Handling Crash and Software Faults Efficiently in Distributed Event Stream Processing

Andrey Brito, Stefan Weigert, Martin Süßkraut, Christof Fetzer
Systems Engineering Group, TU Dresden
Dresden, Germany
Email: {andrey, stefan, suesskraut, christof}@se.inf.tu-dresden.de

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

Abstract—Active replication is a common approach to handle failures in distributed systems, including Event Stream Processing (ESP) systems. However, one weakness of conventional active replication is that replicas, being equal and in the same state, are susceptible to common-mode crashes due to software bugs. We propose a new approach to active replication that assumes a failure model stronger than fail-stop but weaker than models permitting arbitrary failures. We combine transactional memory and extended runtime checking to achieve: (i) low processing latency in failure-free runs by allowing downstream nodes to use speculative results and, thus, to circumvent the overhead added by the extended runtime checks; (ii) reduce the MTTR by enabling localized rollbacks (with word granularity) in several cases. We show that major limitations of n-variant active replication (e.g., multi-threading support, complex and slow recovery) can be overcome and tolerance to software bugs is orthogonal to Byzantine fault tolerance.

Keywords—fault tolerance; event stream processing; speculation; software transactional memory; software bugs.

I. INTRODUCTION

Event Stream Processing (ESP) systems have attracted much attention in last years and have established a new independent research area. An ESP application is composed of an acyclic graph of operators. Typical operators in such a graph are filtering, aggregation, enrichment (e.g., adding information from external sources), and transformation. A common feature among most ESP applications is the ability to receive a huge amount of low-level data and output a much reduced number of higher-level events.

A popular application of ESP is algorithmic trading. In this application, the system monitors stock-market quotes and triggers buy or sell actions when certain conditions are met. This example highlights another common feature in ESP applications: if data cannot be processed quickly, information can lose its value.

As critical ESP applications get deployed, more sophisticated requirements are placed. One such non-trivial requirement is fault tolerance. Operators, however, can be quite complex, requiring events to be processed based on an accumulated state. If failures are to be tolerated, it is essential that the accumulated state in operators is not lost upon failures. Securing this state has severe performance costs. With active replication [1], replicas can process events independently and thereby tolerate failures. However, for the state to be consistent among replicas, all of them must process the same inputs in the same order. Reliable ordered delivery (commonly known as atomic broadcast [2]) requires coordination and, therefore, also inserts processing latency. Nevertheless, at the cost of increased resource usage, active replication provides fast recovery. For this reason, active replication is preferred for time-critical applications.

Furthermore, the robustness of replication is based on the assumption that replicas fail independently. However, if failures are triggered by software bugs, all the replicas may be simultaneously compromised by the same fault. Even if the replicas do not crash, bugs may have non-deterministic effects on the state, causing the replicas to become inconsistent.

A classical solution to prevent software bugs from compromising all replicas of an operator is to have independent versions of the same operator and then vote on the outputs. This approach, named n-version programming [3], has two disadvantages: (i) it is costly, because it requires independent teams (or, at least, different approaches) to implement a single operator, and (ii) because operators use different algorithms, it is not possible to use the state of a correct operator to restore a failed one. More recently, the authors in [4], [5] proposed the usage of automated tools to generate variants of a software component. Nevertheless, recovery is still an open problem. At the end, operators are required to keep logs and checkpoints in order to recover themselves, increasing the overhead in failure-free runs (because logging and checkpointing must be synchronous) and delaying recovery. However, fast recovery, i.e., a short Mean Time To Recovery (MTTR), is essential for high availability [6].

In our work, we also use the automated generation of variant replicas. We aim to generate replicas that are fast (e.g., the original one) and replicas that are robust (e.g., that include some form of strengthening against a certain class of bugs, but often add considerable overhead [7], [8]). However, in contrast to [5], rolling the state back to recover from a problematic input is a simpler procedure. On the one hand, when only the state of the component is
compromised by the bug, our speculation mechanism based on Transactional Memory [9] makes recovery a localized and efficient procedure. On the other hand, when we cannot roll back the operator locally, we can restart it and use the state of a correct variant replica to recover the operator. This is possible because, although the replica is a variant, it is based on the same algorithm and, thus, keeps an equivalent state. Finally, speculation support enables downstream replicas, which depend on the voting of the upstream results, to use inputs that are available early and, later, roll back or commit the computation based on the result of the voting.

In a summary, the contributions of this paper are twofold: (i) we mask the overhead of having strengthened operators (checkers) by letting the original operators provide outputs that will be consumed speculatively in a distributed system; and (ii) we reduce the MTTR (Mean Time To Recovery) by efficiently rolling back only the corrupted portion of the operator’s state when possible.

Roadmap: The rest of the paper is structured as follows. In Section II we provide an example application that we will use to describe our mechanisms throughout the paper. In the same section, we present the system model and background information on the usage of our speculation support. After that, in Section III we detail how the STM-based speculation can be useful in the implementation of n-variant active replication approaches. Experiments that support our claims are presented in Section IV. Finally, we discuss related work in Section V and conclude the paper in Section VI.

II. BACKGROUND AND SYSTEM MODEL

In this section we present our running example. We also overview the usage of speculation and Software Transactional Memory (STM) in event processing. Finally, we discuss our system and failure models.

A. ESP Application Example

In order to illustrate our approach, we will use the prototypical ESP application illustrated in Figure 1. This application has multiple publishers that produce information (e.g., stock quotes). A sequence of ESP operators process this information. The first operator is the Process. This operator executes some analysis on the data and issues events describing some action to be taken. Note that the actions issued may be different if two events are processed in a different order. The next operator is the Enrich operator. This operator receives action events as input and appends to these events data that will be needed to carry out the actions (e.g., non-destructive reads from offline sources like databases). Finally, the Distribute operator cryptographically signs the action events and distributes them to Consumer components that will execute them.

If the application must be able to tolerate faults, the most important step is to guarantee that the accumulated states are not lost upon a failure. Because the Enrich and Distribute operators are stateless, i.e., their outputs depend only on the current input, their replication/recovery is trivial. Thus, we omit their replicas from Figure 1.

In the remainder of the paper we will focus on the Process operator. For this reason, unless specified otherwise, we will denote the replicas of the Process operator simply by replicas. In addition, when referring to the interaction between the Process and the Enrich operators, we will use the term upstream for the former and downstream for the latter.

B. Speculation in Event Stream Processing

In this paper we demonstrate the use of speculation to mask the latency overhead resulting from runtime checking for distributed applications. Our work is based on a speculation mechanism powered by an STM [9]. The STM has a second benefit: it allows fine-grained checkpointing and rollback. This enables us to lower the MTTR for operator failures detected by a checker. In this section, we give an overview of the features of our speculation support and its applications in ESP systems. Details about the implementation can be found in [10].

The speculation support is built as an extension of an STM implementation, TinySTM [11]. STMs provide the illusion that memory accesses are grouped into transactions. Specific to the event stream processing problem, we consider that the whole processing of an event is a transaction. We consider the persistent state of the operator is the memory to be protected. TinySTM allows us to process events speculatively. Then, if we need to rollback the processing of an event, e.g., because of an operator failure, we abort the corresponding transaction. In addition, we use the delivery order of an event as the timestamp of the transaction processing it. We force that transactions with lower timestamps must be committed before others with higher timestamps.

In [10], we show that through speculation we can improve throughput by optimistically processing events in parallel. If the events conflict, the event with the earlier timestamp will not be affected and the later one will be re-executed. Otherwise, both computations can be used. Then, in [12] we show that because the processing of an event appears to be executed atomically, the execution of multi-threaded operators is not affected by non-deterministic scheduling decisions. For this reason, actively replicated multi-threaded operators do not need to agree on the scheduling of the threads. In [12], we also show that we can hide the overhead of atomic broadcast by using optimistic delivery.

In contrast, in this work, we use the same transactional support to address recovery from software faults by allowing replicas to abort a transaction that was considered harmful by some other replica. Because the STM provides memory checkpoints of the positions that were changed through STM calls, this recovery is efficient. In addition, we allow that
if some replica finishes the processing of an event sooner than others, it can forward the results speculatively and downstream operators can use the same speculation support to start processing these results. Downstream nodes can then do an abort if the voting process invalidates the result.

C. System Model

We assume a typical ESP system where nodes are connected by reliable FIFO channels (e.g., TCP) and operators use active replication for fault tolerance. Operator replicas can be put in the same node (to tolerate only software bugs and process crashes) or in separate nodes (to tolerate also node crashes). As traditionally required by active replication, we assume an atomic broadcast protocol as the ones in [2]. This protocol guarantees that messages will be delivered in the same order for all replicas of an operator. As detailed above, this ordering of the messages will be used as timestamp for the transactions that process the events.

We also assume that processing is deterministic in the replicas. In other words, all replicas will emit the same results when fed with the same inputs in the same order. When processing an input, a replica may issue zero or more events. If no output events are generated, the replica outputs a NULL event. All output events that where triggered by an input carry an id composed of the operator id, the replica id and the logical timestamp, which is the timestamp of the transaction that generated it. The transaction is then kept open until the result of the voting in the downstream component allows the replica to commit or abort the transaction.

An operator is defined by three functions: an init() function, which allocates and initializes all resources during system start up; a process() function, which is called for each input event and can emit output events; and a finish() function, which is called during system tear down. All functions are basically C code augmented with library calls that provide accessory functionalities. The compilation of a component includes a pass to automatically instrument the code for the speculation support and a pass for each variant replica that is generated. The automated generation of replicas is discussed in the next sections.

D. Failure Model

Our goal is to tolerate both operator crashes (i.e., fail-stop) and software bugs in the operators. On the one hand, crash failures are detected either locally (e.g., by catching a SIGSEGV signal when a process executes an illegal memory access) or remotely by the lease-based failure detector from [13] (e.g., when the whole node goes down). This failure detector requires that nodes (or processes, depending on the desired granularity) periodically acquire leases with other nodes. If a node does not get its lease renewed, either due to being crashed or too slow, a watchdog (e.g., the software watchdog included in major Linux distributions) forces the node to restart.

Software bugs, on the other hand, are detected through checkers. A checker is a replica of the operator and runs either in a different process or on a different node. Checkers are, however, not exact replicas of the original operator, but variants of it. Different types of checkers are able to detect different classes of software faults. In the next section, we show how to generate checkers for faults caused by common software bugs.

By using checkers, the set of detectable software faults depends on two factors: (i) faults must be benign, we do not tolerate attacks that could control a node and make it behave maliciously; (ii) the fault must be detectable by one of the checker replicas used. Therefore, our failure model is stronger than a crash failure model, but weaker than a failure model that permits malicious adversaries. Note that some Byzantine fault-tolerant protocols assume that failures are independent (e.g., [14]), i.e., these protocols cannot handle benign software bugs that affect all replicas.

We consider that the faults caused by software bugs can also be triggered by malformed inputs. This is an important
consideration as error handling code (i.e., which handles malformed inputs) is likely to contain more bugs [15].

With our failure model, we want to avoid state corruption and incorrect output due to non-deterministic behavior triggered by common software bugs. We believe that tolerating benign software faults has a strong practical motivation. It is more realistic than addressing only crashes, but does not impose the overhead or the complexity needed to handle Byzantine failures, which additionally include hardware failures and malicious replicas. In addition, if an input event causes an operator to fail (e.g., because it is malformed and/or triggers a bug in one of the operator replicas), our model allows the system to rollback and ignore this input. For this reason, we introduce an extra method to the specification of an operator, the handle_exception() method. This method gives the application programmer the opportunity to specify a special action to be executed when an event is ignored (e.g., send a special event filling the potential “gap” in the stream or log the occurrence).

E. Checker Generation

Checker generation is the process of compiling a source code to a binary that contains additional runtime checks that strengthen the operator against some class of bugs. In this work, we consider an example with two instances of an operator. The first is the original operator. The second is a checker replica, which is generated by automatically inserting additional checks for bounds checking during runtime [16]. The goal of the bounds checker is to verify that all memory accesses to a buffer, say $A$, are within its bounds.

The bounds checker prevents that memory accesses to $A$ overflow into unallocated memory or into other buffers. Unfortunately, common languages like C/C++ cannot enforce this checking at compile time. Therefore, our checker adds runtime checks to the code to enforce this. To track the bounds of buffers, allocation operations (like malloc()) are wrapped to associate the returned pointer $p$ with the bounds of the allocated buffer. At runtime, the checker maintains the memory bounds of all buffers in a table. To be able to lookup the buffer bounds of a pointer, each pointer carries the index of its associated buffer in this table. To improve performance, we store the index in the upper 16 bits of the 64 bit pointer value (similar to [17]). The associated bounds are propagated to new pointers derived from $p$ by copying or pointer arithmetic. Finally, a runtime check is inserted at every pointer dereference to check that the pointer is within its associated bounds. If a runtime check detects a fault, a signal is generated to notify the framework. Because of the instrumentation, a checker does an explicit effort to prevent that faults compromise the operator. This effort results in considerable runtime overhead for the checker compared to the original operator. We will show below that by using speculation we can hide these costs.

Besides the bounds checker, other checkers could be designed. For example, a checker could compute a checksum of the relevant objects in the operator’s state and include it in the output. Then, the voting process would be able to detect hardware problems such as bit flips in one of these objects. Another example is to have two versions of the same algorithm, one that is faster, but not reliable, and one that is slow, but reliable. The faster version forwards the result earlier and allows speculative processing downstream. However, if this output is not confirmed by the slower version, the speculative computations can be rolled back.

III. SPECULATION IN N-VARIANT ACTIVE REPPLICATION

In this section we first detail how outputs from different replicas are voted. Then, we explain the process of detecting faults and classify the different types of faults. After that, we provide examples of the benefits of speculation.

A. The Voting Process

As discussed in the previous section, for each input event, an operator outputs at least one output event: the NULL event meaning that it did not issue regular output events. We also discussed that events are processed inside memory transactions, which are kept open until results of all variant replicas are checked. In addition, events carry the logical timestamp of the transaction that issued them. Then, when a set of output events from a replicated operator reaches the neighbor downstream node, a voting process is started. The downstream node groups events by their logical timestamp. In order for an event to be approved by the voting process, all replicas must have issued the same number of events and the events must have the same content. If that is the case, events are acknowledged (i.e., an ACK is sent from the downstream operator to all upstream replicas). Otherwise, if events do not match, they are non-acknowledged (i.e., an NACK is sent). An ACK will cause all the upstream replicas to commit the transaction that generated the corresponding events. On the contrary, a NACK causes the replicas to abort the transaction. If the downstream operator is enabled to use speculation, it can start processing events before the voting process is finished.

A summary of the messages exchanged between upstream and downstream operators is shown in the detail in Figure 1. In the downstream direction, there are 3 types of messages: $e^S$, an event (or a set of them) that should be processed speculatively; NULL, messages denoting that the corresponding transaction did not output events; and, FAULT, denoting that a fault was detected while processing the transaction (detailed below). In the upstream direction, there are only the ACK and NACK messages as explained above.

B. Software-Fault Detection

Regarding detection, we classify software faults according to the place of detection and the severeness of the fault. On
the one hand, faults can be detected locally at the current node, or can be missed locally, but detected by a remote node. On the other hand, a fault is considered minor if it affects at most local-scope variables or operator state variables (i.e., positions that can be recovered by aborting and rolling back the transaction) and major otherwise (e.g., it corrupts the framework’s state). Minor faults require only the abortion of the respective transaction, whereas for major faults the whole process has to restored. Thus, minor faults have lower MTTR than major faults. In the following, we illustrate these faults through some examples.

Recall the Process operator from Figure 1. Assume that the two variant replicas above are being used in conjunction with the original operator. Assume also that a malformed input event causes the original operator to issue a division by zero during its processing. This instruction will cause a signal to be generated (e.g., SIGFPE in Linux), which will be caught by the event-processing framework. This fault is locally-detected and is a minor fault. Note that for a regular ESP framework running this application even if it would catch and treat this signal, there is not much that it could do to fix the potentially inconsistent operator state. Nevertheless, with the STM, the transaction can simply be aborted and rolled back. Then, the problematic input is discarded and a special empty event, named FAULT, is sent as the output for that timestamp.

Now consider the case that the input event leads to the operator trying to write to an array that is part of its state, but to an index that is out of bounds. In this case, it is possible that no exceptions are generated at the original operator and that it executes to its end, producing a potentially incorrect result. Nevertheless, the checker replica would catch the fault and abort the execution, sending the FAULT event as above. As a consequence, when the downstream operator compares the results from the different replicas it sees the divergence and sends a NACK to all replicas. The Process replicas will see the NACK and abort the transactions for that timestamp, discarding the input event that produced the fault. From the point of view of the original operator, this is a remotely-detected fault (because it did not detect it), but a minor one (because it can be fixed with a local abort and rollback).

The NACK can also be used in a preemptive way. For example, the original operator in the example above may have entered an infinite loop after reading a memory position that it was not supposed to read (it has read outside the bounds of the array). This operator will then be blocked in the infinite loop. Meanwhile, the checker variant detects the problem and sends the FAULT event. Because one FAULT event is enough for the voter to decide that a timestamp must be aborted, the voter can already forward the NACK messages regarding that timestamp. Once this NACK reaches the operator, whose processing thread is in an infinite loop, a signal is generated, which will interrupt the processing. Once interrupted, the speculative support learns through the NACK message that the processing for that timestamp is to be aborted anyway and can abort it immediately, proceeding to the next event.

Next, consider that while the original operator wrote to an out-of-bounds position of an array stored on the heap, it corrupted state of the framework. For this situation, no local recovery may be possible. The bounds checker prevents this corruption on the checked replica. Thus, at least the checker replica will be able to survive (rolling back the transaction and ignoring that input) and, detecting that the other replicas are unresponsive, it can help them to recover by providing a correct and actual copy of the operator state. Checker replicas can also identify the addresses where the fault occurred, enabling the distinction of faulty accesses executed under the STM and that can be rolled back with an abort, and faulty accesses done outside the STM (e.g., to local-scope arrays). This information is included in the FAULT event to the downstream node and in the NACK event forwarded to all replicas. Thus, a major fault is a fault where no local recovery is possible.

![Figure 2. State machine of the protocol running at the replicated operator.](image)

![Figure 3. State machine of the protocol running at the downstream voter.](image)

Figures 2 and 3 illustrate the protocols at the replicas and at the voter downstream, respectively. The failure-free paths are drawn in black while the behavior under different failure types is shaded. Note that, in the figure, we use NACK for signaling minor failures and NACK* for major ones.
Although possible, for simplicity the figures do not consider multithreaded executions.

Finally, one special case is with stateless operators. Stateless operators also need to be checked as they, as well, may contain bugs that cause erroneous outputs to be generated. These operators do not have a state that needs to be rolled back when a checker variant detects a problem and there is no need to send NACK messages in case of minor faults. Nevertheless, note that in case of major faults, the complete operator still needs to be rolled back as persistent data on the operator may have been corrupted (e.g., execution parameters assigned statically during initialization).

C. Committing, Rolling Back and Early Processing

Recall the example from Figure 1. Assume that there are two events \( e_1 \) and \( e_2 \) in the input queues of the Process replicas, where \( e_1 \) is to be delivered first. Assume also, that the operator has only one thread and that replicas 1 is fast (e.g., it is the original operator) and replica 2 is slower (e.g., because it implements bounds checking).

First, consider the case of failure-free execution. Replica 1 is the first to finish its processing and forwards an output event \( e'_1 \) to the Enrich operator. Because the Enrich operator is idle, it decides to start processing this input speculatively. It is then speculating that no faults will be detected and \( e'_1 \) will be approved when the voting is finished (as in the dark path in Figure 3). In addition, because replica 1 is also idle (it waits for an acknowledgement from the downstream node), it also decides to speculate. It then starts processing event \( e_2 \) speculatively. If event \( e_2 \) reads from any part of the state that was modified during the processing of \( e_1 \) (i.e., a read-after-write), a dependency between \( e_2 \) and \( e_1 \) is created. In this case, the processing of \( e_2 \) can only proceed when \( e_1 \) is allowed to commit. Note, however, that other kinds of interactions (i.e., read-after-read, write-after-read, and write-after-write) do not create such dependencies.

Later, replica 2 finishes and sends the same output event \( e'_1 \) as replica 1 and the voting process downstream finally approves \( e'_1 \). In addition, the Enrich operator has already started processing \( e'_1 \). If it has finished, it was only waiting for this approval in order to commit the processing. Otherwise, if it is still processing, it will be able to commit as soon as it finishes. Meanwhile, an ACK to both replicas. Replica 1 will then be able to commit the processing of \( e'_1 \) and maybe already send \( e'_2 \). Note that if \( e_2 \) did not depend on \( e_1 \) there would be no reason for replica 1 to wait for \( ACK_{e_1} \) before sending \( e'_2 \). In both cases, the end-to-end processing latency is reduced by having operators making progress while waiting for acknowledgments or voting.

Now consider the case of a locally-detected minor failure. When the replica 1 detects a fault while processing event \( e_1 \), it can immediately abort it and start processing event \( e_2 \). Again, processing latency is reduced by not waiting for the acknowledgments.

If a minor fault is detected remotely, instead of receiving an ACK, replac 1 would receive a NACK. They would consequently abort the execution of \( e_1 \). If \( e_2 \) was dependent on \( e_1 \), it is now allowed to proceed, considering an operator state that ignores event \( e_1 \). On the contrary, if no such dependency exists, the NACK has no impact on the processing of \( e_2 \) (\( e'_2 \) could even had already been sent).

Finally, if a major fault (e.g., a buffer overflow that corrupted the state of the framework) is detected during the processing of \( e_1 \), replica 1 needs to restart and retrieve the state of a correct replica (e.g., the checker replica 2). In this case, speculation is still beneficial by enabling the correct replica 2 to proceed computing event \( e_2 \) and the downstream node to start processing \( e'_2 \), output by replica 2, while replica 1 executes the recovery procedure. Note that, as long as we pause the committing of transactions, the operator state does not change while replica 2 is processing event \( e_2 \) (and other succeeding events) and, thus, there is no need to pause the processing during recovery.

IV. Evaluation

In the following experiments we show how the speculation support reduces the latency costs of active replication enhanced with checkers. The experiments were executed in a machine equipped with 2 quad-core Xeon (8 cores in total), except for the third experiment, which was executed on a quad-core Opteron machine (16 cores in total). Both machines run Linux (Ubuntu 8.04 and Fedora 10, respectively). In the experiments below, we focus on detecting and tolerating software bugs (crash-only fault-tolerance is addressed in [12]). Each operator runs in a process that is connected to the neighbor operators through TCP connections. In addition, we consider an optimization that allows the two replicas to communicate their results directly to each other and thus avoid an additional communication round by not waiting for a message from the downstream voter.

In our first set of experiments, whose results are shown in Table I, we compare the overhead of several versions of two example operators. It is because of the difference between non-checked and checked versions that the proposed

<table>
<thead>
<tr>
<th>Variant</th>
<th>Operator processing time (( \mu s ))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-k</td>
</tr>
<tr>
<td>Original</td>
<td>71</td>
</tr>
<tr>
<td>Checker</td>
<td>344</td>
</tr>
<tr>
<td>Speculative</td>
<td>91</td>
</tr>
<tr>
<td>Speculative checker</td>
<td>356</td>
</tr>
<tr>
<td>GCC Mudflap</td>
<td>139</td>
</tr>
<tr>
<td>Valgrind Memcheck</td>
<td>3150</td>
</tr>
</tbody>
</table>
speculation approach can achieve considerable performance gains. The top-k operator is a stateful operator that is based on a count sketch, a synopsis data structure that summarizes the stream in a bounded amount of memory [18]. The basic filter is a simple stateful filter that forwards all events it receives, no additional computation is carried out. The Original variant of these operators contains no instrumentation for speculation or checking. The Checker variant applies the out-of-bounds checking discussed previously. The Speculative variant performs no checking, but instruments the accesses to the state of the operator for transactional executions. The Speculative checker implements both checking and transactional execution. For comparison we show the overhead introduced by two state-of-the-art third-party checkers: GCC’s Mudflap [19] a memory access checker and Valgrind Memcheck [8] a memory debugger. Note both checking tools do not support speculation and cannot rollback. Nevertheless, they provide a base reference for the runtime overhead of our checker.

In our second experiment, we consider an application similar to the one from Figure 1. It is composed of a replicated Process operator, a non-replicated Enrich operator and a non-replicated Distribute operator. The Process operator implements the top-k filter operation discussed above. This operator forwards only events with values that appear with a frequency higher than a programmed threshold (although in this experiment we set the threshold so that no events are actually dropped). Next, the Enrich computes the moving average of the filtered stream (only slightly heavier than our basic operator in Table 1) and appends it to the events. Finally, the Distribute operator adds a 2048-bit public-key signature to the events before forwarding them to the consumers (processing time of around 200µs).

The results of this experiment are shown in Figure 4. The y-axis represents the elapsed time between the creation of an event in its publisher and its arrival at the respective points (A, B, C, and D, as detailed in Figure 1). In the speculative case, the Process operator propagates speculative output events and the succeeding operators can already process them early (the Enrich is instrumented with the speculation support and the Distribute does not have state that must be rolled back in case of a wrong speculation).

In our third experiment we consider a different application. In this case the Process operator implements a simple neural-network that classifies events according to a pre-configured model. In addition, we consider a stateful Enrich component that uses a top-k to enrich the event with its occurrence frequency. Finally, the Distribute operator serves only as a gateway that filters out speculative events before they reach the consumers. In this example, both Process and Enrich operators will be replicated. Then, in addition to the measurement points A, B, C, and D in Figure 1, we also consider a point C’, which is the output of Enrich*, the checker replica for the Enrich operator.

The results of this experiment are shown in Figure 5. This experiment was executed in the machine with 16 cores to enable each thread to execute in its own core.

Figure 5 clearly illustrates the most important goal of the speculation: the elapsed times up to points C and D are much lower in the speculative version as the variants for the Enrich operator can start processing as soon as they receive the speculative output emitted by the faster replica of the Process operator (point A). In fact, the faster Enrich variant is able to output a speculative result (line C) even before the checker variant of the Process operator finishes. This result, however, can only be forwarded to the consumers (D) after it is validated by the executions of both checker variants. This scenario is especially useful because in practice all operators in an application are to be replicated. Speculation then enables that events go speculatively through a faster path that enables checking in different operators to be done almost in parallel.
Finally, in our last experiment we revisit the first scenario and insert a minor bug in the operator. Then, events (from 25 to 35) trigger the faults and cause both replicas to rollback the processing of the respective events. In addition, speculative events may have reached the output of the Enrich operator, but this will never be the case for final events and, thus, there is a gap in the events emitted by the Distribute operator \((D)\). The results of the experiment are shown in Figure 6.

V. RELATED WORK

Active replication, also known as the replicated state machine approach [1], is a classical technique to provide fault tolerance in distributed systems. In the event processing domain, active replication has been addressed in [12], [20]. In both cases, only crash failures are considered.

To solve the problem of common-mode failures, n-version programming [3] has been proposed. Then, multiple independent copies of the same software component are developed and executed as replicas. The usage of automatically generated versions of a component is discussed in [4], [5]. Our approach is to generate variants with additional error detection. Checkers automatically add error detection code at compile time. In our evaluation we use an out-of-bounds checker similar to [17], but checkers exist for a variety of bugs, e.g., buffer overflow bugs [16], [17], [19], using uninitialized memory bugs [8] and control flow manipulation bugs [21], [7]. All of these and other checkers have in common that they add considerable runtime overhead. In ESP increasing the runtime overhead of operators translates into increased processing latency. We partially hide this latency by using speculation.

Zyzzyva [14] addresses Byzantine fault tolerance and uses some form of speculation to improve performance. There, the speculation consists in having the components to produce outputs whose correctness will not be known until a correct client checks them. The advantage is that there is no need for voting on the failure-free case. On the other hand, components (even correct ones) may be in an inconsistent state until a correct client submits a request and verifies the reply. Then, the client will detect inconsistencies and trigger a complete recovery in the inconsistent nodes that will restore a distributed and trusted checkpoint. Another disadvantage of this approach is that, because it tolerates Byzantine faults, interaction between components, it is much more complex. As an example, a reply from a component contains all the information necessary for the client to validate that different replicas executed the same historic sequences of requests to reach the same result. Speculation in a sense closer to ours is presented in the context of distributed databases by Kemme et al. [22]. There, the system starts processing requests before the final ordering of the messages is computed. Then, it may later roll back if the requests are discovered to be out of order.

There are several approaches to hide the overhead of runtime checks in non-distributed systems (e.g., [23], [24]). First, these approaches aim only in detecting the faults, they do not tolerate them. Second, they partition the execution of the original application in epochs and each epoch is replayed by a checker variant of the application. To apply this approach to distributed systems we abandoned the traditional notion of an epoch and the complete processing of an event forms a processing unit. Finally, we mitigate the checking overhead not only within one operator, but through the whole distributed application.

VI. CONCLUSION

We presented an approach for ESP systems that uses automated code checking and speculation to tolerate both crash failures and software faults caused by common programming bugs. In order to achieve this goal, the system uses the concept of replication with checker variants to detect the faults and a software transactional memory that: \((i)\) provides an speculation support that reduces the performance impacts of replication and software checking (by enabling checks to be executed almost in parallel in different operators); and \((ii)\) allows very fine-grained and efficient rollbacks (which reduce the mean time to repair). As future work, an interesting direction is to consider further types of checkers (and, therefore, broader failure models) and investigate how (at least some) major failures could be transformed into minor ones.

ACKNOWLEDGMENT

This research has been partially supported by the FP7 Integrated Approach to Transactional Memory on Multi- and Many-core Computers (VELOX) project (ICT-216852) supported by the European Commission.

REFERENCES


