Scalable Online Monitoring of Distributed Systems

David Basin, Matthieu Gras, Srđan Krstić, and Joshua Schneider

Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland

Abstract. Distributed systems are challenging for runtime verification. Centralized specifications admit a global view of the system, but their semantics assumes totally-ordered observations, which are often unavailable in a distributed setting. Scalability is also problematic, especially for online first-order monitors, which must be parallelized in practice to handle the complexity of monitoring first-order specifications. We argue that scalable online monitors must ingest events from multiple sources in parallel, and propose a general model for the input of such monitors. Our model only assumes a finitely-precise global clock and allows out-of-order events, which makes it suitable for distributed systems. Based on this model, we extend our previous monitoring framework, which slices a single event stream into independently monitorable substreams. Our new framework receives events from multiple sources and slices them in parallel. We prove our extension correct and empirically show that the maximum monitoring latency significantly improves in situations where slicing was previously a bottleneck.

1 Introduction

Runtime verification (RV) is a technique that verifies systems while they run in their operational environment. RV is realized by *monitors*, which are algorithms that systematically validate a specification by searching for faults in sequences of observations recorded during system execution. The specification language used significantly influences the monitors' efficiency. Monitoring algorithms for propositional languages are very efficient and can process millions of observations per second [6,35]. However, propositional monitors are limited as they distinguish only a fixed, finite set of observations. Specifically, they cannot look for patterns that refer to the observations' parameters. Monitors for first-order languages [5,11,16,28,29,36,37,40] do not suffer from this limitation, but they must be parallelized to reach the performance of propositional monitors [7,27,36–39].

In practice, even small IT systems are often built from multiple interacting subsystems. When monitored, each subsystem provides information about its behavior as a separate observation sequence. Some approaches adopt specification languages that explicitly refer to a collection of observation sequences [21,31], or that have semantics defined on partially-ordered observations [34,41]. However, it is challenging to express global system properties using such specification languages as they couple the system's behavior with its distributed architecture. We instead focus on centralized specification languages that directly provide a global view of the distributed system. These languages often assume totally-ordered observations.

Existing monitors for centralized specifications typically verify a single observation sequence. This *single-source* design limits the monitor's applicability to distributed systems. Without additional information, the multiple observation sequences obtained from such systems induce only a partial order. Checking centralized specifications then becomes intractable, since exponentially many compatible total orders must be checked [9]. Therefore, one needs alternative solutions. Some approaches opt for a *global clock* to tag every event across every subsystem with the time of its occurrence. A global clock abstracts over a collection of local clocks that are present at each subsystem and synchronized using a clock synchronization protocol like NTP [33]. The global clock establishes a total order over observations if the local clocks are sufficiently precise and their mutual drifts are small enough. In practice, it is difficult to achieve both conditions at event-generation time, especially for distributed systems that provide large quantities of information to the monitor at high rates [18].

In this paper we develop a *multi-source* monitoring framework that takes multiple observation sequences as parallel inputs, which improves scalability. The framework can be used to monitor centralized specifications on partially-ordered observations obtained from a distributed system with a *finitely precise* global clock. The clock must only be precise enough so that the verdict for the specification in question is uniquely determined. We account for the fact that observations may have the same creation time (according to the clock) and in such cases restrict the specification language to a fragment that can be monitored [9]. Furthermore, even when the observations are totally ordered, they may be received by the framework in a different order. For centralized or single-source monitors, this can occur if the observations are transmitted over unreliable channels where messages can be delayed, dropped, or reordered [14]. Our multi-source framework faces this issue even with reliable channels, as it interleaves observations from different sources and its internal components exchange information concurrently.

We generalize the concept of a *temporal structure (TS)* (Section 3), which models totally-ordered observations, to a *partitioned temporal structure (PTS)*, which represents partially-ordered observations that may be received out-of-order from multiple sources (Section 4.1). We introduce and explain the assumptions on the observation order in a PTS, which are sufficient to uniquely determine whether the specification is satisfied.

A central part of our work is to extend our scalable monitoring framework [38, 39] with multiple input sources and a reordering step (Section 4.2). The existing framework scales up first-order monitoring tasks for specifications expressed in Metric First-Order Temporal Logic (MFOTL) [11]. The main idea is to slice the input stream into multiple substreams that can be monitored independently. Each substream is monitored in parallel by a standard first-order (sub)monitor, treated as a black box by the framework. When instantiated by a concrete submonitor, the framework becomes an online monitor. We prove that the extended multi-source framework remains sound and complete: the submonitors collectively find exactly those patterns that exist in the input PTS. We extended the implementation (Section 5) and empirically evaluated the result (Section 6). We show that it significantly improves monitoring performance.

In summary, our main contributions are: 1) the definition of the partitioned temporal structure as an input model for multi-source monitors; 2) the extension of our monitoring framework to support multiple sources; 3) its correctness proof, which has been formally verified in the Isabelle proof assistant; and 4) an empirical evaluation showing a significant performance improvement over the single-source framework. Overall, our work lays the foundations for the efficient, scalable, online monitoring of distributed systems using expressive specifications languages like MFOTL.

2 Related Work

Centralized Monitors. Our work builds on top of existing work on parallel black-box monitoring. Basin et al. [7] introduce the concept of slicing temporal structures. They provide composable operators that slice both data and time and support parallel offline monitoring using MapReduce. In prior work [38, 39], we generalized their data slicer and implemented it using the Apache Flink stream processing framework [4]. Another data-parallel approach is parametric trace slicing [36, 37], which supports a restricted form of quantification and uses slicing to improve expressivity, rather than scalability. The above-mentioned data-parallel monitors support only a single input stream.

Basin et al. [9] monitor distributed systems using a single-source, centralized monitor. They preprocess and merge locally collected traces prior to monitoring. Preprocessing assumes that observations with equal time-stamps happen simultaneously and restricts the specification to a fragment where the order of such observations does not influence the monitor's output. Our approach generalizes this idea, whereby [9] becomes a special case.

Monitors that handle missing and out-of-order observations [12, 14] are resilient to network failures, which commonly occur in large distributed systems. These centralized monitors, which support MTL and its variant with freeze quantifiers, are orthogonal to our approach and can be instantiated within our monitoring framework.

Decentralized Monitors. According to the distributed monitoring survey's terminology [25], the organization of our monitoring framework can be seen either as orchestrated or choreographed. In the survey, the notion of a global clock implies infinite precision, while we assume a more realistic global clock with finite precision. Our monitoring framework supports a more expressive specification language than the state-of-the-art alternatives reported on in the survey, which are limited to LTL and predicate detection.

Bauer and Falcone [15] exploit the locality of the observations in monitored subsystems to organize the monitors hierarchically based on the structure of an LTL formula. In contrast, our parallel monitors each monitor the same (global) formula. By decomposing the specification, Bauer and Falcone reduce the communication overhead, but the monitors still must synchronize on every time-point in the trace. Similarly, El-Hokayem and Falcone [23, 24] propose a framework for decentralised monitoring of LTL and (automata-based) regular specifications. However, they focus only on propositional specifications, which limits the expressiveness of their framework.

Leucker et al. [31] describe a concurrent online monitor for multiple non-synchronized input streams. Unlike our work, the authors assume the existence of an infinitely precise global clock and that individual streams are ordered. It is difficult to compare their specification language TeSSLa with ours. TeSSLa refers to multiple input streams directly, while our specification language specifies (global) properties of distributed systems.

Stream Processing. A common mechanism for dealing with out-of-order observations in database and stream processing systems [2, 4, 20, 32] are watermarks [3], which are special markers inserted in data streams to provide a lower bound on the progress of time. Alternatively, a slack parameter [1] can be specified, which denotes the maximum number of positions that any observation can be delayed at a stream operator. It is used to allocate an appropriately sized buffer for each input of the stream operator to perform reordering. Observations delayed more than the slack value are discarded. Punctuations [43] are more general than watermarks in that they indicate the end of

$v, i \models r(t_1, \ldots, t_n)$	if $r(v(t_1), \ldots v(t_n)) \in D_i$	$v, i \models \exists x.\, \varphi$	if $v[x \mapsto z], i \models \varphi$ for some $z \in \mathbb{D}$
$v, i \models t_1 \approx t_2$	if $v(t_1) = v(t_2)$	$v, i \models \bullet_I \varphi$	if $i > 0$, $\tau_i - \tau_{i-1} \in I$, and $v, i-1 \models \varphi$
$v, i \models \neg\varphi$	if $v, i \not\models \varphi$	$v, i \models \bigcirc_I \varphi$	if $\tau_{i+1} - \tau_i \in I$ and $v, i+1 \models \varphi$
$v, i \models \varphi \vee \psi$	if $v, i \models \varphi$ or $v, i \models \psi$		
$v, i \models \varphi \mathsf{S}_I \psi$	if $v, j \models \psi$ for some $j \leq i$, $\tau_i - \tau_j \in I$, and $v, k \models \varphi$ for all k with $j < k \leq i$		
$v, i \models \varphi \mathsf{U}_I \psi$	if $v, j \models \psi$ for some $j \geq i$, $\tau_j - \tau_i \in I$, and $v, k \models \varphi$ for all k with $i \leq k < j$		

Fig. 1. Semantics of MFOTL

some subset of the stream. The semantics of punctuations can vary, e.g., there will be no more observations having certain attribute values in the stream. Heartbeats [42] resemble watermarks and can be seen as special punctuations about temporal attribute values.

3 Preliminaries

We recap the syntax and semantics of Metric First-Order Temporal Logic [11] and summarize our scalable monitoring framework [38], which slices a single temporal structure.

Metric First-Order Temporal Logic (MFOTL). We fix a set of names $\mathbb{E}$ and for simplicity assume a single infinite domain $\mathbb{D}$ of values. The names $r \in \mathbb{E}$ have associated arities $\iota(r) \in \mathbb{N}$. An *event* $r(d_1, \ldots, d_{\iota(r)})$ is an element of $\mathbb{E} \times \mathbb{D}^*$. We further fix an infinite set $\mathbb{V}$ of variables, such that $\mathbb{V}$, $\mathbb{D}$, and $\mathbb{E}$ are pairwise disjoint. Let $\mathbb{I}$ be the set of nonempty intervals $[a,b) := \{x \in \mathbb{N} \mid a \leq x < b\}$, where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$, and $a < b$. Formulas φ are defined inductively, where t_i, r, x, and I range over $\mathbb{V} \cup \mathbb{D}$, $\mathbb{E}$, $\mathbb{V}$, and $\mathbb{I}$, respectively:

$$\varphi ::= r(t_1, \ldots, t_{\iota(r)}) \mid t_1 \approx t_2 \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \bullet_I \varphi \mid \bigcirc_I \varphi \mid \varphi \mathsf{S}_I \varphi \mid \varphi \mathsf{U}_I \varphi.$$

The temporal operators $\bullet_I$ (previous), $\bigcirc_I$ (next), S_I (since), and U_I (until) may be nested freely. We derive other operators: truth $\top := \exists x.\, x \approx x$, inequality $t_1 \not\approx t_2 := \neg(t_1 \approx t_2)$, conjunction $\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$, eventually $\Diamond_I \varphi := \top \mathsf{U}_I \varphi$, and once $\blacklozenge_I \varphi := \top \mathsf{S}_I \varphi$. The set $\mathbb{V}_\varphi$ denotes the set of free variables of φ. We restrict our attention to *bounded future formulas*, where all subformulas of the form $\bigcirc_{[a,b)} \alpha$ and $\alpha \mathsf{U}_{[a,b)} \beta$ satisfy $b < \infty$.

MFOTL formulas are interpreted over *temporal structures (TS)*, which model totally-ordered observation sequences. A temporal structure ρ is an infinite sequence $\langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$, where $\tau_i \in \mathbb{N}$ is a discrete time-stamp, and the *database* $D_i \in \mathbb{DB} = \mathcal{P}(\mathbb{E} \times \mathbb{D}^*)$ is a finite set of events that happen concurrently from the monitored system's point of view. We allow the event source to use finitely precise clocks: databases at different time-points $i \neq j$ may have the same time-stamp $\tau_i = \tau_j$. The sequence of time-stamps must be *monotone* ($\forall i.\, \tau_i \leq \tau_{i+1}$) and *progressing* ($\forall \tau.\, \exists i.\, \tau < \tau_i$).

The relation $v, i \models_\rho \varphi$ defines the satisfaction of the formula φ for a valuation v at an index i with respect to the temporal structure $\rho = \langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$; see Fig. 1. Whenever ρ is fixed and clear from the context, we omit the subscript on $\models$. The valuation v is a mapping $\mathbb{V}_\varphi \to \mathbb{D}$, assigning domain elements to the free variables of φ. Overloading notation, v is also the extension of v to the domain $\mathbb{V}_\varphi \cup \mathbb{D}$, setting $v(t) = t$ whenever $t \in \mathbb{D}$. We write $v[x \mapsto y]$ for the function equal to v, except that the argument x is mapped to y.

Monitors. An *online monitor* for a formula φ receives time-stamped databases that are a finite prefix π of some TS ρ (denoted by $\pi \prec \rho$). The monitor incrementally computes a verdict, which is a set of valuations and time-points that satisfy φ given π. (Typically, one is interested in violations of a specification $\Box\psi$, which can be obtained by monitoring

$\neg\psi$ instead.) A monitor is *sound* if the verdict for π contains (v,i) only if $v,\, i \models_\rho \varphi$ for all $\rho \succ \pi$. It is *complete* if whenever $\pi \prec \rho$ is such that $v,\, i \models_{\rho'} \varphi$ for all $\rho' \succ \pi$, then there is another prefix $\pi' \prec \rho$ for which the verdict contains (v,i). In our formal treatment, we consider the monitor's output in the limit as the input prefix grows to infinity. Thus, a monitor implements an abstract *monitor function* $\mathcal{M}_\varphi : (\mathbb{N} \times \mathbb{DB})^\omega \to \mathcal{P}((\mathbb{V}_\varphi \to \mathbb{D}) \times \mathbb{N})$ that maps a TS ρ to the union of all verdicts obtained from all possible prefixes of ρ. We shall assume that the monitor implementing $\mathcal{M}_\varphi$ is sound and complete. Since φ is assumed to have bounded future, it follows that $\mathcal{M}_\varphi(\rho) = \{(v,\, i) \mid v,\, i \models_\rho \varphi\}$.

Slicing Framework. In prior work, we parallelized online first-order monitoring by slicing [38] the temporal structure into N temporal structures that can be independently monitored. Let $[n]$ denote the set $\{1,\dots,n\}$. For a fixed formula φ and a number N of parallel submonitors, the slicer $\mathcal{S}_g$ is parameterized by a *slicing strategy* $g : [N] \to \mathcal{P}(\mathbb{V}_\varphi \to \mathbb{D})$ satisfying $\bigcup_{k\in[N]} g(k) = (\mathbb{V}_\varphi \to \mathbb{D})$. The slicing strategy specifies the set of valuations $g(k)$ for which the submonitor k is responsible. Next, we describe which events it receives to evaluate φ correctly on all $v \in g(k)$. Given an event e, let *sfmatches*(φ, e) be the set of all valuations v for which there is a predicate subformula ψ in φ with $v(\psi) = e$. (Here v is extended to predicate subformulas, such that $v(r(t_1,\dots,t_{\iota(r)})) = r(v(t_1),\dots,v(t_{\iota(r)}))$, and we assume that φ's bound variables are disjoint from its free variables.) For a database D and a set of valuations R, we write $D \downarrow R$ for the restricted database $\{e \in D \mid \textit{sfmatches}(\varphi,e) \cap R \neq \varnothing\}$. The same notation restricts TS $\rho = \langle \tau_i, D_i \rangle_{i\in\mathbb{N}}$ pointwise, i.e., $\rho \downarrow R = \langle \tau_i, D_i \downarrow R \rangle_{i\in\mathbb{N}}$. Then, it is sufficient if the submonitor k receives the slice $\mathcal{S}_{g,k}(\rho) = \rho \downarrow g(k)$.

The output of the monitor function $\mathcal{M}_\varphi$ on ρ can be reconstructed from the parallel submonitors' output on the N slices. Formally, $\mathcal{M}_\varphi(\rho) = \bigcup_{k\in[N]}(\mathcal{F}_{g,k}(\mathcal{M}_\varphi(\mathcal{S}_{g,k}(\rho))))$, where $\mathcal{F}_{g,k}(X) = X \cap (g(k) \times \mathbb{N})$. In [38], we established this fact assuming a stronger completeness property of the online monitor. However, it can also be shown for the abstract function $\mathcal{M}_\varphi$, which operates on a TS. The intersection with $g(k) \times \mathbb{N}$ is needed to avoid spurious verdicts for some formulas (e.g., those involving equality).

4 Monitoring Distributed Systems

To monitor a distributed system, we must observe the components (processes, machines) that make up the system. There is no defined order between the observations from different components unless their execution is perfectly synchronized. This cannot be assumed in general, however, because one usually desires some parallelism. Yet the semantics of centralized specifications are defined with respect to a total order on observations. Therefore, the first problem that we must solve is the lack of information to reconstruct this total order. Second, distributed systems are often developed to achieve scalability, and online monitors used with such systems should be scalable as well. A monitor that physically combines observations from different sources into a single stream cannot satisfy this requirement: if the workload increases and additional events are generated, the processes working with the single stream will eventually be overloaded. Scalable online monitors must therefore ingest observations in parallel.

We solve the above problems by viewing online monitoring as an instance of distributed stream processing. Observations enter the monitor in multiple parallel streams, called *sources*. We give a general model of sources that captures a variety of distributed monitoring scenarios, while still allowing the efficient monitoring of metric specifications (Section 4.1). The model logically decouples the monitor from the monitored system, which ensures that the system's topology can be chosen independently. We then extend the slicing framework to utilize multiple sources (Section 4.2). The resulting multi-source monitor does not require an infinitely precise global clock, and it scales better than the single-source version, even if the monitored system is not truly distributed.

4.1 Input Model

We model the monitor's input as a *Partitioned Temporal Structure (PTS)*, which we define formally later in this section. We believe that this model is useful beyond our implementation using slicing. The model is based on several assumptions about the nature of the monitoring problem at hand. Below, we explain these assumptions, show how they are reflected in the PTS, and give examples of possible applications.

Assumption 1. We consider the online monitoring of a distributed system. We assume that the given specification has the form $\Phi = \Box \neg\varphi$ and that it is centralized, i.e., its operators and predicates are interpreted over the collection of all events from the entire system.

This basic assumption rules out various monitoring approaches, such as centralized monitors. Moreover, it is not feasible to construct all possible interleavings of concurrent events. We do not rely on system-specific information, such as vector clocks, to reduce the number of interleavings [34], as this would reduce the generality of our approach. Finally, note that centralized specifications cannot be easily split into smaller parts that are handled by local monitors, as illustrated by the following example.

Example 1. Consider a microservice architecture that is running on a cluster of machines. Each service handles incoming requests, which may trigger further requests to other remote services. We want to check that every request operating on sensitive data is originally caused by a user who is authorized to work with the data. Since requests may span arbitrarily many machines, no part of the specification can be checked locally.

We therefore treat the monitored system and the monitor as independent entities. They are connected by M sources, which are parallel observation streams. The sources may correspond the monitored system's components, e.g., the services in Example 1. This is not required by the model, which we will show in a later example.

The next assumption imposes an important restriction: it must be possible to arrive at a definite monitoring verdict even if the observations are only partially ordered.

Assumption 2. There exists a TS $\rho^\star$ that describes the actual sequence of events as they occur in real time. (We do not assume that $\rho^\star$ can be observed directly.) The sources must provide sufficient information about the event order to decide whether $\rho^\star$ satisfies Φ.

Note that the system satisfies the specification Φ iff $\rho^\star$ satisfies Φ. We concretize the event ordering requirement as follows. Every observation carries an *index*, which is a natural number. The indices encode the most precise global ordering that the system provides: if the index of observation o_1 is less than the index of observation o_2, then o_1 must have happened before o_2. At one extreme, the index is simply the position of the observation in $\rho^\star$, i.e., a global sequence number. Then every specification has a definite verdict. A distributed system providing such sequence numbers would need an infinitely precise global clock, which is unrealistic. However, centralized applications, which have access to sequence numbers, can be monitored even more efficiently with a multi-source monitor.

Example 2. Kernel event tracing results in streams with high event rates [22]. We may improve the monitor's throughput by distributing the events over multiple streams (see Section 6). For a single processor, its hardware counters provide global sequence numbers.

Not all specifications can be monitored if indices are coarser than global sequence numbers. For example, the indices could be equal to the time-stamps. We follow the *collapse* approach by Basin et al. [9], where events with the same time-stamp are collapsed into a single instantaneous observation. We generalize the collapse from time-stamps to indices, which unifies the presentation. We then require that monitoring the collapsed stream results in essentially the same output as monitoring $\rho^\star$. To make this precise, we add the indices to $\rho^\star$ itself, which results in the indexed temporal structure $\hat{\rho}^\star$.

Definition 1. *An* indexed temporal structure (ITS) *is a TS over extended tuples* $\langle \alpha_i, \tau_i, D_i \rangle$, *where* $\alpha_i \in \mathbb{N}$ *are indices. Its indices must increase monotonically* ($\forall i.\ \alpha_i \le \alpha_{i+1}$)*, and indices must refine time-stamps* ($\forall i.\ \forall j.\ \alpha_i \le \alpha_j \longrightarrow \tau_i \le \tau_j$)*.*

Definition 2. *The* generalized collapse $\mathcal{C}(\hat{\rho}) = \langle \tau_i^c, D_i^c \rangle_i$ *of an ITS* $\hat{\rho}$ *is characterized by the unique monotone and surjective function* $f : \mathbb{N} \to \mathbb{N}$ *that maps (only) positions with the same index to a common value* ($\forall i.\ \forall j.\ \alpha_i = \alpha_j \longleftrightarrow f(i) = f(j)$)*. Then* $\forall i.\ \tau^c_{f(i)} = \tau_i$ *and* $\forall j.\ D^c_j = \bigcup \{D_i \mid f(i) = j\}$*. We call* $\hat{\rho}$ adequate for φ *iff* $v, i \models_{\mathcal{C}(\hat{\rho})} \varphi \longleftrightarrow (\exists j.\ f(j) = i \wedge v, j \models_\rho \varphi)$ *for all* v *and* i*, where* ρ *is obtained from* $\hat{\rho}$ *by omitting the indices.*

Since $\hat{\rho}^\star$ is the idealized totally-ordered stream, its indices must increase monotonically. Indices must refine time-stamps so that the generalized collapse can be defined. In the context of a distributed system, this means that the components must have access to a finitely precise global clock. This requirement, which may seem quite strong, is necessary when using a metric specification language such as MFOTL under Assumption 2. Note, however, that the granularity of time-stamps with respect to real time is not fixed. Time-stamps and thus indices can be quite coarse as long as it is possible to formalize the specification faithfully, and protocols like NTP provide sufficient synchronization. For instance, we could use NTP-synchronized time in seconds as indices in Example 1, even if the specification requires recent authorization (where "recent" is on the order of seconds, not microseconds). Moreover, there have been results on monitoring with imprecise time-stamps [10].

Monitoring a formula φ on the generalized collapse of an adequate ITS finds the same satisfying valuations as monitoring the ITS itself (modulo the remapping of time-points). If the indices of an ITS are global sequence numbers (e.g., $\forall i.\ \alpha_i = i$), the ITS is adequate for all φ. To get an intuition for other ITS, we focus again on the case where indices are time-stamps (*time-ITS*, $\forall i.\ \alpha_i = \tau_i$). Basin et al. [9] define the notion of *collapse-sufficient* formulas, which are essentially those formulas that can be monitored correctly on a time-based collapse. They provide an efficiently decidable fragment of formulas with this property. (More precisely, a time-ITS $\hat{\rho}$ is adequate for φ iff φ satisfies the properties ($\models\exists$) and ($\not\models\forall$) given in [9], which implies that $\Phi = \square \neg\varphi$ is collapse-sufficient.) Often, a formula can be made collapse-sufficient by replacing subformulas $\blacklozenge_{[0,t]} \alpha$ (note the interval's zero bound) with $\blacklozenge_{[0,t]} \lozenge_{[0,0]} \alpha$, and dually for $\lozenge_{[0,t]}$. More complicated replacements are needed for S and U, however.

Example 1 (continued). We formalize the specification as a collapse-sufficient formula, assuming that authorizations must happen at least one second and at most 60 seconds before use: $\Phi \equiv \square\big((\blacklozenge \lozenge_{[0,0]} \mathsf{start}(u,s)) \wedge \mathsf{use}(s,d) \longrightarrow \blacklozenge_{[1,60]} \mathsf{auth}(u,d)\big)$. The events are formalized by the predicates $\mathsf{start}(u,s)$ (user u starts a session s, i.e., a tree of related requests), $\mathsf{use}(s,d)$ (a request in s affects data d), and $\mathsf{auth}(u,d)$ (u is authorized to use d).

It is common practice in distributed systems to process, aggregate, and store logging information in a dedicated service. The observation fed to the monitor are then taken from this service. In Example 1, the microservices could first send their events to a distributed message broker such as Kafka [30]. As a result, events from different services may be interleaved before they reach the monitor. We therefore allow that individual sources provide observations in a different order than their temporal order. This generalization adds almost no complexity to the monitor's design (Section 4.2): we must anyway reorder the observations, even for correctly ordered streams, to synchronize them across sources. Handling out-of-order observations thus comes almost for free.

Assumption 3. Sources may provide observations in any order. However, the delay of each observation must be bounded.

The latter condition ensures that the monitor does not get stuck. We enforce it by adding watermarks, which are lower bounds on future indices, to the sources. Then, the observations' delay is bounded if the watermarks always eventually increase. In our implementation, watermarks are interspersed between regular observations. We simplify the formal definitions below by assuming that every database has an associated watermark, which is the one most recently seen. Note that an input model with watermarks is strictly more permissive than one without. If we know that the observations will be in the correct order, we can simply set each watermark equal to the next index.

We are now ready to give a formal definition of our input model. We recall the main idea: The monitor's input is a PTS, which partitions some ITS $\hat{\rho}^\star$ into multiple sources. If $\hat{\rho}^\star$ is adequate for the formula φ, it suffices to monitor the generalized collapse $\mathcal{C}(\hat{\rho}^\star)$ via the PTS to achieve the goal of monitoring $\rho^\star$.

Definition 3. *A* partitioned temporal structure (PTS) *is a finite list $\rho_1,\dots,\rho_M$ of $M \geq 1$ streams ρ_k, called* sources. *A source is an infinite sequence of tuples $\langle \alpha_{k,i}, \beta_{k,i}, \tau_{k,i}, D_{k,i}\rangle_{i\in\mathbb{N}}$, where $\alpha_{k,i} \in \mathbb{N}$ is an index, $\beta_{k,i} \in \mathbb{N}$ is a watermark, and $\tau_{k,i}$ and $D_{k,i}$ are as in temporal structures. For all $k \in [M]$, ρ_k must satisfy (P1) monotone watermarks ($\forall i.\, \beta_{k,i} \leq \beta_{k,i+1}$); (P2) progressing watermarks ($\forall \beta.\, \exists i.\, \beta \leq \beta_{k,i}$); (P3) watermarks bound future indices ($\forall i.\, \forall j.\, i < j \longrightarrow \beta_{k,i} \leq \alpha_{k,j}$); (P4) progressing time-stamps ($\forall \tau.\, \exists i.\, \tau \leq \tau_{k,i}$).*

A PTS $\rho_1,\dots,\rho_n$ partitions *an ITS $\langle \alpha_j, \tau_j, D_j\rangle_{j\in\mathbb{N}}$ iff (Q1) sound partition ($\forall k.\, \forall i.\, \exists j.\, \alpha_{k,i} = \alpha_j \land \tau_{k,i} = \tau_j \land D_{k,i} \subseteq D_j$); (Q2) complete partition wrt. indices ($\forall j.\, \exists k.\, \exists i.\, \alpha_{k,i} = \alpha_j \land \tau_{k,i} = \tau_j$) and events ($\forall j.\, \forall e \in D_j.\, \exists k.\, \exists i.\, \alpha_{k,i} = \alpha_j \land \tau_{k,i} = \tau_j \land e \in D_{k,i}$).*

Conditions P1–P3 have already been explained, while condition P4 is inherited from temporal structures. Conditions Q1–Q2 encode that the PTS contains the same information as the ITS. We need both completeness wrt. indices and events (Q2) because the latter is trivially true for empty databases, but we need to ensure that the corresponding index (and time-stamp) occurs in the PTS. Note that for every ITS, there is at least one PTS that partitions it into $M \geq 1$ sources: let $\langle \alpha_{k,i}, \beta_{k,i}, \tau_{k,i}, D_{k,i}\rangle = \langle \alpha_j, \alpha_j, \tau_j, D_j\rangle$ with $j = i \cdot M + k - 1$.

4.2 Slicing Framework with Multiple Sources

Figure 2 shows the slicing framework's dataflow after extending it to multiple sources (here with $M = 2$ sources and $N = 3$ submonitors). Arrows represent streams of elements, and rectangles are stream transducers with possibly multiple inputs and outputs. The

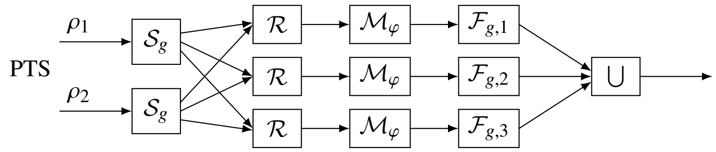

Fig. 2. Dataflow in the multi-source slicing framework (example for $M = 2$, $N = 3$)

input consists of the M sources of a PTS. We apply the slicer $\mathcal{S}_g$ independently to each source, using the given slicing strategy g. The input of $\mathcal{S}_g$ thus carries additional indices and watermarks. Since slicing only affects the databases, we can easily lift it to source streams. Let the stream $\rho_{k,k'}$ be the output of the kth slicer on its k'th outgoing edge, where $k' \in [N]$. The k'th instance of $\mathcal{R}$ (described below) receives an interleaving of the streams $\rho_{1,k'}, \ldots, \rho_{M,k'}$. Stream processor implementations usually do not guarantee a particular order for such an interleaving. This also applies to our implementation (Section 5). Therefore, we assume that the interleaving is nondeterministic, with the only guarantee being one of fairness, namely that every input stream is visited infinitely often. We further assume that the elements in the streams $\rho_{k,k'}$ are tagged with their origin k.

The crucial new component is the *reordering algorithm* $\mathcal{R}$, which fulfills two purposes. First, it collapses databases according to their indices. This has an effect only if indices are time-stamps, i.e., the underlying ITS is time-indexed. Second, $\mathcal{R}$ ensures that the input to the monitor function $\mathcal{M}_\varphi$ is sorted correctly. Even if observations arrive in the correct order at PTS sources, reordering is necessary due to the shuffling between $\mathcal{S}_g$ and $\mathcal{R}$.

The pseudocode for $\mathcal{R}$ is given in Algorithm 1. It uses two global variables, *marks* and *buffer*, both finite associative maps. The expression $\text{keys}(m)$ denotes the set of keys in the associative map m. If $x \in \text{keys}(m)$, then $m[x]$ is the unique value that m associates with x. The map *marks* stores the largest watermark seen so far for each input stream. (Recall that the input to $\mathcal{R}$ is an interleaving of one slice from each input stream.) The map *buffer* maps indices to pairs of time-stamps and databases. Intuitively, *buffer* keeps all indices that may occur in the future as the watermarks have not advanced past them.

The procedure INITIALIZE(M) is called once when the monitor starts, where M is the number of sources. The watermarks are initially zero, which is a lower bound for all indices. The procedure PROCESS(x) is called for every stream element x received by $\mathcal{R}$. The first element of the tuple $x = \langle k, \alpha, \beta, \tau, D \rangle$ identifies the source from which it originates, while the remaining elements are from the sliced PTS. Line 4 restores the invariant for *marks*. In lines 5–9, D's contents are added to the buffer. If the index α is already mapped by *buffer*, we take the union with the previously stored database to implement the collapse. Otherwise, τ and D are inserted into *buffer*. The value θ computed in line 10 is the minimum of all the latest watermarks across all inputs. By condition P3 of PTS (Definition 3), we know that all future indices that $\mathcal{R}$ will receive must be at least θ. Therefore, it is safe (only) to output everything in *buffer* with a smaller index. This happens in lines 11–13. Note that we iterate over the indices in ascending order, which ensures that the output is sorted correctly. The sequence of $\mathcal{R}$'s output elements (which are pairs of time-stamps and databases) forms the stream that is sent to the monitor $\mathcal{M}_\varphi$ in Figure 2.

The following theorem establishes the correctness of the multi-source slicing framework. It has been formalized [8] and verified in the Isabelle/HOL proof assistant.

Algorithm 1 Reordering algorithm $\mathcal{R}$

1: **procedure** INITIALIZE(M)
2: $\quad$ $marks \leftarrow \{k \mapsto 0 \mid k \in [M]\}$, $\quad$ $buffer \leftarrow \{\}$
3: **procedure** PROCESS($\langle k, \alpha, \beta, \tau, D \rangle$)
4: $\quad$ $marks[k] \leftarrow \beta$
5: $\quad$ **if** $\alpha \in \mathrm{keys}(buffer)$ **then**
6: $\quad\quad$ $\langle \tau', D' \rangle := buffer[\alpha]$
7: $\quad\quad$ $buffer[\alpha] \leftarrow \langle \tau', D \cup D' \rangle$
8: $\quad$ **else**
9: $\quad\quad$ $buffer[\alpha] \leftarrow \langle \tau, D \rangle$
10: $\quad$ $\theta := \min\{marks[k] \mid k \in \mathrm{keys}(marks)\}$
11: $\quad$ **for** $i \in \mathrm{keys}(buffer)$ in ascending order, while $i < \theta$ **do**
12: $\quad\quad$ **output** $buffer[i]$
13: $\quad\quad$ delete i from $buffer$

Theorem 1. *Let $\rho_1, \ldots, \rho_M$ be a PTS that partitions $\hat{\rho}^\star$. For all slicing strategies g, the result of the dataflow in Figure 2 (with inputs $\rho_1, \ldots, \rho_M$) is equal to $\mathcal{M}_\varphi(\mathcal{C}(\hat{\rho}^\star))$.*

Note that this theorem holds for all possible partitions of $\hat{\rho}^\star$ and all possible interleavings that can result from the shuffling step. However, it is only a statement about the infinite sequence of verdicts. Each verdict may be delayed by an arbitrary (but finite) amout of time, depending on the watermarks in the input and the shuffling implementation. Theorem 1 does not assume that $\hat{\rho}^\star$ is adequate for φ because it refers directly to the generalized collapse $\mathcal{C}(\hat{\rho}^\star)$. If we additionally know that $\hat{\rho}^\star$ is adequate, we get the same verdicts as if we were monitoring $\rho^\star$ directly, modulo the mapping of time-points.

We conclude with a remark about the time and space complexity of Algorithm 1. Both are unbounded in the worst case because of the problem with unbounded watermark delays mentioned above. However, we obtain a more meaningful result under reasonable additional assumptions. For example, assume that each database in the input has size at most d, that every index occurs at most c times, and that the number of stream elements between an index α and the time that θ (line 10) becomes greater than α is at most z. The parameter c is upper bounded by the *time-point rate* (Section 6) multiplied by M. The parameter z depends on the *watermark frequency* and the *maximum (event) delay* (Section 6), and also on the additional delay introduced by the shuffle step between slicing and reordering.

There are at most z different keys in *buffer* at any given time, each mapping to a database of size at most $c \cdot d$. The space complexity is thus $O(M + c \cdot d \cdot z)$ in the uniform RAM model, where M is the number of sources. By using a self-balancing search tree for *buffer* and hash tables for the databases contained therein, one invocation of PROCESS has an amortized average time complexity of $O(M + d + \log z)$, again in the uniform model. The summand M can be reduced to $\log M$ by using a binary heap to maintain θ instead of recomputing it in every invocation.

5 Implementation

We implemented a multi-source online monitoring framework based on the ideas outlined in Section 4. It extends our previous single-source framework [38, 39] and is available online [8]. The implementation instantiates the submonitors with MonPoly [13], which supports a monitorable fragment of MFOTL [11]. We modified about 4k lines of Scala and Java code (3.2k added and 0.8k deleted). In Section 4, we glossed over many details, e.g., how events are delivered to and exchanged within the framework, which may have an impact on the efficiency and usability of the framework. We explain some of our implementation choices here. Additional details can be found in [26, 38].

Our multi-source framework is built on top of Apache Flink [19], which provides an API and a runtime for distributed stream processing applications with optional fault tolerance. Fault tolerance is important for distributed online monitors since increasing the number of machines on which a monitor runs also increases the risk of failures, which would otherwise disrupt the monitor's operation. The implementation's dataflow corresponds roughly to the dataflow in Figure 2, except that the streams' elements are individual events instead of databases. The events are interleaved with other control elements that carry additional metadata. We use Flink's API to define the logical dataflow graph, whose vertices are operators that transform potentially unbounded data streams. At runtime, operators can have multiple instances as defined by their degree of parallelism. Each operator instance works on a partition, i.e., a substream. Stream elements are repartitioned according to some strategy if the degree of parallelism changes from one operator to the next. In Figure 2, the parallelism changes from M to N at the shuffling step. The slicers' output is a single stream of elements labeled with their destination submonitor. To ensure that the elements reach their intended destination, we use a custom repartitioning strategy that inspects these labels.

We use two types of source operators (TCP and Kafka) with different trade-offs. In Flink, sources are operators without incoming edges in the dataflow graph. Their degree of parallelism, which must be chosen before execution starts, determines the number M of input streams. The TCP source reads simple text streams from multiple sockets by connecting to a list of address and port pairs. It is fast and thus useful for benchmarking the other components, but it cannot provide fault tolerance. Apache Kafka [30] is a stream processor implementing a distributed persistent message queue. Kafka itself has a notion of stream partition, interoperable with Flink's stream partitions. The main advantage of Kafka is that it can provide exactly-once delivery, i.e., it can be guaranteed that every observation is processed exactly once, even if the monitor restarts. This is not possible with the socket source but is necessary for fault tolerance. However, Kafka incurred a significant overhead compared to TCP sockets in our experiments, which then becomes a limiting factor for the whole streaming pipeline at high event rates.

The slicer, submonitors, filtering, and verdict union are nearly unmodified (see [38]). However, there are now multiple instances of the slicing operator. The reordering function $\mathcal{R}$ is a straightforward implementation of Algorithm 1. In our implementation, the *buffer* is simply a hash table, and we access it by probing for increasing indices. A more efficient approach can be used if this becomes a bottleneck. Our implementation currently supports time-points and time-stamps as indices (see Section 4.1). With out-of-order input, only time-stamps are supported, but it should be easy to generalize the framework to time-points. We rely on *order elements*, which are a type of control elements, instead of associating watermarks with every database. For in-order inputs, the order elements are separators between databases, which are inserted by the input parser. In this case, we can synthesize the watermark from the database's time-point or time-stamp. If the

$$\varphi_s \equiv P(x,y) \land \big((\blacklozenge_{[1,3s]} Q(x,z)) \land \blacklozenge_{[1,3s]} R(x,w)\big)$$
$$\varphi_i \equiv \mathit{insert}(u,\mathsf{db1},p,d) \land d \not\approx \mathsf{null} \land \neg \lozenge_{[0,30h]} (\exists u'.\, \mathit{insert}(u',\mathsf{db2},p,d) \lor \mathit{delete}(u',\mathsf{db1},p,d))$$
$$\varphi_d \equiv \Big(\big(\mathit{delete}(u,\mathsf{db1},p,d) \land d \not\approx \mathsf{null} \land \neg \blacklozenge_{[0,30h]} \exists u'.\exists p'.\, \mathit{insert}(u',\mathsf{db1},p',d)\big) \lor$$
$$\big(\mathit{delete}(u,\mathsf{db1},p,d) \land d \not\approx \mathsf{null} \land (\exists u'.\exists p'.\, (\blacklozenge_{[0,30h]} \mathit{insert}(u',\mathsf{db1},p',d)) \lor$$
$$(\lozenge_{[0,30h]} \mathit{insert}(u',\mathsf{db2},p',d))))\Big) \land \neg \lozenge_{[0,30h]} \exists u'.\exists p'.\, \mathit{delete}(u',\mathsf{db2},p',d)$$

Fig. 3. MFOTL formulas used in the evaluation

input is out-of-order, watermarks must be provided as annotations in the input data. The input parser extracts the watermarks and embeds them in newly created order elements.

6 Evaluation

To assess the scalability of our extended framework we organized our evaluation (available online [8]) in terms of the following research questions (RQs).
RQ1: How do the input parameters affect the multi-source framework's scalability?
RQ2: What is the impact of imbalanced input sources on performance?
RQ3: Can multiple sources be used to improve monitoring performance?
RQ4: How much overhead does event reordering incur?

RQ1 and RQ2 assess the impact of input parameters (specifically, the event rate and time-point rate, defined below, as well as the number of inputs and submonitors) on our framework's performance. When events arrive out of order, we additionally control their maximum delay and the watermark frequency. We assess RQ1 by monitoring multiple traces with the same event rate, while for RQ2 we relax this restriction. RQ3 aims to evaluate the overall performance gain of introducing multiple inputs. We aim to validate our hypothesis that the slicer is no longer the performance bottleneck. We also assess the overhead introduced by the newly added reorder function (RQ4).

We run our experiments on both synthetic and real traces. The former are monitored with the collapse-sufficient formula φ_s (Figure 3), which is the common *star* database query [17] augmented with temporal operators. It contains only past temporal operators because these can be monitored more efficiently, which put a higher load on the framework's input and slicers. We have used the trace generator [38] to create random traces with configurable time span, event names, rate, parameter distribution, and time-point rate. The trace's *time span* is the difference between the highest and the lowest time-stamp in the trace. Given a trace and a time-stamp, the *event rate* is the number of events with that time-stamp, while the *time-point rate* is the number of databases with that time-stamp. We synthesize traces with same event and time-point rates at all time-stamps.

The generator is extended to determine the order in which the events are supplied to the monitor by explicitly generating the *emission time* for each event. The emission times are relative to the monitoring start time. For traces received in-order, the events' emission times correspond to their time-stamps decreased by the value of the first time-stamp in the trace. Otherwise, each event's emission time is additionally delayed by a value sampled from a truncated normal distribution $\mathcal{N}(0,\sigma^2)$ over the interval $(0,\delta_{max})$. In our experiments we fix $\sigma = 2$ and vary the *maximum delay* δ_{max} of events. The generator also adds a watermark after fixed time-stamp increments called *watermark periods*.

Experiment groups	$Synthetic_1$	$Synthetic_2$	$Synthetic_3$	$Nokia_1$	$Nokia_2$
Input type	Sockets	Sockets	Sockets	Sockets	Sockets
Formulas	φ_s	φ_s	φ_s	φ_i, φ_d	$\neg\top$
Source distribution	uniform	uniform	uniform, $\left(\frac{2}{3}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}\right)$, $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{6}, \frac{1}{6}\right)$	uniform	uniform
Event order	total, partial	partial	partial	partial	partial
Ingestion order	in-order	out-of-order	in-order	in-order	in-order
No. of input sources	1, 2, 4	1, 2, 4	4	1, 2, 4	1, 2, 4
No. of submonitors	16	16	16	1, 4, 16	16
Acceleration	1	1	1	3k, 5k, 7k	3k, 5k, 7k
Trace time span	60s	60s	60s	a one day fragment from the Nokia trace with 9.5 million events	
Event rate $(1/s)$	500k, 700k, 900k	900k	500k, 700k, 900k		
Time-point rate $(1/s)$	1, 2k, 4k	1	1		
Maximum delay (s)	n/a	1, 2, 4	n/a		
Watermark period (s)	n/a	1, 2, 4	n/a		
Use reorder function	✓	✓	✓	✓	✓, ✗

Table 1. Summary of the parameters used in the experiments

Besides the synthetic traces, we also use a real system execution trace from Nokia's Data Collection Campaign [9]. The trace captures how Nokia's system handled the campaign's data. Namely, it collected phone data of 180 participants and propagated it through three databases: db1, db2, and db3. The data was uploaded directly to db1, while the system periodically copied the data to db2, where data was anonymized and copied to db3. The participants could query and delete their own data stored in db1. The system must propagate the deletions to all databases, which is formalized by formulas φ_i and φ_d (Figure 3). Since the trace spans a year, to evaluate our tool in a reasonable amount of time, we pick a one day fragment (starting at time-stamp 1282921200) containing roughly 9.5 million events with a high average event rate of about 110 events per second.

To perform online monitoring, we use a replayer tool [38] that emits the trace in real time based on its time-stamps or (the generated) explicit emission times. The tool can also be configured to accelerate the emission of the trace proportionally to its event rate, which allows for a meaningful performance evaluation since the trace characteristics are retained. For our multi-source monitor we use one replayer instance per input source. The k input sources are obtained by assigning each event to one of the sources based on a discrete probability distribution called *source distribution*, e.g., for $k = 4$, the source distribution $\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right)$ is the uniform distribution. Both the Nokia and the synthetic traces have explicit time-points, which are used as the partitions' indices. To simulate partially-ordered events we replace the indices with the appropriate time-stamps.

Table 1 summarizes the parameters used in all our experiments. There are six groups of experiments: three using the synthetic and three using the Nokia traces. We perform a separate monitoring run for each combination of parameters within one group.

We used a server with two sockets, each containing twelve Intel Xeon 2.20GHz CPU cores with hyperthreading. This effectively supports up to 48 independent parallel computations. We measure the worst-case latency achieved during our experiments.

In general, monitor latency is the difference between the time a monitor consumes an event and the time it is done processing the event. Thus, at regular intervals, the replayer

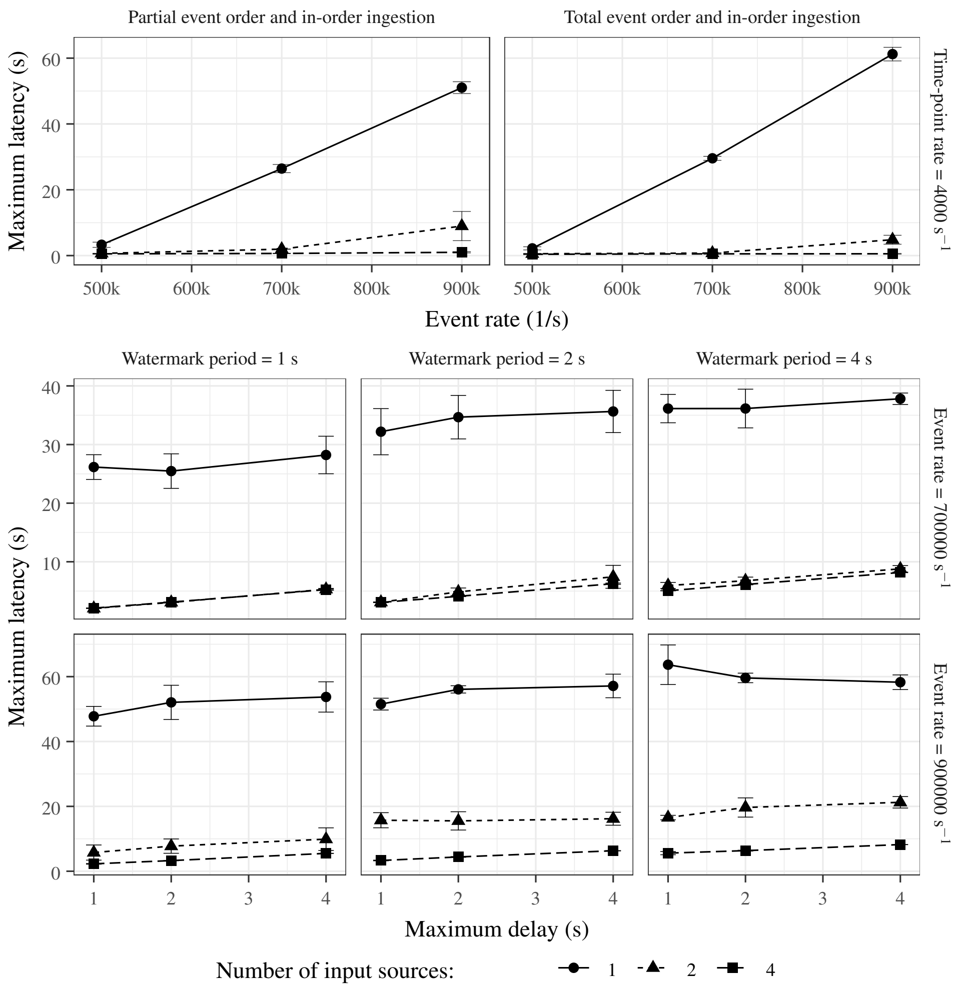

Fig. 4. Results of the *Synthetic*$_1$ (first row) and *Synthetic*$_2$ (second and third row) experiment groups

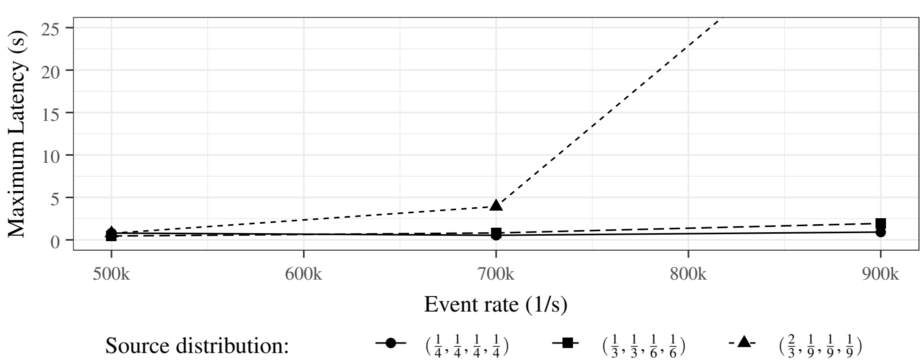

Fig. 5. Results of the *Synthetic*$_3$ experiment group

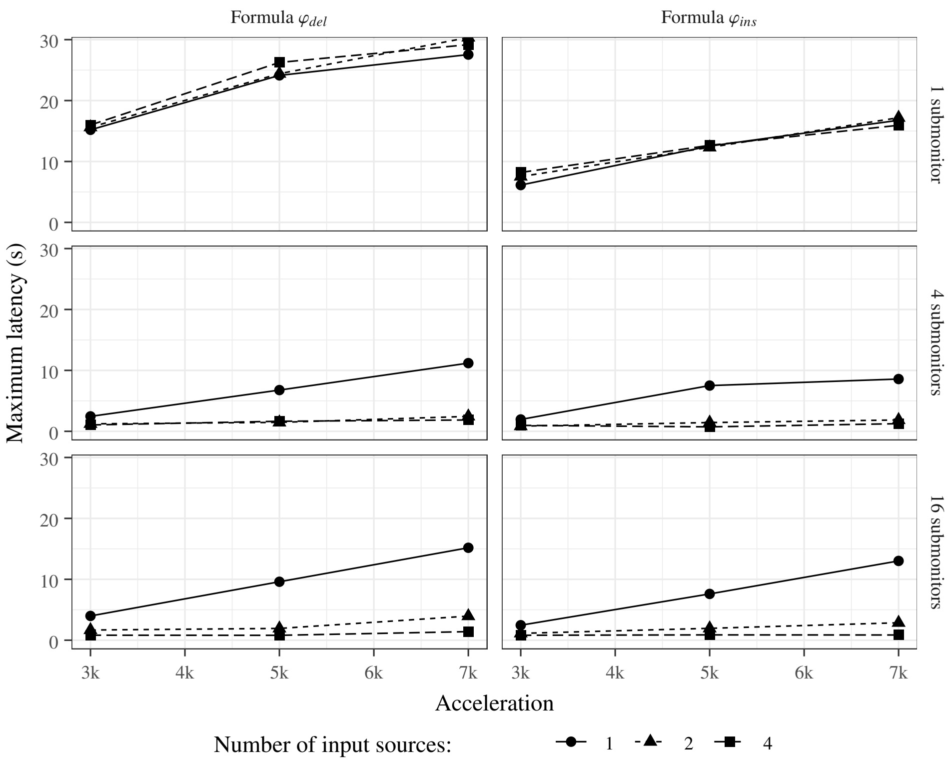

Fig. 6. Results of the *Nokia*$_1$ experiment group

injects a *latency marker*, which is a special event tagged with its creation time and a sequence number local to its source. Each such marker is then propagated by our framework, preserving its order relative to other events from the same input source. It is treated as part of the preceding event, effectively measuring its processing time. The slicers duplicate and forward latency markers to all parallel submonitors, such that each submonitor receives every latency marker from each source. Finally, for every sequence number, the last operator in the framework aggregates all latency markers (coming both from the different input sources and the different parallel submonitors) and calculates the worst-case latency. For a single monitoring run, we report the maximum of the worst-case latency aggregated over the entire run. To avoid spurious latency spikes, some experiments are repeated up to 10 times and the mean value is reported with error bars showing two standard errors.

The results of our experiments are shown in Figures 4–7. The experiments *Synthetic*$_1$ and *Synthetic*$_2$ (Figure 4) answer RQ1. Increasing the number of input sources decreases the worst-case latency, which is particularly evident with high event rates. For instance, when monitoring traces with event rate 900k, we improve the maximum latency by 10 seconds if we double the number of input sources. The relationship between the maximum event rate at a fixed latency and the number of sources appears to be slightly sublinear. We conjecture that this is due to duplicate events that the slicers necessarily emit for some formulas [38]. Therefore, having more slicers increases the framework's total load.

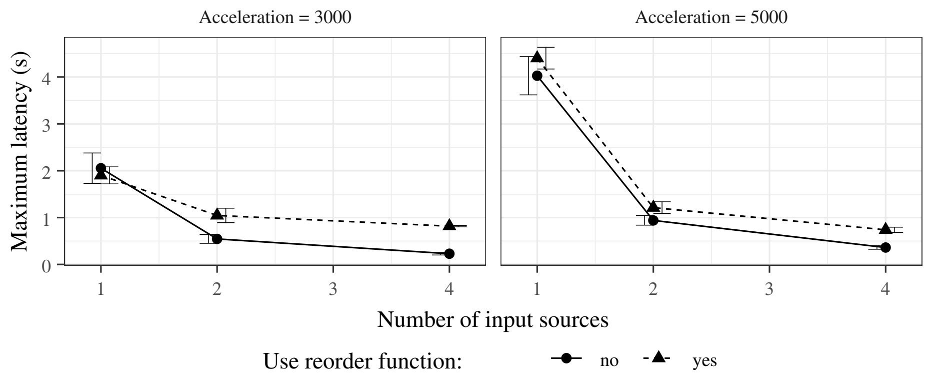

Fig. 7. Results of the *Nokia*$_2$ experiment group

As expected, *Synthetic*$_2$ shows that the watermark period and the maximum delay establish a lower bound on the maximum latency. These parameters determine the minimum amount of time the reorder function must buffer out-of-order events, which our latency measurements capture. We note that the time-point rate has not influenced the monitoring performance in our experiments; we therefore omitted the plots that show different time-point rates and fix the time-point rate to 4000 in *Synthetic*$_1$.

RQ2 is answered by experiment *Synthetic*$_3$ (Figure 5) where we fix the number of input sources to 4 and change the source distribution. The maxium latency is only affected for high event rates and highly skewed source distributions (i.e., when most of the events belong to a single source). Otherwise, our framework shows robust performance.

The results of *Nokia*$_1$ (Figure 6) answer RQ3 and validate our hypothesis that increasing the number of sources can improve monitoring performance in realistic monitoring use cases. Increasing the number of sources is ineffective only when parallel submonitors are themselves the performance bottleneck (e.g., when using only one submonitor).

In *Nokia*$_2$ we monitor the Nokia trace without using the reorder function (RQ4). To retain soundness, we monitor the formula $\neg\top$. The experiment shows that the reorder function introduces negligible overhead: less than 1 second of maximum latency.

7 Conclusion

We have developed the first scalable online monitor for centralized, first-order specifications that is able to efficiently monitor executions of distributed systems. Specifically, we have defined a partitioned temporal structure (PTS) that models an execution of a distributed system, i.e., a sequence of partially-ordered observations received out-of-order. We have extended our monitoring framework to support multiple sources and proved its correctness. Moreover, we empirically show a significant performance improvement over the framework's single-source variant. For example, in our experiments with real data, we were able to more than double the event rate, from an average of about 330k to 770k events per second by using two sources instead of one, while achieving the same maximum latency. As future work, we plan to combine our framework with monitors that inherently support out-of-order observations [14] and make our (now parallel) slicing adaptive [39] with respect to changes in the trace characteristics.

Acknowledgment. We thank Dmitriy Traytel for his helpful comments on a draft of this paper. This research is supported by the US Air Force grant "Monitoring at Any Cost" (FA9550-17-1-0306) and by the Swiss National Science Foundation grant "Big Data Monitoring" (167162).

References

1. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: a new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003)
2. Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S.: Millwheel: Fault-tolerant stream processing at internet scale. Proc. VLDB Endow. **6**(11), 1033–1044 (2013)
3. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The Dataflow Model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proc. VLDB Endow. **8**(12), 1792–1803 (2015). https://doi.org/10.14778/2824032.2824076
4. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The Stratosphere platform for big data analytics. VLDB J. **23**(6), 939–964 (2014)
5. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified Event Automata: Towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer (2012)
6. Basin, D., Bhatt, B.N., Krstić, S., Traytel, D.: Almost event-rate independent monitoring. Form. Methods Syst. Des. **54**(3), 449–478 (2019)
7. Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specifications. Form. Methods Syst. Des. **49**(1-2), 75–108 (2016)
8. Basin, D., Gras, M., Krstić, S., Schneider, J.: Implementation, experimental evaluation, and Isabelle/HOL formalization associated with this paper. `https://polybox.ethz.ch/index.php/s/z8644ywXIak9qAa` (2020)
9. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: Monitoring data usage in distributed systems. IEEE Trans. Software Eng. **39**(10), 1403–1426 (2013)
10. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: On real-time monitoring with imprecise timestamps. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 193–198. Springer (2014)
11. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
12. Basin, D., Klaedtke, F., Zălinescu, E.: Failure-aware runtime verification of distributed systems. In: Harsha, P., Ramalingam, G. (eds.) FSTTCS 2015. LIPIcs, vol. 45, pp. 590–603. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2015)
13. Basin, D., Klaedtke, F., Zălinescu, 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)
14. Basin, D., Klaedtke, F., Zălinescu, E.: Runtime verification over out-of-order streams. ACM Trans. Comput. Log. **21**(1), 5:1–5:43 (2020)
15. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Form. Methods Syst. Des. **48**(1-2), 46–93 (2016)

16. Bauer, A., Küster, J., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer (2013)
17. Beame, P., Koutris, P., Suciu, D.: Communication steps for parallel query processing. J. ACM **64**(6), 40:1–40:58 (2017)
18. Becker, D., Rabenseifner, R., Wolf, F., Linford, J.C.: Scalable timestamp synchronization for event traces of message-passing applications. Parallel Comput. **35**(12), 595–607 (2009)
19. 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)
20. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: Flumejava: easy, efficient data-parallel pipelines. In: Zorn, B.G., Aiken, A. (eds.) PLDI 2010. pp. 363–375. ACM (2010)
21. 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: TIME 2005. pp. 166–174. IEEE Computer Society (2005)
22. Desnoyers, M., Dagenais, M.R.: The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In: OLS 2006 (Ottawa Linux Symposium). pp. 209–224 (2006)
23. El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: Bultan, T., Sen, K. (eds.) ISSTA 2017. pp. 372–375. ACM (2017)
24. El-Hokayem, A., Falcone, Y.: On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. ACM Trans. Softw. Eng. Methodol. **29**(1), 1:1–1:57 (2020)
25. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification, LNCS, vol. 10457, pp. 176–210. Springer (2018)
26. Gras, M.: Scalable Multi-source Online Monitoring. Bachelor's thesis, ETH Zürich (2020)
27. Hallé, S., Khoury, R., Gaboury, S.: Event stream processing with multiple threads. In: Lahiri, S.K., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 359–369. Springer (2017)
28. Havelund, K., Peled, D., Ulus, D.: First order temporal logic monitoring with BDDs. In: Stewart, D., Weissenbacher, G. (eds.) FMCAD 2017. pp. 116–123. IEEE (2017)
29. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Lectures on Runtime Verification, LNCS, vol. 10457, pp. 61–102. Springer (2018)
30. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB. vol. 11, pp. 1–7 (2011)
31. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Runtime verification of real-time event streams under non-synchronized arrival. Software Quality Journal (2020)
32. Miao, H., Park, H., Jeon, M., Pekhimenko, G., McKinley, K.S., Lin, F.X.: Streambox: Modern stream processing on a multicore machine. In: Silva, D.D., Ford, B. (eds.) USENIX ATC 2017. pp. 617–629. USENIX Association (2017)
33. Mills, D.L.: Internet time synchronization: The network time protocol. RFC **1129**, 1 (1989). https://doi.org/10.17487/RFC1129
34. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: IPDPS 2015. pp. 494–503. IEEE Computer Society (2015)
35. Raszyk, M., Basin, D., Krstić, S., Traytel, D.: Multi-head monitoring of metric temporal logic. In: Chen, Y., Cheng, C., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 151–170. Springer (2019)
36. Reger, G., Rydeheard, D.E.: From first-order temporal logic to parametric trace slicing. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 216–232. Springer (2015)
37. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Log. Methods Comput. Sci. **8**(1) (2012)
38. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Scalable online first-order monitoring. In: Colombo, C., Leucker, M. (eds.) Runtime Verification. pp. 353–371. Springer (2018)

39. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Adaptive online first-order monitoring. In: Chen, Y., Cheng, C., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 133–150. Springer (2019)
40. Schneider, J., Basin, D., Krstić, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 310–328. Springer (2019)
41. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) ICSE 2004. pp. 418–427. IEEE Computer Society (2004)
42. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: Beeri, C., Deutsch, A. (eds.) PODS 2004. pp. 263–274. ACM (2004)
43. Tucker, P.A., Maier, D.: Exploiting punctuation semantics in data streams. In: Agrawal, R., Dittrich, K.R. (eds.) ICDE 2002. p. 279. IEEE Computer Society (2002)