TimelyMon: A Streaming Parallel First-Order Monitor

Lennard Reese, Rafael Castro G. Silva, and Dmitriy Traytel

Department of Computer Science, University of Copenhagen, Copenhagen, Denmark

Abstract. First-order monitors analyze data-carrying event streams. When event streams are generated by distributed systems, it may be difficult to ensure that events arrive at the monitor in the right order. We develop a new monitoring tool for metric first-order temporal logic, called TimelyMon, that can process out-of-order events. Using the stream processing framework Timely Dataflow, TimelyMon also supports parallelized monitoring. We demonstrate TimelyMon's good performance and scalability on synthetic and real-world benchmarks.

Keywords: Monitoring · Temporal logic · Stream processing

1 Introduction

First-order monitors detect complex temporal patterns in data-carrying event streams. Metric first-order temporal logic (MFOTL) [5] (Section 2) is a powerful language for expressing the temporal patterns. Monitors like DejaVu [15], MonPoly [6], and VeriMon [2] use (variants of) MFOTL as their input language.

Patterns expressed by MFOTL formulas are interpreted over event streams that are subdivided into time-points. A time-point consists of a time-stamp and a database, i.e., a finite set of events. From the point of view of the monitor, events coming from a single database happen concurrently. Existing monitors either restrict the databases to be singleton sets [7,15] or process entire databases at once and in-order [2,6]. Both these modes of operation limit the existing tools' suitability for monitoring distributed systems in which individual concurrent events may arrive at the monitor out-of-order [3,4]. The plausible workaround based on reordering of the events in a component wrapping the monitor has been proposed [3]. However, the monitor is idle while it waits for delayed events in this approach.

In this work, we develop a monitoring tool for MFOTL that supports the *streaming* mode of operation (Section 3), i.e., the processing of individual events that may arrive out-of-order. Our monitor's output is also presented to the user event-wise and possibly out-of-order. This has the advantage that whenever future temporal operators are involved users may get to see verdicts much earlier than in traditional monitors, which output verdicts for entire time-points at once. A somewhat surprising advantage of the streaming setting is that we can reuse mostly the same data structures for both past and future operators and can even support unbounded future operators without completely stalling the monitor.

An additional challenge for MFOTL monitoring, which we also tackle, is scalability. Typically, MFOTL monitors are sequential algorithms that are quickly overwhelmed by high volume and high velocity event streams. Parallelization is the main approach for tackling such scalability issues. The approaches for parallelizing MFOTL monitors can be grouped in two categories. The *black box* parallelization

approach uses the sequential monitors without modifications (as a black box) to process independent slices of the input [1, 11, 27]. The alternative is *white box* parallelization, in which the monitoring algorithm is itself parallelized [14, 17, 25].

Generic data stream processing frameworks like Apache Flink [10] and Timely Dataflow [24] are useful tools for implementing black and white box parallel monitors as they enforce a program structure that is susceptible to a high degree of parallelism while hiding the pitfalls of parallel programming from their users.

We use Timely Dataflow [23,24] to implement our streaming MFOTL monitor, called TimelyMon, as a white box parallel monitor. Timely Dataflow organizes its programs as a graph of stream transformers, called operators. Given a monitorable [5] MFOTL formula, we map every MFOTL operator occurring in it to a Timely Dataflow operator that inputs the verdicts from the subformulas and outputs the verdicts according to the MFOTL operator's semantics. Parallelized evaluation of Timely Dataflow operators naturally results in out-of-order verdicts for subformulas, which meshes well with our streaming mode of operation.

We demonstrate TimelyMon's reasonable performance and good scalability on a standard first-order benchmark [20] (Section 4) and a real-world example. We also extend the benchmark by incorporating different degrees of the input being out-of-order. TimelyMon is publicly available [26], along with build and usage instructions and our experiments.

Related Work TimelyMon processes out-of-order event streams. In the area of stream runtime verification, pioneered by Lola [12], in-order processing is the norm. An exception is TeSSLa [22], which can process multiple non-synchronized streams, but requires each of them to be in-order. We do not impose any order requirements, but require that the events are labeled with their "true" position so that one has the information to reconstruct the sorted input in principle.

Decentralized runtime monitoring [13] uses multiple monitors to check a centralized specification via multiple decentralized specifications. Each operator in TimelyMon can be viewed as one decentralized monitor with a partial view of the monitored system. Bonakdarpour et al. [9,18] discussed decentralized run-time verification with a focus on fault-tolerance and crash-resiliency and focuses on LTL, whereas TimelyMon supports MFOTL. Timely Dataflow has no built-in fault-tolerance mechanisms, although different approaches have been proposed [21,28].

The traditional MFOTL monitors MonPoly [6], DejaVu [15], and VeriMon [2] all assume in-order inputs. This restriction also extends to the black box parallel monitors using these tools [1,27]. Basin et al. [3] note this limitation and incorporate watermark-based reordering in the black box monitor, with the aforementioned downside of the monitor having to wait for complete input prefixes.

A exception to in-order MFOTL monitors is Basin et al.'s [7] (closed-source) monitor POLÍMON [19], which supports out-of-order processing for metric temporal logic extended with freeze quantifiers. Their streams store precisely one value per register (which the freeze quantifiers refer to) at a time-point, whereas for us one time-point consists of a set of events. Their logic, also called half-order temporal logic [16], is much less expressive than MFOTL. Nonetheless, we use a variant of their observations data structure to optimize our Once and Eventually operators.

2 Preliminaries

We briefly recall metric first-order temporal logic (MFOTL).

Metric First-Order Temporal Logic Let $\mathbb{I}$ be the set of nonempty intervals over $\mathbb{N}$. An interval $[b, b') \in \mathbb{I}$ denotes $\{a \in \mathbb{N} \mid b \leq a \wedge a < b'\}$, where $b \in \mathbb{N}$, $b' \in \mathbb{N} \cup \{\infty\}$, and $b < b'$. We fix a domain $\mathbb{D}$, a set of event names $\mathbb{E}$, and the function $\iota : \mathbb{E} \to \mathbb{N}$ that assigns each $e \in \mathbb{E}$ the arity $\iota(e)$. Additionally, we consider a set of variables $\mathbb{V}$ such that $\mathbb{V} \cap (\mathbb{D} \cup \mathbb{E}) = \varnothing$. MFOTL formulas $\alpha, \beta, \ldots$ are defined as follows:

$$\alpha, \beta \;:=\; e(\bar{t}) \mid t_1 \approx t_2 \mid \neg\, \alpha \mid \alpha \wedge \beta \mid \alpha \vee \beta \mid \exists x.\, \alpha \mid \alpha\, \mathsf{S}_I\, \beta \mid \alpha\, \mathsf{U}_I\, \beta$$

The predicate $e(\bar{t})$ consists of an event name $e \in \mathbb{E}$ and a finite sequence of arguments $\bar{t} = t_1, \ldots, t_{\iota(e)}$, where each $t_i \in \mathbb{V} \cup \mathbb{D}$. Similarly, equality $\approx$ is applied to arguments from $\mathbb{V} \cup \mathbb{D}$. Boolean operators and the existential quantifier are standard. The temporal operators S ("since") and U ("until") are annotated with an interval $I \in \mathbb{I}$. From this minimal MFOTL syntax, we derive additional operators in a standard fashion: $\top := \exists x.\, x \approx x$ ("truth"), $\alpha \to \beta := \neg\, \alpha \vee \beta$ ("implication"), $\forall x.\, \alpha := \neg\, \exists x.\, \neg\, \alpha$ ("universal quantification"), $\Diamond\!\!\!-_I\, \alpha := \top\, \mathsf{S}_I\, \alpha$ ("once"), $\Diamond_I\, \alpha := \top\, \mathsf{U}_I\, \alpha$ ("eventually"), and $\Box_I\, \alpha := \neg\, \Diamond_I\, \alpha$ ("always").

A *database* D is a finite set of events and each event has the form $e(d_1, \ldots d_{\iota(r)})$ for $e \in \mathbb{E}$ and $d_i \in \mathbb{D}$. A *time-stamped database* is a pair of a time-stamp $\tau \in \mathbb{N}$ and a database D. An event stream is an infinite sequence $\rho = \langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$ of time-stamped databases. Each time-stamped database in ρ is called a time-point and is identified by its index i. Time-stamps in the event stream must (1) monotonically increase: $\forall i \in \mathbb{N}.\, \tau_i \leq \tau_{i+1}$ and (2) always eventually make progress: $\forall \tau.\, \exists i.\, \tau < \tau_i$.

MFOTL formulas are evaluated over event streams and valuations for the formula's free variables. The valuation $v : \mathbb{V} \to \mathbb{D}$ maps variables to domain values. The standard semantics of an MFOTL formula α is given by a satisfaction relation of the form $v, i \models \alpha$. We refer to Basin et al. [5, 8] for a formal definition.

The objective for our monitor is to (eventually) compute correct verdicts, also called *satisfactions*, for each time-point i, i.e., satisfying valuations v applied to the formula's free variables. To ensure that the set of satisfactions for any time-point is finite, we syntactically restrict the formulas we consider as is done in some but not all MFOTL monitors [2, 6]. Such restriction allow monitors to use finite relations (or tables) instead of binary decision diagrams [15] or automata [5] to represent sets of satisfactions, which benefits performance. Specifically, we make the following restrictions, which are also present in the MonPoly and VeriMon monitors:

1. negated formulas may occur only on the right-hand side of a conjunction $\alpha \wedge \neg\, \beta$ and moreover β's free variables must be contained in α's in such cases or on the left-hand side of a $(\neg\, \alpha)\, \mathsf{S}_I\, \beta$ or $(\neg\, \alpha)\, \mathsf{U}_I\, \beta$;
2. (negated) equalities between variables may only appear on the right-hand side of a conjunction $\alpha \wedge x \approx y$ (and $\alpha \wedge \neg\, x \approx y$) and moreover x or y (x and y when negated) must be contained in α's free variables in such cases;
3. in formulas of the form $\alpha \vee \beta$, α and β must have the same free variables;
4. in formulas of the form $\alpha\, \mathsf{U}_I\, \beta$ and $\alpha\, \mathsf{S}_I\, \beta$, the free variables of α must be contained in β's. (This also applies when α is a negated formula as in 1.)

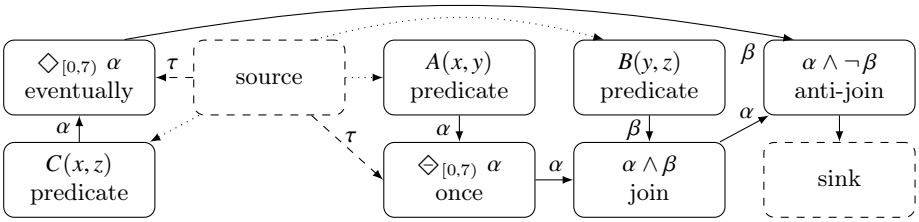


Fig. 1: Dataflow graph for the formula $(\diamondsuit_{[0,7)}\ A(x,y)) \wedge B(y,z) \wedge \neg\,(\Diamond_{[0,7)}\ C(x,z))$

.

3 Monitoring Algorithm Overview

TimelyMon creates a dataflow graph from the MFOTL formula's abstract syntax tree. Usually, the mapping from MFOTL operators to Timely Dataflow operators is one-to-one, but not always. For example, formulas of the form $\alpha \wedge \neg\beta$ are translated into a single, binary anti-join operator. Also, we optimize the derived operators $\diamondsuit$ and $\Diamond$ by using dedicated operators.

Figure 1 shows the dataflow graph constructed for the example formula $(\diamondsuit_{[0,7)}\ A(x,y)) \wedge B(y,z) \wedge \neg\,(\Diamond_{[0,7)}\ C(x,z))$. The inputs of each binary operator are labeled with the role of the incoming input stream from the operator's point of view. Even though $\diamondsuit$ and $\Diamond$ are derived from S and U, we use dedicated, optimized operators for them in the dataflow graph. All temporal operators receive an additional input (denoted by τ) over which time-stamps are communicated.

Our operators exchange three kinds of events. We use MFOTL's time-points as the notion of time in Timely Dataflow. Thus, all exchanged events are labeled with a time-point. The source operator emits individual data events of the form $e(d_1, \ldots, d_{\iota(e)})$ to the predicates operators (dotted connections). Furthermore, the source operator emits time-stamp events of the form (τ_i) mapping a time-point i to its time-stamp τ_i (dashed connections) to temporal operators. The other operators exchange subformula satisfaction events (solid connections). A satisfaction v is represented by a vector of domain values $(d_1, \ldots, d_{\iota(e)})$ that are interpreted as the values assigned by v to the subformula's free variables (listed in some canonical order).

TimelyMon operators make no assumption on the order in which it receives the events. Also, as soon as a satisfaction according to the operator's semantics is found, it is immediately output. Each parallel worker in Timely Dataflow executes the entire dataflow on different portions of the data. We specify for each operator how to distribute the data across workers. For example, the incoming events to be processed by the predicate operators are simply hashed onto the worker's identity: one worker may receive the event $A(1,2)$, another one may receive $B(2,3)$. Later operators may require the redistribution of the data across the workers. For example, the join operator must ensure that the same worker receives the matching satisfaction $(1,2)$ for variables (x,y) and $(2,3)$ for variables (y,z) to correctly compute the resulting satisfaction $(1,2,3)$ for variables (x,y,z). Therefore, our algorithm (re)distributes the events before the join operator by hashing the values from the join key, i.e., the common variables of α and β. Similarly, for the temporal operators we use the variables of α for hashing (unless α has no free variables, in which case we broadcast the trivial α tuples to all workers and hash on β's variables).

We sketch how our algorithm processes each MFOTL operator.

time-point		0	1	2	3	4
time-stamp		1		3	3	7
α	with variable list (x)			(1)	(1),(2)	(1),(2)
β	with variable list (x, y)	(1,2),(3,1)		(2,1)	(4,3)	(2,1),(4,5)
$\alpha \,\mathsf{S}_{[1,5)}\, \beta$	with variable list (x, y)					(2,1)

Fig. 2: Example partial inputs and outputs for $\alpha \,\mathsf{S}_{[1,5)}\, \beta$

Predicate Upon receiving an event $e(d_1, \ldots, d_{\iota(e)})$, the operator for the predicate $e(x_1, \ldots, x_{\iota(e)})$ creates the corresponding satisfaction $(d_1, \ldots, d_{\iota(e)})$. The operator performs matching due to constants and repeated variables that may occur, which may filter out certain events. For example, the predicate $A(x, 5, x)$ matches the event $A(1, 5, 1)$ yielding the satisfaction (1) for the variable list (x). The same predicate does not match events $A(1, 4, 1)$ or $A(1, 5, 2)$—no satisfaction is output.

Existential Quantifier The operator for the existential quantifier $\exists x.\ \alpha$ projects away (i.e., removes) the variable x from the satisfactions it receives for α.

Disjunction The binary operator for $\alpha \vee \beta$ outputs the satisfactions it receives on either input. Recall that α and β are assumed to have the same free variables.

Conjunction We use four operators to handle conjunctions. Whenever equalities are involved we consider two cases: (1) The operator for formulas of the form $\alpha \wedge x \approx y$ or $\alpha \wedge \neg\, x \approx y$ where both x and y are free in α filters α's satisfactions using the (negated) equality constraint. (2) The operator for formulas of the form $\alpha \wedge x \approx y$ where only one of x and y is free in α extends α's satisfactions with the new variable (copying the old variable's value). The regular conjunction $\alpha \wedge \beta$ proceeds symmetrically whenever it receives a new input for α or β. For both inputs, the operator stores all previously received satisfactions grouped by time-point. Upon receiving a new satisfaction for α at time-point i, the operator joins it with all matching previously received satisfactions for β at i and outputs the result. The conjunction with a negated subformula $\alpha \wedge \neg \beta$ is evaluated using a binary anti-join operator. This operator is non-streaming, in the sense that it must wait until it receives all β satisfactions for a time-point i before producing an output at i.

Temporal Operators We exemplify the inner workings of our streaming temporal operators $\alpha \,\mathsf{S}_I\, \beta$ and $\alpha \,\mathsf{U}_I\, \beta$. The operators have three inputs: one for time-stamps and two for the subformulas α and β. Recall that we assume that α's free variables are contained in those of β. We focus on the S case.

Consider the partial trace in Figure 2. It shows the received satisfactions for α and β and the computed output $(2, 1)$ for $\alpha \,\mathsf{S}_{[1,5)}\, \beta$ at time-point 4. The operator works under the assumption that none of the time-points is complete. That is, it may receive new inputs for any of the time-points 0–4. The operator has not yet received any input at time-point 1; even this time-point's time-stamp is unknown. Nonetheless and unlike traditional monitors, the operator was able to already output a satisfaction at time-point 4. To do so, it was sufficient to note the $(2, 1)$ input for β at time-point 2 and the two (1) inputs for α at time-points 3 and 4 together with the quantitative interval check: $7 - 3 \in [1, 5)$. We call such patterns of inputs resulting in S satisfactions β-α-sequences. Because α may have fewer variables than β, a single input for α may be relevant for different β-α-sequences.

Continuing our example, we note that both receiving new inputs for α and β may trigger new output for S. For example, upon receiving the satisfaction (1) at time-point 1 for α the operator can output satisfactions $(1, 2)$ at time-points 2 and 3 and possibly also at 1 if it learns that the time-stamp at time-point 1 is different from 1 (but not at 0 and 4, which fail the interval constraint). Alternatively, upon receiving the satisfaction $(1, 2)$ for β at time-point 1 with time-point 2, we can output the satisfaction $(1, 2)$ at time-points 2 and 3 (but not at 0, 1, and 4).

4 Empirical Evaluation

We evaluate TimelyMon to provide answers to the following research questions:

RQ1: Does our monitor produce the same verdicts as the verified tool VeriMon?
RQ2: How does TimelyMon perform in comparison to MonPoly and VeriMon?
RQ3: How does TimelyMon scale with respect to the number of events?
RQ4: How does TimelyMon scale with respect to the number of workers?
RQ5: How does the order of the events impact TimelyMon's performance?
RQ6: Does TimelyMon output verdicts earlier than MonPoly?

4.1 Experimental Setup

We used the synthetic benchmark generation tool by Krstić and Schneider [20] to conduct our experiments. Our traces were generated for the built-in *temporal three-way conjunctions* family of query patterns $\mathcal{F}_3$. A three-way conjunction is a specific temporal pattern that involves three distinct event types A, B, and C, with integer data values. The family is parameterized by lists of variables v_A, v_B, v_C (called variable patterns) and is given by the following parametric MFOTL formula:

$$(\diamondsuit_{[0,b]}\ A(v_A)) \wedge B(v_B) \wedge (\diamondsuit_{[0,b]}\ C(v_C))$$

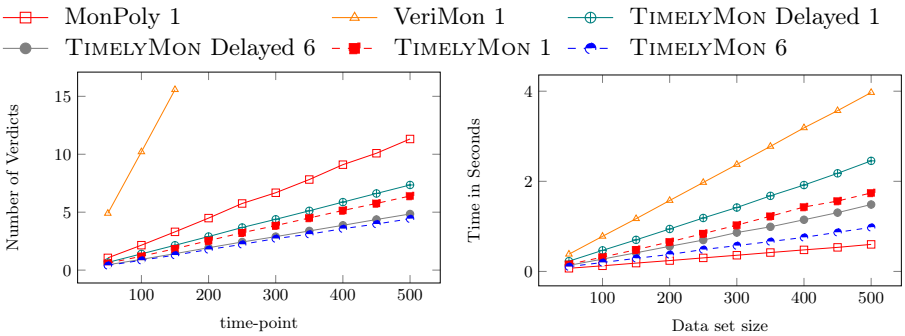
We used the three supported variable patterns: Star $v_A = (w, x)$, $v_B = (w, y)$, $v_C = (w, z)$, Linear $v_A = (w, x)$, $v_B = (x, y)$, $v_C = (y, z)$, and Triangle $v_A = (x, y)$, $v_B = (y, z)$, $v_C = (z, x)$. We also consider the Negated Triangle formula, which includes an anti-join: $(\diamondsuit_{[0,b]}\ A(x, y)) \wedge B(y, z) \wedge \neg\, (\diamondsuit_{[0,b]}\ C(z, x))$. The interval's upper bound b is denoted by the superscript ^{b} on a pattern's name, e.g., Star^7. To benchmark Since and Until we use two additional parametric formulas, which are referred to by combining the variable pattern and the operator, e.g., Star Until or Triangle Since.

$$B(v_A)\ \mathsf{S}_{[0,45]}\ ((\diamondsuit_{[0,45]}\ A(v_B)) \wedge C(v_C)) \qquad B(v_A)\ \mathsf{U}_{[0,45]}\ ((\diamondsuit_{[0,45]}\ A(v_B)) \wedge C(v_C))$$

The benchmark supports adjusting the relative frequency of each event's type. We use 10% for A, 50% for B, and 40% for C. For each variable pattern, we used the benchmark to randomly generated finite prefixes of event streams of different lengths ranging from 50 to 500 time-points with 1 000 events per time-point.

The benchmark has the capability to delay a given fraction of events, which mimics real-world scenarios where there is no guarantee of order. We use this feature but also consider other cases that exhibit a higher degree of disorder. We examine five different orders in which events are given to TimelyMon:

In-order: Events are sent in the correct order, with ascending time-points.
Delayed: Events are sent mainly in order. For each time-point, a subset of the



(a) Scalability of Negated Triangle[7] (b) Scalability of Star[7]

Fig. 3: Load scalability results

data is delayed and moved to a later point in the input. The maximum delay (move distance *md*) and the fraction of delayed events can be specified (standard deviation *sd*) by the user. We use $md = 5$ and $sd = 25\%$.

Reversed: Events are sent in the reverse order, with descending time-points.

Out-of-order (time-points): Entire time-points are shuffled. The monitor receives time-points (consisting of a set of events) in an arbitrary order.

Out-of-order (events): Individual events are shuffled. The monitor receives events (labeled with their time-point and time-stamp) in an arbitrary order.

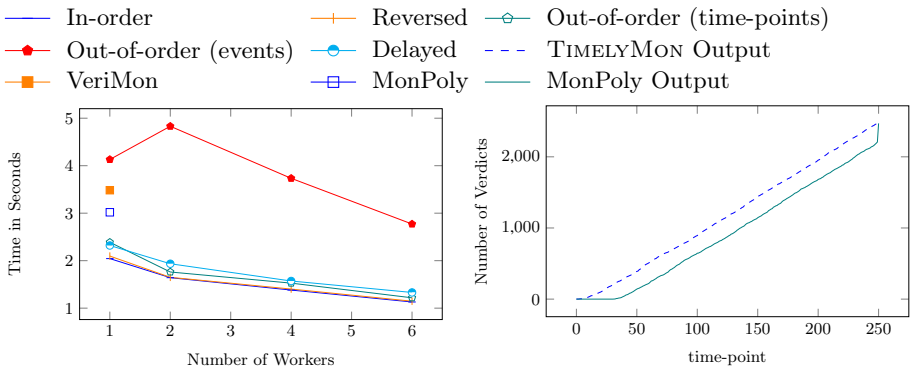
To test TimelyMon with real-world data, we used the Wikipedia API [29] that streams all changes in real-time. The produced stream is mildly out-of-order. Moreover, it does not contain watermarks so that it is not clear by how much individual events may be delayed. We reduce each JSON objects in this stream to a time-stamped event containing the event-type, username, and the bot label. We used a simple python script to interact with the API, and to sanitize and transform the data into the format suitable for TimelyMon. Our objective was to identify users that are not labeled as bots but exhibit bot-like behavior. Our policy outputs users u not labeled as bots (i.e., have label 0) that perform five edits (event *ed*) within (too) close proximity (1-2 seconds) to each other:

$$ed(u,0) \wedge \diamondsuit_{[1,2]}\ (ed(u,0) \wedge \diamondsuit_{[1,2]}\ (ed(u,0) \wedge \diamondsuit_{[1,2]}\ (ed(u,0) \wedge \diamondsuit_{[1,2]}\ ed(u,0))))$$

4.2 Data and Analysis

All experiments were run on an Apple M2 Pro 10 Core CPU at 3.30 GHz, and 16 GB of RAM. For each synthetic experiment we ran 5 executions and recorded the average execution time. For **RQ1**, we observed that there was no difference in the verdicts produced by all monitors (even though TimelyMon's verdicts were ordered differently than VeriMon's and MonPoly's as expected).

A direct comparison between TimelyMon, MonPoly, and VeriMon is only fair when considering one worker and an *in-order* event sequence, as the latter tools are restricted to this setting. TimelyMon scales worse than MonPoly but better than VeriMon in terms of trace length 3b. Therefore, for **RQ2**, we cannot claim any performance gain in equal testing scenarios (i.e., without parallelism).



(a) 250 time-points Linear[7] (b) Number of verdicts; Triangle[30]

Fig. 4: Anticipation and parallel scalability results

For **RQ3**, as shown in Figures 3a and 3b, processing time expectedly increases proportionally with increased numbers of the events. In the one worker scenario, our monitor spends (approximately) five times more time to process 250 time-points than it does for 50 time-points. On the other hand, with six workers, our monitor still has competitive execution times for 500 time-points compared to the other monitors. In response to **RQ4**, in general, increasing the number of workers resulted in a significant reduction of execution time for all benchmarks. For example, the execution time was reduced by almost 40% for the Delayed input order when increasing from 1 to 2 workers in Figure 4a.

Out of the considered tools, only TimelyMon can process events out-of-order. Regarding **RQ5**, out-of-order (events) setting presents the most challenging cases for TimelyMon, as shown in Figure 4a. In this scenario, the execution time increases significantly with a large number of time-points. The larger the set of out-of-order time-points in the partial sequence data structure, the higher the likelihood of gaps and the number of optimised updates. Apart from this corner case, TimelyMon's performance is robust with respect to delaying a fraction of events.

To answer question **RQ6**, we record the numbers of output tuples for TimelyMon and MonPoly after each time-point in the in-order experiment. Figure 4b show the result: TimelyMon is always equal or ahead of MonPoly. We computed the average, minimal and maximal difference by how much TimelyMon leads:

Formula	Average	Minimum	Maximum
Triangle[30]	247	1	302
Star[30]	240	0	292

Due to the event-wise processing and the ability to output as soon as possible, TimelyMon produces output ahead of MonPoly. We also note the advantage of our approach in the real-world scenario, where TimelyMon continuously produces output, whereas MonPoly would need to buffer, sort, and only produce output when receiving watermarks. The API has an enforced timeout; we ran TimelyMon for 13.34 minutes on 6 cores with a peak memory consumption of 54MB.

Acknowledgments This research is supported by a Novo Nordisk Fonden start package grant (NNF20OC0063462). We thank Galina Peycheva for her past-only dynamic-programming-based monitor implementation in Timely Dataflow [25], which served as the starting point for the development of TimelyMon. We thank David Basin, Srđan Krstić, and Joshua Schneider for insightful discussions about different approaches to parallel, out-of-order monitoring.

References

1. Basin, D.A., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specifications. Formal Methods Syst. Des. **49**(1-2), 75–108 (2016). https://doi.org/10.1007/s10703-016-0242-y
2. Basin, D.A., Dardinier, T., Hauser, N., Heimes, L., Huerta y Munive, J.J., Kaletsch, N., Krstić, S., Marsicano, E., Raszyk, M., Schneider, J., Tirore, D.L., Traytel, D., Zingg, S.: VeriMon: A formally verified monitoring tool. In: Seidl, H., Liu, Z., Pasareanu, C.S. (eds.) ICTAC 2022. LNCS, vol. 13572, pp. 1–6. Springer (2022). https://doi.org/10.1007/978-3-031-17715-6_1
3. Basin, D.A., Gras, M., Krstić, S., Schneider, J.: Scalable online monitoring of distributed systems. In: Deshmukh, J., Nickovic, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 197–220. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_11
4. Basin, D.A., Harvan, M., Klaedtke, F., Zalinescu, E.: Monitoring data usage in distributed systems. IEEE Trans. Software Eng. **39**(10), 1403–1426 (2013). https://doi.org/10.1109/TSE.2013.18
5. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015). https://doi.org/10.1145/2699444
6. Basin, D.A., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017). https://doi.org/10.29007/89hs
7. Basin, D.A., Klaedtke, F., Zalinescu, E.: Runtime verification over out-of-order streams. ACM Trans. Comput. Log. **21**(1), 5:1–5:43 (2020). https://doi.org/10.1145/3355609
8. Basin, D.A., Krstic, S., Schneider, J., Traytel, D.: Correct and efficient policy monitoring, a retrospective. In: André, É., Sun, J. (eds.) ATVA 2023. LNCS, vol. 14215, pp. 3–30. Springer (2023). https://doi.org/10.1007/978-3-031-45329-8_1
9. Bonakdarpour, B., Fraigniaud, P., Rajsbaum, S., Rosenblueth, D.A., Travers, C.: Decentralized asynchronous crash-resilient runtime verification. J. ACM **69**(5), 34:1–34:31 (2022). https://doi.org/10.1145/3550483
10. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015), http://sites.computer.org/debull/A15dec/p28.pdf
11. Chen, F., Rosu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer (2009). https://doi.org/10.1007/978-3-642-00768-2_23
12. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA. pp. 166–174. IEEE Computer Society (2005). https://doi.org/10.1109/TIME.2005.26

13. Falcone, Y.: On decentralized monitoring. In: Nouri, A., Wu, W., Barkaoui, K., Li, Z. (eds.) Verification and Evaluation of Computer and Communication Systems - 15th International Conference, VECoS 2021, Virtual Event, November 22-23, 2021, Revised Selected Papers. LNCS, vol. 13187, pp. 1–16. Springer (2021). https://doi.org/10.1007/978-3-030-98850-0_1
14. Hansen, E.H.P.: Streaming Algorithms for Metric First-Order Temporal Operators. Bachelor's thesis, University of Copenhagen (2022)
15. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with bdds. Formal Methods Syst. Des. **56**(1), 1–21 (2020). https://doi.org/10.1007/s10703-018-00327-4
16. Henzinger, T.A.: Half-order modal logic: How to prove real-time properties. In: Dwork, C. (ed.) PODC 1990. pp. 281–296. ACM (1990). https://doi.org/10.1145/93385.93429
17. Jannelli, V.: A White-Box Parallel Monitor for Metric First-Order Temporal Logic. Bachelor's thesis, ETH Zürich (2021)
18. Kazemlou, S., Bonakdarpour, B.: Crash-resilient decentralized synchronous runtime verification. In: 37th IEEE Symposium on Reliable Distributed Systems, SRDS 2018, Salvador, Brazil, October 2-5, 2018. pp. 207–212. IEEE Computer Society (2018). https://doi.org/10.1109/SRDS.2018.00032
19. Klaedtke, F.: Polimon: Checking temporal properties over out-of-order streams at runtime (2024)
20. Krstić, S., Schneider, J.: A benchmark generator for online first-order monitoring. In: Deshmukh, J., Nickovic, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 482–494. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_27
21. Lattuada, A.: You may not need synchronization (in streaming systems). Ph.D. thesis, ETH Zurich (2022)
22. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) SAC 2018. pp. 1925–1933. ACM (2018). https://doi.org/10.1145/3167132.3167338
23. McSherry, F.: Github: Timely dataflow, https://github.com/TimelyDataflow/timely-dataflow/
24. Murray, D.G., McSherry, F., Isard, M., Isaacs, R., Barham, P., Abadi, M.: Incremental, iterative data processing with timely dataflow. Commun. ACM **59**(10), 75–83 (2016). https://doi.org/10.1145/2983551
25. Peycheva, G.: Real-time verification of datacenter security policies via online log analysis. Master's thesis, ETH Zürich (2018)
26. Reese, L., Castro G. Silva, R., Traytel, D.: Development repository of TimelyMon (2024), https://git.ku.dk/kfx532/timelymon/-/releases/RV24_Tool_Paper
27. Schneider, J., Basin, D.A., Brix, F., Krstić, S., Traytel, D.: Scalable online first-order monitoring. Int. J. Softw. Tools Technol. Transf. **23**(2), 185–208 (2021). https://doi.org/10.1007/s10009-021-00607-1
28. Selvatici, L.: A Streaming System with Coordination-Free Fault-Tolerance. Master's thesis, ETH Zurich (2020)
29. Wikimedia Foundation, I.: Event platform/eventstreams, https://stream.wikimedia.org/v2/ui/#/?streams=mediawiki.recentchange