2009 29th IEEE International Conference on Distributed Computing Systems

Minimizing Latency in Fault-Tolerant Distributed Stream Processing Systems

Andrey Brito, Christof Fetzer
Systems Engineering Group
Technische Universität Dresden
Dresden, Germany
Email: {andrey, christof}@se.inf.tu-dresden.de

Pascal Felber
Institut d'informatique
University of Neuchâtel
Neuchâtel, Switzerland
Email: pascal.felber@unine.ch

Abstract

Event stream processing (ESP) applications target the real-time processing of huge amounts of data. Events traverse a graph of stream processing operators where the information of interest is extracted. As these applications gain popularity, the requirements for scalability, availability, and dependability increase. In terms of dependability and availability, many applications require a precise recovery, i.e., a guarantee that the outputs during and after a recovery would be the same as if the failure that triggered recovery had never occurred. Existing solutions for precise recovery induce prohibitive latency costs, either by requiring continuous checkpoint or logging (in a passive replication approach) or perfect synchronization between replicas executing the same operations (in an active replication approach). We introduce a novel technique to guarantee precise recovery for ESP applications while minimizing the latency costs as compared to traditional approaches. The technique minimizes latencies via speculative execution in a distributed system. In terms of scalability, the key component of our approach is a modified software transactional memory that provides not only the speculation capabilities but also optimistic parallelization for costly operations.

1. Introduction

Most modern computing systems produce huge amounts of information, making the storage of all generated data unrealistic, if not impossible. Often, data has to be processed in real time to sort the important items that must be preserved from those that can be discarded. We refer to this task as Event Stream Processing (ESP). The main goal of ESP is to process and extract useful information from large amounts of real-time data. The resulting information, with greatly reduced size but much higher value, can then be stored and/or used to trigger actions [1].

ESP systems have attracted significant attention in recent years and have established a new field of research. Besides throughput and processing latency, availability is a major requirement for ESP systems. An ESP system typically runs on network of computers. According to studies on the availability and failure rates in large computer clusters, one can expect that several computers crash every day. Only recently have researchers started to design and implement fault-tolerant distributed event stream processing systems [2].

In order to implement fault tolerant ESP systems, transient information like internal states and decisions must be recoverable after a crash. To that end, one can apply a combination of logging and checkpointing: message logs can be used to replay messages and in this way, regenerate internal state. Of course, message logs might grow without bounds and hence, one typically combines logging with checkpointing of the internal state to be able to prune the logs. One trivial approach is to log messages at the source components that inject the event streams into a system, and use checkpointing within the stateful processing components. However, this approach will not work for ESP systems that are non-deterministic and have external interactions.

To understand how ESP systems work, we need to look at the types of operations involved and how these operations interact with each other. ESP applications are usually architected in the form of an acyclic graph of computing operators (see Figure 1). In such a graph, operators execute different types of computations, for example: filtering, transformation/conversion, enrichment (e.g., addition of offline information), aggregation (i.e., combine multiple events in a time/count window in order to generate a single event with higher-level information, e.g., averages), join (combination of events from multiple streams to produce a new event with the combined information), and union (merge multiple streams into a single one that contains all individual events).

Each of these operations has different requirements. For example, filters normally discard or forward events based only on their attributes. Thus, such operators do not depend on any local state (i.e., they are *stateless*) and are completely deterministic. Computing the average (an aggregate) of all the events that occur within a given time interval, however, depends on the current local state of the operator (i.e., the events seen so far), but not directly on the order of the events. A time window aggregation is therefore typically *stateful*, but deterministic. At the other end of the scope, join operators are both stateful and non-deterministic. They may depend both on a local state (i.e., which events have already been seen and are waiting to be matched) and on the order of the events (e.g., if one event from stream S_1 can be matched with more than one event from stream S_2, then the first event from S_2 that arrives will trigger the join).

For stateless deterministic components, fault tolerance can be provided by being able to replay the stream. This can be implemented by having upstream components save all produced events until the next downstream components confirm that the event has been processed and forwarded further. For stateful deterministic components, it is necessary to replay all the events to reconstruct the state, not only the ones that were being processed. In general, a good approach is to have stateful components execute periodic checkpoints of their local state so that upstream components only need to keep the events that have been processed after the last checkpoint. This approach reduces the size of the log and speeds up the recovery by not requiring the complete sequence of past events to be replayed. Finally, non-deterministic components (both stateless or stateful) require much more expensive mechanism, the simplest approach being to do a complete checkpoint before sending any event.

Because of the prohibitive cost of providing fault-tolerant non-deterministic operators, fault-tolerant stream processing systems often assume they can be replaced with weaker deterministic versions [2], [3]. In [2], the authors explicitly address only the deterministic operators. In [4], the authors present how *precise recovery*[1] can be achieve for non-deterministic component through different fault tolerance approaches (e.g., passive standby, active standby). However, all the solutions have in common that they are extremely costly in how they secure the non-deterministic decisions before sending events downstream (e.g., by only forwarding checkpointed tuples when using passive replicas, by having consensus among replicas for every non-deterministic decisions with active replication).

Despite the lack of efficient fault-tolerant support, non-deterministic operators are important parts of an ESP application. In general, non-determinism is caused by dependence on time (either execution or arrival times), dependence on arrival order of tuples on different input streams, or usage of non-determinism in processing (e.g., Monte-Carlo simulations, which are based on random numbers). As a consequence, a simple union operator that joins two streams and is followed by an order-sensitive component must log the order in which events were selected from the input streams. While aggregations are insensitive to ordering if the aggregation window is based on the event timestamps, aggregation windows based on system time depend on the arrival times of the events. Furthermore, for count-based windows, the order will always be important. Thus, many basic ESP applications may include non-deterministic operations and the respective non-deterministic decisions must consequently be logged if the system must provide precise recovery from faults.

1. Precise recovery means that the output results after recovering from a failure will be the same as if no failure had occurred.

It is important to also note that ESP systems must interact with external systems like databases and Web services. Hence, precise recovery is required to make sure that external systems do not see conflicting events. In this work, we present the novel approach used in StreamMine [5], [6], a distributed event stream processing framework in development at the Technische Universität Dresden, to limit the logging costs of non-deterministic fault-tolerant operators. StreamMine has the distinguishing feature of using speculation to parallelize stateful components [7], which allows us to send events speculatively *before* the log is committed to disk. The result is that all the logging operations required along the computation graph can be executed in parallel, reducing the processing latency by several orders of magnitude.

This paper is structured as follows. In the next section, we introduce a simple example application with non-deterministic operators that we use through the paper to illustrate our mechanisms, and we describe the system model, including details of the workings of StreamMine. In Section 3, we discuss the underlying speculation support. We evaluate the benefits of our approach in Section 4 and compare it with related work in Section 5. Finally, we conclude in Section 6.

2. Background and system model

We introduce in this section important background information about fault-tolerant even stream processing, including an example application used throughout this paper. We then discuss the system model considered in this work.

2.1. Distributed event stream processing

As previously mentioned, event stream processing applications are usually composed of a collection of components (computing operators) organized in an acyclic graph. In distributed settings, components are located on multiple machines and events are forwarded over the network between the components.

In the paper, we will consider a typical ESP example application (see Figure 1) divided into 5 processing steps. In the first step, events are generated by `Publisher` components (e.g., financial events in a stock exchange). These two event streams are merged and processed (e.g., events are analyzed and aggregated with previous events) by the stateful operator `Processor`. The third processing step takes place at the `Enrich` component, which adds some information to the event (e.g., data retrieved from a database). This step is costly but, being stateless and

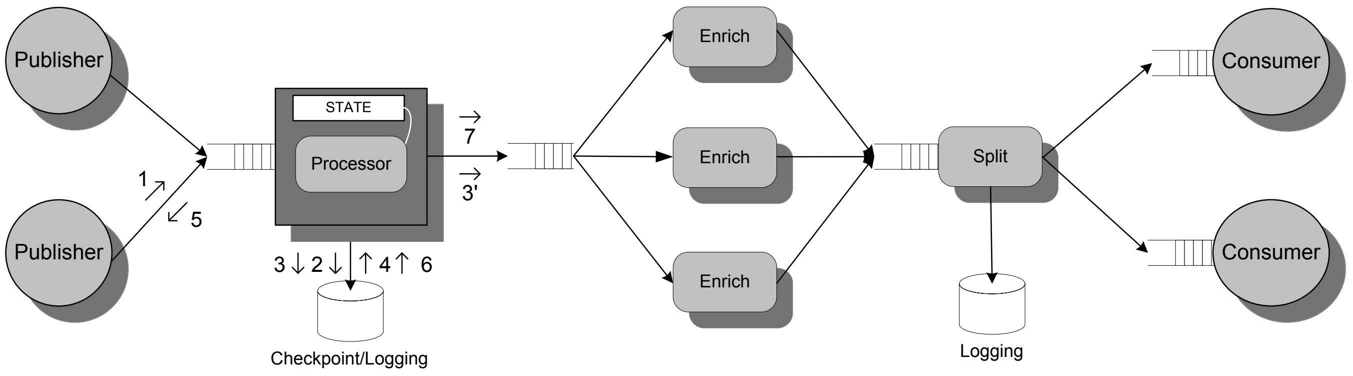


Figure 1. A simple prototypical ESP application graph.

unaffected by the order of the events, it can be parallelized by simply replicating the component. Next, the `Split` component balances the processing load by splitting the stream among several consumers. Finally, the events are consumed in the last step.

2.2. Fault-tolerant distributed ESP

Consider again our example of Figure 1. The `Processor` operator has a local state that was constructed based on the events received so far. In order to provide fault tolerance, we must be able to reconstruct this state in case of a failure. The reconstruction of the state is based on events received from the two sources and, because the order of the events received from each source matters, the input events (or at least their ids) are logged as they are received. In addition, it is also possible that the computations depend on some non-deterministic decisions, for example, random numbers or physical time during the processing of the event. These non-deterministic decisions are also logged. Furthermore, to avoid that the log grows indefinitely the `Processor` operator is also periodically checkpointed. When a failure occurs, the operator restores its latest checkpoint and replays the messages stored in the log, then it asks the upstream nodes to replay messages starting at the last logged messages from each source. Alternatively, if the operator logs only the ids of the messages, it requests a replay based on the messages that had been processed until the latest checkpoint and use the logged ids so that messages from different sources can be processed in the same order.

The protocol for the logging procedure above is as follows: (i) when the `Processor` receives an event from one of the sources (e.g., event 1 in Figure 1) it logs the event (e.g., through control message 2 in the figure); (ii) next, during the actual processing of the event, other non-deterministic decisions may be taken and they also need to be logged (e.g., through message 3); (iii) eventually, the logged input events (step i) will get stable in the storage (as notified by message 4) and if the complete events are logged, the operator may notify (through control message 5) that such event will never be requested to be replayed again and the upstream node may remove that event from its output buffer; (iv) finally, the non-deterministic decisions used during the processing will also get stable (as notified by message 6) and the results of the processing can be sent downstream (illustrated by event 7). Note that normally the node must wait until the log is stable on disk, otherwise events could be processed in different order during replay (or use different non-deterministic decisions) and the operator may reach a state different from the one expressed by the previous outputs. Thus, although the new state would be also valid, the inconsistency could have critical consequences.

Still during recovery process, the `Processor` component may output duplicate events that had been already emitted before the failure. Nevertheless, because any ordering or additional non-determinism was logged before outputs were sent, the duplicates will have the exact same information as their first instances (including ids) and can be silently dropped.

Now consider that the `Split` operator fails. If this operator picks events from the inputs based on arrival order and randomly selects one downstream node in order to balance the load, the input order must be logged together with the random decision in order to ensure a replayable output. In this case, however, the operator is stateless and no state must be rebuilt. The log keeps then only the events that still may need to be replayed. Therefore, no checkpoint is necessary and logs are truncated based on the acknowledgments from downstream nodes (i.e., similar to message 5 in the case of the `Processor` operator). This example highlights the challenges of adding fault tolerance to distributed ESP applications.

2.3. System model

We assume an ESP application as an acyclic graph of computing operators. Each operator can be configured as being speculative or not. When executing, all events sent by non-speculative operators are final. A final event is an event that will not change anymore. In case of an failure, if a final event reappears it will have the exact same content as the original one and can be silently dropped. On the other hand, a speculative operator runs under control of a software transactional memory (STM), which will be detailed later. During their execution, speculative operators may issue events, which may be final or speculative. A speculative event is an event that is not guaranteed to be final, i.e., in case of a failure, it may not reflect the actual state of the operator that generated it, and it may be later revoked.

The specification of an operator is independent of its configuration and is done by providing an optional initialization method (which is called during the system start up, to allocate and initialize resources), a required processing method (which will be called for each event that arrives in one of the input streams), and an optional termination method (called once before system shutdown). In case of a fault-tolerant application, the initialization method should be able to handle the recovery process, with the help of methods provided by the ESP system. The operator functions are plain C source code enriched with some library methods provided by the system. The insertion of the instrumentation instructions in the speculative components, which is required for speculation support, is done using TANGER [8] at compile time.

One major limitation regards the usage of "non-speculative external actions", i.e., operations such as I/O that cannot be performed speculatively. Due to the very nature of the speculation, which may require rollbacks and re-executions, non-speculative external actions should not be performed in speculative operators, or explicit undo functions must be provided.

The specification of a component also includes its network address for listening to connections, as well as the addresses of its downstream neighbors. Except when explicitly mentioned, operators are operating system processes that communicate with their neighbors via TCP connections. These processes are independent and can be placed on the same host or across distributed machines.

2.4. The logging algorithm and its costs

The logging algorithm used in StreamMine works as follows. When executing an operation in a component (using one or more input events), the processing function issues an asynchronous storage request for its non-deterministic decisions. Later, when the processing is finished and the resulting events are ready to be sent, they are blocked until the non-deterministic decisions have been committed to disk. The storage requests are handled by a set of threads that can write to different data stores in parallel in order to maximize throughput. Thus, if the user configures N storage points (e.g., local disks, NFS mounted disks), there will be one thread per point plus 1 extra thread that collects the requests while the others are busy, thus, using $N + 1$ threads total.

When a thread finishes writing a set of decisions, it releases its associated storage point and hands it to the thread that was collecting requests, before taking over the role of waiting for new requests. Eventually, when all the storage requests associated with some output events are written to the stable storage, these events may be forwarded as final. In case of speculative components, the events are output as soon as they are generated, but they are tagged as speculative.

To get an idea of the logging costs, we have experimented with a simple system composed of just two components. Each component consumes one event at a time and outputs one event per operation. For each event processed, the component needs to log a 64-bit value as decision. In Figure 2, we show the impact of the logging in a non-speculative system for different configurations. We use three logging configurations in which the system is equipped with one, two, and three local hard drives (referred to as *1 disk*, *2 disks*, and *3 disks*, respectively). In addition, we have simulated two logging configurations, where we assume a very fast disk that is able to store a set of event logs in 10 ms and 5 ms (referred to as *Sim 10* and *Sim 5*). These last two configurations serve the purpose of simulating multiple disks in the machine used for our experiments: a SUN T1000 machine with eight cores and four hardware threads per core (and thus, parallel processing capabilities), but with a single disk that would represent a bottleneck.

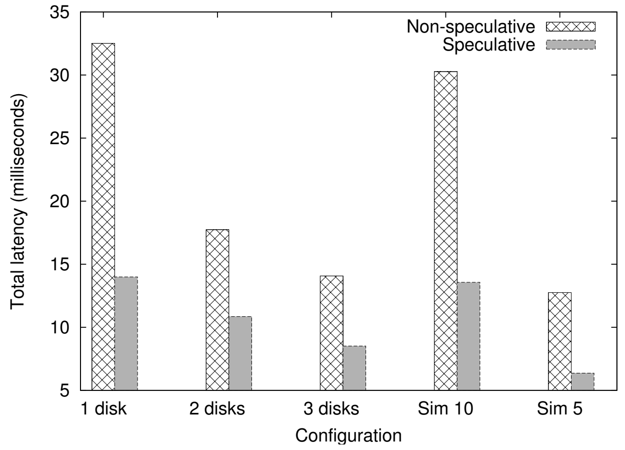


Figure 2. End-to-end latency for a network with two component for different logging configurations.

In this experiment, the two components are in two threads of the same process and share the same logging queues and storage. One can observe that the costs of logging in a network with only two components is already very high. The speculative approach provides considerable gains because the logs of components can be written in parallel. A more detailed evaluation will be presented in Section 4.

3. STM-based distributed speculation

It this section, we give an overview of how software transactional memory (STM) is used to support speculation in stream processing operators. STM is an optimistic concurrency control mechanism that speculatively executes lightweight transactions in multiple threads and, upon conflict, automatically aborts and restarts the offending transaction(s). For details on the STM and on the usage of speculation to optimistically parallelize stream operators, we refer the reader to [7], [9]. Without speculation, operators have one or more working threads that collect events from the input queue and call the operator's `process()` method, passing the input event as parameter and receiving an array of output events as result. The processing method carries out some computation and can generate output events. With speculation, a transaction is created using as arguments the operator's `process()` method and the input events. The modified software transactional memory will execute the function under transactional support, which means that all memory accesses (e.g., read, write, memory allocation, memory release) that affect the local state of the component will be intercepted by the STM. Memory accesses for addresses with local scope do not need to be intercepted.

During the processing of an event, all writes are buffered and no modification is performed to the actual data until the transaction commits. Each time a memory location is written, the STM looks in a shared metadata region, which we call the "lock array" (see [9] for details), whether a concurrent transaction is also updating the same location. If that is not the case, the transaction registers itself as a writer in the lock array. When reading a memory location, the transaction will also check whether a concurrent update is being performed. If no conflict is detected until commit time and the transaction is final, it applies all buffered updates to the shared memory and release all its locks. If the transaction is speculative, it waits in pre-commit stage and does not unregister itself from the lock array.

By default, a transaction cannot commit if at least one of the events it used is still speculative. When a transaction generates events, the STM checks if the transaction is speculative. If that is the case, any event generated by the transaction is also marked as speculative. Once the transaction receives the authorization to commit (by getting an appropriate final message, and/or not depending on any other still open transaction, as we will discuss later), it can write its buffered updates. Thereafter, the events generated during the processing are forwarded as final.

Now consider that a transaction is *still open*, i.e., it finished processing and is just waiting for the commit authorization, when a new speculative event arrives. If the transaction that processes the new event does not conflict with the first one, the reads and writes take place as detailed above and there is no interference between the two transactions. Otherwise, if the second transaction wants to write to a memory location already written by a previous transaction, the conflict will be detected when looking into the lock array. Conflict resolution typically involves aborting one of the offending transactions, possibly after waiting some time in case the conflict would be transient.

In the context of this work, transactions remain open because they are waiting for the confirmation that any preceding logging operation in other components has successfully committed. Taking this into account, we allow a later transaction to read or overwrite buffered values of a preceding transaction that has completed its execution but is still open. The success of the second transaction is then conditional to that of the first one. Even if the later transaction receives a commit authorization, it must wait for the preceding transaction to commit. In addition, if the first transaction aborts, the second one must also abort.

If the operator is configured to process multiple events at the same time, i.e., optimistic parallelization is enabled, upon conflict we abort and restart the transaction associated with the event that arrived last.

3.1. Example

In order to illustrate how speculation works and how the fine grained control enabled by the STM can help, consider an expanded version of our example from Figure 1. In this expanded version, the publishers are in fact graphs of operators that can also generate speculative events. Assume also that the operator `Processor` executes some kind of classification, e.g., assigns events to one of several classes and outputs how many elements are currently on that class.

Consider now the case that publisher P_1 generates a speculative version of an event E_1, namely E_1' (e.g., some information used during the computation of E_1 needs to be logged and is not yet stable). Then, publisher P_2 generates a final event E_2 (i.e., no non-determinism, and thus, no logging, was present during the computation of this event). These two events cause transactions to be created in the operator. First, assume no collisions occur. During the processing, event E_1' will be assigned a class and an output will be generated, but also flagged as speculative. After that, event E_2 is processed and is assigned a different class and an output is generated with the current counter for the class. Because, event E_2 is not speculative and did not use any state that had been speculatively modified (i.e., E_1' modified another class), the output will not be speculative, even if it is out of order.

Alternatively, consider that there is a collision, i.e., events E'_1 and E_2 belong to the same class. In this case, the output caused by E_2 will be also marked as speculative because it is not guaranteed that upon a failure the same E'_1 will be generated and, thus, the same counter output would be generated for E_2. Later, if a failure occurs in the subgraph P_1 another (maybe again speculative) event E''_1 may be generated. Upon reception of E''_1, the transaction for E'_1 will be updated and if the differences are relevant, i.e., the old values were used during the computation, the transaction will rollback and re-execute. If during the re-execution, the classification of E_1 changes, the output for E_2 must also be regenerated. Therefore, the transaction for E_2 is also be re-executed.

The example above illustrates the cases where creating a simple dependency relation between an output event and a pending log operation does not suffice: (i) if a speculative input of a component taints all component's outputs until the speculation is confirmed, more events would be marked as speculative, tainting even more downstream components, possibly delaying application's outputs even when they are in truth not affected; (ii) in an application with tens (or even hundreds) of nodes, the failures will be frequent and rollbacks should be done only when really needed and as efficiently as possible.

4. Evaluation

As previously explained, the greatest potential of speculation in fault-tolerant ESP systems is the ability to parallelize the writing of the logs to stable storage. We have seen already in Figure 2 that in a network with two operators, the logging costs are approximately halved in the execution with speculation activated. To evaluate the impact of speculation in a more complex processing graph such as the one in Figure 1, we analyze the impact of the checkpoint in a graph with 2 to 7 operators that need to log their decisions. This experiment was executed in a single SUN T1000 machine, with each operator running as a separate process connected to its neighbors through a TCP connection. The machine has enough hardware threads for not having contention on the processor and, to avoid bottlenecks on disk accesses, we used the simulated checkpoint as discussed in Section 2.

As expected, the latency for the speculative executions is nearly constant regardless of the number of operators (see Figure 3). In a real distributed scenario, for each remote TCP connection there would be a latency increase from a few hundreds of microseconds in LAN settings to up to tens of milliseconds in WAN settings. Nevertheless, these added latencies would still be shorter to the delays produced by the logging and, thus, the graph would have a similar shape.

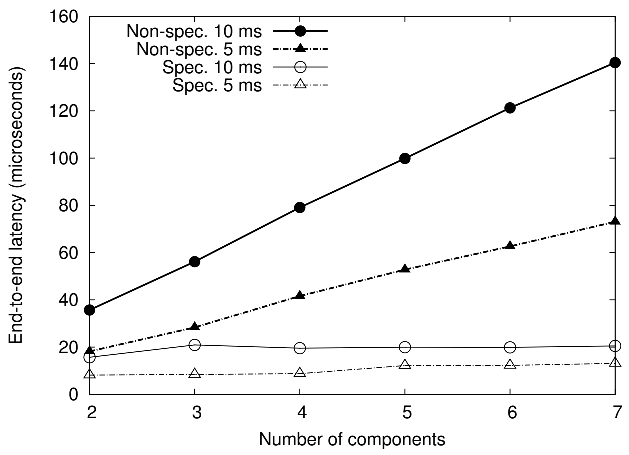


Figure 3. End-to-end latency for a network with different number of operators and logging times.

In the experiments above, the processing costs inside the operators were very low, i.e., in the range of microseconds. The optimistic parallelization of operators by StreamMine becomes even more interesting when processing costs are high. This is the case, for example, when data requires sophisticated analysis such as deriving models through computer learning algorithms, or when costly stream analysis operators (e.g., top-k [10]) are used.

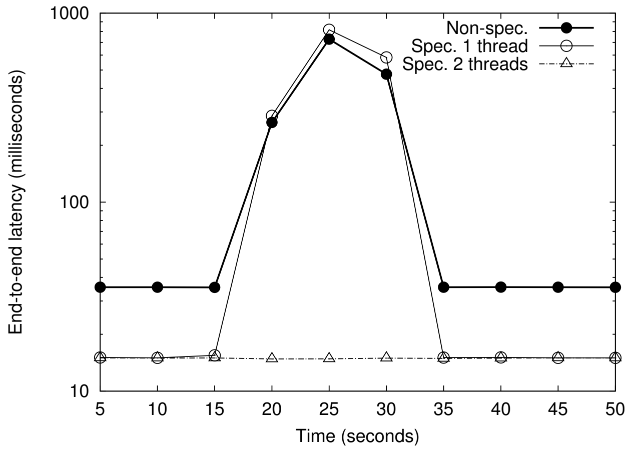


Figure 4. Evolution of the end-to-end delay for an execution in which the event inter-arrival time drops below the sequential processing time during interval $[15, 25]$ from the start of the execution.

We first consider a scenario with a short burst in the event rate, when the inter-arrival time of the events becomes slightly lower than the processing costs: for a 10 seconds interval (from time 15 to time 25) the processing costs of the events is 10% higher than the inter-arrival time. In such a case, the latency will grow sharply, and the system needs a long time to recover as seen in Figure 4. Nevertheless, if we enable optimistic parallelization (by simply increasing the maximum number of threads an operator may use), the processing latency remains unaffected.

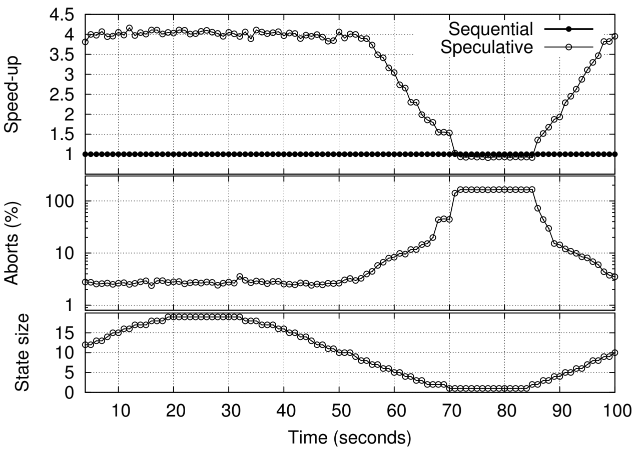


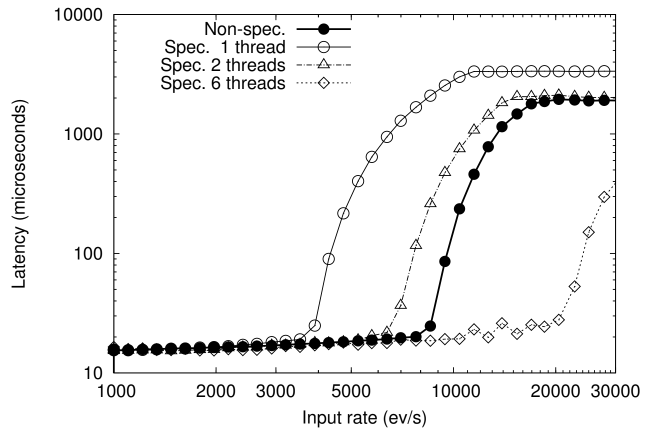
Figure 5. Local speed-up and abort rate in a parallelized operator for varying amounts of available parallelism in the workload.

It is important to notice that the benefits shown in Figure 4 can only be enjoyed if the operator semantics and/or the load allow such parallelism. This is common, for example, in stream analysis operators that uses *sketches* to represent the stream and extract information (e.g., to answer continuous queries). A sketch (e.g., count sketch [10]) is a limited representation of the stream. When an event is received, only parts of the sketch need to be updated or read. In addition, because the part accessed may depend on runtime information, opportunities for parallelism cannot be explored by automatic static analyzers.

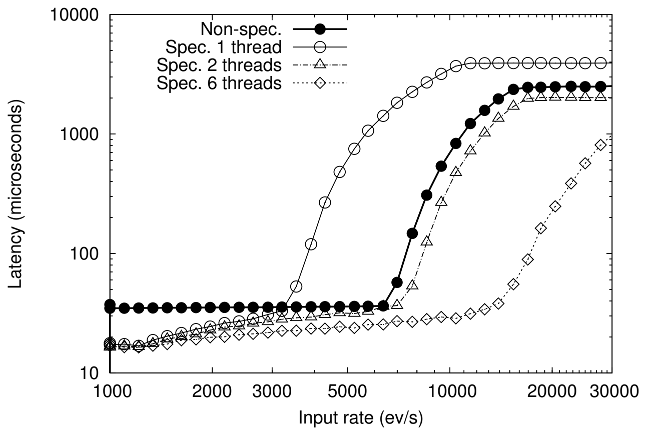
On the other hand, if the semantics of the operator or the current workload do not allow for parallelism, even with the optimistic parallelization enabled the system still offers a performance equivalent to the non-parallelized execution, as depicted in Figure 5. In this experiment, we vary the number of fields in the component state, as depicted in the lower part of the figure: if the state contains only one field (the minimum values in this sub-figure), any two concurrent operator executions will interfere because they modify the same field and, thus, there is no available parallelism; in contrast, if there are several fields, the probability that some events can execute in parallel without interferences quickly raises. The middle part of the figure shows the amount of computations that had to be rolled back because of interferences or missed data dependencies. The upper part shows the speed-up when up to 8 threads could be used by the speculative component. As can be observed by the increase in the abort rate (middle figure), a high number of speculative executions are aborted when there is no parallelism. If resources should be used with parsimony, the system can be configured to trade promptness to explore parallelism (i.e., how fast parallelism can be found and exploited) against the amount of resources wasted when there is none.

The speculation benefits for handling logging latencies and exploiting available parallelism with sketch operators can be seen by looking at how the system responds to different event input rates (Figures 6 and 7). In this scenario, we consider two operators, a union and a sketch (in this case the count sketch from [10]), and evaluate how this application responds to different intensities of workloads. The union operator is computationally cheap, it simply merge the two streams into a single one and does not need to be parallelized. The count sketch, however, requires a considerable amount of computation and is, thus, parallelized using the speculative approach. The end-to-end latency in this application when only the union does logging is shown in the left-hand side of Figure 6. The case in which both components do logging is shown at the right-hand side. The latency added by having the sketch operator do logging is clear before the system is overloaded with messages (around 2500 events per second). Furthermore, the throughput response of the system is shown in Figure 7. This figure also highlights the overhead of the speculation: with a single thread, the speculative operator is almost half as fast as the non-speculative. Nevertheless, if the sketch operator is allowed to use more threads this overhead is compensated by the parallelization. For the scenario where both components do logging, the throughput is not affected by the additional logging.

In the last experiment, illustrated in Figure 8, we evaluated the costs of a speculative execution and a rollback and consequent re-execution (fixed transaction creation and commit costs are not considered here). Two sets of three curves are depicted. The upper set of curves correspond to an operation that does expensive computations (around 800 μs, referred to as T1), while the lower set corresponds to a much cheaper operation (around 1 μs). In both cases, an increase in the number of shared-memory accesses (horizontal axis) reduces the ratio of computation to memory accesses. Note that we do not limit that number of non-shared memory access (e.g., local variables), only the memory accesses that are potentially shared. As shown in the figure, there is a constant overhead per memory access. Nevertheless, the rollback and re-execution of the task takes approximately the same time as the first execution, which confirms our argument that the rollback is fast. In fact, in this experiment we consider that an execution reads random positions from a large local state and therefore, that a re-execution will not benefit from caching effects of the first execution (which would be the case if the new event caused only small changes in the computation).



(a) Only Union does logging



(b) Both operators do logging

Figure 6. Latency response for different input rates with speculation for parallelism and reduced logging costs in an application with 2 operators.

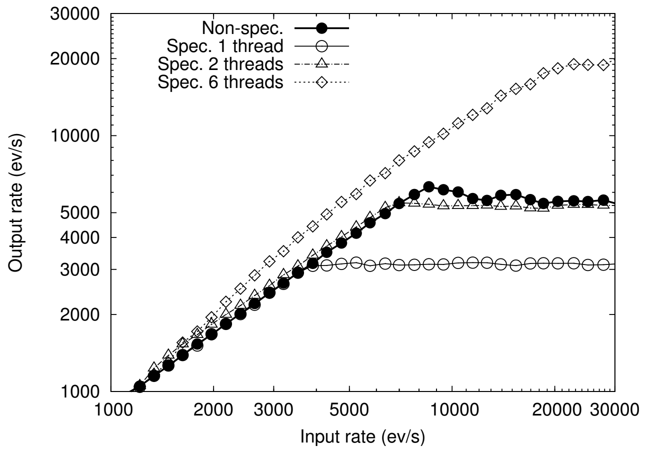


Figure 7. Throughput response for different input rates with speculation for parallelism and reduced logging costs in an application with 2 operators.

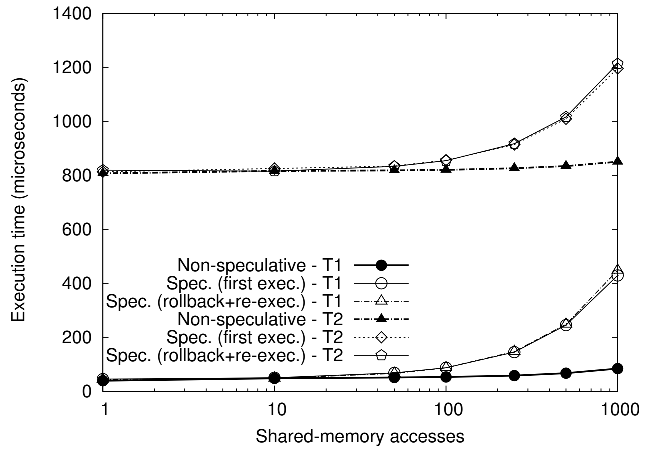


Figure 8. Comparison among the execution times of non-speculative, speculative and rollback followed by re-execution.

Finally, we want to point out one more scenario. In some cases, an ESP system might be permitted to output speculative results. For example, let us assume that the ESP application analyzes huge amounts of events from a large number of sensors and writes the analysis result to an external resource like a file or database. Furthermore, another application periodically looks at this resource to take some action. If we allow that the records written to the file be marked as "speculative" and we let the reading application filter out speculative records that are not finalized (e.g., with the help of a library), the total processing latency will be independent of the logging latency. Using this approach, processing latencies can be as low as a few tens of microseconds with all components running on a single multi-core machine or a few hundreds of microseconds when running in a cluster.

5. Related work

The problem of highly available distributed event stream processing application is a topic that has only recently been addressed. The most closely related work is from Flux [2] and Borealis [4]. Flux applies the well-known process-pair approach and focuses on deterministic operators. In Borealis, the authors classify operator graphs in four classes (repeatable, convergent-capable, deterministic, and arbitrary) and present four types of recovery protocols (amnesia, passive standby, upstream backup, and active standby). They advocate that many applications do not require precise recovery and they classify how precise each of the recovery protocols is for the different operator graph types. Finally, they discuss how three of the protocols could be adapted to provide precise recovery for arbitrary (which is the only class that includes non-determinism) graphs. In the passive standby approach, based on checkpoints, the operator can only forward checkpointed tuples downstream. The active standby approach, based like Flux on the process-pair model, requires that primaries send the non-deterministic decisions to the secondaries and then wait for the acknowledgment before sending events downstream. In this paper, we address the problems of the precise recovery of arbitrary operator graphs and we provide a constant latency cost by using speculation in the execution of the operators.

In general, rollback-recovery protocols using checkpoints and logs are well understood [11]. Closely related to this work is optimistic logging, which also assumes that nodes do not need to wait for logging to become stable. Nevertheless, optimistic logging was never popular in practice for several reasons: (1) complex garbage collection protocols to maintain the required checkpoints (it may require multiple checkpoints to be kept); (2) multi-host coordination to commit output to the external world due to the asynchronous logging of the determinants; and, (3) complex rollback protocols that can need multi-host coordination or may cause multiple rollbacks to be triggered.

Optimism is also the base of Virtual time [12]. There, the author defines the meaning of speculation in a distributed system, where instead of waiting for some message, nodes can simply continue computing in the hope that later messages will not have earlier timestamps. In order to make it possible, the nodes must keep checkpoints of the states and log the messages they send or receive. In addition, they can only discard these messages when a global consensus shows that some old data is not necessary anymore. StreamMine differs from Virtual Time (as well as from previous work on optimistic logging) mainly in two ways. First, the usage of the software transactional memory provides a way to not only keep fine-grained checkpoints, but also to rollback only when strictly necessary (e.g., even if a message changes, the rollback and re-execution are not necessary when the change does not affect a field that was read during the computation). Second, because ESP applications are acyclic graphs, the global consensus on a commit time (as with the thresholds for garbage collection or the maximum available recovery time, in the case of optimistic logging) is not necessary, it is unilaterally determined by the upstream node that sent the speculative messages. Other systems use the concept of speculation to achieve parallelization in different contexts. Pedone *et al.* [13] use optimistic delivery for atomic broadcast in distributed databases. The work from Ferretti [14] uses speculation to hide latency in game servers but requires application provided information to detect which events are important for consistency. The system presented in [15] enables an application relying on synchronous disk writes to proceed before the writes are stable. In this work, the overlap between writing and processing is enabled by having the operating system to track the dependencies and hold any externalization of data that is causally dependent on the not yet stable write.

The parallelization of costly operations in an ESP is addressed by Flux [16] using a partition-compute-combine approach. Ivanova and Risch [17] also rely on this approach. The main limitation is that it only works with operators that can be processed according to this divide and conquer pattern. Further, even in such cases, the merging phase can be complex and limit the total speed up according Amgdahl's law. Streamflex [18] assumes stateless operators that can be parallelized by simple replication. Borealis handles operator scalability and latency management by applying coordinated load shedding [19].

Software transactional memory [20] was introduced as a synchronization mechanism that is easier to use and potentially more scalable than locks. Instead of having the programmer worry about acquiring and releasing locks, blocks of code that must execute atomically are simply encapsulated within specific programming constructs (transactions). STM uses speculation to execute transactions in parallel and detects conflicts at runtime, possibly rolling back and restarting the execution of some transaction(s). We extend the STM model by allowing transactions to perform speculative operations that introduce dependencies with later transactions. Then, not only executing a transaction in a way that appears to be atomic is necessary, but also that the order that transactions commit also obey the application timestamps of the event. In addition, transactions may be paused (e.g., because it is still too early to commit them) and later, revalidated and committed by another thread.

6. Conclusions

In this work, we have addressed the problem of latencies in a fault-tolerant event stream processing (ESP) system, caused by the logging of messages and non-deterministic decisions. Non-deterministic decisions need to be logged to facilitate the replay of messages in the case of a failure. This problem is of great importance as most event stream systems need to perform non-deterministic operations, e.g., picking events from one of two input streams or performing actions based on random data. We have shown that with the help of speculation, which is implemented as a layer below the regular event processing operators, one can minimize the impact of logging. The logging operations throughout the stream processing graph can be performed in parallel without compromising the precision of recovery or increasing their costs. If the ESP system is permitted to output speculative messages, our approach permits to hide all latencies introduced by logging.

We have also shown the impact on the latency of parallelizing computationally expensive operators in a processing graph via speculative execution. We have illustrated this aspect with the help of a sketch-based operator. Sketch-based operators are suitable for optimistic parallelization, but not necessarily for more traditional compiler-based techniques. In summary, speculation has proved to be a valuable technique for ESP systems and is a key feature in StreamMine, an ESP processing framework currently in development at TU Dresden.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 216181 (STREAM Project).

References

[1] 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)*. Madison, USA: ACM Press, New York, NY, June 2002, pp. 1–16.

[2] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2004, pp. 827–838.

[3] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," *ACM Trans. Database Syst.*, vol. 33, no. 1, pp. 1–44, 2008.

[4] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 779–790.

[5] "Streammine," http://streammine.inf.tu-dresden.de, November 2008.

[6] C. Fetzer, A. Brito, R. Fach, and Z. Jerzak, "Streammine," in *Handbook of Research on Advanced Distributed Event-Based Systems, Publish/Subscribe and Message Filtering Technologies*, A. Hinze and A. Buchmann, Eds. Hershey, PA, US: IGI Global, 2009.

[7] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, "Speculative out-of-order event processing with software transaction memory," in *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*. New York, NY, USA: ACM, 2008, pp. 265–275.

[8] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, and H. Sturzrehm, "Transactifying applications using an open compiler framework," in *TRANSACT*, August 2007.

[9] 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.

[10] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, 2004.

[11] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[12] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[13] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, no. 4, pp. 1018–1032, July-Aug. 2003.

[14] S. Ferretti, "A synchronization protocol for supporting peer-to-peer multiplayer online games in overlay networks," in *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*. New York, NY, USA: ACM, 2008, pp. 83–94.

[15] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the sync," *ACM Trans. Comput. Syst.*, vol. 26, no. 3, pp. 1–26, 2008.

[16] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *Proceeding of the 19th Internationsal Conference on Data Engineering*, 2003, pp. 25–36.

[17] M. Ivanova and T. Risch, "Customizable parallel execution of scientific stream queries," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 157–168.

[18] 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*. ACM Press, New York, NY, October 2007, pp. 211–228.

[19] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: efficient load shedding techniques for distributed stream processing," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 159–170.

[20] N. Shavit and D. Touitou, "Software transactional memory," in *Symposium on Principles of Distributed Computing*, 1995, pp. 204–213. [Online]. Available: http://citeseer.ist.psu.edu/shavit95software.html