Speculative Out-Of-Order Event Processing with Software Transaction Memory

Andrey Brito
TU Dresden, Germany
andrey.brito@inf.tu-dresden.de

Christof Fetzer
TU Dresden, Germany
christof.fetzer@inf.tu-dresden.de

Heiko Sturzrehm
University of Neuchâtel, Switzerland
heiko.sturzrehm@unine.ch

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

ABSTRACT

In event stream applications, events flow through a network of components that perform various types of operations, e.g., filtering, aggregation, transformation. When the operation only depends on the input events, one can trivially parallelize its processing by replicating the associated components. This is not possible, however, with stateful components or when there exist dependencies between the events. Parallel versions of a number of simple stream mining operators have been designed, but, in general, complex and user-defined operators are limited by single thread performance. In this paper, we propose leveraging the processing capabilities of multi-core processors to improve the efficiency of stateful components using optimistic parallelization techniques (as provided by transactional memory). We show that, even though some speculative event executions might need to be disregarded, the overall throughput increases noticeably in the general case and latency can be reduced by pre-processing out-of-order events. Moreover, we show how simple conflict predictors can boost the parallelism even more and reduce the amount of resources used for a given level of parallelism.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming, parallel programming*

General Terms

Algorithms, Design, Performance

Keywords

Event stream processing, complex event processing, software transactional memory

1. INTRODUCTION

Event stream processing (ESP) applications [2] operate on data streams with the goal of identifying meaningful events or summarizing large sets of low-level events into fewer, more significant ones. Events typically traverse a network of components or modules, each of which takes a stream as input, processes it (e.g., by filtering, aggregating, or transforming data), and produces a resulting output stream. Scalability is a big challenge for ESP applications that have to process infinite streams of events that arrive at a high rate (e.g., network monitoring applications).

When event processing is stateless, that is, the output stream only depends on the input events, the operation of a component can be trivially parallelized by executing multiple instances of the component. An example of such stateless processing would include basic filters (e.g., remove invalid events from the stream) and simple transformations (e.g., normalize event values).

When processing is stateful, one cannot simply improve performance by replicating the components. First, multiple copies of the same component would need to maintain a consistent replicated state, which is in general non trivial and usually adds significant overhead. Second, events must most often be processed in a specific order, either because they have dependencies with one another or because the effect on the component's state depends on the processing order. It is important to note that, even when events arrive in the right order at stateful component, they typically *cannot* be processed in parallel.

In this paper, we consider ESP systems with components connected in a cascade (see Figure 1). We are interested in parallelizing the operations of the components by exploiting the processing capabilities of multi-core architectures. Some components are stateless and can be trivially parallelized, but they may reorder the events. Other components are stateful and must typically (but not necessarily) process events in order. We assume that the order of events is determined when they enter the system, e.g., by associating monotonically increasing logical timestamps with each of them.

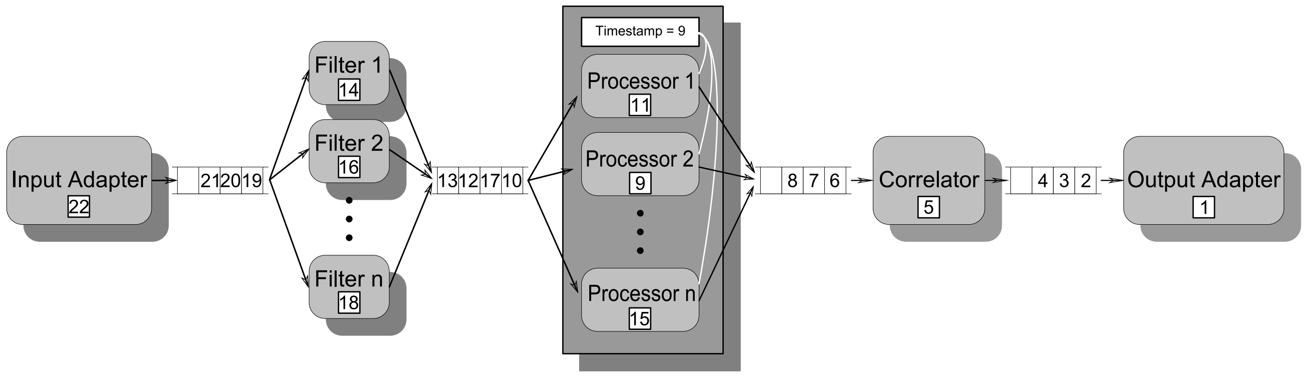

Figure 1: A simple typical ESP application network.

Example 1. *Consider the simple ESP application network shown in Figure 1. The stream is first processed by an input adapter that transmits events in order to a filter component. As the filter is stateless (e.g., it parses an XML event and converts them into a native format), one can have multiple instance of the component that process events in parallel. The processing time might not be the same for each event and the outgoing stream might be "slightly" out of order. For instance, we observe in the figure that event* 17 *has overtaken event* 12. *The unordered stream enters a stateful "processor" component (e.g., stream query operator) that uses speculative execution to account for out-of-order executions. It allows events to be processed optimistically but does not output them until all preceding events have been completed. The stream then traverses a correlator component that forwards events in order to the output adapter.*

The approach used in this paper consists of using speculative execution to process events in parallel that should normally be processed sequentially, even when they are received out-of-order. In short, we use an underlying software transactional memory (STM) [6] infrastructure to optimistically process the events in the context of transactions. STM provides the concept of transactions on the programming language level. It has become very popular recently as an approach to develop concurrent programs for many-core CPUs. So far, however, STMs do not provide features to pre-order transactions that execute in parallel. Hence, we have extended the design of the STM so that we can pre-assign commit timestamps to transactions, effectively imposing an order in which they need to complete. Transactions will execute in parallel but may have to delay their completion (i.e., their commit) when ordering is required.

The key intuition behind our approach is that an STM will dynamically detect dependencies between events, if any, and will sequentialize the processing only when necessary (possibly delaying, or aborting and restarting some transactions). Without speculative execution, ESP components would not only have to execute events sequentially one at a time, but may have to *wait* idle when events are not received in the right order. As we shall see, the increased parallelism of our approach yields substantial performance benefits.

There is a thin line between being optimistic and being overly optimistic. Speculative execution of a transaction that has a high likelihood of having to ultimately abort (e.g., because it is "too much" out-of-order) can prove to be counterproductive. We also study the problem of conflict prediction, i.e., mechanisms for deciding whether or not it makes sense to speculatively process an event at a given time. Experimental evaluation shows that a good conflict predictor can have a significant influence on event throughput.

This paper makes the following contributions. We propose a novel approach for speculative execution of events with dynamic conflict detection in an ESP system. To the best of our knowledge, this is the first application of transactional memory for out-of-order event processing. We study the conditions under which parallel processing can improve throughput over sequential execution and we introduce the notion of conflict predictor as a way to drive the parallelization process toward the most promising executions. We have implemented an event processing engine that uses a modified version of TinySTM [5]. We have conducted an experimental evaluation of the performance of our implementation on several workloads.

The rest of the paper is organized as follows. We discuss the related work in Section 2 and outline the system model in Section 3. Section 4 describes our speculative execution algorithm and introduces the notion of a conflict predictor. Section 5 presents our experimental evaluation and Section 6 concludes the paper.

2. RELATED WORK

In recent years, several works have addressed the scalability of event stream processing, but all of them take an approach different from ours. For example, Koparanova and Risch [7] have developed GSDM (GRID Stream Data Manager) to handle processing of data produced in real time by a large amounts of sensors that receive signals from space. Their approach is to use data partitioning to split the events between several replicated components and then merge the results. This approach is, however, applicable only for computations in which the processing can be split into several units that do not require any synchronization (e.g., in their case, Fast Fourier-Transforms). In addition, it needs components to split and merge the data and these components need to consider the semantics of the computation. Other works, like Borealis [1] or more recently StreamFlex [11], use the assumption that components are normally stateless and scalability can then be easily achieved through simple replication.

Another research direction focuses on the problem of efficient correlation of events. Correlation is strongly dependent on state and therefore difficult to parallelize. SASE [14] is an example of an event stream processing system that focuses on matching event patterns efficiently.

With respect to out-of-order processing of events, one classical approach is to buffer an event until it is known that no prior events may arrive. CEDR (Complex Event Detection and Response) [3] tries to balance insensitivity to event arrival and system performance. It uses a temporal model that deals with out-of-order events by allowing results to be output and later be retracted and revised in case a relevant event arrives. The main motivation is that in some contexts an early, but conceivably wrong, result is more valuable than having buffers and delays to cope with late events. Similarly, Li and colleagues [8] propose an extension for SASE that allows a pattern to be tardily matched when some late event arrives.

The idea of optimistically executing tasks that may have undetected dependencies has been around for a long time. Steffan and Mowry proposed Thread Level Data Speculation (TLDS) [12] as a way to benefit from multiprocessor computers when the programs are not designed for explicit parallelism. They suggested that compiler support could enable activities in a sequential program to be executed simultaneously and then committed from the less speculative activity to the most speculative one. If some dependencies are detected the speculative activity is terminated and restarted.

Software Transactional Memory (STM) [6] is another approach to parallelize programs. On the one hand, TLDS creates parallel tasks from sequential code, and on the other hand, STMs are mainly used to help synchronizing already parallel code. Good synchronization is crucial to performance, but implementing it correctly and efficiently requires strong programming skills. STMs facilitate concurrent programming by enabling to specify blocks of code (transactions) that must execute atomically and in isolation without worrying about how synchronization is implemented. Transactions are executed optimistically under the control supervision of the STM and, upon conflict, may be rolled back and re-executed. Modern STMs typically monitor accesses the memory and use some form of global time base to efficiently guarantee that data accesses are consistent [10, 13, 5].

The increase of available processing cores per machine has motivated the development of STMs and in some contexts they already outperform programs that were laboriously synchronized by hand [4]. In this work, we use TinySTM [5], an open-source STM, to implement speculative event processing.

3. BACKGROUND AND SYSTEM MODEL

We consider event stream processing (ESP) systems that consist of a network of components, which manipulate the events as they flow through them. Each of the components has a certain number of input and output queues depending on the operation it has to fulfill. Components behave as black boxes: their internal operation is not visible to other components upstream or downstream. In this section, we discuss how speculative processing is implemented and how new stateful components can be developed.

3.1 Enhanced ESP Components

A stateful component enhanced with support for speculative execution is similar to a regular component from the outside: it supports input and output queues. The main difference is that events in the input queues may be unsorted. In the output queue, they will be sorted according to their timestamps.

An enhanced ESP component has several threads working in parallel. The number of threads typically depends on the processing capabilities (number of cores) of the system that hosts the component. Each thread can access the input queues and retrieve events to be processed. The manipulation of the event is performed in the context of a transaction, which means that modifications are invisible to other threads until the transaction commits.

The STM-enhanced components use an underlying time-based STM (TinySTM [5]) that utilizes a shared commit counter to maintain consistent snapshots of memory locations read by transactions without incurring the cost of incremental validation. Commit timestamps are essentially used to linearize transactions and detect whether the content of a memory location can be safely accessed (i.e., is consistent with the transaction execution order). We rely on the use of commit timestamps in our speculative parallelization approach.

Unlike the classical behavior of an STM, transactions cannot complete in any order and threads do not automatically commit their transactions. Instead, transactions have pre-assigned commit timestamps (determined according to the timestamp of the associated events). A thread checks if a transaction can commit by comparing its timestamp with the current commit counter of the component. If both are equal, the transaction commits and the event can be sent to the output queue. Otherwise, the whole transaction is suspended and inserted in a waiting list. Each time a new event is processed, the list is checked to see if a waiting transaction can now be committed. It may happen that a transaction in the waiting list is aborted due to a conflict with another transaction; in that case it is restated.

As for a regular ESP component, the developer of an enhanced ESP component must provide a function `execute()` that implements the actual event processing. In addition, s/he may provide two functions specific to transactional operation: `onCommit()` executes upon successful completion of a transaction, while `postCommit()` runs after commit. The main difference between both functions is that executions of the former are serialized (necessary, for instance, when inserting the processed event in the output queue) while mul-

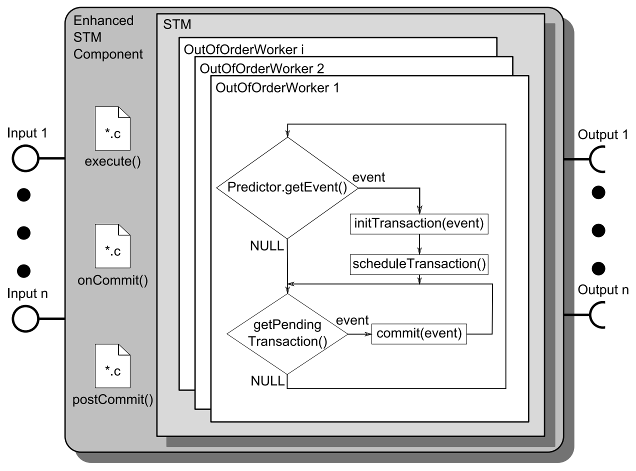

Figure 2: Enhanced ESP component internals.

tiple instances of the latter can execute in parallel (e.g., to gather statistics or perform cleanup operations). This difference has strong implications on performance, as we shall discuss later. Figure 2 illustrates the operations supported by the enhanced ESP component.

3.2 Assumptions

We base the development of our system on some assumptions about the way ESP components behave and about the environment. First, the events receive a logical timestamp as they enter the system, these timestamps need to be unique and continuous (i.e., no gaps). To keep that assumption valid throughout the system, each time an event is discarded (e.g., a filter that drops an irrelevant event) a null event inserted to carry the timestamp through the system. The null events are specially important if the components can be deployed in distributed computer nodes, as there is no way to distinguish a discarded event from a late one.

Second, the algorithms written by the user to process the events should obey certain constraints. The `execute()`, `onCommit()`, and `postCommit()` functions should guarantee progress (lockout-free). In addition, the `execute()` function cannot execute external actions as these cannot be rolled back in case that the transaction aborts.

Third and last, we assume a node has sufficient memory to keep (out-of-order) events in memory until they can be processed and committed.

4. STM FOR OUT-OF-ORDER EVENT PROCESSING

In this section, we first give a brief overview on how the underlying TinySTM[1] works. More detailed information can be found in [5]. We then describe the changes we needed to make to support event processing and how we use this extended STM implementation for out-of-order event processing.

[1]Available as open source from `http://tinystm.org`.

4.1 Software Transactional Memory Basics

STMs facilitate optimistic synchronization of threads: transactions are executed speculatively by multiple threads and, if a read/write or a write/write conflict is detected, one of the affected transactions is aborted (rollback) and retried. Since only memory changes are rolled back, most STMs do not permit the execution of actions with external effects such as I/O within a transaction. All operations on memory (read, write, allocation, release) are executed through the STM.

TinySTM has two modes of operation: *write-through* and *write-back*. With the write through approach, memory writes are intercepted and a copy of the original value is performed before a location is written to for the first time. Successive accesses to this location will directly read and update the modified values. In case the transaction is aborted, the copy is used to restore the original values. This approach has the drawback of exposing intermediary values from non-committed transactions to non-transactional code. In some applications, this could lead to problems like endless loops or crashes. In what follows, we assume that the STM uses a write-back approach.

When using write-back, memory writes within a transaction are intercepted and redirected to a private copy specific to that transaction, leaving the original value intact. Read accesses to a previously updated locations are similarly redirected to the private copy. Later, if the transaction commits, the original values are overwritten by the local copies.

To make sure that a transaction always operates on a *consistent snapshot* of the memory, TinySTM keeps track of several items. It maintains meta-data associated with memory locations. The meta-data is either a version number, which corresponds to the logical timestamp of the last committed transaction that wrote the location, or a lock, which indicates that location has been modified by an active transaction. The mapping from memory addresses to meta-data items is achieved using a hash function and was designed to reduce the probability of false conflicts.

Each transaction also maintains a read set and a write set to keep track of all the memory locations that have been accessed, as well as the list of the memory blocks that have been allocated or released.

When a transaction tries to read or write a memory location that is locked, it aborts in the hope that the other transaction will be able to complete while the first one restarts its execution. When a transaction completes without aborting, it tries to commit. This will succeed if, at commit time, all positions read by the transaction form a consistent snapshot. In that case, modified values are copied to main memory and the associated meta-data is updated to reflect the commit timestamp of the transaction. Freed memory blocks are also permanently deleted. If the transaction aborts, modified values are disregarded and allocated memory blocks are freed. The transaction is then restarted.

4.2 Event-Processing STM Features

Using TinySTM for event processing required some changes. The main new feature is the support for an application-given commit order. Each event is processed by a transaction and therefore, we want to commit transactions according to the logical time stamp of the processed event. Furthermore, we needed to improve TinySTM to support priorities and a better conflict resolution. The priority of a transaction is determined according to the logical timestamp of the event: transactions that should commit first should be able to "win" conflicts with other lower-priority transactions. For similar reasons, an aborted transaction should not be retried unless all earlier transactions it conflicted with have already been processed. If a transaction was aborted because of a conflict with a higher-priority transaction, it does not make sense to retry the transaction immediately. Aborted transactions go to a waiting list and will be reprocessed after the conflicting higher-priority transaction(s) have committed.

Another major change is the way a transaction is specified. Instead of considering a transaction as a block of code, we define it as a function call. Thus, the transaction contains the function name and the input parameters. If the transaction is aborted and later re-executed, the function will be called again with the same parameters. Local variables will then be reset while the state of the component will be protected by the STM. The insertion of wrappers to protect these accesses is done during the translation of the higher-level user specification of the component into lower-level functions.

In addition to the above changes, we had to perform a few more adaptations to TinySTM:

Conflict Resolution: When a lower-priority transaction reads a memory location that is being written to by a higher-priority transaction, as well as in the case of write-write conflicts, the lower-priority transaction should abort as it will have no chance to commit. We identify such conflicts and resolve them based upon the transactions' priorities. Higher priority transactions can abort lower priority ones, but not the opposite.

Work stealing: Threads and transactions are not coupled because retries are not immediate, a transaction may remain open for a long time and be continued by another thread.

Ordering: Reads must be validated at commit time for all transactions, even if they are read-only. Unlike regular transactions for which only a consistent snapshot of the memory is required, the values read by event transactions must still be valid at commit time.

External actions: In TinySTM, a failed transaction will keep retrying until it commits. As a consequence, statements that appears after the transactional code will be executed only after the transaction commits. In our case, the transaction may be put aside for a long time and control flow may return to the user code long before commit. We have thus introduced two special functions: `onCommit()` and `postCommit()`. The former contains actions that must be ordered with regard to other transactions (e.g., inserting ordered events in a queue) while the latter contains actions that should be executed after the commit but with no strict ordering requirements (e.g., clean up actions, monitoring).

Speculation: Not all transactions should be immediately processed. Initiating the processing of a transactions too early may lead to bad performance. To handle this issue, transactions are processed only after passing a test from a "predictor" that tries to determine good parallel schedules. We discuss this issue and the predictors in more detail in Section 4.5.

Garbage collection: Conflict resolution may cause transactions executed by one thread to hold references to transactions running on other threads. As these references could be invalidated when disposing of the transaction structures (transaction descriptors, read sets, writes sets) after commit, we use an epoch based garbage collection to ensure that such data will not be deleted too early.

4.3 STM Interface

To better describe our out-of-order event processing, we briefly discuss the main interface functions of our underlying STM:

`initTransaction()` initializes a transaction descriptor with the exception of the start timestamp that is determined upon the first transactional access (e.g., a read or write).

`scheduleTransaction()` starts processing a given event. If the transaction cannot commit immediately (e.g., if not all preceeding events have been committed yet), it is inserted in a wait list for a later commit and/or retry, freeing the current worker to execute another transaction.

`tryCommitTransaction()` tries to commit a given transaction. If the commit fails, the transactions is marked to be retried.

`getPendingTransactions()` searches in the wait list for a transaction associated with the current timestamp. If found and the transaction needs to be retried, the transaction is restarted and its event processing function is executed.

The `startTransaction()`, `scheduleTransaction()`, and `getPendingTransactions()` functions are used by the speculative event processing worker discussed below. The `tryCommitTransaction()` function is responsible for checking if a transaction may commit immediately or needs to wait. If the transaction timestamp is adequate, the read set will be validated. Validation needs to be performed only if some other transaction has committed during the processing of the event.

If the validation fails, the transaction is restarted. Otherwise, it commits and its modifications to the memory are made visible to other threads and locks are released.

Upon commit, the `onCommit()` actions are executed before the component's timestamp is incremented to strictly order their execution with respect to the commit order. Finally, the component's timestamp is incremented, potentially unlocking other transactions in other threads. The wait list is searched for transactions to reactivate and, finally, the `postCommit()` actions are executed.

4.4 A Speculative Event Processing Engine

After compiling the user specification of the component, a regular event processing engine is composed of the following components: (1) initialization code that initializes state variables; (2) the set of variables that form the state of the component; (3) a number of input and output queues, which are connected to the upstream and downstream components, respectively; and, (4) a function that continuously takes events from the input queues, processes them, and possibly insert events into the output queues.

For a speculative event processing engine, the initialization function additionally initializes the STM and a set of speculative workers. The function that implements the processing is further divided into four components: (1) a function implementing the speculative worker threads (see discussion below); (2) a function that processes input events, potentially producing output events; (3) an `onCommit()` function that is called by the STM and inserts output events in the output queues; (4) a `postCommit()` function that frees task structures and does other non-critical activities such as logging or statistics maintenance.

The speculative worker is illustrated in Figure 2 and works as follows (see simplified pseudo-code in Algorithm 1). It first retrieves an event from the input queue with the help of a predictor (see below). The goal is to select the task with the lowest timestamp that has a good likelihood to be committed. If such a task is found, it is processed by the worker thread in the context of a new transaction. If the commit fails because of a conflict with a higher-priority transaction, the transaction will be automatically retried or inserted in the wait list. Before processing the next event, the worker checks the wait list for pending transactions to be committed or retried.

Algorithm 1 SpeculativeWorker()

```
1: loop
2:     task ← predictor.getEvent()
3:     if task ≠ null then
4:         initTransaction(task)
5:         scheduleTransaction(task)
6:     end if
7:     task ← getPendingTransaction()
8:     while task ≠ null do
9:         tryCommitTransaction(task)
10:        task ← getPendingTransaction()
11:    end while
12: end loop
```

4.5 Conflict Predictors

In a practical system, it is difficult to distinguish between being optimistic and overly optimistic. Being overly optimistic can lead to excessive collisions and, consequently, to performance values below sequential executions. To address this problem, we introduce conflict predictors. Conflict predictors (or for short, predictors), can give some information about the likelihood of an event to generate a conflict. When experimenting with our system, we have limited speculation to 10 logical timestamps in the future (w.r.t. last committed transaction). This works as a predictor that evaluate events that are more than 10 logical timestamps ahead of the last committed timestamp as overly speculative. For our workload and 4 worker threads this was enough to get good performance improvements. In the general case, however, using inadequate predictors could degrade performance to the level of sequential execution or worse just because of conflicts between a small subset of events.

Predictors can be classified according the type of prediction they are able to make, we divide them in two classes: predictors that classify events as suitable for speculation or that should be processed in order (i.e., they return a boolean value); and predictors that return the predicate that should be satisfied for the events to be executed without conflicts (e.g., event e_{t+j} should be executed only after event e_{t+i}, with $i < j$ and assuming event e_t is the last committed event). The system uses boolean predictors to classify which events will be processed optimistically and which will be processed only when their time is reached. Predicate-based predictors are used to link an event with the one(s) it depends on. This allows delaying the processing of an event until after the other events it depends upon.

An additional dimension that we use to classify predictors is the way they are built. We distinguish three classes: user-provided, generated by static analysis, and dynamic. User-provided event predictors benefit from user knowledge about the system. They can exploit semantic information that would not be available otherwise and do not need to be complex to achieve good performance. We performed several experiments using a trivial user-provided predictor that just limits the speculation to a number of steps in the future. The predictor is a simple function that takes as parameter the event to be evaluated and the current timestamp; it returns true if the timestamp of the event is not too far from the current timestamp. Such a simple predictor has shown good speedup in our experiments.

Static analysis on the processing rules can also be used to generate predictors. For example, static analysis might be used to generate a predictor that pessimistically predicts the possible conflicts.

Finally, dynamic predictors would operate based on statistics collected by the system in runtime. One such predictor could be based on estimations on the density of dependencies, and thus collision, to estimate a recommendation value for the optimistic processing of an event. Another example of dynamic predictor, could be one that increments or decrements the number of logical time ticks in the future that are acceptable for speculation based on the current rate of aborts. Such a predictor is useful with operators that build sketches of data, like a histogram. In the case of a simple histogram, for example, the probability of a conflict depends only on the ratio between the number of speculations and the number of buckets. With the dynamic predictor, the speculation horizon would converge to this value. In Section 5, we evaluate the effect of simple predictors on the performance of our speculative worker.

4.6 Algorithm Correctness

To understand why our algorithm works, consider first that the events in the incoming stream may be out-of-order but there should not be any gaps. A missing event would also prevent the non-parallel ESP application from processing the stream and hence we do not take into account the loss of events.

Now assume that a set of non-conflicting events arrives. As the events do not conflict, none of the associated transaction will fail (because the STM guarantees that non-conflicting transactions cannot abort one another). As there are no gaps, the transaction with the next commit timestamp will eventually be processed and commit, and let the next one commit and so on.

We have to address the problem of conflicting transactions. We should first step back and consider the properties of the underlying STM. By design, there cannot be deadlocks as, upon conflict, one of the conflicting transaction aborts and retries (even if it has already completed its execution and is just waiting for its turn to commit). Livelocks can happen if a set of transactions repeatedly aborts one another. It is the responsibility of the *contention manager* to ensure that such scenarios cannot happen. The contention manager is a module that take two conflicting transactions and decide which may continue and which should abort [9]. In our implementation, we rely on a "priority" contention manager that sets the priority of transactions according to the timestamp of the associated event. Upon conflict, the lower priority transaction is aborted. This contention manager has the useful property that it bounds the number of retries of transactions. If a transaction with the timestamp to be committed next arrives at a component after a number of other conflicting transactions with higher timestamps, it will be the highest priority transaction in the component and will eventually commit.

It follows that, given an underlying STM implementation that is deadlock- and livelock-free, the speculative execution engine processes events without encountering deadlocks or livelocks.

5. EVALUATION

5.1 Performance evaluation

We shall illustrate the operation of our ESP system and our speculative execution algorithm on the sample application network of Figure 1. This test approach consists of 5 components, with an event source (the input adapter), a stateless component (the filter), a stateful component (the processor), a correlator and an event sink (the output adapter). Events generated by the source have monotonically increasing logical timestamps with no gaps. Events are shuffled when traversing the parallelized filter component and are received out-of-order by the processor, which processes them speculatively and reorders them upon commit. In our evaluation, the output adapter also performs verifications on the order and values of the events processed by the previous modules, in particular it checks that the results generated by the processor component are correct. All tests were run on an 8-core Intel Xeon machine at 2 GHz running Linux 2.6.18-4 (64-bit).

We consider 6 scenarios for our performance analysis. The non-speculative scenarios use a sequential processing component. Thus, an event can only be processed after the event with the immediately preceding logical timestamp is committed. The speculative scenarios use the STM-equipped processor with 4 worker threads. Thus, up to four events can be processed at a time. In these experiments, with exception of the one with ordered event input, we used a predictor that allowed an event to be speculatively processed only if the difference between its timestamp and the last committed timestamp was less then 10. For the speculative execution with sorted input, we set this value to 500 to evaluate the maximum achieved parallelism.

The ordering of events depends on the number of filter worker threads. If there is a single filter thread, no order inversion occurs. The unordered version uses 4 filter threads. As the threads concurrently take events from the common input queue and insert them in the output queue, the order will be changed.

Algorithms 2 and 3 illustrate the two kinds of processing functions we are using in the different scenarios. These algorithms resemble sketching operators, like histograms. Algorithm 2 will cause an event e_{t+20} to always conflict with event e_t. Algorithm 3 may cause additional conflicts with events in the form e_t and e_{t+5} with e_{t+1} and e_{t+6}, respectively. A busy waiting loop is used to ensures that the processing has a predefined minimum duration.

Algorithm 2 PROCESSWITHCONFLICT1(*Event e*)

```
1: $startTime \leftarrow$ getTime()
2: $pos \leftarrow e.ts$ % STATE_SIZE
3: updateState($state[pos]$)
4: while getTime() $<$ $startTime +$ TASK_SIZE do
5:         {Ensure a minimum duration for the task.}
6: end while
```

Algorithm 3 PROCESSWITHCONFLICT2(*Event e*)

```
 1: $startTime \leftarrow$ getTime()
 2: $pos \leftarrow e.ts$ % STATE_SIZE
 3: updateState($state[pos]$)
 4: if (CONFLICTS $= 1 \wedge e.ts$ % $10 = 0) \vee ($CONFLICTS $=$
    $2 \wedge e.ts$ % $5 = 0)$ then
 5:     {Also update next state entry (create conflicts).}
 6:     updateState($state[(1 + pos)$ % STATE_SIZE$]$)
 7: end if
 8: while getTime() $<$ $startTime +$ TASK_SIZE do
 9:         {Ensure a minimum duration for the task.}
10: end while
```

The 6 scenarios were defined as follows.

Ord. Seq. Ordered event input, with sequential processor: in this case, there is only one filter component (and thus, no order inversion) and single thread processing component. The processing component task is illustrated in Algorithm 2. The STATE_SIZE value has no effect on the computation.

Unord. Seq. Unordered event input, with sequential processor: there are multiple filters and the processing component is still single threaded. The processing component task is illustrated in Algorithm 2. Again the STATE_SIZE value has no effect.

Ord. Spec. Ordered event input, with speculation: again, only one filter and the processing component try to optimistically parallelize the event to be processed.

The processing component task is illustrated in Algorithm 2. The `STATE_SIZE` value, which determines the frequency of conflicts, is set to 1000. In this case, there are no conflicts between events being processed but they are processed in parallel and committed in order.

Unord. Spec. 1 Unordered event input, with speculation and 0% conflicts: there are multiple filters and the processing component tries to optimistically parallelize events and may process them out-of-order. The processing component task is illustrated in Algorithm 2, with the `STATE_SIZE` value set to 20, which results in no conflicts in the horizon allowed by our default predictor.

Unord. Spec. 2 Unordered event input, with speculation and 10% conflicts: there are multiple filters, with optimistic parallelization and out-of-order processing. The processing component task is illustrated in Algorithm 2, with the `STATE_SIZE` value set to 20 and `CONFLICTS` set to 1.

Unord. Spec. 3 Unordered event input, with speculation and 20% conflicts: there are multiple filters, with optimistic parallelization and out-of-order processing. The processing component task is illustrated in Algorithm 3, with the `STATE_SIZE` value set to 20 and `CONFLICTS` set to 2.

The results of the experiments are illustrated in Figures 3, 4 and 5. For ordered executions, one can observe in Figure 3 that, even though the speculative version has no conflicts and a very far horizon for speculation, the throughput of the non-speculative version is initially much higher. This is due to the overhead of the STM and the synchronization costs on the ordered commit. However, this overhead becomes much less significant as the task size grows. Figure 4 also shows that the overhead of parallelization becomes negligible with longer tasks and the speed-up of the parallel processor improves almost linearly with the number of processor workers.

For unordered executions, the performance of the non-speculative version is not better than the speculative ones even with shorter task durations. A deciding factor in this case is that the non-speculative version must wait until the proper event arrives. This is not required by the speculative version. As the size of the tasks grows, the ordered and unordered non-speculative versions tend to perform similarly. This can be explained by the fact that if the processing of the tasks is long enough, by the end of the processing of the i^{th} event, the $(i+1)^{th}$ event will have already arrived and no waiting will be necessary.

The difference between the three unordered speculative experiments can be seen more clearly in Figure 4. With longer tasks the various synchronization costs and STM overheads tend to be negligible and the difference in the amount of useful work appears clearly: the performance of the 20%-conflict version is about 26% lower than the speculative version with non-conflicts; the performance of the 10%-conflict is about 18% lower than the no-conflict one. Finally, the 0% version (*Unord. Spec. 1*) tends to have a performance

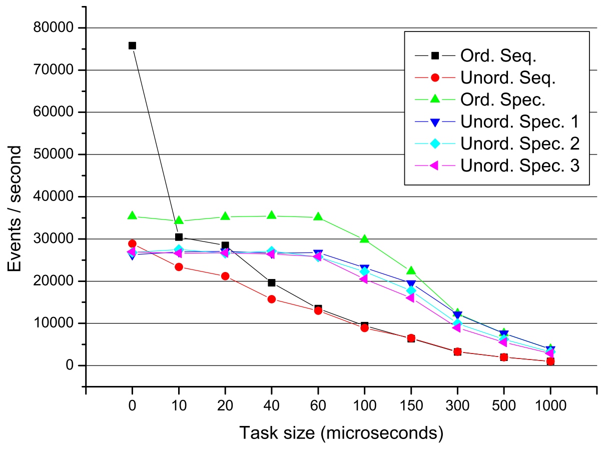

Figure 3: Average throughput of the various configurations.

similar to the ordered one with a farther speculation horizon (*Ord. Spec.*), because with large task sizes events are more ordered. In this case, the scheduling variations have less impact and thus there are rarely two events with conflicting timestamps (e.g., e_t and e_{t+21}) being processed by different threads at the same time.

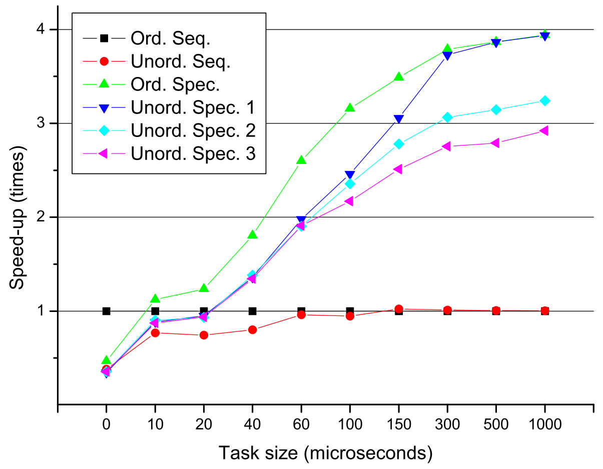

Figure 4: Speed up.

Another metric that can be very significant in event stream processing is latency. The end-to-end latency, from the input adapter to the output adapter, is depicted in Figure 5. These graphs show that the latency for the non-speculative version is far lower for ordered events. But this advantage is quickly disappears when the processing length grows and the parallelization becomes profitable. For the same reason, the right-hand side of the graph shows a considerable difference in latency in the non-speculative executions in comparison to the speculative ones.

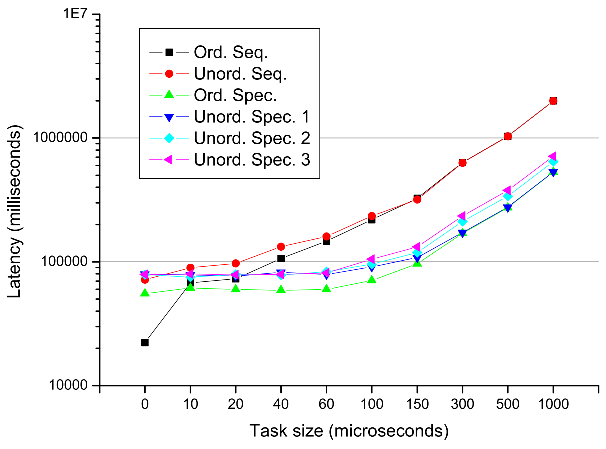

Figure 5: Average end to end latency with the various configurations.

5.2 Predictor analysis

To illustrate the impact of predictors, we analyzed the performance of 5 very simple predictors in two scenarios that could exhibit conflicts in optimistic executions. The first scenario is similar to the fourth scenario in Section 5.1, with the conflicts defined by Algorithm 2 for `STATE_SIZE` set to 20. The second scenario defines conflicts similarly to the sixth scenario in Section 5.1, with `STATE_SIZE` set to 20 and `CONFLICTS` set to 2 in Algorithm 3. The 5 predictors were defined as follows.

Predictor 0 returns *true* for every call. This predictor is useful if conflicts are rare, otherwise, as discussed previously, the system may enter a steady state in which all events are processed sequentially.

Predictor 1 returns *true* only if the event is within 20 logical clock ticks from the timestamp of the last committed event. It predicts perfectly the conflicts in our first scenario, but limits its return to a boolean value indicating if a conflict will occur or not in case the evaluated event is processed immediately.

Predictor 2 returns *true* if the event will conflict according to the collisions specified in Algorithm 3, for `STATE_SIZE = 20` and `CONFLICTS = 2`. This predictor is able to perfectly predict the conflicts for our second scenario, but as with Predictor 1, limits its evaluation to a boolean value.

Predictor 3 works as predictor 1, but instead of returning $false$ when the event is likely to conflict, it returns the event that should be committed before the evaluated event can be processed, so that no conflicts would ever occur. For example, the evaluation of event e_{23} would return 3, indicating that event e_{23} is likely to conflict with event e_3, and thus, should wait for it to be committed before it is processed. This predictor returns *true* if the event will not generate a collision, i.e., the events which could conflict were already committed.

Predictor 4 has the same knowledge as predictor 2 but, as predictor 3, returns a value indicating the event that should be committed before the event that is being evaluated can be processed without generating conflicts.

The speed-up achieved by the predictors in the first scenario is depicted in Figure 6. In the figure, we have a curve for each of 5 possible event processing task lengths. For tasks with short processing times we observe a decrease in performance when using predictors. This is indeed expected, because with such settings the best results are obtained with sequential processing as the relative overhead of speculation is significant. The best approach would be not to do any speculation at all and use a predictor that returns *true* only if the event is the next to be committed. With such a pessimistic predictor the results would be similar to predictor 0, but with much less CPU utilization.

When increasing the duration of the tasks, we observe noticeable improvements for the more sophisticated predictors. As expected, predictor 3 has better performance as it is capable of telling when a conflict is going to occur and what should be done to avoid it without sacrificing any non-conflicting parallelism. The usage of predictor 4 leads to sub-optimal performance as it predicts more collisions than are actually happening.

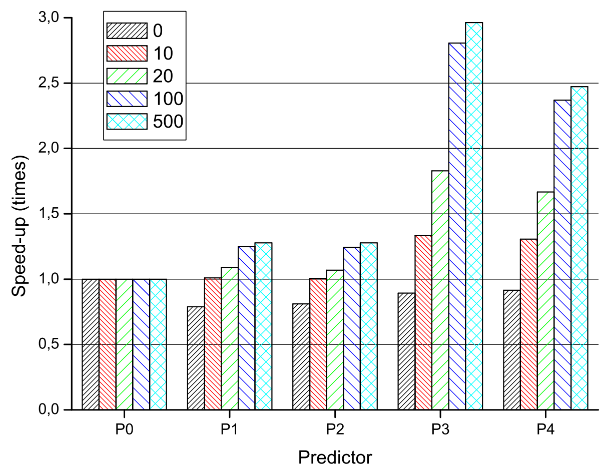

Figure 6: Speed up for different task sizes with Scenario 1.

Results for the second scenario are depicted in Figure 7. The same reasoning as for the first scenario applies for tasks with short processing times. For longer tasks, one can clearly see that predictor 4 produces higher speed-ups, as expected. To give an idea of the amount of useful work, with predictor 0 all events are aborted once (and then retried when their timestamp is reached); with predictor 3 there are 40% aborts; with the fourth predictor there are no aborts.

Although having the perfect predictors lead to the best results, one can observe in the graphs that sub-optimal predictors also enable impressive improvements. Predictor 3 in

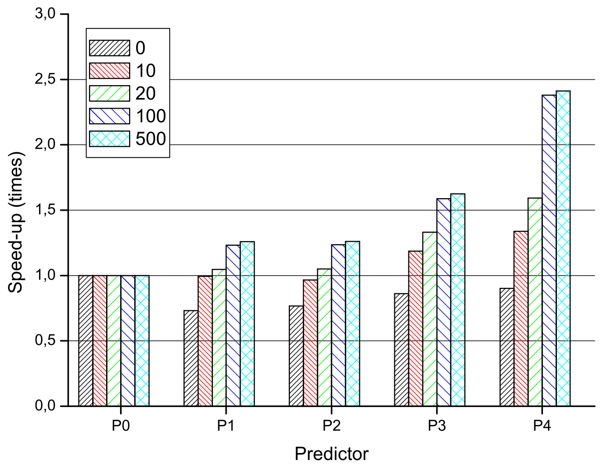

Figure 7: Speed up for different task sizes with Scenario 2.

the second scenario is a good example: even thought there are still 40% unforeseen conflicts, the overall speed up is still above 1.5. As a matter of fact, one cannot expect perfect predictors to be available for most practical cases. Some predictions may depend not only on event parameters, but also on the current state of the system, which could change between the time the event is evaluated until it is processed. With perfect predictors there would be no need for speculation support. Thus, exactly because of the inability to perfectly predict conflicts, support from a speculation infrastructure is required. The infrastructure we developed can dynamically monitor the processing and reevaluate events when conflicts occurs and, in spite of that, exploit as much parallism as the predictor is able to identify.

6. CONCLUSIONS

We have designed a speculative execution environment for event stream processing (ESP) components. Events that are received out of order and/or conflict with one another are optimistically processed in parallel. To ensure that the system remains in a consistent state despite parallelization, we use an underlying software transactional memory (STM) that was extended to account for the specificities of ESP. In particular, the STM can pre-assign timestamps to transactions to drive the commit order. Evaluation of our system confirms that good performance improvements can be achieved through speculation even if some computations may have to be disregarded and reexecuted.

We have also proposed using application specific conflict predictors to drive the system towards more efficient executions. These predictors can be specified by the user or generated automatically by static or dynamic analysis. We showed that even very simple predictors (e.g., limiting how far in the future the speculation should go) can improve the speed-up and that they do not need to be always correct to be useful.

7. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, CA, January 2005.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widow. Model and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'02)*, pages 1–16, Madison, USA, June 2002. ACM Press, New York, NY.

[3] R. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: a vision for event stream processing. In *Proceedings of the third biennial conference on Innovative data systems research (CIDR'07)*, Asilomar, USA, January 2007.

[4] D. Dice and N. Shavit. What really makes transactions faster? In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing.* Jun 2006.

[5] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[6] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[7] M. Koparanova and T. Risch. High-performance grid stream database manager for scientific data. In *Across Grids 2003*, pages 86–92. Springer-Verlag Berlin Heidelberg, 2004.

[8] M. Li, M. Liu, L. Ding, E. A. Rundensteiner, and M. Mani. Event stream processing with out-of-order data arrival. In *ICDCSW '07: Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, page 67, Washington, DC, USA, 2007. IEEE Computer Society.

[9] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul 2005.

[10] M. F. Spear, V. J. Marathe, W. N. S. III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *20th Intl. Symp. on Distributed Computing (DISC)*, 2006.

[11] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throughput stream programming in java. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 211–228. ACM Press, New York, NY, October 2007.

[12] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 2, Washington, DC,

USA, 1998. IEEE Computer Society.

[13] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *International Symposium on Code Generation and Optimization (CGO)*, 2007.

[14] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international Conference on Management of Data (SIGMOD'06)*, pages 407–418, Chicago, USA, June 2006. ACM Press, New York, NY.