. This allows conflict detection to occur in parallel with speculative thread execution and in parallel with value commit. As Raman et al. [2010] points out, the conflict detection of the two systems is different: in STMLite, each speculative thread computes read and write signatures that are compared at commit time. STMLite allows speculative threads to compete for commit ordering; in contrast, due to its focus on loop parallelization, the SMTX system requires speculative threads to have a total ordering. The authors mention that the system can be easily modified to loosen this restriction. Both systems use lazy version management but as explained earlier SMTX's subsequent loads to the same location do not require one to scan the write-buffer due to the use of virtual memory.

9.3. Trade-Offs

Generally speaking, a system that utilizes EVM might be more complicated to implement than one using LVM. Since stores are written in-place (eagerly), the designer has to also consider *WAR* and *WAW* dependencies, apart from *RAW* dependencies. Nevertheless, both design choices (LVM and EVM) are advantageous in different cases. Recall from Section 2.2.4 that EVM has a cheap commit phase (updates are in-place) but an expensive rollback. Contrarily, a LVM system has a cheap rollback (just discard local buffers) but a more expensive commit. Thus, a rule that may be wise to follow is that EVM is probably more appropriate for applications with minimal runtime conflicts (since commit is cheap) and LVM more appropriate for applications with a high number of conflicts (since rollback is cheaper). This pattern was also observed by Garzarán et al. [2005] in a study where separation of task state (version management) and merging of task state (commit) were analyzed. In TLS systems implementing LVM, loads will potentially have to scan the store buffer to get the last write to a given location. Previous work in the literature has shown that this can be avoided by taking advantage of the OS process isolation [Ding et al. 2007; Raman et al. 2010].

There are various ways that Version Management and Conflict Detection can be combined to implement a TLS system but there are advantages and disadvantages to every choice. LVM requires the TLS system to maintain a private buffer per thread in order to track the address and value of a potential update operation. A Load will first need to check its thread's speculative buffer for the value before attempting to read from main memory (although this can be avoided by taking advantage of OS process isolation [Raman et al. 2010]). If LCD is used for LVM, checks for conflicts could occur at the end of a speculative thread's execution or at any predefined intermediate state of the speculative data. This technique obviates the cost of checking for conflicts on every memory access. This is beneficial only if conflicts are rare, otherwise any speculative execution (both in terms of time and space) from the conflict point until the detection point will be wasted. Eagerly detecting conflicts will endure the extra cost of checking upon every memory access but would reduce the risk of wasted speculative work in case of frequent conflicts. The two combinations, LVM with LCD and LVM with ECD, have been evaluated in Cintra and Llanos [2005]. EVM updates memory directly so a private per thread buffer is still required for the original values in case of a conflict. When a TLS system is implemented using ECD, checks for conflicts occur on every memory access and action is taken as soon as a violation arises. Commits occur in-place and rollback is initiated to restore memory in case of a conflict. Eager conflict detection prevents wasted speculative work but carries the overheads of checking for violations on every memory access (as with LVM and ECD). Examples of such systems have been evaluated in Oancea et al. [2009] and Yiapanis et al. [2013]. Note that with EVM it is necessary to use ECD because the thread requires exclusive access to the locations if it is going to write to them directly.

Conflict detection granularity introduces a trade-off between precision and size of auxiliary data for conflict detection. A system that implements a coarse-level of granularity will be susceptible to false sharing and thus imprecise. This allows for a more compact auxiliary data structure. For example, `STMLite`, `MiniTLS`, `Lector`, and `SpLIP`, all use signatures (hash-based representation for reads and writes) to optimize the space for their shadow data structure. Choosing the right set of hash functions and the proper size for signatures is crucial in software systems to ensure minimal overhead and few false positives. `SMTX` is an example that uses fine-grained checking (value-based checking at the instruction/word-level).

The main overhead in software TLS arises from maintaining the speculative state. Certain hardware solutions that apply LVM can overcome this problem by taking advantage of the cache coherence protocol [Gopal et al. 1998]. For example, cache lines are modified with additional information, an extra bit and a pointer, to facilitate marking of speculative data. The bit, known as the "load bit," is set when a processor loads a value not produced by itself (i.e., a potential RAW violation). The extra pointer indicates the next most speculative cache line that has a version of that datum in case a value needs to be forwarded to a remote cache. Furthermore, L1 cache is normally private to the processor and thus buffering occurs at the L1 cache lines by default.

Not only buffering is cheap but also marking simply needs to write a bit and change a pointer. Software approaches, on the other hand, rely on additional data structures to maintain marking information. Thus, each potential speculative load or store will produce at least an extra load and store from the hardware point of view (to read and insert that item to the data structure). These data structures, when shared, are normally accessed via locks or *CAS* operations to avoid data races between different threads. A possible way to avoid using locks is to separate the speculative from the nonspeculative execution as discussed in the previous section. Data structures need to be designed very carefully and be highly optimized to lower any software-related costs.

9.4. Other Program Points of Speculation

The article (and most of the TLS systems) mainly focused on *loop-level* speculation as there is significant parallelism found there, especially in outermost loops of applications [Bridges et al. 2007; Thies et al. 2007; Tian et al. 2008; Zhong et al. 2008]. TLS is not restricted to loops and can be applied to different levels as well. One notable example is *method-level* speculation [Chen and Olukotun 2003; Pickett and Verbrugge 2006; Ioannou et al. 2010], which was also successfully applied in scripting languages [Martinsen et al. 2013] (see Section 9.5.1). In method-level speculation speculative tasks are formed by methods (subroutines) as opposed to loop iterations in loop-level speculation. Each method executes on a different speculative thread and can further spawn a method found on its path on a different child speculative thread, forming nested speculation. The speculative order is defined by the order in which methods would have been called in the original unmodified sequential program. In order to expose more method-level parallelism, it is important to use *return value prediction* [Hu et al. 2003]. Predicting the value to be returned by a method that has not yet executed (or finished execution) allows the program to continue speculative execution past that method's call site.

From the TLS system's point of view this is almost the same as in loop-level speculation: speculative tasks that need to be executed in parallel and commit in a specified order. Data dependencies must be preserved the same way as in loop-level speculation. The only addition required to be implemented is support for value prediction. `SMTX` already supports value speculation but other systems could be easily modified to provide that support. The code also needs to be transformed in such a way so that tasks are formed by methods. Liu et al. [2006] describe a TLS compiler infrastructure that is able to leverage the code structure (both loop iterations and subroutines of any nesting level) to generate tasks for speculative parallelization.

Other sources of speculation have also been considered. For instance, the work in Johnson et al. [2004] considers all basic block boundaries as possible thread spawn points and uses a min-cut algorithm based on expected dependencies to find the best selection of threads. Similarly, the work in Marcuello and González [2002] considers all control quasi-independent points (i.e., points more or less guaranteed to postdominate a certain program point) in the program as possible thread spawn targets.

9.5. Other TLS Applications

Speculative parallelization is not only applicable to scientific and general-purpose applications or applications running only on multicore machines. Various work has explored different domains where parallelization is beneficial. In this section, two notable examples are discussed: the application of TLS in scripting languages and the application of TLS in distributed systems. Appendix A.2 also briefly discusses the application of TLS in Graphics Processor Units (GPUs).

9.5.1. Scripting Languages. JavaScript lets developers add interactivity at the client side of a Web application. However, JavaScript is an interpreted language (which is slower than compiled languages) and due to its sequential nature it cannot take advantage of parallel multicore processors. Thus, optimizing JavaScript execution for Web applications is vital [Ratanaworabhan et al. 2010; Richards et al. 2010; Martinsen and Grahn 2011].

Martinsen et al. [2013] experimented by adding TLS functionality to JavaScript applications in order to take advantage of existing multicore processors. For the experiments, the SquirrelFish JavaScript engine was extended with TLS support. SquirrelFish was modified so that an instance of it executes as a thread. Speculation occurs at the method level in which case every function is executed on a different thread. Nested speculation is also supported in which case a speculative function can further spawn another speculative function encountered on its path. This forms a parent-child relationship between speculative functions in which case a child is more speculative than its parent and children functions must commit within the context of their parents to maintain speculative order (based on the original sequential order of the program).

Their system operates as follows: a Web application sends its compiled bytecode instructions to the modified SquirrelFish engine for execution as normally. When a function call is encountered, the state of the engine before the fork point is saved (i.e., lazy version management) and a new thread containing a new SquirrelFish engine will be spawned to execute the function. The parent thread continues execution from the function's continuation point. This process is repeated for each function encountered that could be a suitable candidate for speculation. The state preserved contains a list of previously modified global variables, a list of states from each thread, the register content, and a list of currently executing threads (ordered by speculation order). To detect data dependence violations the system checks for write and read conflicts between global variables, object property ID names, and function calls' unsuccessful return value predictions. Each global variable is shadowed by a list of reads and writes from all speculative threads along with thread IDs and speculation order of each thread. On every read/write the shadow lists are checked for data dependence violations (i.e., eager conflict detection). When a violation occurs, the offending thread and all more speculative threads are squashed, values are restored from saved state, and reexecution begins. When a speculative thread reaches the end of its execution, all its modifications of global variables and object property IDs must be committed back to its parent thread. The commit cannot be completed before child threads from this thread have returned and committed their values back to their parent thread.

The results are promising showing good speedups against competing JavaScript engines. Nevertheless, the authors observe a small number of rollbacks in the type applications they experimented, which may not always be the case for general-purpose applications.

9.5.2. Distributed Systems. Datasets across all domains are increasing in size exponentially [Lynch 2008]. The need for efficient processing of such large volumes of data gave rise to recent developments in the area of distributed computing such as MapReduce [Dean and Ghemawat 2004] and Hadoop [2005] that allow processing of large-scale data on share-nothing clusters of commodity hardware. These frameworks are primarily used for scientific and datacenter applications that can be broken up to independent tasks and avoid the high internode communication cost of cluster computing. General-purpose applications are not very poplar in such environments since they are characterized by irregular data access patterns and complex control flow, which entails high internode communication cost for remote accesses of shared data.

Kim et al. [2010] propose Distributed Software Multithreaded Transactional memory (DSMTX), a framework with optimized communication mechanisms that allows general-purpose applications that support speculation to be executed on nonshared memory clusters.

DSMTX uses the same Application Programming Interface (API) as SMTX [Raman et al. 2010] allowing existing applications parallelized using this paradigm to be executed on clusters without any modifications (apart from functions to initialize and finalize Message Passing Interface (MPI)). DSMTX is implemented on top of MPI and provides a unified virtual address space on top of the message-passing machine in order to allow each thread to initialize its memory state without an address translation, as on a shared memory machine.

Since the memory system of a cluster is physically distributed across multiple nodes without a globally shared address space, remote data must be explicitly sent and received between a producer-consumer pair using a message-passing protocol such as MPI. DSMTX aims to optimize this aspect since nearly all operations of MTX require communication among the workers, try-commit unit, and commit unit. To facilitate more efficient communication two mechanisms were implemented. First, a mechanism that transfers data to workers only when needed. This is achieved by adding access protections to each worker thread's heap space at the beginning of parallel execution. This results in a page fault when a thread accesses a memory location, effectively causing a transfer of data from the commit unit to the worker. Second, to avoid memory locations updated by threads executing different stages of the pipeline on the same page, DSMTX operates at a finer level of granularity, namely, *word*-level granularity.

DSMTX also optimizes the way MPI sends and receives data. MPI primitives for these operations are expensive in relation to the amount of data transferred. DSMTX provides enhanced message queues that allow produced values to be buffered and transferred only when the buffer fills up to a profitable size.

Apart from enabling speculative DSWP-style and other TLS techniques on both shared memory systems and message-passing systems, as the authors point out, DSMTX may also be useful for emerging manycore architectures that discard chip-wide cache coherence [Howard et al. 2010]. These architectures pose challenges similar to those found in clusters. DSMTX can add value to these platforms by enabling scalable parallelization of a variety of codes.

9.6. Diversity in TLS Approaches

TLS approaches are often tuned to the workload and architecture at hand. The reason for such diversity illustrated in this article is the range of behaviors present in various architectures and workloads as well as the natural improvements of TLS.

One important aspect that has driven TLS design is the multicore evolution. Early TLS systems were targeting high-end multiprocessor machines and Fortran applications drawn from a number of scientific and engineering research areas. As multicore machines now became pervasive there is an increasing need in exploiting the abundant computing resources made available by these technological advances. Thus, later TLS research was directed toward multicores as well as supporting applications from a greater spectrum (*Architecture* and *Workload Support* columns, respectively, in Table III). The next paragraphs provide some instances where TLS systems were optimized to support different workloads and architectures.

Early TLS systems were optimized for loops over fixed-size array-based data structures. The evolution of programming languages and multicores as well as the need for general-purpose applications support led engineers to develop TLS systems that are also suitable for parallelizing programs that use dynamic data structures that may

Table III. Workload, Architecture, and Synchronization Methods that TLS Systems Described in this Article are Applicable For

Work	Workload Support	Architecture	Synchronization Method
LRPD test [Rauchwerger and Padua 1995]	Static Memory	Alliant FX SMP—Intel i860	No synchronization
Gupta and Nim [Gupta and Nim 1998]	Static Memory	IBM G30 SMP—PowerPC	Locks
Softspec [Bruening et al. 2000]	Static Memory	Digital Alpha Server 800—Alpha	Memory barriers
Rundberg and Stenström [Rundberg and Stenström 2001]	Static Memory	SPARC V8	Locks
R-LRPD [Dang et al. 2002]	Static Memory	ccUMA HP-V2200 Server—PS-RISC	No synchronization
Cintra and Llanos [Cintra and Llanos 2003, 2005]	Static Memory	SPARC V9	Memory barriers
CorD [Tian et al. 2008, 2010]	Dynamic Memory Allocation	IA-32, ×86-64	Central manager thread
SpLIP [Oancea et al. 2009]	Dynamic Memory Allocation	IA-32, ×86-64	No synchronization
STMLite [Mehrara et al. 2009]	Dynamic Memory Allocation	SPARC V9	Central manager thread
SMTX [Raman et al. 2010]	Dynamic Memory Allocation	IA-32, ×86-64	Central manager thread
MiniTLS [Yiapanis et al. 2013]	Dynamic Memory Allocation	SPARC V9	CAS, LL/SC
Lector [Yiapanis et al. 2013]	Dynamic Memory Allocation	SPARC V9	CAS, LL/SC

grow (or change structurally in other ways) during speculative execution and loops with arbitrary control flow (*Workload Support* column in Table III).

Raman et al. [2010] introduced SMTX to provide software speculation support for DSWP [Ottoni et al. 2005b] in order to exploit the finer-grained parallelism available in general-purpose applications.

Tian et al. [2010] proposed several optimizations to support heap-intensive programs that operate on linked dynamic data structures in the context of state separation based speculative parallelization [Tian et al. 2008].

The LRPD test [Rauchwerger and Padua 1995] was not designed to efficiently handle misspeculations. Its primary focus was for applications where a potential data dependency does not eventually manifest at runtime. Even though this approach provides a scalable system, it also generates wasted work and limits the potential to exploit parallelism when data dependencies do exist in an application. Thus, in order to reduce wasted work and provide support for a wider variety of applications, later TLS systems were designed in such a way so that there is an occasional communication between threads to verify the absence/presence of conflicts. A number of approaches utilize a shared mutable shadow data structure, which requires the use of mutual exclusive locks, to achieve this communication. Later TLS systems were optimized to specific architectures in an attempt to replace these locks with lighter forms of synchronization (*Synchronization Method* column in Table III). Cintra and Llanos [2003] carefully designed their system by placing memory fences in key points of their implementation so that it guarantees correctness even when using the most relaxed memory consistency model of the *SPARC* architecture. Yiapanis et al. [2013] implemented a non-locking TLS system that uses CAS instructions rather than locks to access the shared data structure in a more efficient way. This design is applicable to all architectures that support *CAS* or *Load-Linked/Store-Conditional (LL/SC)* instructions. Oancea et al. [2009] eliminate the use of atomic *CAS* operations and memory fences in their design by taking advantage of the caching behavior of the ×*86* architecture. Recent TLS approaches optimize the bottleneck arising from using shared shadow structures by separating the speculative from the nonspeculative state, thus effectively removing a shared access point and abolishing locks (Section 6.3).

Table III shows the workload types and architectures that the TLS systems described in this article are tuned for as well as their synchronization model.

As different workloads and systems evolve and certain areas become more popular, TLS is expected to also follow directions along those lines. Some notable examples discussed in this article are the application of TLS in scripting languages (Section 9.5.1), distributed systems (Section 9.5.2), and GPUs (Appendix A.2).

Advances in STM (Appendix A.3) have also influenced software TLS design. For example, eager version management implemented by Oancea et al. [2009] and `MiniTLS` [Yiapanis et al. 2013], had been already successfully tested under various STM implementations [Saha et al. 2006; Harris et al. 2006]. Nonetheless, hardware TLS with eager version management had been previously explored in Steffan et al. [2000] and Cintra et al. [2000]. Further, `STMLite` [Mehrara et al. 2009] extends a state-of-the-art transactional memory model with support for speculative parallelization.

Other design parameters can be tweaked depending on the workload, as explained in Section 9.3. For example, lazy version management is more suitable on applications with frequent data dependencies across threads as opposed to eager version management that better addresses scenarios where data dependencies are rare. A system that detects conflicts at the object level is more suitable for workloads where memory accesses to the same objects by multiple threads are rare. A system that detects conflicts eagerly is more suitable for workloads with frequent data dependencies. A potentially optimal TLS system should be able to adapt to different scenarios at runtime. Profiling the runtime application could provide indications as to which parameters would be more suitable for a given scenario and adapt the system accordingly. Research in STM shows that such a methodology can potentially offer performance improvements to existing methods with relatively low adaptivity costs [Spear 2010]. Finally, future TLS will continue to refine and innovate as systems and workloads evolve.

9.7. Leveraging Sequential Programs for Multicores

Sometimes lack of parallelism may be due to artificial constraints imposed by sequential execution models. For example, Bridges et al. [2007] indicate that even though many programs allow for a range of legal outcomes to exist, there is no means for that to be specified by the software developer. As a result, the compiler is forced to maintain the single correct output that a sequential application specifies, even when others are more desirable. The authors propose some simple extensions to the sequential programming model to overcome this limitation by allowing the developer to annotate the code and help the compiler to extract further parallelism by ignoring certain dependencies. Two annotations are proposed: *Y-branch* and *Commutative*.

The Y-branch indicates a branch that can be taken with a specified probability irrespective of whether the corresponding conditional evaluates to true. The authors give an example of a dictionary for a compressor program that needs to be refreshed every so often.

The commutative annotation is applied on functions, and informs the compiler that invocations to these functions can be scheduled in any order (but in a synchronized manner) even though dependencies may exist between them. A common example is the random number generator function.

These extensions allow a software developer to develop in a sequential programming model, but still obtain performance from parallelization.

10. SUMMARY

Static compiler approaches to automatic parallelization can be successful but fail to parallelize code where sufficient information is not known until runtime. Initial attempts to runtime thread-level parallelism applied the Inspector/Executor model where an inspector phase determines at runtime whether a loop is suitable to run in parallel, an action accomplished by the executor phase. Overheads associated with the inspection phase of that model directed research in runtime parallelization toward a more promising solution: *speculative parallelization* (also known as *TLS*).

Speculative parallelization executes a code in parallel optimistically (without knowledge whether the code can be parallelized) and provides mechanisms to maintain correctness during runtime. This entails ways to schedule speculative threads, ways to monitor accesses by speculative threads, ways to detect undesirable actions, and prevent wrong results to be maintained in main memory. Part I of this article, “Fundamentals of Speculative Parallelization,” elaborated on those mechanisms and explained different ways they can be implemented.

Part II, “Advanced Topics in Speculative Parallelization,” discussed the advances in compiler-driven speculative parallelization for thread-level parallelism. The discussion was broken down based on the main design choices on implementing a TLS system: execution model, metadata, version management, conflict detection, commit/rollback, and work scheduling for speculative threads. Each of these sections described how different work in the literature implements a particular design dimension. Table IV summarizes the design choices that each surveyed TLS system follows. The article concluded with a critical discussion on designing TLS systems.

A. APPENDIX—FURTHER BACKGROUND READING

A.1. Compiler Transformations and Automatic Nonspeculative Parallelization

Although the focus of this article is on techniques for parallelizing irregular applications (where access patterns are unknown until runtime), a plethora of work has also been produced on regular (typically numerical) application parallelization. When the access patterns of an application can be calculated at compile-time (offline), it is sometimes possible for the compiler to prepare the code for parallelization. Several techniques have been proposed and applied over the years targeting mainly loop parallelization. The compiler examines the memory accesses across loop iterations to determine if they are independent of one another (that is, no flow of values or reuse of memory locations across iterations). If loop iterations are independent, the code can run in parallel without any additional transformations. Even when loop iterations are not independent, there are certain transformations, such as *privatization* and *reduction*, that can be applied to remove the dependencies and enable parallelization. A compiler that performs dependence analysis and transformations for parallelization is called a *parallelizing compiler*.

Excellent resources for learning about data dependence analysis as well as automatic parallelization techniques include Kennedy and Allen [2002], Midkiff [2012], as well as the survey of compiler transformations by Bacon et al. [1994].

Compile-time and runtime parallelization techniques are not mutually exclusive. Certain offline techniques can be used to modify the code in a way that is more runtime parallelization friendly. For instance, optimizations such as *loop distribution* and *loop invariant hoisting* can help to decouple an *inspector* from its *executor* loop by reducing the control flow affecting data accesses between the two [Rauchwerger and Padua 1994b].

Table IV. Design Choices for Main Work in the Literature of Speculative Parallelization

Work	Execution Model	Versioning	Conflicts	Commit	Rollback	Scheduling	Supports Pointers and/Dynamic Allocation
`LRPD` test [Rauchwerger and Padua 1995]	Speculative	Lazy	Lazy	Serial	Parallel	Static/Dynamic	NO
Gupta and Nim [Gupta and Nim 1998]	Speculative	Lazy	Lazy	Serial	Parallel	Static /Dynamic	NO
`Softspec` [Bruening et al. 2000]	Speculative	Eager	Lazy	Parallel	Parallel	Static /Dynamic	NO
Rundberg and Stenström [Rundberg and Stenström 2001]	Speculative	Lazy	Eager	Parallel	Parallel	Static/Dynamic	NO
`R-LRPD` [Dang et al. 2002]	Speculative	Lazy	Lazy	Serial	Parallel	Sliding Window	NO
Cintra and Llanos [Cintra and Llanos 2003, 2005]	Speculative	Lazy	Eager	Serial	Parallel	Sliding Window	NO
`CorD` [Tian et al. 2008, 2010]	Speculative	Lazy	Lazy	Serial	Parallel	Static/Dynamic	YES
`SpLIP` [Oancea et al. 2009]	Speculative	Eager	Eager	Parallel	Serial	Static/Dynamic	YES
`STMLite` [Mehrara et al. 2009]	Speculative	Lazy	Lazy	Parallel	Parallel	Static/Dynamic	YES
`SMTX` [Raman et al. 2010]	Speculative/ DSWP	Lazy	Lazy	Parallel	Parallel	Static/Dynamic	YES
`MiniTLS` [Yiapanis et al. 2013]	Speculative	Eager	Eager	Parallel	Parallel	Sliding Window	YES
`Lector` [Yiapanis et al. 2013]	Speculative/ Inspector	Lazy	Eager	Serial	Parallel	Sliding Window	YES

A.2. Hardware Support for Speculative Parallelization

One of the main overheads of a software TLS system arises from maintaining the speculative state. Loads and stores need to be examined globally using locks, before they are inserted in their local data structures. As a consequence, many clock cycles are wasted simply to synchronize and track such information. And even if that was not enough, during rollback, values may need to be locked and restored causing further clock cycles. Work in TLS has shown that using hardware might avoid these redundant cycles by taking advantage of certain properties of the coherence protocol [Gopal et al. 1998]. For example, cache lines can be modified with additional information, an extra bit and a pointer, to facilitate marking of speculative data. The bit, known as the "load bit," is set when a processor loads a value not produced by itself (i.e., a potential RAW violation). The extra pointer indicates the next most speculative cache line that has a version of that datum in case a value needs to be forwarded to a remote cache. Furthermore, L1 cache is normally private to the processor and thus buffering occurs at the L1 cache lines by default. Not only buffering is cheap but also marking simply needs to write a bit and change a pointer. Software approaches, on the other hand, rely on additional data structures to maintain marking information. Thus, each potential speculative load or store will produce additional instructions from the hardware point of view (to read and insert that item into the data structure). These data structures are normally accessed via locks or *CAS* operations to avoid data races between different threads. Maintaining a speculative state is the major slowdown in software TLS [Yiapanis 2013]. Nevertheless, there are other operations that get accelerated when implemented in hardware. For instance, spawning threads in hardware uses significantly fewer processor cycles in comparison with software. Squashing a thread might simply boil down to flushing a processor's private cache. Committing values to main memory can be part of the hardware mechanism for propagating local cache values to memory. Notable TLS architectures and further reading on that topic include Oplinger et al. [1997], the Hydra TLS project [Hammond et al. 1998], the STAMPede project [Steffan et al. 2005], Renau et al. [2005a, 2005b], Marcuello and González [1999], and Cintra et al. [2000].

Advances in GPUs attracted a lot of attention in recent years as a way of parallelizing general-purpose applications due to their efficient way of exploiting thread-level parallelism. Zhang et al. [2013] propose a runtime system, `GPU-TLS`, for parallelizing loops speculatively on GPU architectures. Under GPU-TLS, a potentially parallel loop is broken down into several subloops and each subloop is coupled with a GPU kernel. Speculative threads are scheduled using the sliding window scheduling mechanism and the correct values are committed in parallel to main memory. The authors employ a hybrid dependency checking scheme in which small groups of threads are eagerly checked between them while there is a global lazy dependency checking at the end of the speculative execution. Their system also applies *value forwarding* to move data already produced to more speculative threads that might require them.

A.3. Transactional Memory

Speculative execution resembles some of the mechanisms required in *Transactional Memory (TM)* [Herlihy and Moss 1993], another form of optimistic execution. In TM, parallel portions of applications are executed concurrently as transactions and access shared data simultaneously. In the TLS paradigm, a sequential program is first transformed to parallel and then executes speculatively. In TM, the input program is already parallel but instead of using locks to protect against data races, synchronization is achieved by transactions. *Transactional* threads in TM have the same guarantees as in TLS apart from the requirement to commit in a predefined total order (the sequential order). As such, transactional memory can be seen as a less restrictive form of speculation when compared to TLS. `STMLite` [Mehrara et al. 2009], presented earlier in this article, is one example of a TM system adapted for TLS execution. Harris et al. [2010] provide an overview of the state-of-the-art in the design and implementation of TM systems. In contrast to transactional memories that typically target "short critical sections" (few instructions), TLS systems usually target "long-running transaction" (outer-loop iterations, which are potentially several thousands of instructions).

Rock, developed by Sun Microsystems, was intended to be the first general-purpose processor to support TM, but it was never commercialized. Nevertheless, TM finally made its way to hardware as part of the Intel Haswell processor and Blue Gene/Q [Wang et al. 2012].

REFERENCES

David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran, and Wen mei W. Hwu. 1998. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 227–237.

David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler transformations for high-performance computing. *Computing Surveys* 26, 4 (1994), 345–420.

Hans-J. Boehm. 1996. Simple garbage-collector-safety. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 89–98.

Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. 2007. Revisiting the sequential programming model for multi-core. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 69–84.

Matthew Bridges. 2008. *The VELOCITY Compiler: Extracting Efficient Multicore Eexecution from Legacy Sequential Codes*. Technical Report. Princeton University.

Derek Bruening, Srikrishna Devabhaktuni, and Saman Amarasinghe. 2000. Softspec: Software-based speculative parallelism. In *Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*.

Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 227–238.

Ding Kai Chen, Josep Torrellas, and Pen Chung Yew. 1994. An efficient algorithm for the run-time parallelization of DOACROSS loops. In *Proceedings of the International Conference on Supercomputing (ICS)*, 518–527.

Michael K. Chen and Kunle Olukotun. 2003. The Jrpm system for dynamically parallelizing Java programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 434–446.

Marcelo Cintra and Diego Llanos. 2005. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems* 16, 6 (2005), 562–576.

Marcelo Cintra and Diego R. Llanos. 2003. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

Marcelo Cintra, José F. Martínez, and Josep Torrellas. 2000. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 13–24.

Ron Cytron. 1986. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, 836–844.

Francis Dang, Hao Yu, and Lawrence Rauchwerger. 2002. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, 20–29.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of Conference on Symposium on Opearting Systems Design and Implementation (OSDI)*.

Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. 2007. Software behavior oriented parallelization. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 223–234.

María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. 2005. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions in Architecture and Code Optimization* 2, 3 (September 2005), 247–279.

Sridhar Gopal, T. Vijaykumar, James Smith, and Gurindar Sohi. 1998. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, 195–215.

Manish Gupta and Rahul Nim. 1998. Techniques for speculative run-time parallelization of loops. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 1–12.

Apache Hadoop. 2005. Apache Hadoop. http://hadoop.apache.org/. (2005). Accessed February 2, 2015.

Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data speculation support for a chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 58–69.

Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory* (2nd ed.). Morgan and Claypool Publishers.

Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. 2006. Optimizing memory transactions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 14–25.

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, 289–300.

Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. 2010. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 108–109.

Shiwen Hu, Ravi Bhargava, and Lizy Kurian John. 2003. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelsim* 5 (2003), 1–21.

Nikolas Ioannou, Jeremy Singer, Salman Khan, Paraskevas Yiapanis, Adam Pocock, Polychronis Xekalakis, Gavin Brown, Mikel Luján, Ian Watson, and Marcelo Cintra. 2010. Toward a more accurate understanding of the limits of the TLS execution paradigm. In *Proceedings of the IEEE International Symposium on Workload Characterization*.

Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. 2012. Speculative separation for privatization and peductions. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 359–370.

Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. 2004. Min-cut program decomposition for thread-level speculation. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 59–70.

Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. 2010. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 3–14.

Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. 2006. POSH: A TLS compiler that exploits program structure. In *Proceedings of the International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 158–167.

Mikel Luján, Phyllis Gustafson, Michael Paleczny, and Christopher A. Vick. 2007. Speculative parallelization—Eliminating the overhead of failure. In *Proceedings of the 3rd International Conference on High Performance Computing and Communications (HPCC)*, 460–471.

Clifford Lynch. 2008. Big data: How do your data grow? *Nature* 455, 7209 (2008), 28–29.

Pedro Marcuello and Antonio González. 1999. Exploiting speculative thread-level parallelism on a SMT processor. In *Proceedings of the International Conference on High-Performance Computing and Networking*, 754–763.

Pedro Marcuello and Antonio González. 2002. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, 55–67.

Jan Kasper Martinsen and Hakan Grahn. 2011. A methodology for evaluating JavaScript execution behavior in interactive web applications. In *Computer Systems and Applications (AICCSA)*. 241–248.

Jan Martinsen, Hakan Grahn, and Anders Isberg. 2013. Using speculation to enhance javaScript performance in Web applications. *IEEE Internet Computing* 17, 2 (2013), 10–19.

Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 166–176.

Samuel P. Midkiff. 2012. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool Publishers.

Erik M. Nystrom, Hong-Seok Kim, and Wen-Mei W. Hwu. 2004. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of 11th Static Analysis Symposium (SAS)*, 165–180.

Cosmin Oancea, Alan Mycroft, and Tim Harris. 2009. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 223–232.

Jeffrey Oplinger, David Heine, Shih Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. 1997. *Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor*. Technical Report CSL-TR-97-715. Stanford University.

Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005a. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 105–118.

Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005b. Automatic thread extraction with decoupled Software Pipelining. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, 105–118.

Christopher J. F. Pickett and Clark Verbrugge. 2006. Software thread level speculation for the Java language and virtual machine environment. In *Proceedings of the International Conference on Languages and Compilers for Parallel Computing (LCPC)*, 304–318.

Manohar K. Prabhu and Kunle Olukotun. 2005. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 142–152.

Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 65–76.

Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 114–123.

Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. 2008. Performance scalability of decoupled software pipelining. *ACM Transactions on Architecture and Code Optimization* 5, 2, Article 8 (2008), 8:1–8:25 pages.

Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled software pipelining with the synchronization array. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 177–188.

Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps)*.

Lawrence Rauchwerger. 1998. Run-time parallelization: Its time has come. *Parallel Computing* 24, 3–4 (1998), 527–556.

Lawrence Rauchwerger and David Padua. 1994a. *Speculative Run-Time Parallelization of Loops*. Technical Report CSRD-827. Center for Supercomputing Research and Development, University of Illinois.

Lawrence Rauchwerger and David Padua. 1994b. The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th International Conference on Supercomputing (ICS)*, 33–43.

Lawrence Rauchwerger and David Padua. 1995. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 218–232.

Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck, and Josep Torrellas. 2005a. Thread-level speculation on a CMP can be energy efficient. In *Proceedings of the International Conference on Supercomputing*, 219–228.

Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. 2005b. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *Proceedings of the Internatonal Conference on Supercomputing*, 179–188.

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 1–12.

Peter Rundberg and Per Stenström. 2001. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism* 3 (2001), 1–28.

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 187–197.

Joel H. Salz and Ravi Mirchandaney. 1991. The preprocessed doacross loop. In *Proceedings of ICPP*, 174–178.

Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. 1989. The doconsider loop. In *Proceedings of ICS*. 29–40.

Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. 1991. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers* 40, 5 (1991), 603–612.

Michael F. Spear. 2010. Lightweight, robust adaptivity for software transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 273–283.

Michael F. Spear, Virendra J. Marathe, William N. Scherer, and Michael L. Scott. 2006. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the International Conference on Distributed Computing (DISC)*, 179–193.

Gregory Steffan. 2003. *Hardware Support for Thread-Level Speculation*. Doctoral dissertation. Carnegie Mellon University Pittsburgh, PA.

Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. 2005. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems* 23, 3 (2005), 253–300.

Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2000. A scalable approach to thread-level speculation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1–12.

Peiyi Tang and Pen-Chung Yew. 1986. Processor self-scheduling for multiple nested parallel loops. In *Proceedings of the International Conference of Parallel Processing*, 528–535.

William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. 2007. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 356–369.

Chen Tian, Min Feng, and Rajiv Gupta. 2010. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 62–73.

Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. 2008. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st Annual International Symposium on Microarchitecture (MICRO)*, 330–341.

Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative decoupled software pipelining. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 49–59.

Amy Wang, Matthew Gaudet, Peng Wu, Jose Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. 2012. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 127–136.

Paraskevas Yiapanis. 2013. *High Performance Optimizations in Runtime Speculative Parallelization for Multicore Architectures*. Ph.D. dissertation. School of Computer Science, University of Manchester.

Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown, and Mikel Luján. 2013. Optimizing software runtime systems for speculative parallelization. *ACM Transactions on Architecture and Code Optimization* 9, 4, Article 39 (2013), 39:1–39:27.

Chenggang Zhang, Guodong Han, and Cho-Li Wang. 2013. GPU-TLS: An efficient runtime for speculative loop parallelization on GPUs. In *Proceedings of the International Symposium on Cluster, Cloud, and Grid Computing (CCGRID)*. 120–127.

Hongtao Zhong, Mojtaba Mehrara, Steven A. Lieberman, and Scott A. Mahlke. 2008. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the International Conference on High-Performance Computer Architecture (HPCA)*, 290–301.

Chuan-Qi Zhu and Pen-Chung Yew. 1987. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering* 13, 6 (1987), 726–739.

Received August 2014; revised July 2015; accepted September 2015