Language-Level Checkpointing Support for Stream Processing Applications

Gabriela Jacques-Silva[‡*], Buğra Gedik[♠], Henrique Andrade[♠], Kun-Lung Wu[♠]

[‡] Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
gjsilva@crhc.illinois.edu

[♠] Thomas J. Watson Research Center
IBM Research
{bgedik,hcma,klwu}@us.ibm.com

Abstract

Many streaming applications demand continuous processing of live data with little or no downtime, therefore, making high-availability a crucial operational requirement. Fault tolerance techniques are generally expensive and when directly applied to streaming systems with stringent throughput and latency requirements, they might incur a prohibitive performance overhead. This paper describes a flexible, light-weight fault tolerance solution in the context of the SPADE *language and the System S distributed stream processing engine. We devised language extensions so users can define and parameterize checkpoint policies easily. This configurable fault tolerance solution is implemented through code generation in* SPADE*, which reduces the overall application fault tolerance costs by incurring them only for the parts of the application that require it. In this paper we focus on the overall design of our checkpoint mechanism and we also describe an incremental checkpointing algorithm that is suitable for on-the-fly processing of high-rate data streams.*

1 Introduction

Stream processing has emerged as a paradigm to analyze streaming data such as audio, video, sensor readings, and business data in real-time. Data streams may have extremely high rates. Traditional solutions for data analysis, such as database management systems are usually not suitable, since they first store and then process data. Databases enforce ACID properties, which impact performance; therefore, they are not, in many cases, able to process data at high rates with low latency.

Developers build streaming applications as data-flow graphs. These graphs consist of interconnected *stream operators* that implement analytics over incoming data streams. Users deploy streaming applications as continuous queries, since data sources (e.g., sensors) constantly produce new information. For applications to generate semantically correct results even in the presence of failures, it is essential to employ fault tolerance techniques.

For instance, sensor-based patient monitoring applications require rigorous fault tolerance, since data loss or computation errors may lead to catastrophic results. On the other hand, there are applications that do not have such strict requirements. One such example is an application that discovers pairs of caller/callee by data-mining a set of VoIP streams [24]. In case of failures, VoIP packets may be lost or a user can get disconnected from the VoIP system. The application can still infer the caller/callee pairs, although with less confidence. Such class of applications is called *partial fault-tolerant*. Moreover, in some streaming applications it may be better to produce partial results within a time bound than to produce complete results too late [4]. In systems that aim to provide maximum data throughput, resources must be spent with parsimony. Therefore, the one-size-fits-all fault tolerance is not the best approach for streaming applications.

To achieve massive parallelism and scalability, stream processing systems distribute the application over nodes in a cluster. Hence they are subject to the failure model of distributed systems [8], where messages can be omitted, duplicated, nodes can crash and the network can become partitioned. A successful technique to provide fault tolerance to large-scale distributed systems is synchronous checkpointing [9]. While it can be applicable to streaming applications, more specialized approaches can be more appropriate. A streaming application is one particular type of a parallel application. While the latter, in general, has clear synchronization points (barriers), the former is constantly changing its state based on the incoming data. Additionally, a typical streaming application has real-time constraints. Stopping the application to coordinate a checkpoint operation directly affects the timing requirements. Many fault tolerance techniques were developed specifically for stream computing [12, 15, 25]. However, they do not consider the semantics of the application and apply fault tolerance throughout the system. This results in unnecessary performance overhead and an overall degradation of performance when contrasted with a more targeted approach.

In this paper we describe the design of checkpointing techniques for *partial fault-tolerant* streaming applications. We deployed such techniques in SPADE [11], the declarative stream processing language of System S – a massively scalable and distributed stream processing middleware under development at IBM Research. To provide flexibility to the user, we do not enforce a single checkpointing policy for the whole application. More specifically, we allow the application developer to annotate his SPADE application source code, so that he can choose

*Sponsored by CAPES/Brazil.

which parts of his application should be fault-tolerant. The use of language-level annotations is a natural approach to specify such policies, since the developer knows his application semantics and failure behavior. To carry out the behavior chosen by the user, we take advantage of SPADE'S code generation framework to automatically produce the extra code required by the fault tolerance policies.

Our main contributions are: (i) a framework for applying fault tolerance policies for streaming applications via language annotations; (ii) an incremental checkpointing algorithm for sliding window-based stream operators; (iii) a code generator that outputs specialized checkpointing code based on the stream operator type and instance.

The rest of this paper is organized as follows. Section 2 discusses the general architecture of System S and its fault tolerance guarantees. Section 3 describes the SPADE language, its programming model, and its code generation framework. Section 4 describes our checkpointing techniques for user-defined operators and window-based operators. Section 5 demonstrates the applicability and benefits of our technique and includes several performance studies using a real-world manufacturing application as well as synthetic applications. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 System S

As mentioned in Section 1, System S is a distributed large scale stream processing middleware. One of its characteristics is to be non-transactional, since it does not have atomicity or durability guarantees. This is typical in stream processing systems where applications are continuously running and quickly producing results.

In System S, independent executions of an application with the same input may generate different outputs. There are two main reasons for this non-determinism. First, stream operators often consume data from more than one source. If the data transport subsystem does not enforce message ordering across data coming from different sources, there is no guarantee in terms of which message the operator will consume first. Second, operators can use time-based windows. Some stream operators (e.g., aggregate and join) produce output based on the data within specified window boundaries. For example, if a programmer declares a window which accumulates data over 20 seconds, there is no guarantee that two different runs receive the same amount of data in the defined time interval.

System S deploys each application as a job. A job is composed of multiple *Processing Elements* (PEs), which are containers for the operators that make up an application data-flow graph. A PE hosts one or more *stream operators*. To run a job, the user contacts the *Job Manager* (JMN), which is responsible for dispatching the PEs to remote nodes. The JMN contacts a *Resource Manager* to check for available nodes in the system. Then, JMN contacts the *Master Node Controller* (MNC) on the remote nodes, which instantiates the PEs locally. Once the PEs are running, the *Stream Processing Core* (SPC) is responsible for carrying out the stream connections and transporting data between PEs.

The System S middleware has many self-healing features. As a central component, JMN plays a fundamental role in this. Besides dispatching PEs, JMN also monitors their life-cycle. Each MNC monitors which PEs are alive in its local node and sends this information to JMN. If a PE fails, JMN detects it and re-dispatches the PE in the same node. If the PE has crashed due to a node failure, JMN may restart the PE in a different node. During the recovery time, the behavior of the PEs connected to the crashed PE differs. The behavior depends on the specific position of the PE in the data-flow graph, as it is shown by Figures 1 and 2.

Figure 1 shows an example of a graph with 6 PEs. PE 1 sends the same data to PE 3 and PE 4. PE 4 also consumes data from PE 2. PE 5 and 6 consume data from PE 3 and 4, respectively. Figure 2 shows the consequence of a failure in PE 4. As expected, PE 6 does not have input streams to process, therefore, it does not produce any data. The behavior differs for PE 1 and PE 2, since they are data producers (also referred as *source PEs*). SPC *discards* all the new data PE 2 sends, given that there is no PE to consume it. PE 1 still has one live connection, so it continues to send new data to PE 3, but it stops sending data to PE 4. Once PE 4 is reintegrated, the connections are reestablished. At this point, PE 2 stops discarding data and PE 1 resumes sending *new* data to both links. More details on SPC and System S failure behavior can be found in our earlier work [1, 14].

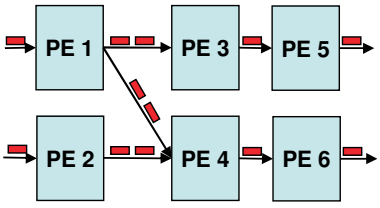

Figure 1: SPC normal behavior

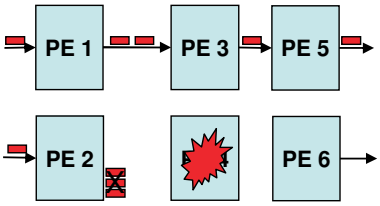

Figure 2: SPC under PE failure

3 SPADE

SPADE [11] is a language and a compiler for creating distributed data stream processing applications to be deployed on System S. SPADE offers: (i) a language for flexible composition of parallel and distributed data-flow graphs; (ii) a toolkit of type-generic built-in stream processing operators; (iii) an extensible operator framework, which supports the addition of new type-generic and configurable operators to the language, as well as new user-defined operators (UDOPs) used to wrap existing, possibly legacy analytics; (iv) a broad range of edge

adapters used to ingest data from outside sources and publish data to outside destinations, such as network sockets, databases, file systems, etc.

The SPADE language provides a stream-centric, operator-based programming model. The stream-centric design implies a programming language where an application writer can quickly translate the flows of data from a block diagram prototype into the application skeleton by simply listing the stream data flows. The second aspect, i.e., operator-based programming, is focused on designing the application by considering the smallest possible building blocks that are necessary to deliver the computation an application has to perform. In summary, SPADE programs are data-flow graphs, where operators are connected via streams and serve as basic computational units performing stream transformations. A *tuple* is the basic unit of data in a stream. Each tuple has a set of attributes, built from SPADE basic types (e.g., integer, string).

A key distinction between SPADE and other stream processing middleware is its emphasis on *code generation*. Given an application specification in the SPADE language, the SPADE compiler generates *specialized* application code based on the computation and communication capabilities of the runtime environment. This specialization includes many aspects, including *code fusion*. Code fusion is the ability to place multiple operators inside a single PE. Instead of using the regular stream transport, the streams are converted into function calls [2]. Fusion results in higher operator throughput, but it has an important implication on fault tolerance. If an operator in the fused group crashes due to a software bug, this results in the crash of the whole set of operators hosted by a PE.

4 Checkpoint Design

As described in Section 3, SPADE has an operator-based programming model. To checkpoint an operator, it is important to define the behavior of the operator under failure and the state it should have once it recovers. To minimize the performance overhead, we have to develop techniques that are specific to individual operator types and that can be customizable by different instances a developer might employ in the context of one application. Here is where we can take advantage of the SPADE'S compiler and code generation framework. With the knowledge of the application, SPADE can generate code that uses specialized checkpoint techniques for each operator instance based on their parameters. For *each* operator we have to take into account the following three aspects: (i) the minimal operator state required in order to recover after a failure, (ii) whether the operator is able to produce semantically correct results after a recovery from a failure, (iii) whether the restored state contains stale data. We assume a fail-stop model for an operator failure.

Regarding operator state, SPADE has both stateless and stateful operators. An example operator that can be either stateless or stateful is a *functor*. Functors perform tuple-level manipulations such as filtering, projection, and mapping. For every incoming tuple, the functor generates an output, unless the input tuple does not satisfy the filter predicate. If the filter predicate does not involve any variables other than the attributes of the current tuple (e.g., a stream attribute in a tuple is greater than a value), there is no need to save state. If the operator crashes and restarts, it can still filter tuples with the same predicate. The developer can also customize the functor operator to update state variables when it receives a tuple (e.g., to compute the running average for an attribute). In this case, a functor is stateful and the state variables should be checkpointed. Note that the variable value can be affected by tuple loss (e.g., maximum value of an attribute). This should be considered when deploying the checkpoint technique.

Depending on the operator type, checkpointing the internal operator state and restarting it may not be sufficient to provide correct operator semantics. One such example is the *barrier* operator. Barriers synchronize logically related streams. The operator emits an output tuple every time it has at least one tuple from all of its inputs. As described in Section 2, when a PE fails, tuples may be lost during the recovery time. For a barrier operator to provide correct results after recovery, we need to save in-flight tuples. If tuples are lost, there is no guarantee that the logical pairing the operator produces is correct. For this type of operator, it is mandatory to apply other techniques that save in-flight tuples, such as upstream backup [12]. The difference of our technique is that through code generation we can enforce in-flight tuple buffering only for the operators that require such semantics.

In streaming applications, it is possible that the operator state is valid only during a certain time frame. One example is the *aggregate* operator. The aggregate operator groups and summarizes the incoming tuples according to an operation (e.g, sum, average, among others). It performs the operation over all the tuples that are inside a window boundary. One option available to the developer is to parameterize the window behavior based on size (`x`) and an output interval (`y`). The size of the window can be defined as all the tuples accumulated over the last `x` seconds. As new tuples arrive, the operator discards tuples older than `x` seconds. At every `y` seconds, the operator computes the aggregate function based on the current contents of the window. On the event of a failure, the restored state of an aggregate contains all the tuples that are inside the window at the time of the checkpoint. This means that on recovery the middleware must handle stale data. In normal operation, some of these tuples would have been discarded due to the arrival of new tuples. Therefore, the recovery routine has to eliminate the expired tuples.

In the next sections we show how we modified the SPADE infrastructure to support checkpointing and how we added fault tolerance to *user-defined* operators as well as the built-in *join* operator, which is an example of a windowed operator. Our checkpointing technique applies to most other windowed operators; however, due to space constraints, this paper focuses only on the join.

4.1 SPADE Support

To selectively provide fault tolerance, we allow the user to define in his source code what parts of his ap-

```
stream CountStream(count: Int, str: String)
 := Source()
 ["ctcp://src.somewhere.com:789"]{}

stream AverageStream(avg: Int, str: String)
 := Udop(CountStream)["Avg"]{}
 -> checkpoint=10 # 10 seconds
```

Figure 3: Checkpoint annotation in SPADE

plication should be fault-tolerant by employing language constructs. After the user annotates his application, the SPADE compiler generates code that saves the state of the selected operators with a pre-established frequency. For built-in SPADE operators, the compiler automatically generates checkpoint methods. The state of each operator is assumed to be independent from each other. When operators are fused in the same PE, the state of each of them is saved at their specified frequency. We do not save them all at the same time to allow maximum possible throughput. Due to performance overheads, the user may not want to checkpoint an operator with a large state at the same frequency of an operator with small state. The state independence also applies to PEs. During a PE recovery, the states of other interconnected PEs are not rolled back. This is consistent with the recovery model outlined in Section 1.

Figure 3 shows an example of how to specify that an operator should be checkpointed. The example has a *source* (`Source`) and a *user-defined* (`Udop`) operator. The source stream (`CountStream`) produces tuples with an integer (`count`) and a string (`str`), extracted from the packets coming from a TCP client located at `src.somewhere.com:789`. The user-defined operator (UDOP) creates an `AverageStream` where each tuple contains an integer (`avg`) and a string (`str`). Note that the UDOP contains the `checkpoint` keyword and the associated checkpoint frequency in seconds (`10`).

The SPADE compiler generates code both for the operators and the PEs (with or without fusion). For the operators, it generates extra code so it can implement the checkpointing policy. The extra code is dependent on the operator type and configuration. Further details on the generated code is given in Sections 4.3 and 4.4. The compiler modifies the configuration of the PEs to selectively enable checkpoint services for its operators.

4.2 Runtime Support

The PE execution flow changes if any of the operators in a PE has the `checkpoint` keyword. Figure 4 shows the PE operation steps with checkpointing. A PE is comprised of a `PEWrapper`, which manages all the operators it contains (OP1-OP3). When the PE starts, it initiates a thread (`PECheckpoint`) that is responsible for carrying out the checkpoint policy (step 1). `PECheckpoint` verifies which operators should checkpoint and builds a priority queue with their next checkpoint timeout. In step 2, the thread removes the next operator to checkpoint from the queue (`getExpiringOperator()`) and sleeps until it is time to save the next state snapshot. When the thread wakes up, it invokes the `getCheckpoint()` method of the operator (step 3). This method contains the logic to serialize the operator state. While the `getCheckpoint()` method executes, the operator cannot process any new incoming tuples. Once the method call returns, the thread saves the serialized state to the storage subsystem via the `OPState` class (step 4). `OPState` saves the new state in a temporary file first, which is later renamed to a permanent file through an atomic `rename()` operation.

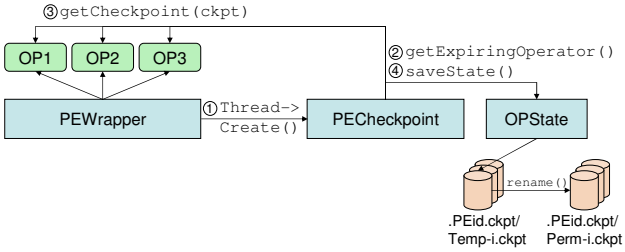

Figure 4: PE checkpoint operation

The PE recovery is similar. When the PE starts up, it searches for the checkpoint files based on the PE and the operator IDs. These identifiers are constant throughout the lifetime of the PE. At first, the restore procedure checks the integrity of the checkpoint file via a hash value computed and stored by the `OPState` class. If the file is corrupted, the procedure discards the state and the operator restarts with a fresh state. One option is to maintain different versions of checkpoint files and restore an older checkpoint in case of corruption of the latest one. File corruption is handled differently when the operator rebuilds its state from multiple files. Such case is described in Section 4.4 in the context of the join operator. When the checkpoint file is valid, the restore routine invokes the method `restoreCheckpoint()` in the operator class. Similar to the serialization function, the de-serialization implementation is specialized for each operator type.

4.3 User-Defined Operator

In SPADE, the user has the capability to extend the basic set of built-in operators via user-defined operators (UDOPs). With UDOPs, the developer can use external libraries and implement customized operations for his particular application. SPADE generates skeleton code so the user can easily handle tuples from System S streams, process them with the specialized code, and send them over as a System S stream to other operators.

To checkpoint UDOPs, the SPADE compiler adds checkpoint method interfaces to the generated skeleton. The user has to fill-in the methods with the appropriate serialization logic. This approach is similar to the technique employed in the Fault-Tolerant CORBA standard for application-level state checkpointing [18]. The `PECheckpoint` thread automatically invokes the serialization methods at the specified frequency.

Figure 5 shows the generated interfaces and an example of how to add the serialization code. This is part of the SPADE output for the code shown in Figure 3. In this example, the state of the UDOP has two member variables, namely `avgCount` and `numCount`. The user receives a reference to a serialization buffer object (`SBuffer`), used both for the state saving and restoring methods. The user has to serialize/de-serialize the data to/from the buffer in

```
void UDOP_Avg::getCheckpoint(
 SBuffer &checkpoint) {
  AutoMutex am(mutex);
  checkpoint << avgCount << numCount;
}
void UDOP_Avg::restoreCheckpoint(
 SBuffer &checkpoint) {
  AutoMutex am(mutex);
  checkpoint >> avgCount >> numCount;
}
```

Figure 5: UDOP checkpoint interface

the same order. Because other methods can modify the member variables during checkpoint, they must be protected by a mutual exclusion construct.

4.4 Join Operator

The *join* operator correlates two streams. The streams are paired up based on the join predicate and the window configuration. Two different windows (one per incoming stream) group the tuples from each incoming stream. Each stream can have a different window configuration. The window keeps the input tuples in the order of arrival. Once the operator receives an input tuple from *stream 1*, it evaluates the predicate condition against all tuples in the window of *stream 2*. If the predicate evaluates to true, the operator pairs the matching tuples and sends them downstream. After the pairing stage, the operator inserts the input tuple into its corresponding window. If the window is full, the oldest tuple is discarded, i.e., the window *slides*.

A join operator can have an arbitrarily large window. For example, the application described in Section 5.1 has three join operators. The sizes of two of them are 512,000 and 128,000 tuples. Tuples in join operators may accumulate over a long period of time, depending on the stream input rate. If the operator crashes and there is no checkpoint, the operator produces few outputs for a long time, since it has to fill up its windows in order to produce matches at full rate. With checkpoint, we can recover most of the window content. Therefore, the operator is able to produce matches right after the restore operation.

If for the join operator we use the same checkpoint technique employed for UDOPs, all the tuples inside the window should be serialized. This results in the serialization of large chunks of data, which introduces a prohibitive performance overhead. To overcome this problem, we devised an incremental checkpointing technique for sliding window-based operators, such as joins. Incremental checkpointing algorithms for stream computing require performing a checkpoint maintenance operation per tuple that arrives to an operator [25]. For low-cost checkpointing, both tuple insertion and serialization should be light-weight operations.

Incremental Checkpoint Algorithm

In a sliding window configuration, as new tuples arrive, the older ones are evicted from the window. This behavior can be implemented with a *double-ended queue* data structure. New tuples are inserted at the tail of the queue and old tuples are removed from the head of the queue. Between two checkpoints, the state of the operator can be described by two possible configurations. The first is if the window only has new tuples, meaning that the total number of new tuples since the last checkpoint is greater than the size of the window. In this case there is no common state between the last checkpoint and the current one. If the number of new tuples is less than the size of the window, the serialization time can be decreased by not repeating this operation for tuples that are part of both the last and the current checkpoint.

To incrementally save sections of the same window, we use independent checkpoint files. Saving a sliding window in one file is not appropriate. To match the position of the tuples in the window, we would need to change the position of the tuples in the file. This is not an efficient approach, since we need to rewrite the whole file to disk.

Using page protection hardware to detect which memory pages of the operator process we should save, such as in Libckpt [19], is also not suitable. As mentioned in Section 3, SPADE can fuse a set of operators within a single PE. This means that all operators are running in the same memory space; hence, memory pages of all the operators are being modified and serialized. This approach breaks the language semantics and records unnecessary state.

One option to save sections of the window is to save one tuple per file, serializing only the new tuples at every checkpoint interval. However, this can generate thousands of files, incurring a prohibitive overhead. Another approach is to save all new tuples in a single file. Due to system non-determinism (e.g., network), it is highly likely that there is a different number of new tuples at each checkpoint interval. This results in an undefined number of files for the state of each join window. This creates problems in both recovering the state and garbage collecting the files. Without garbage collection, we may have to inspect a large number of files to determine the correct state upon recovery.

To overcome the aforementioned problems, we devised a circular buffer data structure that divides the sliding window into fixed *groups of tuples*. As a result, we can control the number of checkpoint files and avoid the garbage collection problem. At every checkpoint interval, we verify which groups have new tuples and save their contents to disk, serializing both new and old tuples in a group. We limit re-serialization by dividing the window into smaller groups. Groups that did not change between two checkpoint intervals do not need to be re-saved. Since it is a sliding window, only groups with more recent tuples change.

Each position in the circular buffer contains the following data: (i) a checkpointing file name; (ii) a dirty bit, which indicates if group should be serialized to disk; (iii) the current number of tuples in the group; (iv) the window index of the most recent tuple in the group, so we can correctly index the double-ended queue data structure. To decrease the performance overhead, our algorithm updates the circular buffer only at every checkpoint interval.

The number of positions in the circular buffer data structure is based on the number of tuples we want to save per checkpoint operation. We divide the window size by

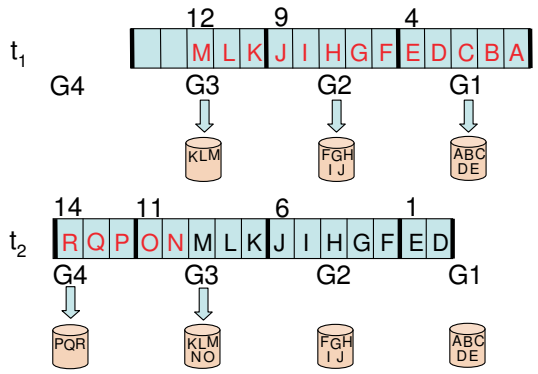

Figure 6: Join checkpointing operation

the number of tuples per checkpoint file and one extra position. The extra position accounts for the window slide.

Figure 6 shows an example of a count-based 15-tuple sliding window. In this example, we divide the window in groups of 5 tuples. This results in 4 checkpoint groups. At checkpoint time $t1$, the window contains 13 new tuples (A–M). G1 and G2 contain 5 tuples each, while G3 contains 3. The checkpoint routine evaluates the circular buffer structure and finds that G1-G3 are dirty. At this point, it serializes the tuples based on the indexes maintained by the circular buffer. Like in UDOPs, no tuple processing is allowed during serialization. After the serialization, the checkpoint routine cleans the dirty bits in the circular buffer and saves the tuple contents to disk.

At time $t2$, the window has 5 new tuples. G3 has tuples N–O and G4 has tuples P–R. G1 lost 3 tuples (A–C) due to the window slide. The checkpoint method checks that G3 and G4 are dirty, and serialize *all* their contents to disk (K–R). Even though G1 lost tuples, its corresponding file is not updated. This file is invalid only after G1 loses all its tuples. The checkpoint thread reuses this file after the window slides by a whole group size.

Restore Algorithm

To recover the operator state, we read all files related to a window. Since we use a circular buffer, the first valid tuple of the window can be in any group. We start to rebuild the window from the file that has the oldest write timestamp. The oldest file corresponds to the window segment that was not updated for the longest time; hence it is the beginning of the sliding window. Restoring all the tuples from the oldest file may result in a window bigger than its maximum size. Thus, we discard all tuples that exceed the total size of the window.

As mentioned in Section 4, another factor we have to consider during state restore operations is stale data. If the operator had not crashed, some of the tuples from the beginning of the window would have been discarded. We eliminate the stale data by estimating how many tuples would have been discarded during normal operation. The estimate is used to remove the top N tuples from the window. We compute the number of stale tuples by the following formula:

$$N_{tuples} = (T_{recovery} - T_{serialization}) * N_{tuples/sec}$$

$T_{recovery}$ is the time after the de-serialization routine completes. The time of operator state serialization ($T_{serialization}$) and the number of tuples per second ($N_{tuples/sec}$) are retrieved from the checkpoint file. Both data are obtained at runtime and are serialized with the tuples to the checkpoint file. Note that we calculate an *approximation* of the number of stale tuples, since there may be a variance on the input tuple rate.

Our recovery routine also handles corrupted checkpoint files. As the operator state is divided in different files, even if one of the files is corrupted by a disk failure, we can still recover the operator state. The implication of a corrupted file is the loss of, at most, the same number of tuples contained in a checkpoint group.

Code generation

To enable checkpointing, the SPADE compiler generated modified join code. The join operator code has two main methods – one for each input port. These methods are changed to include a mutual exclusion variable and a counter of new tuples per window. This is the only code added to the operator critical path. For *per-group*[1] join operators [2] – where the operator allocates a new sliding window depending on the tuple key attribute content – we add code to dynamically allocate our circular buffer data structure and the new tuple counter.

The new counter helps to estimate the operator input rate and to update the indices contained in the circular buffer. Once the checkpoint method runs, it slides the circular buffer data structure by the number of new tuples. The circular buffer code is generic and does not need to be specialized for each join configuration. The only parameters it needs are the window size and the number of tuples each checkpoint group has. This number can be estimated based on the tuple size and operator input rate and can be learned during the operator profiling phase [10].

For incremental checkpointing, the checkpoint method interface changes. Since we need independent serializable buffers, we add extra checkpoint methods at the operator interface. The checkpoint thread invokes the correct checkpoint/restore method according to the operator type.

For per-group join operators we automatically generate a specialized class that associates serialization buffers and per-group keys. Since the key type is dependent on the tuple type, which is defined at the language level, the checkpoint thread does not know the key type. This specialized class abstracts the serialization buffer key-based access to the checkpoint thread. In Section 5.2 we show the overhead imposed by our technique both for join operators with one sliding window and for per-group joins.

5 Experimental evaluation

To evaluate our technique, we conducted two different experiments. The first employs checkpointing in a UDOP that is part of a real-world manufacturing application. The

[1] A per-group operator emulates the existence of several/disjoint instances of the operator processing tuples associated with one logical group. For example, joining stock market transactions related to IBM independently from transactions related to Google. Note that, in this example, the per group key attribute is the company attribute.

application builds statistical models over streaming data. The experiment objective is to evaluate how the application behaves under a UDOP failure by quantifying the crash impact on the accuracy of the application output. The second experiment evaluates the performance overhead of our checkpoint technique for join operators. We devised synthetic applications with different join configurations and evaluate how different checkpoint parameters impact the operator performance.

5.1 Application Output Deviation

To quantify the impact of our checkpoint technique on the application output, we used an application called FAB [23]. FAB generates two statistical models from several sensors embedded in semiconductor manufacturing tools used in IBM's chip manufacturing facilities. The models are built to predict the wafer yield from the input sensors. One of FAB's outputs is a *quality metric* (QM), which compares the value predicted by the statistical model and the ground truth, i.e., the actual wafer yield metric. For our runs, the actual wafer yield was collected along with sensor reading from the real manufacturing environment during the production of 9000 wafers.

For this experiment, we use FAB's QM to quantify how our checkpoint technique performs when facing crashes in the operator that creates and maintains the incrementally built classification model. This operator is implemented as a UDOP, which generates model parameters based on information accumulated during runtime. If the UDOP crashes and no checkpointing is implemented, it loses all its collected information. Therefore, after a UDOP restore, it rebuilds its model parameters from scratch, losing valuable historic data previously used to fine-tune the classification model. When checkpointing is in place, all the UDOP state variables are maintained. After recovery, the UDOP produces model parameters from the same state it had before the crash. For this application, we do not buffer tuples while the operator is recovering. If there is any input tuple sent to the UDOP while it is offline, the tuple is discarded. Note that while there is the potential for data loss, it does not critically affect the accuracy of the classification model.

FAB has 79 operators. For the purpose of this experiment, we run one operator per PE. If we inject a crash fault in the statistical model operator, all the other 78 operators will continue to run. The PEs are distributed across 10 nodes, each with 4 Intel Xeon 3 GHz processors. The average runtime of the application is 30 minutes. However, we consider that one experiment run is complete only when all the sensor inputs available have been processed. To quantify the output deviation of the application, we ran FAB in the following scenarios: (i) FAB *without* UDOP checkpoint and *no* PE crash (S1); (ii) FAB *with* UDOP checkpoint at every 1 second and PE crash (S2); (iii) FAB *without* UDOP checkpoint and PE crash (S3).

We ran each scenario 40 times. For S1, all runs produced the same predicted wafer yield. Even though FAB contains non-deterministic operators (e.g., time-based operators), they did not affect the QM output during our runs. We used this output as the baseline to compute output deviation under the PE crash scenarios.

Since the time of the failure can impact the output deviation due to the amount of accumulated state in the classification model, each run has a different crash time. For S2, we randomly pick a time between the beginning of the application and the average application runtime (30 minutes). At that time, we inject a fault in the statistical model PE via a *kill* command. We restore the PE after 2 to 5 seconds, which is the estimated failure detection time in System S. Once all input is processed, we finish the application and collect all its results (e.g., predicted wafer yield) and injection data (e.g., crash time). S3 employs the same fault injection parameters as S2, so we can compare outputs produced with equivalent failure times.

S1 produces 107 wafer yield predictions and outputs the aforementioned QM for each one of them. We compare each QM produced by the run without PE crash and the run with PE crash. The difference between the two QMs is called *prediction error*. To evaluate the overall output deviation between failed runs and the golden run we computed the *root mean squared error* (RMSE) of the prediction errors. The RMSE shows how far from the correct answer the failed runs are on average. Note that we compare wafer yield prediction errors starting from the failure point onward. Every prediction produced before the failure is discarded from the RMSE computation.

The RMSE for S2 is 4.80, while for S3 is 7.79. Our experiments show that the prediction error distribution for the runs without checkpoint have longer tails than the runs with checkpointing. This means that runs without checkpoint produced QM samples very distant from the correct value. This can be seen in Figure 7, which shows the *cumulative distribution function* (CDF) for the prediction errors in one of our runs. The prediction errors are in log scale. For this run, the checkpoint scenario generated results where the predicted value was at most 6 points away from the correct value. On the other hand, the checkpoint free scenario produced predictions that reached up to 52 points away from the correct value.

The performance overhead for checkpointing FAB's statistical model operator is negligible, since it did not change the tuple processing rate. It also did not affect the results produced by the non-deterministic operators downstream. We observed that by checking that all the measured QMs before operator crash were identical to the ones produced by the checkpoint-free run.

5.2 Checkpoint Performance Overhead

To quantify the performance overhead of our checkpointing technique for join operators, we devised two synthetic SPADE applications. Each application contains four operators. Two of them are source operators, which send data to a join operator. The join operator correlates the data and sends the output to a sink operator. The sink writes the join output to a file. The objective is to stress the join operator and evaluate the checkpoint technique under high data loads.

The two applications differ by their join window con-

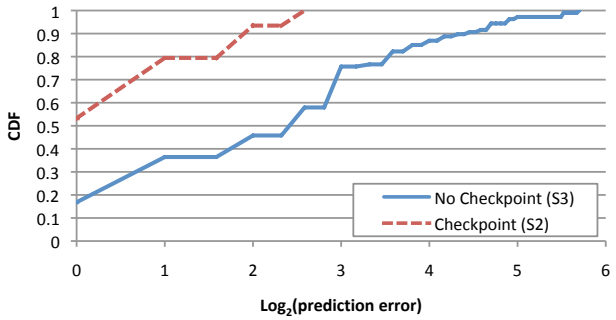

Figure 7: CDF of prediction error

figuration. They both correlate *stream 1* (S1) with *stream 2* (S2), however, the second application uses the *per-group* window modifier for S1. The attributes in S1 are two integers and one 70-byte string. S2 contains one integer and one 10-byte string. The join matches the streams when the integer of S1 is equal to the one in S2. The join output stream contains all attributes from S1 plus the string attribute from S2.

The operators are hosted by 4 PEs, which are placed in 4 different nodes interconnected by Gigabit Ethernet. Each node runs Linux on 4 Intel Xeon 3GHz processors. Both source and sink operators read/write data to their local hard disk. The join operator saves its checkpoint files in a shared file system, in this case NFS. The NFS server is shared with approximately another 200 nodes.

Single-window Join

For the first application, we evaluate the checkpoint performance varying the window size and the group size. The window of S1 is parameterized with the following sizes: 8192, 16384, 32768, 65536 and 131072. The window of S2 is fixed with size 0. Tuples from S2 are compared against S1 to generate an output tuple, but the operator does not maintain S2 tuples in its internal window buffers. Therefore, the checkpoint routine serializes only tuples from the first window. The statistical distribution for both data sources is uniform. We evaluated the checkpoint overhead for this scenario under two checkpoint frequencies: 1 and 10 seconds. To assess how the window group size impacts the performance, we divided the window of S1 in 4 different group sizes. Each window size was divided in the following number of tuples/group: 512, 1024, 2048 and 4096. We ran each configuration 30 times, where each execution lasted 200 seconds.

The checkpoint overhead is compared to the operator performance when running without checkpoint (IO rate with checkpoint / IO rate without checkpoint). Each window configuration has a different Input/Output (I/O) rate, which is shown in Table 1. The input rate is the sum of the inputs from S1 and S2. In all configurations, the input rate is evenly distributed (approximately 50% for each input stream). The rate is in tuples/second. Note that The input rate decreases as the size of the window increases. Because the operator has a greater number of tuple comparisons to make, it cannot process as many tuples as it could in the case of smaller windows. This creates back pressure on the output ports of the source operators, leading to lower input rates.

Window Size	Input Rate	Output Rate
8192	9350.94	25451.04
16384	5054.39	27180.69
32768	2503.25	25643.36
65536	970.19	14099.27
131071	889.64	13062.59

Table 1: I/O rates for single-window join in tuples/sec

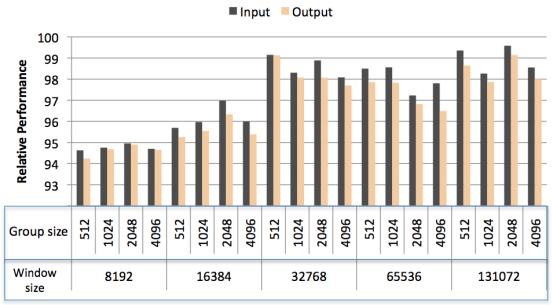

Figure 8: I/O relative performance for single-window join with checkpoint interval of 1 second

Figure 8 shows how much the checkpoint operation at every 1 second affects the I/O rate of the join operator. When the window size is 8192, the checkpoint overhead is greater than 5% in all configurations. As Table 1 shows, the input rate for this operator is 9350.94 tuples/sec. This means that the number of new tuples is more than half of the window, implying that, most of the time, we have to save the full window. Since there is few common state between checkpoint intervals, our technique does not have good performance. As the window size increases, we can see the benefit of our algorithm. For a window of size 32768, the configuration with 512 tuples/group shows less than 1% performance overhead. A reason for this scenario to have better performance than configurations with bigger group size is that a group of 512 tuples results in less data re-serialization. Note that as the number of groups increases, the overhead of managing more files increases as well. However, this overhead did not affect the performance here, since it had the least performance penalty.

Figure 9 shows the I/O rates for a checkpoint interval of 10 seconds. The I/O rates are higher compared to the 1 second scenario, since the checkpointing routine spends less time in serialization. When the checkpoint interval is higher, the performance is better for bigger group sizes. As the checkpoint interval increases, it is expected that a configuration with higher number of tuples/group performs better, since it needs to manage a number of files.

Per-group Join

The second application uses the join operator with the *per-group* modifier for S1. The window has size 1024 tuples for S1 and 0 for S2. The *per-group* modifier generates one window of size 1024 based on a key attribute of the stream. In this case, the attribute is one of the integers of S1. For this experiment, the stream data has 50 different integers for S1 and 600 different integers for S2 uniformly

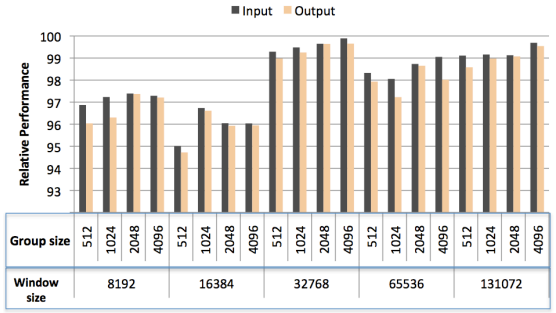

Figure 9: I/O relative performance for single-window join with checkpoint interval of 10 seconds

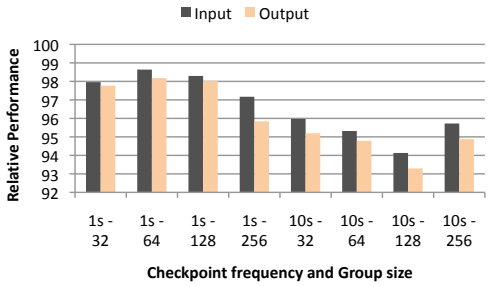

Figure 10: I/O relative performance for per-group join

distributed. This generates 50 different windows of size 1024. Therefore, at every checkpoint interval, we have to serialize new tuples in 50 different windows. Here, each window is divided in 4 different groups sizes. The numbers of tuples/group are 32, 64, 128 and 256. The checkpoint intervals are 1 and 10 seconds. We ran each configuration 30 times, where each run lasted 200 seconds.

Figure 10 shows the I/O relative performance for the per-group join. The averages for input and output for the configuration without checkpoint are 2026.52 tuples/sec and 75856.67 tuples/sec, respectively. When the checkpoint interval is 1 sec, the group size of 64 is the configuration with lower performance impact. The number of input tuples/sec decreases by 1.37%, while the output rate decreases by 1.83%. When the checkpoint interval is 10 secs, the configuration with better I/O performance is with a group size of 32. However, our data shows that the checkpoint thread is able to start a serialization, in average, at every 10.5 secs. Not meeting the checkpoint interval deadline results in a higher loss of tuples during recovery, since there will be more stale data on the files. When the group size is 256, the performance impact is higher, but it gets the operator state at every 10.09 secs. In general, the performance impact is higher when the join operator has the *per-group* modifier. A per-group join has more checkpointing data structures to maintain and more files to save; hence, the serialization phase takes longer.

We conducted experiments to evaluate the *per-group* join checkpointing when the key attribute of the source data follows a Zipf distribution. Word frequency is an example of distribution that follows Zipf's law [17]. In our scenario, this means that there are windows that have a greater number of new tuples than others. Our data shows that the I/O performance impact is lower than the *per-group* join under uniform distribution (2-3% overhead). However, the checkpoint thread is not able to meet the checkpoint deadlines. Instead of serializing the operator state at every 1 sec, it serializes it at every 6 secs in average. This happens because a lot of data must be rewritten to disk for windows with low tuple insertion rate. For example, we have to re-save a whole chunk of data even when there is only one new tuple in the group. Our results suggest that, when the stream data does not follow a uniform distribution, we should divide the window corresponding to different keys with different group sizes.

6 Related Work

Fault tolerance in stream processing has been a very active research area over the last few years. Many techniques add fault tolerance by changing the communication substrate. We argue that this slows down the maximum throughput of the system and should only be applied in selected parts of the application. Balazinska et al. [3] proposes a protocol called DPC. When a failure occurs, stream operators may produce tentative tuples that need to be corrected later on. The developer has to know how to correct results that were produced with tentative tuples. Our aim is to abstract out the fault tolerance from the user by providing a language level abstraction. Additionally, DPC may require operator checkpointing to recover from a failure. Our techniques can complement DPC.

Another set of fault tolerance techniques are based on operator replication. Passive standby, active standby [12] and process-pairs [21] were adapted to the streaming context. Hwang et al. [12] describes *upstream backup*, which enforces tuple backup in upstream operators. Tuples are replayed in case of failure. This technique also changes the communication substrate. Cai et al. [6] proposes a hybrid replication based technique. Replicas are brought up-to-date via checkpoints of the active replica.

LSS [25] is a checkpoint technique closer to our approach. The authors assume that data loss is acceptable during the failure/recovery process of a streaming application. LSS provides an API that should be embedded in the operator code right after a tuple is processed. We automate this process by using SPADE'S code generation framework to output specific checkpoint methods based on the operator type. Hwang et al. [13] proposes *delta checkpoints* for both aggregate and join operators. Our work differs by proposing different checkpoint techniques depending on the operator failure semantic. SGuard [15] employs memory management middleware to track application-level memory pages. It uses copy-on-write to perform asynchronous checkpoints. Our technique lets the user choose which operators should apply a checkpointing scheme, decreasing the overall performance overhead. Our incremental checkpoint technique also handles corrupted checkpoint files without requiring a replicated file system like SGuard does.

There are works that aim to provide application fault tolerance at the language level [7, 20]. Szentiványi et al [22] uses aspect-oriented programming features for building fault-tolerant applications. CATCH [16] is a compiler-

based approach to transparently checkpoint applications. CATCH is a process level technique and does not apply to our concept of operator-based checkpointing. Bronevetsky et al [5] introduces a pre-compiler that instruments MPI programs for automatic checkpointing. This approach aims at parallel applications, which have a different behavior than streaming applications.

7 Conclusions

Large-scale stream processing is becoming a paradigm for developing long-running applications that will monitor, control, and extract knowledge from the critical infrastructure in many operational areas. From applications in manufacturing, responsible for monitoring and acting on sophisticated tools and fabrication processes [23], to business domains such as algorithmic trading [2], to surveillance and fraud detection systems, personal healthcare and public health systems. These few examples demonstrate how critical it is to develop mechanisms to ensure that these applications and the components that make up the middleware supporting them have the means to stay up and operating continuously, even in the presence of software and hardware failures. In this paper, we have made two contributions towards this goal. First, we have shown how language primitives coupled with code generation can provide a flexible mechanism for specifying well-targeted state checkpointing to large-scale applications. Second, we have shown how incremental checkpointing can be carried out for a stream join operator, which is a fundamental building block in applications that carry out data correlation on live streams.

We are now investigating and prototyping a mechanism for specifying high-availability segments in stream processing applications. The aim is to provide a *targeted* way of designating portions of the application as critical and, therefore, express the need to replicate them. We are also looking into experimentally assess the consequences of Byzantine failures in stream operators.

References

[1] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *Workshop on Data Mining Standards, Services and Platforms, DM-SSP*, Philadelphia, PA, 2006.

[2] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. Scale-up strategies for processing high-rate data streams in System S. In *Int. Conf. on Data Engineering (ICDE 2009)*, 2009.

[3] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1):1–44, 2008.

[4] N. Bansal, R. Bhagwan, N. Jain, Y. Park, D. Turaga, and C. Venkatramani. Towards optimal resource allocation in partial-fault tolerant applications. In *INFOCOM 2008*, pages 1319–1327, April 2008.

[5] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *PPoPP 2003*, pages 84–94, San Diego, CA, 2003.

[6] Z. Cai, V. Kumar, B. F. Cooper, G. Eisenhauer, K. Schwan, and R. E. Strom. Utility-driven proactive management of availability in enterprise-scale information flows. In *Middleware*, pages 382–403, 2006.

[7] R. Cmelik, N. Gehani, and W. Roome. Fault-tolerant concurrent C: a tool for writing fault tolerant distributed programs. In *FTCS-18*, pages 56–61, Jun 1988.

[8] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, 1991.

[9] E. N. Elnozahy and M.-J. S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, 2004.

[10] B. Gedik, H. Andrade, and K.-L. Wu. A code generation approach to optimizing high-performance distributed data stream processing. Under submission, 2009.

[11] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. In *ACM SIGMOD International Conference on Management of Data*, pages 1123–1134, New York, NY, 2008.

[12] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *21st Int. Conference on Data Engineering*, Tokyo, Japan, 2005.

[13] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE 2007*, pages 176–185, Istanbul, Turkey, April 2007.

[14] G. Jacques-Silva, J. Challenger, L. Degenaro, J. Giles, and R. Wagle. Self healing in System S. *Cluster Computing*, 11(3):247–257, 2008.

[15] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant Stream Processing Using a Distributed, Replicated File System. In *VLDB 2008*, Auckland, New Zeland, 2008.

[16] C.-C. J. Li and K. W. Fuchs. CATCH - compiler-assisted techniques for checkpointing. In *FTCS-20*, pages 74–81, Jun 1990.

[17] W. Li. Random texts exhibit Zipf's-law-like word frequency distribution. *IEEE Trans. on Information Theory*, 38(6):1842–1845, Nov 1992.

[18] Object Management Group. Corba fault tolerant chapter, v3.03. Technical Report 04-03-21, OMG, 2004.

[19] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *USENIX Winter*, pages 213–224, 1995.

[20] R. D. Schlichting and V. T. Thomas. Programming language support for writing fault-tolerant distributed software. *IEEE Trans. Computers*, 44(2):203–212, 1995.

[21] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *ACM SIGMOD Int. Conf. on Management of data*, pages 827–838, Paris, France, 2004.

[22] D. Szentiványi and S. Nadjm-Tehrani. Aspects for improvement of performance in fault-tolerant software. *Pacific Rim International Symposium on Dependable Computing, IEEE*, 0:283–291, 2004.

[23] D. S. Turaga, O. Verscheure, J. Wong, L. Amini, G. Yocum, E. Begle, and B. Pfeifer. Online FDC control limit tuning with yield prediction using incremental decision tree learning. In *Sematech AEC/APC*, 2007.

[24] O. Verscheure, M. Vlachos, A. Anagnostopoulos, P. Frossard, E. Bouillet, and P. S. Yu. Finding "who is talking to whom" in VoIP networks via progressive stream clustering. In *ICDM'06*, pages 667–677, Washington, DC, 2006.

[25] Q. Zhu, L. Chen, and G. Agrawal. Supporting fault-tolerance in streaming Grid applications. In *IPDPS 2008*, pages 1–12, Miami, FL, April 2008.