Aggregate Update Problem for Multi-clocked Dataflow Languages

Hannes Kallwies, Martin Leucker, Torben Scheffel, Malte Schmitz, Daniel Thoma
Institute for Software Engineering and Programming Languages, University of Lübeck, Lübeck, Germany
{kallwies, leucker, scheffel, schmitz, thoma}@isp.uni-luebeck.de

***Abstract*—Dataflow languages have, as well as functional languages, immutable semantics, which is often implemented by copying values. A common compiler optimization known from functional languages involves analyzing which data structures can be modified in-place instead of copying them. This paper presents a novel algorithm to this so called *Aggregate Update Problem* for multi-clocked dataflow languages, i.e. those that allow streams to have events at disjoint timestamps, like e.g. Lucid, Lustre and Signal. Unrestricted multi-clocked languages require a static triggering analysis on how events and hence data values are read, written and replicated. We use TeSSLa as a generic stream transformation language with a small set of operators to develop our ideas. We implemented the solution in a TeSSLa compiler targeting the Java VM via Scala code generation which combines persistent data structures and mutable data structures for those data values which allow in-place editing. Our empirical evaluation shows considerable speedup for use cases where queues, maps or sets are dominant data structures.**

I. Introduction

Dataflow programming [1], [2] is a programming paradigm that is gaining popularity for concurrent and distributed applications, e.g. log data analysis and runtime verification. It aims at describing transformations from input to output data streams. In dataflow languages one can, based on a set of input streams, iteratively define new streams by applying basic operations on other defined streams. Streams consist of events, i.e. values at a certain timestamp. Operations on streams can be simple function applications, e.g. for adding the current events of two other streams, or an operator for retrieving the previous event of another stream, which could for example be used for aggregating data of a stream over time. During the execution of a dataflow program, the events of the input streams arrive successively and the events of the output streams are then one by one calculated according to the operators by which they were defined. Events may be of simple types like booleans or numbers but also carry more complex data structures such as sets or arrays.

Dataflow (and likewise functional) languages usually have so called immutable semantics. This implies the execution of functions is free of side effects: By modifying a value an altered copy of the original value is created with the old one preserved, i.e. the old value can be read or written again. This copying is especially critical for aggregate data structures (like arrays) as copying them is costly.

This work was funded in part by the projects ReNuBiL (https://renubil.de) and DOING eBus.

Consider the example in Figure 1 using sets to aggregate input events over time. (The example is given in TeSSLa [3] which is used as a generic stream transformation language in this paper. A detailed introduction follows in section II.)

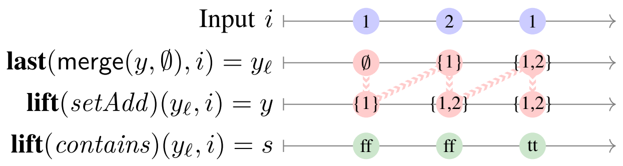

Input i
$\textbf{last}(\mathsf{merge}(y, \emptyset), i) = y_\ell$
$\textbf{lift}(setAdd)(y_\ell, i) = y$
$\textbf{lift}(contains)(y_\ell, i) = s$
1
2
1
∅
{1}
{1,2}
{1}
{1,2}
{1,2}
ff
ff
tt

Fig. 1. Simple TeSSLa specification with complex data structures.

Think of i to be an input stream of numbers. y is defined as the stream resulting from adding (function *setAdd*) the current value of i to y_ℓ's current event. y_ℓ is the stream reproducing the last event of y (or the empty set in case y had no previous event) always when a new event on stream i arrives. Finally s is the stream containing *true* or *false* dependent on whether the number on i is already contained in the set from y_ℓ or not (function *contains*).

In summary, this example aggregates the input events in a set and checks whether the current event has already appeared in the past. Semantically, every time a new input on stream i occurs, the set on stream y_ℓ is copied and modified. The original set on stream y_ℓ remains unchanged and is used again for calculation of stream s - and a straight-forward implementation would do so as well.

However, it is not efficient to implement a dataflow language by copying a data structure every time it is written to avoid side effects on the original data structure. Sometimes the old structure is not needed anymore after the modification. If this is the case the updates could directly be performed in-place on the original data structure instead of copying. Such a kind of update is then called *destructive update*, because the old value of the data structure is destroyed with the modification.

In the example from Figure 1 one could first calculate s and then update the set from y_ℓ destructively (i.e. in-place) to receive y's event.

Yet performing updates destructively is not always possible. The general problem of the efficient implementation of updates on complex data structures is known under the term *Aggregate Update Problem* [4]. Several solutions to this problem exist, mainly from the field of functional languages. Principally the approaches can be categorized into the following three groups [5]:

1) The first group of solutions aims at using a type system for ensuring data structures are only used once so they can be modified in-place. The fundamental approach is the usage of linear types [6]. In their basic form they are quite restrictive, as it is for example forbidden to read a given variable twice, but there are certain relaxations, which enforce the existence of a single reference exclusively at the write.

2) The second approach deals with the problem at runtime by using persistent data structures [7] which store only a reference to a base data structure and the difference. However this approach typically requires some kind of garbage collection for the occurring orphaned objects. Even though these data structures show quite good results they are still inferior to in-place updates in most cases.

3) The third method is to perform a static code analysis to determine when data structures can be modified in-place. In functional as well as in dataflow languages it often depends on the calculation order of certain variables or streams if an update can be made in-place (see example in Figure 1). First approaches in the field of functional languages assume a fixed scheduling of the operations [8], [9], [10]. In [11] the benefit of finding an optimizing schedule was described.

With approach 1) the programmer explicitly states which variables should be implemented destructively. Note that in stream languages (like TeSSLa) considerations about mutability are always a matter of the schedule, which was chosen by the compiler. This means the user has to think over all possible schedules to decide if it is possible to choose a mutable type for a certain variable. In this paper we show how a combination of approaches 2) and 3) can be applied to dataflow language implementations. The presented solution aims at maximizing the number of mutable data structures without manual intervention by the user.

In the field of dataflow languages one can distinguish between *single-* or *mono-clocked* and *multi-clocked* ones [12], [13]. While in single-clocked transformations every stream has an event at every calculation step (imagine a single global clock), in the multi-clocked case only some streams may have an event at a certain calculation step (every stream has its own clock indicating its events).

Working with multi-clocked stream transformations, it is often desirable to access the last event of a certain stream if it has no event at the current timestamp (event reproduction). Most dataflow languages that support multi-clocked streams bring operators for this use case (e.g. Lustre's **current** or TeSSLa's **last**). However previous work on the Aggregate Update Problem for dataflow languages did not handle such an operator or was completely restricted to single-clocked streams [14], [15], [5].

In this paper we provide an algorithm for the Aggregate Update Problem. We implement our approach for the TeSSLa language, mainly to explain the development. Since TeSSLa is able to express every future-independent multi-clocked stream transformation function [3], including those reproducing events, the approach can be adapted to other common data stream languages such as Lustre [16], Signal [17] and similar specification languages such as Lola [18] and Striver [19].

For the development of the algorithm one has to take care that data structures which were updated in-place are not reused at future timestamps. Concerning stream languages this requires a careful analysis of a stream's triggering behavior (i.e. when it has events) which is presented in this paper.

The concrete implementation of our algorithm translates TeSSLa to Scala but the same scheme could also be used for translation to other imperative languages, provided that implementations of persistent data structures are available.

A. Related Work

The idea of translating dataflow languages to imperative code has already been described for languages like Lustre, Signal and similar [16], [20], [21]. The efficient combination of calculation blocks according to their clock behavior is studied in [20]. The immutable (a.k.a. functional) semantics of data structures is a common property of dataflow languages but also of functional languages. The arising problem of how such persistent data structures can be implemented in an efficient way, commonly known as Aggregate Update Problem [4], has hence already been intensively studied for the field of functional languages, but also for dataflow languages to some degree. A good overview over approaches and related topics in the field of functional languages can be found in [11]. In general the three previously enumerated approaches for solving this problem exist and have already partly been adjusted to dataflow languages: [14] applies the idea of static analysis to find out which data structures can be modified in-place to the dataflow language LabView. [15] also presents an approach in this direction, but in difference rejects compilation if a perfect ordering so that every update is possible in-place cannot be found. [5] also deals with the problem of copy avoidance for arrays in a Lustre-like language. In contrast to the other approaches they use a semilinear type system giving the programmer the opportunity to decide which data structures use which memory locations and hence are modified in-place.

B. Contribution

Our approach differs from the ones above in several aspects:

1) It automatically finds the optimal ordering of the stream definitions such that the maximum number of data structures can be implemented in a mutable way, according to our mutability criterion. It does not require the user to define the execution order nor to annotate which variables can be implemented mutable.

2) It does not perform full copies but uses persistent data structures if an in-place update is not always possible.

3) It is able to handle stream transformations which are not restricted to a single clock and allow access to the last event of a stream if it does not have an event at this timestamp. Though in most dataflow languages such transformations can be expressed (e.g. by Lustre's **current** operator) the mentioned publications are not able to detect mutability in connection with such an operator or don't support it at all.

Hence, to the best of our knowledge we provide a more general solution to the Aggregate Update Problem of dataflow languages in the unrestricted case.

II. The TeSSLa Specification Language

In this section we will give a short informal introduction to the TeSSLa language. A full definition of its syntax and semantics can be found in [3]. In contrast to other datastream languages like Lustre, TeSSLa operates on timed event streams. This means there is no fixed grid of timestamps where events can occur but every event has a timestamp attached which can also be used for calculations. A timed event stream can be seen as a function mapping a timestamp from a time domain $\mathbb{T}$ to a value from the stream's data domain $\mathbb{D}$ or $\perp$ if the stream has no event at the timestamp. For the set of all streams over data domain $\mathbb{D}$ we write $\mathcal{S}_{\mathbb{D}}$. A TeSSLa specification defines a relation between input and output streams. Every monotone, continuous and future-independent transformation can be expressed by a TeSSLa specification [3]. A specification φ consists of a set of equations assigning an expression to every output stream name. Valid expressions are names of input or output streams or one of the following six basic operators applied to other expressions:

nil is the empty stream with no events. **unit** is a stream that contains only a single event at timestamp 0 with the unit value $\Box$. **time**(s) is a stream with an event at every timestamp where s has an event, but with the timestamp as value. **lift**$(f)(s_1, \ldots, s_n)$ is a stream which has an event with value $v = f(v_1, \ldots, v_n)$ if $v \neq \perp$ and v_i is the value of s_i at this timestamp or $\perp$ if s_i has no event. $f(\perp, \ldots, \perp) = \perp$ must hold for all lifted functions. **last**(v, r) is a stream which has an event every time r has one with the strictly last value of v. If v is uninitialized (i.e. has no last value) when r has an event the last expression also has no event. **delay**(d, r) is a stream which has an event with the unit value $\Box$ t time units after the last reset (event of r or the stream itself) if no further resets occurred between this event and its reset and t is the value of d at this timestamp.

Recursive definitions in the equation system are only allowed if they go through the first parameter of a **last** or **delay** expression, because the current value of streams defined by one of these expressions is solely dependent on the previous and not the current value of the streams used as their first argument.

We further add the following syntactic sugar: A constant value is implicitly converted to a stream with this constant value at timestamp 0 and no other events. E.g. *true* equals the stream with no events except one with value *true* at timestamp 0.

Using the basic operators we can also derive the utility function merge: $\mathsf{merge}(x, y) := \textbf{lift}(f_{\text{merge}})(x, y)$ with $f_{\text{merge}}(a \neq \perp, b) = a$ and $f_{\text{merge}}(\perp, b) = b$, which combines events from two streams, prioritizing the first stream.

A TeSSLa specification is called flat, if only stream names are used as sub-expressions inside the basic operators. Clearly, every specification can be transformed into a flat one, by introducing fresh identifiers for sub-expressions.

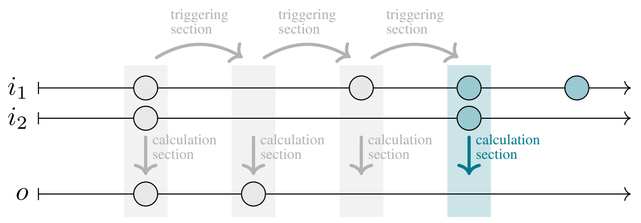


Fig. 2. Basic functionality of the generated monitor program: For each timestamp where any stream bears an event the calculation section is processed.

For the example from Figure 1 the following specification results from desugaring and flattening: $u = \textbf{unit}$, $\emptyset = \textbf{lift}(f_\emptyset)(u)$, $m = \textbf{lift}(f_{\text{merge}})(y, \emptyset)$, $y_l = \textbf{last}(m, i)$, $y = \textbf{lift}(setAdd)(y_l, i)$, and $s = \textbf{lift}(contains)(y_l, i)$ with $f_\emptyset(\Box) = \emptyset, f_\emptyset(\perp) = \perp$.

III. Translation to an Imperative Language

In this section we describe how a TeSSLa specification generally can be translated into an imperative program which successively receives the values of input streams and calculates the values of output streams. We call this kind of program a monitor.

A TeSSLa specification defines a transformation between input and output streams. This transformation is monotone: If a set of input streams is related to a set of output streams, then the extended input streams, i.e. those with the old input streams as prefixes, are also related to extended output streams.

This behavior of TeSSLa specifications allows a monitor to receive the input events of all input streams in a chronological order and calculate the output events successively. Furthermore TeSSLa allows us to calculate the events of output streams at timestamp t, once all input events up to t have arrived, which is called future-independent.

Therefore our generated code consists of two sections. One for calculating the output events at a certain timestamp, the *calculation section*, and one for subsequently calling the calculation section for all timestamps, where any output stream bares an event, we call it the *triggering section*.

As long as input events arrive, the triggering section determines the next timestamp where any output stream may carry an event and calls the calculation section to calculate the output stream values at this timestamp and possibly print them (see Figure 2).

In the first step the TeSSLa specification is flattened, i.e. an equivalent specification is created where every operator does only contain stream variables as sub-expressions. In the rest of this paper we always refer to the flattened specification.

In our monitor program we successively calculate the current value of all streams. Therefore we introduce a variable `v` for every stream v. The type of `v` is the type of v.

Streams defined via a **last** or **delay** expression must store additional information: For **last** expressions the value of the last event of its first argument stream has to be stored, for **delay** expressions the timestamp of the next potential event. Therefore we introduce additional variables:

1) For every stream v used as first argument of a **last** expression we introduce a variable `v_last` which has the type of v and is always carrying the value of the last event of v.

2) For every stream v defined via a **delay** expression we introduce a variable $\texttt{v}_{\texttt{nextTs}}$ containing the next potential timestamp of v. Its domain is $\mathbb{T} \cup \{\bot\}$.

Every time the calculation section of our generated code is called for a timestamp t, these variables are calculated according to t. Every time a new input event arrives, the values of the input stream are set accordingly, as well.

Since the calculation of variables depends on the current value of other streams, i.e. the calculation of other variables, it is necessary to follow a certain order in the determination of the values. Consider again the example from Figure 1. The current value of streams s and y at a certain timestamp t is dependent on the current value of stream y_ℓ. y_ℓ on the other hand is not dependent on the current value of y or s (but on the last value of y). So our monitor has to calculate first the value of variable $\texttt{y}_{\texttt{l}}$ and afterwards `s` and `y`.

For determining the order in which the stream values have to be calculated, we use a graph-representation of the TeSSLa specification, which will also play a role in further steps of the translation.

Definition 1 (TeSSLa Usage Graph)**.** *Let φ be a flat TeSSLa specification. The tuple (G, S), where G is a directed graph $G = (V, E)$ and $S \subseteq E$ a set of special edges, is called* usage graph *of φ, iff*
– *V is the set of all input and output streams of φ,*
– *$(u, v) \in E$, iff u is used in the expression defining v and*
– *$(u, v) \in S$, iff v is defined by a **last** or **delay** expression and u is used as first parameter.*

The TeSSLa usage graph of the example from Figure 1 is depicted in Figure 3. The special edges are drawn in a dashed style. Note that for the graph generation the flattened specification is used.

For the translation of the specification we need an order, where all variables belonging to a certain stream are calculated before they are used in another calculation. However, since a **last** or **delay** expression is unaffected by the first stream's current event, this does not hold for streams which are used as first argument in a **last** or **delay** (special edges). We can define an order, matching these requirements:

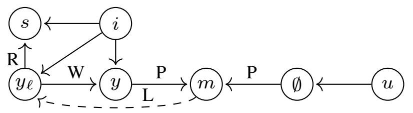


Fig. 3. TeSSLa usage graph of the flattened specification from Figure 1 with edge classification according to Definition 3.

Definition 2 (Translation Order)**.** *Let φ be a flat TeSSLa specification with usage graph $((V, E), S)$. A total order $(V, <)$ is called* translation order *of φ, iff there is no $(u, v) \in E \backslash S$ with $v < u$.*

Such an order can easily be found by determining a topological sorting of the graph. Such an order always exists, since dependency cycles in a TeSSLa specification must, by definition, go through the first argument of a **last** or **delay** expression. However, the order is not necessarily unique.

Now, to generate the monitor program, there is code appended to the triggering and calculation section for each output stream following a previously defined translation order. The concrete code is dependent on the type of expression which is translated.

A. Calculation Section

For the calculation section the translation is straight-forward for most operators. We give examples for **lift**, **last** and **delay** in this section. Besides the already mentioned variables we state `ts` to be the current timestamp for which the calculation section is called. $\bot$ is a special null value for variables. The single operators are translated as follows:

$s = \textbf{lift}(f)(s_1, \ldots, s_n)$ is translated to
if $\bigvee_{\texttt{i}} \texttt{s}_{\texttt{i}} \neq \bot$ **then** $\texttt{s} := \texttt{f}(\texttt{s}_1, \ldots, \texttt{s}_\texttt{n})$

If any of the input streams of a **lift** expression has an event at the current timestamp, the belonging function is calculated on these streams. The result of the function application is used for a new event on stream s. The translation of a **lift** expression also requires a translation of the lifted function f to `f` which is callable from the code and is done as expected.

$s = \textbf{last}(v, t)$ is translated to **if** $\texttt{t} \neq \bot$ **then** $\texttt{s} := \texttt{v}_{\texttt{last}}$

If s is defined via a **last** expression, then stream s has the same value as the last event of v, if t also has an event at the current timestamp. This is done by assigning $\texttt{v}_{\texttt{last}}$ to s if `t` is different from $\bot$.

$s = \textbf{delay}(d, r)$ becomes **if** $\texttt{s}_{\texttt{nextTs}} = \texttt{ts}$ **then** $\texttt{s} := \Box$

For all streams s which are defined by a **delay** expression, the additional variable $\texttt{s}_{\texttt{nextTs}}$ contains the timestamp where s raises the next event; its value is calculated in the triggering section. If the calculation section is passed at this timestamp, s is set accordingly.

By subsequently adding up these code parts following the determined translation order, one receives the calculation section. With it, the current state of all output streams can be determined for a certain timestamp t, provided that all input stream variables represent the state at t and the $\texttt{s}_{\texttt{last}}$ and $\texttt{s}_{\texttt{nextTs}}$ variables are set correctly.

At the end of the calculation all $\texttt{s}_{\texttt{last}}$ variables are set for the next calculation step: **if** $\texttt{s} \neq \bot$ **then** $\texttt{s}_{\texttt{last}} := \texttt{s}$

Finally, for every stream s the variable `s` is reset for the next timestamp: $\texttt{s} := \bot$.

B. Triggering Section

The triggering section is accountable for calling the generated calculation section for every timestamp, where any input or output stream bears an event. In TeSSLa, streams may have events at timestamps, where input streams also have events. Events at other timestamps can exclusively be generated by **delay** expressions. Note that this is a major difference to other dataflow languages, where events may only occur at the ticks of a base clock, so a dedicated triggering section is not needed in their translation.

In detail the triggering section works as follows: As long as new (ordered) input events arrive at the monitor, their timestamp, say t, is recorded and the events are stored to the corresponding input stream variables. If the timestamp, which has just arrived, is not equal but greater than the last one t', the calculation section is first triggered for t'. After that for every **delay** expression in the TeSSLa specification a timestamp t_i, $i \in \{1, \dots, n\}$, is calculated, where the **delay**'s next event would occur, if no previous reset would take place. The calculated values are then stored in the according $\mathtt{x}_{\mathtt{nextTs}}$ variables. If $\min\{t_1, \dots, t_n\} < t$ holds, the calculation section is triggered again for $\min\{t_1, \dots, t_n\}$, since one can be sure there is no further event between t' and $\min\{t_1, \dots, t_n\}$ and thus also no possibility to reset the corresponding **delay**. The procedure is then repeated until $\min\{t_1, \dots, t_n\} \geq t$, then the monitor continues consuming input events. When receiving the end of the input t is set to ∞.

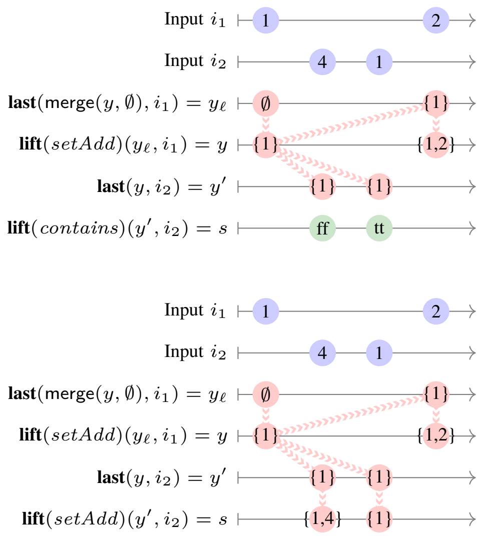
By combining this triggering section with the calculation section a TeSSLa monitor is obtained.

IV. Aggregate Update Optimization

As in the example from Figure 1 streams may sometimes carry non-trivial data structures such as sets, arrays etc. When functions that modify these data structures, such as *setAdd* in the example, are lifted to such streams, the original event on the argument stream remains unchanged. However a new, independent event with the modified data structure arises on the result stream. This behavior is called persistent (or functional) semantics and is common for dataflow languages. To preserve this semantics in the generated imperative program one might copy these data structures before modifying them. As already pointed out this is not necessary in the example from Figure 1 if the calculation of stream s takes place prior to the calculation y, because in this case the modified event from stream y_ℓ is not needed anymore and hence the modification can be done in-place. This indicates that whether or not updates can be done in-place is a matter of the chosen translation order.

Consider also the two examples from Figure 4 which are slight modifications of our previous example. In the upper example we have a second input stream i_2. y accumulates all events from the first input stream i_1 in a set. y' reproduces the last event of y when a new event on i_2 occurs. s finally indicates whether the current event on i_2 is contained in the set from y'. Again all updates in this example can be done in-place because the set from stream y is only modified to create a new event on this stream. The old one is never accessed again after this update, if the calculation of streams y' and s is done in advance. In contrast the update cannot be done in-place in the lower example in Figure 4. There the stream s does not only result from a read access to the set from y but from a modification of it. After adding 4 to the set $\{1\}$ exactly this set is required again in the next timestamp to add 1. This makes a direct modification of the set impossible. It has to be copied.

If every update of a data structure can be done in-place we use mutable data structures in our generated code, otherwise persistent ones. In the following we present our algorithm for



Input i_1
1
2
Input i_2
4
1
last(merge(y, $\emptyset$), i_1) = y_ℓ
$\emptyset$
{1}
lift($setAdd$)(y_ℓ, i_1) = y
{1}
{1,2}
last(y, i_2) = y'
{1}
{1}
lift($contains$)(y', i_2) = s
ff
tt
Input i_1
1
2
Input i_2
4
1
last(merge(y, $\emptyset$), i_1) = y_ℓ
$\emptyset$
{1}
lift($setAdd$)(y_ℓ, i_1) = y
{1}
{1,2}
last(y, i_2) = y'
{1}
{1}
lift($setAdd$)(y', i_2) = s
{1,4}
{1}

Fig. 4. Two TeSSLa specifications with complex data types.

determining which data structures can be implemented in a mutable way and which not. The presentation of the algorithm is structured as follows:

A. The foundation of the algorithm is the specification's usage graph. In the first step we introduce a classification of the graph's edges.

B. For our analysis it is crucial to know which streams may carry the same data structure, therefore we define a notion of aliased stream variables.

C. In order to do this it is necessary to statically know which streams imply events on which other streams. We determine boolean formulas for all streams indicating their triggering behavior to reason about implications of these streams.

D. Based on this we give a criterion which variables may be implemented mutable and which not.

E. Finally we present an algorithm to find the optimal translation order (according to our criterion), i.e. the one with the most variables implementable in a mutable way.

A. Edge Classification

We start with classifying the edges of the usage graph (Definition 1) as follows. Note: we only do that for edges passing complex data types e.g. sets, where we have to decide over mutability. Other edges are not categorized.

Definition 3 (Edge Classes)**.** *Let (u, v) be an edge in the TeSSLa usage graph. If the stream belonging to u has a complex datatype, we call (u, v) a*

- **Write edge**, *if the expression which defines v performs a write access to the current value of stream u.*
- **Read edge**, *if the expression which defines v performs a read access to the current value of stream u.*
- **Last edge**, *if v is defined by a* ***last*** *expression, where u is used as first argument.*
- **Pass edge**, *if the value of stream u may be passed to v unchanged (e.g. by a* merge *expression).*

For an edge (u, v) we write $u \xrightarrow{W} v$, $u \xrightarrow{R} v$, $u \xrightarrow{L} v$, $u \xrightarrow{P} v$ to indicate these edge types.

For combination of edges we use regular expressions. E.g. $u \xrightarrow{W+L} v$ indicates a write or last edge. $u \xrightarrow{P^*L} v$ means a path of arbitrary many pass edges followed by a single last edge and $u \xrightarrow{W^+} v$ a path of arbitrary many write edges but at least one. An example of the edge classification can be found in Figure 3.

B. Aliasing Analysis

The **last** expressions are able to reproduce the same event more than once in the future. This is a problem if the reproduced data structure gets modified after the first reproduction. See for example the lower part of Figure 4. There stream y' reproduces the event from y twice though it is already modified after the first reproduction (and hence cannot be mutable). We call **last** expressions which may reproduce events from their value stream more than once replicating **last**s. In the following we define the triggering behavior of streams, i.e. the set of timestamps where the streams have events and based on this replicating **last**s.

Definition 4 (Triggering Behavior)**.** *Let φ be a specification and $\mathcal{S}$ the set of all streams in φ. For a tuple of concrete input streams $\mathcal{I} \in \mathcal{S}_{\mathbb{D}_1} \times \cdots \times \mathcal{S}_{\mathbb{D}_n}$, the function $ev_{\mathcal{I},\varphi} \colon \mathcal{S} \to 2^{\mathbb{T}}$ yields the set of all timestamps where stream s has an event: $t \in ev_{\mathcal{I},\varphi}(s) \iff s(t) \neq \bot$.*

Definition 5 (Replicating Last)**.** *Stream s from specification φ defined as $s = \textbf{last}(v, t)$ is called* replicating **last**, *iff $\exists_{\mathcal{I}} \colon ev_{\mathcal{I},\varphi}(s) \not\subseteq ev_{\mathcal{I},\varphi}(v)$.*

That means we define a **last** as replicating if it may produce an event without a new event arriving at its value stream. Based on this classification we define a notion of aliasing variables. First, two variables are considered as aliasing-safe if we can prove they never carry the same event at the same timestamp:

Definition 6 (Aliasing-Safe Variables)**.** *Two variables u, v from a specification φ are* aliasing-safe*, $u \not\simeq v$, if for all common ancestors c and all paths p_u and p_v from c to u and v (either p_u from c to u and p_v from c to v or p_u from c to v and p_v from c to u) in the usage graph we can find*
$p_u = c \xrightarrow{(P^*L)^+} u_1 \xrightarrow{(P^*L)^+} \dots \xrightarrow{(P^*L)^+} u_n \xrightarrow{(P^*L)^+} u$ *and*
$p_v = c \xrightarrow{P^*L} v_1 \xrightarrow{P^*L} \dots \xrightarrow{P^*L} v_n \xrightarrow{P^*} v$ *such that*
– $\forall_{i \in \{1 \dots n\}} \colon \forall_{\mathcal{I}} \colon ev_{\mathcal{I},\varphi}(u_i) \subseteq ev_{\mathcal{I},\varphi}(v_i)$ *and*
– *all* ***last****s from p_v are non-replicating.*

According to this definition, two variables u and v are aliasing-safe if they either have no common ancestor or the paths from their common ancestor are of different lengths (counting the last nodes). If they have no common ancestor, they can never carry the same event. If u and v have a common ancestor c, but the path from c to u contains at least one more last node, we can be sure the event from c always reaches u at a later timestamp than v. For this consideration of course we also have to make sure the triggering behavior of the last nodes

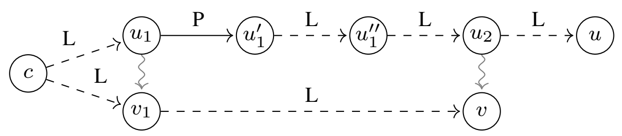


Fig. 5. TeSSLa usage graph where v and u are aliasing safe. $x \rightsquigarrow y$ indicates $\forall_{\mathcal{I}} : ev_{\mathcal{I},\varphi}(x) \subseteq ev_{\mathcal{I},\varphi}(y)$

from the two paths is not unrelated. n last nodes on the longer path must imply triggering of the corresponding nodes on the shorter path to make sure the events on the longer path cannot outpace the ones on the shorter path. Therefore we check that the set of timestamps when u_i has an event ($ev_{\mathcal{I},\varphi}(u_i)$) is a subset of the timestamp set for v_i ($ev_{\mathcal{I},\varphi}(v_i)$) for all possible inputs, $ev_{\mathcal{I},\varphi}(u_i) \subseteq ev_{\mathcal{I},\varphi}(v_i)$.

See for example the usage graph from Figure 5. If the event from c gets reproduced in u_1 it is also reproduced in v_1 because of the triggering implication: If u_1 has an event then also v_1. The event from u_1 could then be handed through to u''_1 without an event in v. However if it is further passed to u_2 the very same element must also be reproduced in $v_2 = v$, again because of the triggering implication. This means if an event from c finally reaches u it has already been in v at a previous timestamp. Because all the **last**s on the path to v were non-replicating this event may also not be reproduced again and so v and u can never carry the same event at the same timestamp. They are aliasing-safe.

If we cannot prove variables u, v to be aliasing-safe, we consider them as potential aliases $u \simeq v$.

C. Static Approximation of the Aliasing Analysis

Determining $ev_{\mathcal{I},\varphi}(v)$, i.e. the set of timestamps where a certain stream has events is of course highly dependent on the input data. Even reasonings about input-independent containment of such sets are a semantic property of a TeSSLa specification and the lifted functions and hence undecidable. Thus for deciding which variables are potential aliases we use a relaxation of this rule: We utilize an algorithm to determine an over-approximation by calculating formulas $ev'_\varphi(u), ev'_\varphi(v)$, s.t. $ev'_\varphi(u) \to ev'_\varphi(v) \in \text{TAUT} \Rightarrow \forall_{\mathcal{I}} \colon ev_{\mathcal{I},\varphi}(u) \backslash \{0\} \subseteq ev_{\mathcal{I},\varphi}(v)$. In the final algorithm this approximation may cause some variables to be implemented with persistent data structures while mutable ones would be possible. This approximation is valid because usage of persistent data structures always leads to a correct, although possibly slower, implementation. We exclude 0 from the left set since we check the containment exclusively for **last** streams on the left side, which never have events at timestamp 0.

For the set of all stream names V the approximation $ev'_\varphi \colon V \to \mathbb{B}^+(V)$ maps a stream name to a positive boolean formula of stream names, i.e. a formula with conjunctions and disjunctions of sub-formulas but without negations. This formula expresses when stream v has an event in relation to events on input streams and other streams where we are not able to determine an exact formula describing when it has events.

$ev'_\varphi(s)$ for a stream $s \in V$ is determined as follows:

- $ev'_\varphi(s) = \mathit{false}$ if s is defined as **nil**
- $ev'_\varphi(s) = ev'_\varphi(x)$ if s is defined as **time**(x)
- $ev'_\varphi(s) = ev'_\varphi(x_1) \wedge \dots \wedge ev'_\varphi(x_n)$ if s is defined as **lift**$(f)(x_1, \dots, x_n)$ and $f(v_1, \dots, v_n) \neq \bot \iff \forall v_i \colon v_i \neq \bot$
- $ev'_\varphi(s) = ev'_\varphi(x_1) \vee \dots \vee ev'_\varphi(x_n)$ if s is defined as **lift**$(f)(x_1, \dots, x_n)$ and $f(v_1, \dots, v_n) \neq \bot \iff \exists v_i \colon v_i \neq \bot$
- $ev'_\varphi(s) = ev'_\varphi(y)$ if s is defined as **last**(x, y) and x is always initialized.
- $ev'_\varphi(s) = s$ in all other cases

nil never has an event, hence its corresponding formula is false. **time**(y) and **last**(x, y) always have an event, iff y has one, though for **last** this only holds if we can be sure that the stream x is always initialized, which is the standard case. It can easily be checked by a simple graph analysis where it is tested if every value parameter of a **last** node has a direct connection to a **unit** node without a filtering operation in between.

For the **lift** we distinguish two cases:

1) An event is produced iff all of the inputs have an event. This is the case for all basic operators $(+, *, ...)$.

2) An event is produced iff any of the inputs has an event. This is for example the case for f_{merge} from Section 2.

Most built-in functions correspond to one of these groups and the correspondence is known a-priori. However the accuracy of the approximation could be further extended by determining general boolean formulas for arbitrary functions.

All other expressions, including input streams, **delay**s etc., are not expressed as formulas of other streams but build atoms of the formulas (see the last rule).

To check if an event on stream u implies an event on stream v one can check if $ev'_\varphi(u) \to ev'_\varphi(v)$ is a tautology. In that case v always has an event if u has one. The other direction does not hold, this may lead to missed optimizations but not to faulty behavior of the monitor, as we pointed out above.

Applying the function ev'_φ on the streams y_ℓ and m from the example in Figure 1 (after flattening) we get $ev'_\varphi(y_\ell) = ev'_\varphi(i) = i$. And for m: $ev'_\varphi(m) = ev'_\varphi(y) \vee ev'_\varphi(\emptyset) = (ev'_\varphi(y_\ell) \wedge i) \vee ev'_\varphi(u) = (i \wedge i) \vee u$. The formula $i \to (i \wedge i) \vee u$ is a tautology, which means every time y_ℓ has an event the same holds for m and so y_ℓ is a non-replicating **last**.

D. Mutability Criterion

Based on the notion of potentially aliasing variables we now define a *mutability set* M_φ for a flat TeSSLa specification φ. Such a set contains all variables which may be implemented in a mutable way.

Definition 7 (Mutability Set)**.** *Let φ be a TeSSLa specification, (G, S) with $G = (V, E)$ the corresponding usage graph and $(V, <)$ a translation order. The set $M_\varphi \subseteq V$ is called* mutability set *for specification φ, if*

1) *There are no nodes s, s', t, t' with $t \neq t', s \simeq s'$ and $s \xrightarrow{W} t$, $s' \xrightarrow{W+L} t'$ in G and $s \in M_\varphi$ (no double write/reproduction)*
2) *There are no nodes s, s', t, t' with $s \simeq s'$ and $s \xrightarrow{W} t$, $s' \xrightarrow{R} t'$ with $t < t'$ in G and $s \in M_\varphi$ (no read after write)*
3) *There are no edges $s \xrightarrow{P+W+L} t$ with $s \in M_\varphi \iff t \notin M_\varphi$ (consistent mutability)*

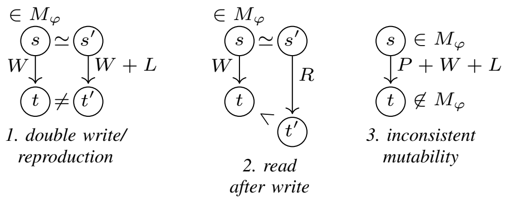


Fig. 6. Forbidden patterns for a mutability set M_φ.

This criterion is straight forward: One may not write, read or reproduce a mutable data structure at a later timestamp, if it was already written. The last rule is included because in our approach a conversion from mutable to persistent data structures is very costly. So if one stream is implemented with persistent type all streams which get an event passed from it also have to be implemented in a persistent way. The three rules are visualized in Figure 6.

We call a mutability set M_φ optimal, if there is no other mutability set M'_φ with $|M'_\varphi| > |M_\varphi|$. Note that such an optimal mutability set is not necessarily unique.

E. Overall Algorithm

We now present an algorithm for finding the maximal set of variables in our specification that can be implemented in a mutable way. Finding the optimal mutability set depends on the chosen translation order. For our example in Figure 1 two possible translation orders yielding different mutability sets are depicted in Figure 7. Since only rule 2 in Definition 7 depends on the translation order, we first consider rules 1 and 3 and then find an optimal translation order fulfilling rule 2.

The algorithm initializes an empty set $P \subseteq V$ holding all variables that have to be implemented in a persistent way and fills this set throughout its execution. A simplified pseudo-code representation can be found in Figure 8. Our algorithm is divided into four major steps:

Step 1. According to rule 3 of Definition 7 (consistent mutability) there are stream variables in the specification which either have to be implemented persistent all together or mutable to avoid type conversions in the generated code. So in the first step we determine these variable families. For managing these sets a Union-Find data structure is beneficial.

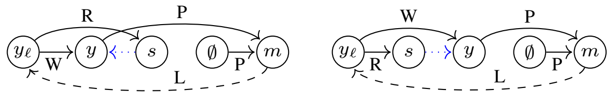


Fig. 7. Two possible translation orders for the example from Figure 1 (left to right, flattened). Left $M_\varphi = \emptyset$ because the read access of y_ℓ is after its write, but right yields $M_\varphi = \{\emptyset, m, y, y_\ell\}$. The dotted blue line indicates a read-before-write constraint. For better readability nodes i and u are not depicted.

We iterate over all pass, write and last edges (u, v) and union together their families. In the following steps we always add whole variable families to P.

Step 2. In the second step we check rule 1 of Definition 7 (no double write/reproduction). Therefore we iterate over all write edges in our graph. For each write edge $u \xrightarrow{W} v$ we follow paths of last and pass edges leading to u in the inverse order and from all nodes we traverse we follow last and pass edges downwards again to find streams potentially carrying the same data structure at the same timestamp. For this we apply the rules for aliasing from Definition 6 using the over-approximation of the set inclusion of the ticking behavior.

If during stepping down we reach another write or last edge $u' \xrightarrow{W+L} v'$ with $u \simeq u'$ we include the variable family of u to P due to violation of rule 1 from Definition 7.

Step 3. At last we also have to care about rule 2 of Definition 7 (no read after write). We do this in the following way: First for every write edge $u \xrightarrow{W} v$ we again identify the aliases u' of u (In the pseudo-code we combined the alias search of step 2 and step 3). Then, if an edge $u' \xrightarrow{R} v'$ with $u \simeq u'$ exists, we memorize the read-before-write constraint (v', v) in a set E', since the variable family of u can only be mutable if the read operation from v' is calculated prior the write operation of v in the generated code.

Step 4. Finally, after the whole graph is traversed we add the memorized read-before-write constraints (E') as additional edges to the usage graph (see blue edges in Figure 7). After this, every cycle in the TeSSLa usage graph without last/delay edges (S) runs through one of these added edges indicating a read-before-write constraint. If the graph does not contain cycles, a valid translation order can be found by a linear sorting of the graph. We now want to extend our set P to P' with as less variables as possible such that after removing all read-before-write edges where the family of the written node is in P' (persistent structures may be written before they are read) the graph is cycle free.

To do this we combine the edges (v'_i, v_i) where the written/read variables belong to the same variable family, which is the family of v_i, to sets and weight every set according to the number of variables contained in the family. Then we search the set of edge sets with the lowest accumulated weight, such that the graph is cycle-free after removing these edges. We add the variable families that belong to the removed edges to P and receive the minimal set P' this way. Finally we return $M_\varphi = V \backslash P'$ with V being the set of all variables.

1) Correctness and Optimality: The given algorithm is correct in the sense that the resulting mutability set M_φ suffices Definition 7 and optimal as it yields the greatest possible mutability set. Note that the greatest possible mutability set according to Definition 7 not necessarily contains every variable that can principally be implemented mutable. This is undecidable in general.

Correctness follows from the following consideration. If the resulting mutability set M_φ would not be correct, then at least one rule of Definition 7 would have to be violated:

```
**Input**: TeSSLa usage graph $((V, E), S)$
$P \leftarrow \emptyset$ /* persistent vars */ ,   $E' \leftarrow \emptyset$ // edges from read/write constraints
$F \leftarrow$ new UnionFind($V$) // families of variables for consistent mutability
**for each** $u \xrightarrow{W+P+L} v \in E$ **do** $F$.union($u$,$v$) // Step 1
**for each** $u \xrightarrow{W} v \in E$ **do** // Traverse all write edges for Step 2 and 3
  // Go up to all ancestors c and down to all descendants u
  **for each** ancestor $c$ of $u$ reachable with P/L edges **do**
    **for each** $u' \in V$ reachable from $c$ with P/L edges **do**
      $p_u = c \xrightarrow{(P+L)*} u, \quad p_{u'} = c \xrightarrow{(P+L)*} u'$
      **if** $u$ is potential alias of $u'$ **then** // Perform check by paths analysis
        // Step 2: Add all variables from family of $v'$ to $P$
        **if** $\exists v' \in V \backslash \{v\} : u' \xrightarrow{W+L} v' \in E$ **then** $P \leftarrow P \cup F(v')$
        **for each** $v' \in V$ with $u' \xrightarrow{R} v' \in E$ **do**
          $E' \leftarrow E' \cup \{(v', v)\}$ // Step 3
Find minimal $P' \supseteq P$ s.t. // Step 4
  $G' = (V, (E \cup E') \backslash (S \cup \{(u, v) \in E' \mid F(v) \subseteq P'\}))$ is cycle–free
**Output**: $M_\varphi = V \backslash P'$
```

Fig. 8. Pseudo-code of the combined algorithm.

In step 2 of the algorithm we iterate through every write edge $u \xrightarrow{W} v$. Then we find all potential aliases u' of u. Since potential aliases are always connected by a common ancestor (Definition 6) stepping up and down in the graph, as we do it in our algorithm, will yield us all these aliases. If u' is written or reproduced by a last edge we add the variable family of u and u' to our set of persistent variables P, so they will not be in M_φ and hence rule 1 of Definition 7 cannot be broken.

For every write edge $u \xrightarrow{W} v$ we also identify potential aliases u' of u in step 3, where u' is read by another node v': $u' \xrightarrow{R} v'$. For each of these read nodes v' we add an additional edge (v', v) to our final graph in step 4. From step 4 we get an ordering of the graph s.t. all read nodes can be ordered in front of the corresponding write nodes if the accessed variable is still in M_φ. Hence a breach of rule 2 is impossible as well.

Finally rule 3 cannot be violated since we only add whole variable families to our sets P and P', and in step 1 of the algorithm we union together all nodes connected by pass, write and last edges to a variable family.

So we conclude our algorithm yields a valid mutability set according to Definition 7. The returned set is also optimal: In the algorithm we add the variable family of a variable to the set of persistent variables P, if and only if this variable violates rule 1 or 3. It is not possible to make any of these variables mutable, no matter of the chosen ordering, since the concerned rules 1 and 3 are independent of the ordering. So these variables are never contained in any mutability set. In the fourth step we also add variable families to the persistent set P', where a read-before-write constraint is violated for our chosen ordering. However we choose exactly the ordering which implies as less as possible variables to be added to P' among all possible orders. Hence our set P' is as small as possible, while the returned mutability set is as big as possible.

Summarizing, the algorithm yields a correct and minimal mutability set according to Definition 7. Hence implementing the data strucures from this set in a mutable way does not lead to incorrect behavior of the generated monitor.

2) Complexity: The individual steps of the algorithm are located in different complexity classes. Those outside of complexity class P are the implication check of two streams used in the aliasing analysis in step 2,3 and finding the optimal ordering in step 4.

The implication check of two streams reduces to the implication check of two positive boolean formulae in our algorithm. This problem is known to be CONP complete [22]. However the formulas may have an exponential size in terms of the specification length in the worst case.

Finding the optimal ordering lies in NP since one can check if a better solution than a given one exists by guessing a set of edges to remove and checking if this yields a higher number of mutable variables. The problem is further NP complete which can be shown by reduction from the very similar Feedback-Arc-Set problem [23].

V. Evaluation

We have implemented the algorithm as described above in a TeSSLa-to-Scala compiler. We use the standard Scala mutable and persistent data structures. We have evaluated our approach both on generic synthetic examples, where random input data is produced directly by the generated monitor, and in the second part on four real-world specifications and data. We use the synthetic evaluation to examine which impact the size of the data structures and length of the input trace have on the speedup. The real-world scenarios are included to evaluate the performance-gain in practice. Compilation took less than half a minute for all mentioned specifications. We ran our measurements on a 64-bit Linux with an Intel Core i7 with manually fixed clock-rate of 2 GHz and 8 GB RAM.

In the following we compare the optimized monitors which use mutable data structures with the unoptimized monitors which use exclusively persistent data structures. Those can be seen as baseline since they are the natural choice when no dedicated optimization algorithm is used.

A. Evaluation on Synthetic Data

These typical use cases may need unbounded data structures:

Seen Set. A set keeps track of values that have occurred in the past. If the new value is already contained in the set, it is removed, if not it is added. Additionally the specification prints out whether the element has already been contained.

Map Window. We store the last n data values which occurred on a stream. In our implementation we use a map as a ring buffer, depicting a position index to its value. Further we print out the n^{th} last value at every new input that arrives.

Queue Window. We implemented the same behavior as in Map Window but with a queue data structure where every new input event is enqueued at back and the first element of the queue is printed and removed.

We use these specifications since they are standard use cases of the mentioned data structures without additional code parts that do not rely on data structures. Thus the results give an idea of the maximal reachable speedup through the optimization and how it differs among the supported data structures.

As one would expect, the runtime difference between the optimized and the non-optimized monitor depends on how big the data structures get. Therefore we examined three variations of the upper specifications where the data structures only grow up to a certain size: We used a variation for small (max. 10 elements), medium (max. 200 elements) and large (max. 10,000 elements) data structures for every specification. We ran the examples for sample traces of length 10^4 to 10^9. For the examples with small data structures even up to length 10^{10}.

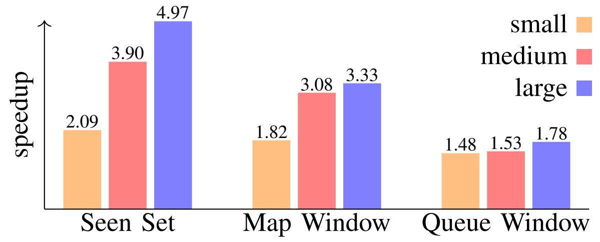


Fig. 9. Comparison of the measured speedups for different specifications and data structure sizes.

Figure 9 depicts the speedups of the optimized over the non-optimized implementations.

As Scala is compiled to Java bytecode the generated monitor is optimized during execution by the JVM's JIT compiler. This effects a non-linear development of the speedups with growing execution time and trace size. Since the executions also contain some non-deterministical influences, e.g. garbage collection, we have taken the median of three runs for the calculation of the speedup. Figure 9 show the speedup for the longest evaluated trace length, where it has already quite stabilized.

In general one can see that the speedup is higher for large data structures than for small ones. This is due to the growing overhead of modifying big persistent data structures in comparison to in-place modification of mutable ones. The highest speedup can be reached for the Seen Set example with almost 5 for large sets. For the other structures we reach a maximum speedup of 3.3 (Map Window) and 1.8 (Queue Window), again for the large data structures.

For medium and small data structure sizes the speedup is lower. The one for medium ones lies between 1.5 and 3.9 while for small data it is between 1.5 and 2.1. For all data structure sizes the highest speedup is always achieved for the Seen Set example, the lowest for the Queue Window.

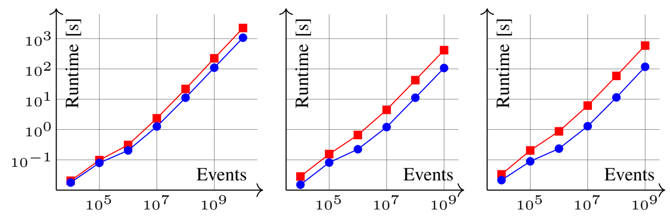


Fig. 10. Seen Set for small, medium and large set sizes (left to right) for the ●– optimized and the ■– non-optimized case.

In Figure 10 the runtime of the optimized and non-optimized Seen Set example is depicted over time. In all cases (small, medium, large data sets) the speedup stabilizes around a trace length of 10^6. Furthermore one can observe that the runtime of the optimized monitor is hardly influenced by the set size, while in the non-optimized case it is. This explains why the speedup is growing with the data structure sizes.

However, for the Queue Window example this effect is weaker, which results in a more constant speedup for all data structure sizes. This is caused by the different persistent implementations of set and queue: The persistent queue is realized as two lists, one is used for appending elements, the other one for removing elements; if the list for removing elements runs empty the other one is reverted. In difference to a persistent set which is implemented as an adjusted Hash-Array Mapped Trie (HAMT) [24], [25] this requires less restructuring after a modification. Hence the persistent queues are more efficient compared to their mutable counterpart than sets.

B. Evaluation on Real-World Scenarios

In the previous section we examined how the speedup alters for different trace lengths and data set sizes on synthetic specifications. In this section we show how big the speedup of the optimized compiler is for real-world applications.

We examined four different specifications on two data sets. The existing specifications were formalized in TeSSLa specifically for this evaluation. The first two specifications were executed on a 14 GB database log from the Runtime Verification Competition 2014 [26], originally from [27]. The log contains information about database operations performed on a distributed database system (inserts, deletes, updates, script executions etc.) in about one year. The data originates from Nokia's Lausanne Data Collection Campaign 2010 [28], [29]. The second two specifications belong to energy consumption analysis of a university building that was monitored in the ReNuBiL (https://renubil.de) project. Aim of this project is to decide when unneeded power from the university net can be fed into batteries of a car-sharing network or sourced from there in case of a higher need. Since we only had log data of about one month we extended the data to one year by repeating the measured data points. The four specifications are defined as follows:

DBTimeConstraint. Here we check the constraint from the RV Competition [26]: If data was added to database db3 then it had to be added to db2 during the last 60 seconds. We check this by maintaining a map with the insertion times of db2.

DBAccessConstraint. This checks a typical database constraint: A record may not be accessed before it was inserted or after it was deleted in a database. We use a set of all currently inserted IDs to check this.

PeakDetection. In this specification we detect peaks in the current power consumption data which may distort further calculations on the data. Therefor we check if a value is 40 % lower or higher than the medium of the values from a quarter hour ago to a quarter hour in the future. For this we require a queue to calculate the moving average.

SpectrumCalculation. Here we calculate a spectrum how the values of the power consumption are distributed in a map data structure which are in the end used to calculate how often the measured power consumption is above a certain threshold.

TABLE I
MEASURED RUNTIMES AND SPEEDUPS ON THE REAL-WORLD SCENARIOS

Specification	**Op.**	**Non-op.**	**Speedup**
DBTimeCons.	171 s	216 s	1.3
DBAccessCons.(full)	233 s	> 1 h	> 15.5
DBAccessCons.(33 %)	59.2 s	127 s	2.1
PeakDetection	7.56 s	14.0 s	1.9
SpectrumCalc.	1.04 s	2.07 s	2.0

Our measurements on these real-world scenarios are shown in Table I (median over three runs). The measured speedups are mostly smaller than the maximal speedups that were measured in the synthetic setting but still significant. This is due to the constant time which is required for reading from the hard drive (70 s for the first two specifications) and a larger amount of calculations which are not related to data structures. The input format of the database log is more complex as it contains record of varying type and size and requires more parsing than the input data for the latter examples, which only consists of timestamp and data value.

For the second specification DBAccessConstraint we ran the specification also on 33 % of the trace to keep the data structures smaller. For the optimized version it was no problem to check the full trace, it terminates after 3.9 minutes. For the unoptimized monitor, the memory consumption grew rapidly, probably because of garbage collection trailing behind. The operating system of our test environment started swapping and the monitor did not terminate within one hour.

VI. Conclusion

In this paper we showed an optimization of code generated from a TeSSLa specification by the usage of mutable data structures where possible and persistent ones in the other cases. By this we have combined two traditional approaches for the Aggregate Update Problem: Static Analysis for finding data structures that can be directly mutated and the use of persistent ones. TeSSLa has, like other data flow languages, the possibility to access past events more than once in future timestamps. This has to be taken in account to avoid the access of data structures after their modification. While traditional approaches restricted the language to avoid such scenarios, we presented an algorithm to check whether an event is replicated and written more than once in the future.

Our algorithm is based on the analysis of the streams' triggering behaviors and checking implications on them. Since this is in general undecidable, we presented an approximation, that is coNP-hard. Also the finding of a perfect translation order is NP-complete. However for typical specifications our implementation showed no unusual long compilation time. We have implemented our optimized translation and evaluated it on several examples where it showed a significant performance gain. As such, we presented a viable compiler optimization based on static analysis.

Artifact Appendix

A. Abstract

Our artifact is packaged as a Docker image for x86-64 architectures. It provides shell scripts to compile and execute the synthetic as well as the real-world benchmarks described in the paper. In case of the synthetic benchmarks the traces are generated in memory during the benchmark's execution. For the real-world benchmarks the real-world traces are included in the image. The artifact further contains the source code of the implemented compiler phase and additional examples.

B. Artifact Check-List (Meta-Information)

- **Algorithm:** Code generation from TeSSLa specifications[1], Aggregate Update Problem
- **Program:** TeSSLa compiler
- **Compilation:** Scala compiler, sbt (contained in the Docker image)
- **Transformations:** Optimization implemented as a TeSSLa compiler phase
- **Binary:** Optimized TeSSLa compiler included as binary, runtime environment provided as Docker image
- **Data set:** Synthetic data generated at runtime, Nokia (RV competition) and ReNuBiL traces provided in the Docker image
- **Run-time environment:** JVM in the Docker image
- **Hardware:** x86-64 architectures
- **Execution:** Shell scripts provided in the Docker image
- **Output:** Measured runtime and output is printed to stdout
- **Experiments:** Synthetic and real-world examples as described in the paper.
- **How much disk space required (approximately)?:** 15 GB
- **How much RAM required (approximately)?:** Please provide at least 10 GB RAM to the Docker image, otherwise it might crash.
- **How much time is needed to prepare workflow (approximately)?:** You need to install Docker and download the provided Docker image.
- **How much time is needed to complete experiments (approximately)?:** 10 hours
- **Publicly available?:** The Docker image is publicly available on Zenodo including the source code, the benchmark scripts and the real-world trace data.

[1]https://www.tessla.io/

C. Description

1) How Delivered: Our artifact is packaged as a Docker image for x86-64 architectures. It provides shell scripts to compile and execute the synthetic as well as the real-world benchmarks described in the paper.

2) Hardware Dependencies: Our artifact is packaged as a Docker image for x86-64 architectures. As the image contains the real-world trace data, it is approximately 15 GB large if extracted. Please provide at least 10 GB RAM to the Docker image, otherwise it might crash.

3) Software Dependencies: Our artifact is packaged as a Docker image. Please ensure that you use at least Docker version 20 or newer. The docker image contains all further software dependencies and provides shell scripts to execute the benchmarks.

4) Data Sets: For the synthetic benchmarks the traces are generated in memory during the benchmark's execution. For the real-world benchmarks the traces are included in the image. See section V of the paper for a discussion of the different data sets of the synthetic and the real-world benchmarks.

D. Installation

1) Download the Docker image archive `aggregate_update_artifact_ubuntu.tar.gz` from https://zenodo.org/record/5710526#.YZagm3vMJgc.
2) Load the Docker image archive: `docker load -i aggregate_update_artifact_ubuntu.tar.gz`
3) Ensure that Docker allocates sufficient memory: Please provide at least 10 GB RAM to the Docker image, otherwise it might crash.
4) Start the Docker image: `docker run -ti aggregate_update_artifact_ubuntu`

E. Experiment Workflow

- The synthetic benchmarks *Queue Window*, *Map Window* and *Seen Set* are provided in the directory `/app/art-benchmarks`. The script `runAll.sh` executes all benchmarks described in the paper. The runtimes of the optimized and the non-optimized executions for the different benchmarks, different trace lengths and different data structure sizes (small, medium and large) are subsequentially printed to stdout. The TeSSLa monitors are provided pre-compiled in the Docker image, but you can also use the script `compileAll.sh` in the same directory to compile them manually from the specifications.

- The *Nokia benchmark (RV competition)* is provided in the directory `/app/real-world-scenarios/Nokia`. The two subdirectories `accessConstraint` and `timeConstraint` contain the benchmarks *DBAccessConstraint* and *DBTimeConstraint*, respectively. Each contains a script `run.sh` which compiles the TeSSLa monitors from the TeSSLa specifications and executes them on the traces contained in the directory, too. The runtimes of the optimized and the non-optimized executions as well as the monitor outputs are subsequentially printed to stdout. (Note: Make sure for this experiment, Docker has access to about 10 GB RAM)

- The *ReNuBiL benchmark* is provided in the directory `/app/real-world-scenarios/Renubil`. The two subdirectories `PeakDetection` and `Spectrum` contain the benchmarks *PeakDetection* and *SpectrumCalculation*, respectively. Each contains a script `run.sh` working as described for the Nokia benchmark above.

F. Evaluation and Expected Result

As described in the previous section the benchmarks measure the runtime and print it to stdout. The speedups follow from the runtime and should roughly match those discussed in section V of the paper.

G. Experiment Customization

For all the benchmarks mentioned in the previous section the TeSSLa source code is provided in the mentioned directories. It can be tweaked and the monitor can be recompiled as described above. Further example specifications with complex data structures are located in the `src/examples` directory.

H. Notes

For further instructions on how to run the benchmarks, adjust the experiments and compile the TeSSLa compiler phase from scratch see the `README.md` provided in the Docker image.

References

[1] W. R. Sutherland, "The on-line graphical specification of computer procedures." Ph.D. dissertation, Massachusetts Institute of Technology, 1966.

[2] T. B. Sousa, "Dataflow programming concept, languages and applications," in *Doctoral Symposium on Informatics Engineering*, vol. 130, 2012.

[3] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, "TeSSLa: Temporal Stream-based Specification Language," in *SBMF*, ser. LNCS. Springer, 2018, https://doi.org/10.1007/978-3-030-03044-5_10.

[4] P. Hudak and A. G. Bloss, "The aggregate update problem in functional programming systems," in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, M. S. V. Deusen, Z. Galil, and B. K. Reid, Eds. ACM Press, 1985, pp. 300–314, https://doi.org/10.1145/318593.318660.

[5] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet, "A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler," in *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2012, LCTES '12, Beijing, China - June 12 - 13, 2012*, R. Wilhelm, H. Falk, and W. Yi, Eds. ACM, 2012, pp. 51–60, https://doi.org/10.1145/2248418.2248426.

[6] P. Wadler, "Linear types can change the world!" in *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, M. Broy, Ed. North-Holland, 1990, p. 561.

[7] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," in *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, J. Hartmanis, Ed. ACM, 1986, pp. 109–121, https://doi.org/10.1145/12130.12142.

[8] P. Hudak, "A semantic model of reference counting and its abstraction (detailed summary)," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986*, W. L. Scherlis, J. H. Williams, and R. P. Gabriel, Eds. ACM, 1986, pp. 351–363, https://doi.org/10.1145/319838.319876.

[9] A. G. Bloss, "Update analysis and the efficient implementation of functional aggregates," in *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, J. E. Stoy, Ed. ACM, 1989, pp. 26–38, https://doi.org/10.1145/99370.99373.

[10] A. Deutsch, "On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications," in *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, F. E. Allen, Ed. ACM Press, 1990, pp. 157–168, https://doi.org/10.1145/96709.96725.

[11] A. V. S. Sastry, W. D. Clinger, and Z. M. Ariola, "Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates," in *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, J. Williams, Ed. ACM, 1993, pp. 266–275, https://doi.org/10.1145/165180.165222.

[12] M. Nebut, "An overview of the signal clock calculus," *Electron. Notes Theor. Comput. Sci.*, vol. 88, pp. 39–54, 2004, https://doi.org/10.1016/j.entcs.2003.05.005.

[13] V. Papailiopoulou, L. Madani, L. du Bousquet, and I. Parissis, "Extending structural test coverage criteria for lustre programs with multi-clock operators," in *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, ser. Lecture Notes in Computer Science, D. D. Cofer and A. Fantechi, Eds., vol. 5596. Springer, 2008, pp. 23–36, https://doi.org/10.1007/978-3-642-03240-0_6.

[14] S. Abu-Mahmeed, C. McCosh, Z. Budimlic, K. Kennedy, K. Ravindran, K. Hogan, P. Austin, S. Rogers, and J. Kornerup, "Scheduling tasks to maximize usage of aggregate variables in place," in *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, ser. Lecture Notes in Computer Science, O. de Moor and M. I. Schwartzbach, Eds., vol. 5501. Springer, 2009, pp. 204–219, https://doi.org/10.1007/978-3-642-00722-4_15.

[15] U. Beaugnon, A. Cohen, and M. Pouzet, "In-place update in a dataflow synchronous language: A retiming-enabled language experiment," in *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2016, Sankt Goar, Germany, May 23-25, 2016*, S. Stuijk, Ed. ACM, 2016, pp. 40–49, https://doi.org/10.1145/2906363.2906379.

[16] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems," in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987.* ACM Press, 1987, pp. 178–188, https://doi.org/10.1145/41625.41641.

[17] T. Gautier and P. Le Guernic, "SIGNAL: A declarative language for synchronous programming of real-time systems," in *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, ser. Lecture Notes in Computer Science, G. Kahn, Ed., vol. 274. Springer, 1987, pp. 257–277, https://doi.org/10.1007/3-540-18317-5_15.

[18] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "LOLA: runtime monitoring of synchronous systems," in *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 2005, pp. 166–174, https://doi.org/10.1109/TIME.2005.26.

[19] F. Gorostiaga and C. Sánchez, "Striver: Stream runtime verification for real-time event-streams," in *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, ser. Lecture Notes in Computer Science, C. Colombo and M. Leucker, Eds., vol. 11237. Springer, 2018, pp. 282–298, https://doi.org/10.1007/978-3-030-03769-7_16.

[20] P. Amagbégnon, L. Besnard, and P. Le Guernic, "Implementation of the data-flow synchronous language SIGNAL," in *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA,*

June 18-21, 1995, D. W. Wall, Ed. ACM, 1995, pp. 163–173, https://doi.org/10.1145/207110.207134.

[21] N. Halbwachs, P. Raymond, and C. Ratel, "Generating efficient code from data-flow programs," in *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP'91, Passau, Germany, August 26-28, 1991, Proceedings*, ser. Lecture Notes in Computer Science, J. Maluszynski and M. Wirsing, Eds., vol. 528. Springer, 1991, pp. 207–218, https://doi.org/10.1007/3-540-54444-5_100.

[22] P. A. Bloniarz, H. B. H. III, and D. J. Rosenkrantz, "Algebraic structures with hard equivalence and minimization problems," *J. ACM*, vol. 31, no. 4, pp. 879–904, 1984, https://doi.org/10.1145/1634.1639.

[23] R. M. Karp, "Reducibility among combinatorial problems," in *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, Eds. Springer, 2010, pp. 219–241, https://doi.org/10.1007/978-3-540-68279-0_8.

[24] M. J. Steindorfer and J. J. Vinju, "Optimizing hash-array mapped tries for fast and lean immutable JVM collections," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, J. Aldrich and P. Eugster, Eds. ACM, 2015, pp. 783–800, https://doi.org/10.1145/2814270.2814312.

[25] P. Bagwell, "Ideal hash trees," École polytechnique fédérale de Lausanne, Tech. Rep., 2001.

[26] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zalinescu, and Y. Zhang, "First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014," *Int. J. Softw. Tools Technol. Transf.*, vol. 21, no. 1, pp. 31–70, 2019, https://doi.org/10.1007/s10009-017-0454-5.

[27] D. A. Basin, M. Harvan, F. Klaedtke, and E. Zalinescu, "Monitoring data usage in distributed systems," *IEEE Trans. Software Eng.*, vol. 39, no. 10, pp. 1403–1426, 2013, https://doi.org/10.1109/TSE.2013.18.

[28] I. Aad and V. Niemi, "Nrc data collection and the privacy by design principles," *Proc. of PhoneSense*, pp. 41–45, 2010.

[29] J. K. Laurila, D. Gatica-Perez, I. Aad, J. Blom, O. Bornet, T. M. T. Do, O. Dousse, J. Eberle, and M. Miettinen, "From big smartphone data to worldwide research: The mobile data challenge," *Pervasive Mob. Comput.*, vol. 9, no. 6, pp. 752–771, 2013, https://doi.org/10.1016/j.pmcj.2013.07.014.