Adaptive Online First-Order Monitoring

Joshua Schneider, David Basin, Frederik Brix, Srđan Krstić, and Dmitriy Traytel

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

Abstract. Online first-order monitoring is the task of detecting temporal patterns in streams of events carrying data. Considerable research has been devoted to scaling up monitoring using parallelization by partitioning events based on their data values and processing the partitions concurrently. To be effective, partitioning must account for the event stream's statistics, e.g., the relative event frequencies, and these statistics may change rapidly. We develop the first parallel online first-order monitor capable of adapting to such changes. A central problem we solve is how to manage and exchange states between the parallel executing monitors. To this end, we develop state exchange operations and prove their correctness. Moreover, we extend the implementation of the MonPoly monitoring tool with these operations, thereby supporting parallel adaptive monitoring, and show empirically that adaptation can yield up to a tenfold improvement in run-time.

1 Introduction

Online monitoring is a well-established runtime verification approach. System requirements are formalized as properties of an event stream that represents observations of a running system's behavior. An online monitor detects property violations in the event stream.

In practice, monitors must cope with high-volume and high-velocity event streams arising in large-scale applications. To meet these scalability demands, researchers have exploited parallel computing infrastructures [6, 11, 12, 15, 22–24], e.g., by splitting (or slicing) the event stream into smaller substreams that can be processed independently by monitors acting as black boxes. However, since monitoring is not an embarrassingly parallel task, slicing may need to duplicate events. Another performance bottleneck arises from slices with significantly more events than others. In prior work [24], we reduced duplication and distributed events evenly by leveraging insights from database research [1, 10]. We gave an algorithm that slices based on the event stream's characteristics, like the relative rates of event types or data values occurring disproportionately frequently in events. Provided the stream's characteristics are known and stable, this approach scales well.

Example 1. Consider a (simplified) policy for a document management system: *a document must be updated to its latest revision before being sent to a user*. The event stream contains *update* events parameterized with a document ID and *send* events parametrized with a document ID and a user ID. The above policy relates *update* and *send* events with the same document ID value and specifies that the former must precede the latter.

Let us first assume that we observe many update events to different documents. Then it makes sense to split the event stream based on a partition of the document ID values. Each parallel monitor instance (*submonitor*) would receive *send* and *update* events with document ID values from one partition. However, such a slicing strategy would not yield balanced substreams if the event stream changes to consist exclusively of send events that all have the same document ID, e.g., 955; this may occur, for example, if this is an

important document sent to many users. Only one submonitor will continue receiving events. To counter this suboptimal utilization, we would like to partition the user IDs instead of the document IDs. To continue outputting correct verdicts, the state of the submonitor that was previously responsible for the document ID 955 and thus was the only one to observe its update events in the past, must be transferred to all other submonitors.

The example illustrates that a slicing strategy based on outdated stream characteristics may lead to unbalanced substreams, degrading the monitor's performance. Hence, the slicing strategy used must be changed during monitoring to adapt to the stream's changing characteristics. Furthermore, this adaptation necessitates that the submonitors exchange parts of their state and thus can no longer be treated as black boxes.

Contribution. In this paper, we design, prove correct, implement, and evaluate state migration functionality for a monitor for properties expressed in the monitorable fragment of metric first-order temporal logic (MFOTL) [7]. This rich specification language (Section 2.1) can express complex dependencies between data values coming from different events in the stream.

We significantly extend an existing stream slicing framework (Section 2.2) with the ability to dynamically change the slicing strategy (Section 3). Moreover, we develop operations to migrate the state of a simplified MFOTL monitor (Section 4), modeled after the state-of-the-art monitor MonPoly [7, 8]. Concretely, we provide two operations that together achieve the state exchange: split for splitting a monitor's state according to a new slicing strategy and merge for combining parts of the states coming from different submonitors. These operations are conceptually straightforward. For example, split partitions the monitor's state based on the data values it stores. However, the operations' interaction with the monitor's invariants is intricate. To establish correctness, we have mechanically checked our proofs using the Isabelle proof assistant. A separate paper reports on our related formal verification of the simplified MFOTL monitor [26], which we extend here. We have also extended MonPoly with implementations of split and merge and evaluated its performance with adaptive slicing strategies (Section 5). Our formalization and the evaluation are available online [25].

In summary, our main contributions are: (1) the development of an abstract framework for adaptive monitoring; (2) the design of state migration operations for an MFOTL monitor; and (3) the implementation and evaluation of state migration in MonPoly. Our evaluation shows how adaptivity can substantially improve monitoring performance and enable the monitoring of high-velocity event streams.

Related Work. Basin et al. [6] introduce the concept of slicing for MFOTL monitors. They provide composable operators that slice both data and time and support scalable monitoring on a MapReduce infrastructure. Another data-parallel approach is parametric trace slicing [22, 23], which supports only a restricted form of quantification and focuses on expressiveness, rather than on scalability or performance. Other monitors [3, 5, 11, 12, 19] decompose the specification for task-parallel execution, which limits their scalability. In prior work [24], we generalized Basin et al.'s data slicing [6] and implemented it using the Apache Flink stream processing framework [2]. The above works are limited in that they consider a single static strategy only. We develop a mechanism that lifts this restriction for first-order monitoring, making it possible to react to changes in the event streams. Note that we do not tackle the orthogonal problems of deciding *when* to change

the slicing strategy and finding the *best* strategy for a given stream. The former requires a state migration mechanism already in place, while the latter requires deciding MFOTL, since for an unsatifiable formula, the best strategy is to drop all events.

Stream processing systems implement generic operations on data streams. They achieve scalability by exploiting data parallelism. The Flux operator [27] redistributes values between two parallel stages of a data stream pipeline. It adaptively changes the routing if long-term imbalances are detected. This requires state migration for downstream operators whose state must be consistent with the incoming data. Flux specifies an abstract interface to extract and implant state partitions, which our splitting and merging functions implement for a concrete monitoring operator. Megaphone [16] is a refined mechanism for state migration in Timely Dataflow [21]. Unlike other approaches, it avoids stopping the execution and excessive data duplications during migration. The mechanism is generic, hence a viable low-level streaming abstraction for our work.

Other works study adaptive controllers for distributed stream processing. The scheduler by Aniello et al. [4] continuously optimizes a task topology based on CPU load and communication traffic measurements. The granularity of tasks is much coarser than the data parallelism in our slicing approach. The DS2 controller [17] performs dynamic scaling, i.e., it selects an optimal degree of parallelism, which is orthogonal to the question of *how* to parallelize a task such as monitoring. DS2 assumes that every event can be partitioned based on a single key, which is not the case for MFOTL monitoring. In the context of complex event processing, Mayer et al. [20] optimize the assignment of overlapping temporal windows to machines. Their controller must determine the target machine at the start of each window because windows cannot be migrated in their model. A generic algorithm for deciding when to trigger adaptation is described by Kolchinsky et al. [18].

The Squall engine [28] implements various parallel join operators on data streams, including the (hash-)hypercube scheme [1]. This scheme, which we have also applied to monitoring [24], yields an optimal slicing strategy for conjunctive database queries [10]. The theta join operator by Elseidy et al. [14] can migrate its state with minimal overhead and without blocking, but only at the cost of relaxing the state's consistency. In comparison to all of these other stream processing systems, our approach supports adaptation for a much more expressive specification language (MFOTL), albeit with a larger overhead.

2 Preliminaries

We recap the syntax and semantics of metric first-order temporal logic (MFOTL) [7] and an approach to its parallel monitoring based on slicing event streams [24].

2.1 Metric First-Order Temporal Logic

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 call $1, \ldots, \iota(r)$ the *attributes* of the name r. 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 constructed inductively, where t_i, r, x, and I range over $\mathbb{V} \cup \mathbb{D}$, $\mathbb{E}$, $\mathbb{V}$, and $\mathbb{I}$, respectively:

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

$v, i \models r(t_1, \ldots, t_n)$	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 $v, i+1 \models \varphi$
$v, i \models \varphi \lor \psi$	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

Along with Boolean operators, MFOTL includes the metric past and future temporal operators $\bullet$ (*previous*), S (*since*), $\bigcirc$ (*next*), and U (*until*), which may be nested freely. We define other standard Boolean and temporal operators in terms of this minimal syntax: truth $\top := \exists x.\ x \approx x$, falsehood $\bot := \neg\top$, conjunction $\varphi \land \psi := \neg(\neg\varphi \lor \neg\psi)$, universal quantification $\forall x.\ \varphi := \neg(\exists x.\ \neg\varphi)$, eventually $\Diamond_I\, \varphi := \top\, \mathsf{U}_I\, \varphi$, always $\Box_I\, \varphi := \neg \Diamond_I \neg\varphi$, once $\blacklozenge_I\, \varphi := \top\, \mathsf{S}_I\, \varphi$, and historically $\blacksquare_I\, \varphi := \neg \blacklozenge_I \neg\varphi$. Abusing notation, $\mathbb{V}_\varphi$ denotes the set of free variables of the formula φ. We restrict our attention to *bounded future formulas*, i.e., those in which all subformulas of the form $\bigcirc_{[a,b)}\, \alpha$ and $\alpha\, \mathsf{U}_{[a,b)}\, \beta$ satisfy $b < \infty$.

MFOTL formulas are interpreted over streams of time-stamped events. We group finite sets of events that happen concurrently (from the event source's point of view) into *databases* $\mathbb{DB} = \mathcal{P}(\mathbb{E} \times \mathbb{D}^*)$. An (*event*) *stream* ρ is thus an infinite sequence $\langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$ of databases $D_i \in \mathbb{DB}$ with associated time-stamps τ_i. We assume discrete time-stamps, modeled as natural numbers $\tau \in \mathbb{N}$. We allow the event source to use a finer notion of time than the one used as time-stamps. In particular, databases at different indices $i \neq j$ may have the same time-stamp $\tau_i = \tau_j$. The sequence of time-stamps must be non-strictly increasing ($\forall i.\ \tau_i \leq \tau_{i+1}$) and always eventually strictly increasing ($\forall \tau.\ \exists i.\ \tau < \tau_i$).

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 stream $\rho = \langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$; see Fig. 1. Whenever ρ is fixed and clear from the context, we omit the subscript on $\models$. The valuation v is a mapping $\mathbb{V} \to \mathbb{D}$, assigning domain elements to the free variables of φ. Overloading notation, v is also the extension of v to the domain $\mathbb{V} \cup \mathbb{D}$, 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.

An *online monitor* for a formula φ receives time-stamped databases that incrementally extend a finite stream prefix π. It computes a stream of verdicts, i.e., the valuations and time-points that satisfy φ given π. (Typically, one is interested in violations, but they can be obtained by monitoring the negated formula instead.) Formally, it implements a *monitor function* $\mathcal{M}_\varphi : (\mathbb{N} \times \mathbb{DB})^* \to \mathcal{P}((\mathbb{V} \to \mathbb{D}) \times \mathbb{N})$ that maps π to the set of all verdicts output by the monitor after observing π. The monitor function must satisfy

$$\begin{aligned} \text{Monotonicity: } & \forall \pi, \pi'.\ \pi \preceq \pi' \implies \mathcal{M}_\varphi(\pi) \subseteq \mathcal{M}_\varphi(\pi') \\ \text{Soundness: } & \forall \pi.\ \mathcal{M}_\varphi(\pi) \subseteq \{(v, i) \mid i \leq |\pi| \land \forall \rho \succeq \pi.\ v, i \models_\rho \varphi\} \\ \text{Completeness: } & \forall \pi, \rho, i.\ \pi \preceq \rho \land i \leq |\pi| \land (\forall \rho' \succeq \pi.\ v, i \models_{\rho'} \varphi) \implies \\ & \exists \pi' \preceq \rho.\ (v, i) \in \mathcal{M}_\varphi(\pi'), \end{aligned}$$

where $\preceq$ denotes the prefix-of relation both between stream prefixes and between stream prefixes and infinite streams. Monotonicity prohibits the monitor from retracting its verdicts. Soundness requires it to only output satisfying valuations. Completeness forces the monitor to eventually output them. Monitor functions are not unique because we allow satisfactions to be emitted later than the point at which they become certain.

2.2 Slicing Framework

In prior work, we parallelized an online monitor by slicing the event stream into N substreams that can be independently monitored [24]. For a fixed formula φ, their (*joint data*) *slicer* is parameterized by a *slicing strategy* $f : (\mathbb{V}_\varphi \to \mathbb{D}) \to (\mathcal{P}(\{1,\ldots,N\}) - \{\varnothing\})$, which specifies which of the N submonitors are responsible for processing a given valuation. Thus, each submonitor indexed by $k \in \{1,\ldots,N\}$ is responsible for a subset $f[k] = \{v \mid k \in f(v)\}$ of φ's valuations, called a *slice*. Because $f(v) \neq \varnothing$, for every valuation there is at least one slice responsible for it.

We focus on slicing strategies that consider each variable in $\mathbb{V}_\varphi$ separately. Assume that N is a product $\prod_{x\in\mathbb{V}_\varphi} p_x$ of positive integers p_x. We say that the variable x is sliced if $p_x > 1$. In this case, we must provide a partitioning function $f_x : \mathbb{D} \to \{1,\ldots,p_x\}$, e.g., a hash function. The resulting slicing strategy is $f(v) = \{q(\langle f_{x_1}(v(x_1)),\ldots,f_{x_n}(v(x_n))\rangle)\}$, where q is a bijection between the Cartesian product $\prod_x\{1,\ldots,p_x\}$ and $\{1,\ldots,N\}$.

Which events must the submonitor k receive? We assume that φ's bound variables are disjoint from its free variables. Given an event $e = r(d_1,\ldots,d_n)$, $\mathrm{matches}(\varphi,e)$ is the set of all valuations $v \in \mathbb{V}_\varphi \to \mathbb{D}$ for which there is a subformula $r(t_1,\ldots,t_n)$ in φ where $v(t_i) = d_i$ for all $i \in \{1,\ldots,n\}$. For a database D and a set of valuations R, called a *restriction*, we write $D \downarrow R$ for the *restricted* database $\{e \in D \mid \mathrm{matches}(\varphi,e) \cap R \neq \varnothing\}$. The same notation restricts streams $\rho = \langle \tau_i, D_i\rangle_{i\in\mathbb{N}}$ pointwise, i.e., $\rho \downarrow R = \langle \tau_i, D_i \downarrow R\rangle_{i\in\mathbb{N}}$ (analogously for stream prefixes π). The submonitor k receives the stream prefix $\pi \downarrow f[k]$.

The output of a single monitor after processing π can be reconstructed from the submonitors' joint output: $\mathcal{M}_\varphi(\pi) = \bigcup_{k\in\{1,\ldots,N\}}(\mathcal{M}_\varphi(\pi \downarrow f[k]) \cap (f[k] \times \mathbb{N}))$. (In prior work [24], we established this fact assuming a stronger completeness property. However the weaker formulation used in this paper suffices.) The intersection with $f[k] \times \mathbb{N}$ is needed to avoid spurious verdicts for some formulas, notably those involving equality.

3 Adaptive Slicing

The slicing approach to scalable monitoring achieves good performance only if the events are distributed evenly and with minimal duplication [24]. Therefore, it is crucial to choose a good slicing strategy, which usually depends on the statistics of the events in the stream.

Consider the following extension of the Example 1. Documents now depend on resources, which may be modified over time. Any document d sent out by the system must be based on the latest version of the resources it depends on. Events $dep(d,r)$ define dependencies between documents d and resources r. We assume for simplicity that dependencies are never removed. An event $mod(r)$ indicates that the resource r has been modified. The MFOTL formula corresponding to the negation of this policy is

$$send(d) \wedge (\blacklozenge\, dep(d,r)) \wedge \neg\big(\neg mod(r)\ \mathsf{S}\ (update(d) \wedge \blacklozenge\, dep(d,r))\big). \tag{1}$$

Both variables d and r can be used for slicing. Some predicates do not refer to r (e.g., $send(d)$), while $mod(r)$ does not refer to d. Therefore, any slicing strategy will result in some duplicated events. If documents are delivered much more frequently than resources are modified, it makes sense to slice only d. This would distribute the bulk of *send* events as much as possible, while having a negligible overhead due to duplicated *mod* events.

In contrast, we should slice r if modifications are much more frequent. Thus, the optimal strategy is influenced by the relative frequency of the different event types.

The optimal strategy can vary for different parts of the stream if statistics, like relative frequencies, change. Yet the existing slicing framework (Section 2.2) is parametrized by a fixed strategy. We propose a generalization where the strategy may change. In a setting with varying statistics, our generalization can lead to a substantially lower maximum load for the parallel monitors than what any fixed strategy can achieve. As an extreme case, consider a temporal separation between events pertaining to documents and resources in the above example. For instance, 1000 document deliveries might alternate with 1000 resource modifications. On average, both event types are equally likely. Hence, the strategy selected by the existing framework achieves a maximum load of around $1/\sqrt{N}$ for N parallel monitors, e.g., 25% for $N = 16$. This strategy, based on the hypercube algorithm [1], hashes the value assigned by an event to one of the two variables into $\sqrt{N}$ buckets. Each bucket pair identifies a slice, and every event is replicated to all slices with a compatible bucket assignment. However, alternating between slicing one of the variables, according to which event is currently present, yields a maximum load of $1/N$ (6.25% for $N = 16$).

Unfortunately, adjusting the slicing strategy in the middle of a stream may result in incorrect verdicts. Monitors for temporal specifications keep state that depends on previously observed events. If the subset of events sent to a monitor instance changes, its state becomes inconsistent with respect to that subset. Continuing with the example, let there be two slices, A and B. At time t, resource events are distributed by slicing variable r, and in particular the event $mod(123)$ is routed to slice A. At time $t' > t$, the slicing strategy has changed to variable d. Now any delivery of a document d, where d depends on resource 123 and is routed to slice B, will *not* be detected as a violation of the policy, i.e., as a satisfying valuation of the MFOTL formula (1).

There are two solutions to this problem. First, we could use the old and the new strategy in parallel, during a suitably sized interval around the adaptation point, as in temporal slicing [6]. The main drawback is that at least one of the strategies is suboptimal during the adaptation period, whose length has a lower bound that depends on the formula. Moreover, temporal slicing is ineffective if there are unbounded past temporal operators.

Second, we could instead migrate the parallel monitors' state. We shall proceed this way, taking measures to ensure the state's consistency with the updated slicing strategy. Upon strategy changes, each monitor instance first splits its state into fragments, each fragment corresponding to a slice of the new strategy. Then, the fragments of all monitors are reshuffled according to their destination slice, where they are merged. Splitting and merging must ensure that the resulting state is equivalent to one that would have been obtained if the new strategy had already been applied to the previously processed events.

Figure 2 shows the high-level control flow of our adaptive parallel monitor $\mathbf{A}(\mathbf{M}_\varphi)$. The algorithm is generic in that it wraps the actual monitor implementation $\mathbf{M}_\varphi$, which can be non-parallel. Parallelism is achieved by spawning N instances of $\mathbf{M}_\varphi$ as independent submonitors. The variables k and k' range implicitly over the submonitors. $\mathbf{M}_\varphi$ refines the monitor function $\mathcal{M}_\varphi$ and makes its state explicit, which allows us to describe the adaptive functionality. We model $\mathbf{M}_\varphi$ as function mapping a pair consisting of a time-stamped database and the current state to a list of new verdicts and the successor state. The initial state is denoted s_φ^{init}. We additionally require a *splitting function* $\text{split}(R, s)$ and

Algorithm 1: Adaptive monitor $\mathbf{A}(\mathbf{M}_\varphi)$

initialization: $i \leftarrow 0$; submonitor states $s_k \leftarrow s_\varphi^{\text{init}}$ for all $k \in \{1,\dots,N\}$
for every input $\langle \tau, D \rangle$ **do**
  if $f_{i+1} \neq f_i$ **then**
    adaptation:
    all submonitors k' compute fragments $F_{k,k'} := \text{split}(f_i[k'] \cap f_{i+1}[k],\, s_{k'})$ for all k
    all submonitors k receive fragments $F_{k,k'}$ from all k'
    $s_k \leftarrow \text{merge}(F_{k,1},\dots,F_{k,N})$ for all k
  end
  parallel monitoring:
  compute slices $D_k := D \downarrow f_{i+1}[k]$ and send to k for all k
  all submonitors k perform a monitoring step $\langle X_k, s_k \rangle \leftarrow \mathbf{M}_\varphi(\langle \tau, D_k \rangle, s_k)$
  receive verdicts X_k from all k and output $\bigcup_k (X_k \cap (f_{i+1}[k] \times \mathbb{N}))$; $i \leftarrow i+1$
end

Fig. 2: High-level operation of the adaptive monitor

a *merging function* $\text{merge}(s_1, s_2, \dots)$. These are specific to the monitor implementation. The splitting function takes a restriction R and a state s, and returns the state fragment corresponding to R. The associated merging function takes a nonempty, finite list of split states and combines them into a single state corresponding to the union of the restrictions.

The adaptive monitor $\mathbf{A}(\mathbf{M}_\varphi)$ is parametrized by an infinite sequence $\langle f_i \rangle_{i \in \mathbb{N}}$ of strategies. For every $i \geq 1$, f_i defines the strategy for slicing the i-th input database. Initially, the strategy f_0 is used. Whenever the strategy changes between the i-th and the $(i+1)$-th input, i.e., $f_{i+1} \neq f_i$, adaptation occurs and each submonitor continues with a new state. Let s_k be the state of submonitor $k \in \{1,\dots,N\}$ right before adaptation. Its new state is

$$s'_k = \text{merge}(\text{split}(f_i[1] \cap f_{i+1}[k], s_1), \dots, \text{split}(f_i[N] \cap f_{i+1}[k], s_N)).$$

We require some properties of $\mathcal{M}_\varphi$, $\mathbf{M}_\varphi$, split, and merge to show that $\mathbf{A}(\mathbf{M}_\varphi)$ has the same input–output behavior as $\mathcal{M}_\varphi$. The monitor function $\mathcal{M}_\varphi$ must be *slicable*: $\mathcal{M}_\varphi(\pi \downarrow R) \cap (R \times \mathbb{N}) = \mathcal{M}_\varphi(\pi) \cap (R \times \mathbb{N})$ for all π and R. This implies that the verdicts for which a slice is responsible are detected at the exact same time points. The remaining properties are expressed in terms of a state invariant W. The intuitive meaning of $W(\pi, R, s)$ is that the state s is consistent with prefix π with respect to the valuations in R. Formally, W is called a *monitoring invariant* if it satisfies the following conditions, where $\cdot$ concatenates a stream prefix and a time-stamped database:

1. $W(\varepsilon, R, s_\varphi^{\text{init}})$ for all R, where ε denotes the empty prefix.
2. For all π, R, and s, $W(\pi, R, s)$ implies that the verdicts output by $\mathbf{M}_\varphi(\langle \tau, D \rangle, s)$ are equal to $\mathcal{M}_\varphi(\pi \cdot \langle \tau, D \rangle) - \mathcal{M}_\varphi(\pi)$ when both sets are intersected by $R \times \mathbb{N}$, and the successor state s' satisfies $W(\pi \cdot \langle \tau, D \rangle, R, s')$.
3. For all π, R_k, R'_k, and s_k (where $1 \leq k \leq N$), $R'_k \subseteq R_k$ and $W(\pi \downarrow R_k, R_k, s_k)$ for all $i \in \{1,\dots,N\}$ imply $W(\pi \downarrow (\bigcup_k R'_k), \bigcup_k R'_k, \text{merge}(\text{split}(R'_1, s_1), \dots, \text{split}(R'_N, s_N)))$.

Lemma 1. *The adaptive parallel monitor $\mathbf{A}(\mathbf{M}_\varphi)$ described above is functionally equivalent to $\mathcal{M}_\varphi$ if $\mathcal{M}_\varphi$ is slicable and there exists a monitoring invariant W.*

4 Monitor State Migration

The exact mechanism for state migration, which we need for adaptivity, depends on the monitor algorithm and the structure of its state. Here we provide a high-level account of a simplified version of the MonPoly algorithm with finite relations. Our presentation differs from the original description [7] by evaluating subformulas more eagerly. We also give the state an explicit representation. This allows us to define concrete splitting and merging operations. We verify the resulting adaptive monitor by proving the conditions outlined in Section 3.

We omit the past operators $\bullet$ and S in this section and refer to our paper describing the algorithm in depth for more details [26]. Our machine-checked formalization [25] includes the algorithm for the full language with the split and merge operations.

4.1 Monitoring Algorithm

Like MonPoly, our simplified algorithm $\mathbf{M}_\varphi$ is restricted to a fragment of MFOTL for which all subformulas of φ have finitely many satisfying valuations. We call a formula *monitorable* if negation is applied only to the right operand of $\wedge$ and to the left operand of U, and $\mathbb{V}_\beta \subseteq \mathbb{V}_\alpha$ holds for all subformulas $\alpha \wedge \neg\beta$, $\mathbb{V}_\alpha = \mathbb{V}_\beta$ for subformulas $\alpha \vee \beta$, and $\mathbb{V}_\alpha \subseteq \mathbb{V}_\beta$ for subformulas $\alpha \,\mathsf{U}_I\, \beta$ and $\neg\alpha \,\mathsf{U}_I\, \beta$. Not all finitely satisfiable formulas are monitorable. In many practically relevant cases it is possible to obtain a monitorable formula that is equivalent to φ [7]. For example, $\neg\beta \wedge \alpha$ can be rewritten to $\alpha \wedge \neg\beta$.

We present the monitor's *state* as an extension of the abstract syntax tree of the formula that it evaluates. We write a superscript after each operator to denote the state component associated with the operator. For example, $M_1 \wedge^Z M_2$ is a state corresponding to a formula $\alpha \wedge \beta$, where M_1 is the state for α (and M_2 for β), and Z is the state component of the conjunction. In general, monitor states M are constructed inductively as follows, where $\otimes \in \{\wedge, \wedge\neg, \vee\}$ and $\mathbb{U} \in \{\mathsf{U}, \neg\mathsf{U}\}$ are the monitorable operator–negation patterns.

$$M ::= r(t_1, \ldots, t_{\iota(r)}) \mid t \approx c \mid M \otimes^Z M \mid \exists x.M \mid \bigcirc_I^{(b,T)} M \mid M \,\mathbb{U}_I^{(Z,U,T)}\, M$$

The meta-variables have the following types (X^* is the type of finite lists over X, and $\mathbf{R}$ is the type of relations, i.e., finite sets of finite tuples over $\mathbb{D}$):

$$r \in \mathbb{E},\ t_i \in \mathbb{V} \cup \mathbb{D},\ Z \in \mathbf{R}^* \times \mathbf{R}^*,\ I \in \mathbb{I},\ b \in \{\top, \bot\},\ T \in \mathbb{N}^*,\ U \in (\mathbb{N} \times \mathbf{R} \times \mathbf{R})^*.$$

The algorithm $\mathbf{M}_\varphi$ performs a bottom-up evaluation of the formula φ for each incoming database of events. The result of evaluating a subformula ψ is a list of finite relations over its free variables $\mathbb{V}_\psi$. These relations contain the valuations v satisfying the subformula for increasing indices i, i.e., those v with $v, i \models \psi$. We thus obtain the monitor's output, all (v, i) with $v, i \models \varphi$, incrementally at the root φ. The evaluation of subformulas with future operators is delayed until the most recent time-stamp in the input has advanced by a sufficient lookahead, which is determined by the upper bound on the interval. Therefore, the result of evaluation is a list of relations: Several indices (or none) may be resolved at once if their lookahead has been reached (is still missing).

We choose to evaluate subformulas as much as possible with respect to the lookahead. All binary operators have a state component $Z = \langle z_1, z_2 \rangle$ that stores the results from one

operand while the other is delayed. For example, if the left operand is three indices ahead, z_1 contains the corresponding three results and z_2 is empty. For temporal operators, the list T stores the time-stamps of not yet evaluated indices. The flag b marks whether $\bigcirc$ has been evaluated on the first index. The state component U associated with a subformula $\alpha \, \mathbb{U}_I \, \beta$ is a list of triples $\langle \tau_i, R_i, R'_i \rangle$. It corresponds to a contiguous interval $i \in [n', n)$ of input indices, where n' is the index of the next result to be computed for the subformula, and n is the index of the next input to the monitor. The relation R_i contains all valuations v such that $v, k \models \alpha$ for all $k \in [i, n)$. The relation R'_i contains all valuations v for which there exists a $j \in [i, n)$ such that $\tau_j - \tau_i \in I$, $v, j \models \beta$, and $v, k \models \alpha$ (or $v, k \not\models \alpha$ in the negated case) for all $k \in [i, j)$. In the initial state, all lists Z, T, U are empty, and b is set to $\perp$.

We now describe how $\mathbf{M}_\varphi$ processes a new input $\langle \tau_n, D_n \rangle$. Evaluation of predicates and equalities is straightforward: A single relation is produced, and the state remains unchanged. For all other operators, the algorithm first evaluates and updates the substates recursively. For existential quantifiers $\exists x.M$, the recursively computed relations are projected onto $\mathbb{V}_M - \{x\}$. For the Boolean connectives in $\otimes$, the two lists of results r_1, r_2 (which may be empty) are appended to the corresponding z_1, z_2 that were stored in the previous state, resulting in z'_1, z'_2. The first $\min\{|z'_1|, |z'_2|\}$ elements of each list are removed and combined pointwise into the result of the connective by applying standard relational operations: a natural join $\bowtie$ for $\wedge$, an antijoin $\triangleright$ for $\wedge\neg$, and a union for $\vee$. The state component Z is updated to the remainder of the lists.

The state component T of the temporal operators is maintained by appending τ_n and by removing a time-stamp from the front for every computed result. Evaluation of $\bigcirc_I$ discards the very first result of its operand, as indicated by b. Apart from this initialization, the operators forward the results, unless the corresponding time-stamp difference is not in the interval I. In this case, the result is replaced by the empty relation. The difference is computed using T. The state component $U = U_0$ of an operator $\mathbb{U}_{[a,b)}$ is updated as follows, where z'_1, z'_2, and $\ell = \min\{|z'_1|, |z'_2|\}$ are obtained as above. Let A_k, B_k, and τ'_k be the k-th element, $1 \leq k \leq \ell$, in z'_1, z'_2, and $T \cdot \tau_n$, respectively. For every k, U_{k-1} is updated to obtain U_k, where $\langle \sigma_i, R_i, R'_i \rangle$ is the i-th tuple in U_{k-1}:

1. Replace every R'_i by $R'_i \cup (B_k \bowtie R_i)$ (by $R'_i \cup (B_k \triangleright R_i)$ if the left operand is negated) if $\tau'_k - \sigma_i \in I$.
2. Replace every R_i by $R_i \bowtie A_k$ (by $R_i \cup A_k$ if negated).
3. Append $\langle \tau'_k, A_k, B_k \rangle$ if $0 \in I$. Otherwise, append $\langle \tau'_k, A_k, \varnothing \rangle$.

We now consider the tuples $\langle \sigma_i, R_i, R'_i \rangle$ in U_ℓ. Let m be the largest index such that $\sigma_m + b < \tau_n$. The result of the operator is the list $\langle R'_1, \ldots, R'_m \rangle$. The updated state component U is U_ℓ without the first m elements.

4.2 Splitting and Merging

MonPoly's state can be viewed as consisting of two parts. First, its *shape* comprises the arrangement of nodes in the abstract syntax tree, the lengths of the lists associated with the nodes, and the flags' and time-stamps' values. Second, the state has *content*, namely the relations stored in it. The key insight, which we use to define splitting and merging operations, is that the shape is independent of slicing, while the content has

$$\mathrm{split}(P,M) = \begin{cases} \mathrm{split}(P,M_1) \otimes^{\mathrm{split}(P,Z)} \mathrm{split}(P,M_2) & \text{if } M = M_1 \otimes^Z M_2 \\ \exists x.\, \mathrm{split}(\mathrm{lift}(P,x),M_1) & \text{if } M = \exists x. M_1 \\ \bigcirc_I^{(b,T)} \mathrm{split}(P,M_1) & \text{if } M = \bigcirc_I^{(b,T)} M_1 \\ \mathrm{split}(P,M_1)\, \mathbb{U}_I^{(\mathrm{split}(P,Z),\mathrm{split}(P,U),T)}\, \mathrm{split}(P,M_2) & \text{if } M = M_1\, \mathbb{U}_I^{(Z,U,T)}\, M_2 \\ M & \text{otherwise.} \end{cases}$$

$$\mathrm{split}(P,\langle z_1,z_2\rangle) = (\mathrm{map}(\lambda X.\, \mathrm{split}(P,X),z_1), \mathrm{map}(\lambda X.\, \mathrm{split}(P,X),z_2))$$

$$\mathrm{split}(P,U) = \mathrm{map}(\lambda\langle\tau,A,B\rangle.\langle\tau,\mathrm{split}(P,A),\mathrm{split}(P,B)\rangle,U)$$

$$\mathrm{split}(P,X) = \{x \in X \mid P(x)\}$$

Fig. 3: Splitting operations for MonPoly's state

a direct interpretation in terms of MFOTL's semantics. We exploit the fact that the shape of the state is determined purely by the sequence of time-stamps observed by the submonitors. Note that slicing has no effect on this sequence. Therefore, the states of all submonitors at a given point in time have identical shapes. In contrast, we have $v,i \models_{\rho\downarrow R} \varphi$ iff $v,i \models_\rho \varphi$ for all $v \in R$, which is the property that allows data slicing in the first place (see the proof of [24, Prop. 1]). The inclusion of a valuation in the content associated with the submonitor for R thus depends only on the full stream ρ (if that valuation can be extended to some $v \in R$). We can reorganize the state's content to reflect updated restrictions R by distributing valuations according to their consistency with R.

The splitting function for MonPoly's state is shown in Fig. 3. We use standard functional operators on lists, in particular map and zip. The splitting function is applied recursively to all parts of the state while preserving its shape. We overload it for the different state components. Only the relations in the state are affected by splitting. The operation keeps all valuations that are consistent with the restriction R, which we represent as a predicate function P (the first argument of split). There are two reasons for this modified representation. First, restrictions are usually infinite sets and so they cannot be passed explicitly in an implementation. Second, the finite relations stored in the monitor's state cover only a subset of the formula's free variables, possibly extended by bound variables. For example, consider the state $A(x,y) \wedge^{Z_1} (B(x)\, \mathsf{U}^{(Z_2,U,T)}\, C(x))$. The relations in the list U are unary because they assign values to x only. A valuation $(x \mapsto a)$ contained in such a relation is compatible with a restriction R iff there exists a valuation $v \in R$ with $v(x) = a$. This generalizes in the obvious way to relations of higher arity. Thus P must be true for a valuation iff it is compatible with R. We can always define such a P in theory, but an implementation will provide a specialized function for the specific slicing strategies that it uses. The lift functional lifts a predicate function P to a context with a bound variable x. Therefore, $\mathrm{lift}(P,x)(v)$ is true iff P is true for v with x removed from its domain.

The merge function $\mathrm{merge}(s_1,s_2,\dots)$ combines the list of states by repeatedly applying a binary merge in arbitrary order. The binary merge function mrg_2 is shown in Fig. 4. Here, $\mathrm{map2}(f,A,B)$ abbreviates $\mathrm{map}(f,\mathrm{zip}(A,B))$. We assume that the two inputs to mrg_2 have the same shape. Some parts of the state, like the time-stamp lists T, can thus be merged by simply taking the value from either state. This works because the shape is

$$\mathrm{mrg}_2(M_a, M_b) = \begin{cases} \mathrm{mrg}_2(M_{1a}, M_{1b}) \otimes^{\mathrm{mrg}_2(Z_a, Z_b)} \mathrm{mrg}_2(M_{2a}, M_{2b}) & \text{if } M_i = M_{1i} \otimes^{Z_i} M_{2i} \\ \exists x.\, \mathrm{mrg}_2(M_{1a}, M_{1b}) & \text{if } M_i = \exists x. M_{1i} \\ \bigcirc_I^{(b_a, T_a)} \mathrm{mrg}_2(M_{1a}, M_{1b}) & \text{if } M_i = \bigcirc_I^{(b_i, T_i)} M_{1i} \\ \mathrm{mrg}_2(M_{1a}, M_{1b}) \, \mathbb{U}_I^{(\mathrm{mrg}_2(Z_a, Z_b), \mathrm{mrg}_2(U_a, U_b), T_a)} \mathrm{mrg}_2(M_{2a}, M_{2b}) & \text{if } M_i = M_{1i} \, \mathbb{U}_I^{(Z_i, U_i, T_i)} M_{2i} \\ M_a & \text{otherwise.} \end{cases}$$

$$\mathrm{mrg}_2(\langle z_{1a}, z_{2a} \rangle, \langle z_{1b}, z_{2b} \rangle) = \langle \mathrm{map2}(\cup, z_{1a}, z_{1b}), \mathrm{map2}(\cup, z_{2a}, z_{2b}) \rangle$$

$$\mathrm{mrg}_2(U_a, U_b) = \mathrm{map2}(\lambda \langle \tau_a, A_a, B_a \rangle, \langle \tau_b, A_b, B_b \rangle . \langle \tau_a, A_a \cup A_b, B_a \cup B_b \rangle, U_a, U_b)$$

Fig. 4: Binary merging operations for MonPoly's state

not affected by slicing, as we have argued before. Relations are merged by taking their union. This makes sense intuitively because the desired effect of mrg_2 and merge is to be consistent with the union of the states' restrictions.

Theorem 1. *There exists a slicable monitor function $\mathcal{M}_\varphi$ with a corresponding monitoring invariant for the functions $\mathbf{M}_\varphi$,* split*, and* merge *described in this section.*

We prove the existence of the invariant in our formalization [25]. Together with Lemma 1, which has also been formally verified, we obtain the correctness of the adaptive monitor.

Corollary 1. *The adaptive parallel monitor $\mathbf{A}(\mathbf{M}_\varphi)$ constructed from $\mathbf{M}_\varphi$,* split*, and* merge *is functionally equivalent to the monitor function $\mathcal{M}_\varphi$.*

5 Implementation and Evaluation

We have extended the MonPoly monitoring tool [8] with the state split and merge functionalities, adding about 960 lines of OCaml. The source code is available online [25]. Note that MonPoly implements optimizations that are beyond the scope of this paper. Specifically, subformula evaluation is less eager [7], with binary operators evaluating the right subformula only when the left subformula can be evaluated. MonPoly treats subformulas of the form $\Diamond_I \varphi$ and $\blacklozenge_I \varphi$ in a special way by greedily reusing intermediate computations of the (associative) union in a sliding window bounded by the interval I [9]. Also, MonPoly filters out events and time points with no events when they do not influence the monitor's output [6]. Still, our implementation takes all of these optimizations into account, with the exception of the empty time-point filtering, which we leave as future work.

We have also extended our online slicing framework [24] to enable dynamic changes to the slicing strategy. The extended framework can synchronously redistribute the parallel submonitors' states. The redistribution consists of splitting the states of all submonitors and forwarding the splits to the appropriate monitors before all of them resume monitoring. The framework uses Apache Flink [2] to achieve low latency stream processing with fault tolerance. However, we directly invoke the monitors on prepared files for the purpose of this evaluation, due to Flink's limited state migration capabilities.

$$\begin{aligned} \mathit{star} &= \big((\blacklozenge_{[0,10\,\mathrm{s})} P(a,b)) \land Q(a,c)\big) \land \Diamond_{[0,10\,\mathrm{s})} R(a,d) \\ \mathit{linear} &= \big((\blacklozenge_{[0,10\,\mathrm{s})} P(a,b)) \land Q(b,c)\big) \land \Diamond_{[0,10\,\mathrm{s})} R(c,d) \\ \mathit{triangle} &= \big((\blacklozenge_{[0,10\,\mathrm{s})} P(a,b)) \land Q(b,c)\big) \land \Diamond_{[0,10\,\mathrm{s})} R(c,a) \end{aligned}$$

Fig. 5: MFOTL formulas (after negation) used in the evaluation

We have validated our approach and evaluated the performance of our implementation by answering the following research questions:

RQ1: Does dynamically adapting the splitting strategy improve the performance?

RQ2: How scalable is the adaptive monitoring with respect to the stream event rate and the degree of parallelism, i.e., the number of submonitors?

RQ3: How much overhead is incurred by a single adaptation?

To answer the above questions we designed a parametric testbed for measuring the performance of both non-adaptive and adaptive monitoring [25]. For n adaptation steps, the testbed takes a list of $n+1$ stream statistics and creates an event stream that consists of $n+1$ parts, each conforming to the respective statistics. Given an input formula, the testbed performs two monitoring runs: a *non-adaptive run*, which uses a slicing strategy optimized [24] for the first part of the stream to monitor the entire stream, and an *adaptive run*, which uses stream statistics for each part of the stream to construct a sequence of optimized slicing strategies. Each part of the stream is sliced according the appropriate slicing strategy. The number of slices is equal to the degree of parallelism, which is configurable. Alternatively, we could consider the entire stream's aggregate statistics to compute a strategy for the non-adaptive run. However, we believe that our setup is more suitable for comparing the two online monitoring approaches, whereas the alternative assumes complete knowledge of the stream—a trait often associated with offline monitoring.

We fix the number of adaptations n to one. Hence, we define two stream statistics for the corresponding parts of the streams. We focus our evaluation on single adaptations in order to properly isolate and measure the effects of specific changes in the stream statistics on the monitoring performance. While having multiple adaptation steps is certainly more realistic, this would not contribute to answering our research questions as the results would be harder to interpret and the space of possible stream statistics would be much larger. When monitoring a formula containing future subformulas, the monitor often needs to establish a lookahead. During this process, the monitor exhibits better performance as it performs simple updates to its state without outputting any verdicts. To prevent this behavior from effecting our measurements, we add an additional prefix to our streams as a warmup, generated with identical statistics as the stream's first part.

We monitor the three formulas shown in Figure 5 (named *star*, *linear*, and *triangle*) over streams with different event rates and stream statistics. The different variable patterns in the formulas cover common patterns in database queries [10], which we additionally extend with temporal operators. Given a stream $\rho = \langle \tau_i, D_i \rangle_{i \in \mathbb{N}}$, its *event rate* at time τ is the total number of events in one time unit, i.e., $|\{e \in D_i \mid \tau = \tau_i\}|$. *Stream statistics* consist of *relative relation rates* (i.e., the fraction of events in a time unit with a certain name) and *heavy hitter values* (i.e., event attribute values that occur frequently).

We implemented a stream generator that takes a random seed and stream statistics, and synthesizes a random stream that conforms to the supplied statistics. Specifically, it produces streams containing events with the names P, Q, and R. The event rate and the

	stream statistics for part 2	description	event rate for formula		
			star	*linear*	*triangle*
S1	$r_P = 0.01, r_Q = r_R = 0.495$	reduce relation rate for P	2500	1300	1300
S2	$r_P = r_Q = 0.495, r_R = 0.01$	reduce relation rate for R	2500	1300	1300
S3	$r_P = r_Q = 0.01, r_R = 0.98$	reduce relation rates for P and Q	2500	1300	1300
S4	d_a = Zipf, $z_a = 10$, $s_a = 1000$	add a single heavy hitter value	75	1300	1300
S5	default	remove a single heavy hitter value	75	1300	1300
S6	d_a = Zipf, $z_a = 10$, $s_a = 2000$	change the heavy hitter value	75	1300	1300
S7	d_a = Zipf, $z_a = 2$, $s_a = 1000$	add more heavy hitter values	75	1300	1300
S8	d_c = Zipf, $z_c = 10$, $s_c = 1000$	change the heavy hitter variable	75	400	700
S9	$d_a = d_c$ = Zipf, $z_a = z_c = 10$, $s_a = s_c = 1000$	add more heavy hitter variables	75	700	700

Fig. 6: Stream statistics used in our experiments (omitted parameters have default values)

rate of verdicts is configurable. Each of the three events has two integer attributes. The generator can also synthesize streams with configurable relative relation rates and force some event attribute values to be heavy hitters. Attribute values are sampled with two possible distribution types. Infrequent values are drawn from the uniform distribution over the set $\{0,1,\ldots,10^9-1\}$. Heavy hitter values are drawn from a Zipf distribution that can be defined per variable. Its probability mass function is $p(x) = (x-s)^{-z}/\sum_{n=1}^{10^9} n^{-z}$ for $x \in \{s+1, s+2, \ldots, s+10^9\}$, i.e., the larger the exponent $z > 0$ is, the fewer values in the variable valuation have a large relative frequency. The parameter s is the start value, which can also be configured to control the specific heavy hitter values. We call variables with heavy hitter values *heavy hitter variables*. To prevent excessive monitor output, all Zipf-distributed values of R events are shifted (i.e., increased by 10^6), whereas events that cause the monitor to output a verdict have their values always drawn uniformly.

Figure 6 summarizes the stream statistics (in terms of the parameters supplied to the stream generator) used in our experiments. The total time span of each stream across all parts is 1000 seconds. The parameters r_P, r_Q, and r_R are the relative relation rates for relations P, Q, and R, respectively, each with the default value $1/3$. The parameters d_a, d_b, and d_c are the distribution types for values occurring in valuations of the variables a, b, and c respectively. Values are distributed uniformly by default. For a Zipf-distributed variable x, z_x and s_x define its Zipf exponent and the starting value. We distinguish between nine representative changes in the stream statistics, labelled S1–S9 in the leftmost column in Figure 6. For S1–S4, all parameters assume default values in the streams' first part. For S5–S9, the first part is generated using the default parameters, except that $d_a = \text{Zipf}$, $z_a = 10$, $s_a = 1000$. The second column shows the parameters for the second part. The third column describes the change informally. Such changes of the stream statistics can have a large impact on a monitor's performance. For example, the monitoring time can differ in orders of magnitude between the two stream parts generated by S4. This is due to the size of the intermediate results that the monitor computes for the subformulas. Their size can grow significantly if a heavy hitter is added (consider the satisfying valuations of $(\blacklozenge_{[0,10s)} P(a,b)) \land Q(a,c)$ when a is a heavy hitter variable). To overcome this problem, we have chosen the event rates such that it takes at most 25 seconds to monitor each slice. We searched and sampled monitoring times for event rates between 10 and 6000 events per second. The chosen event rates are summarized in the last three columns of Figure 6.

We measured the execution time for monitoring each slice of each stream part and each run, as well as the time to split and merge states during the adaptive run. Each run is

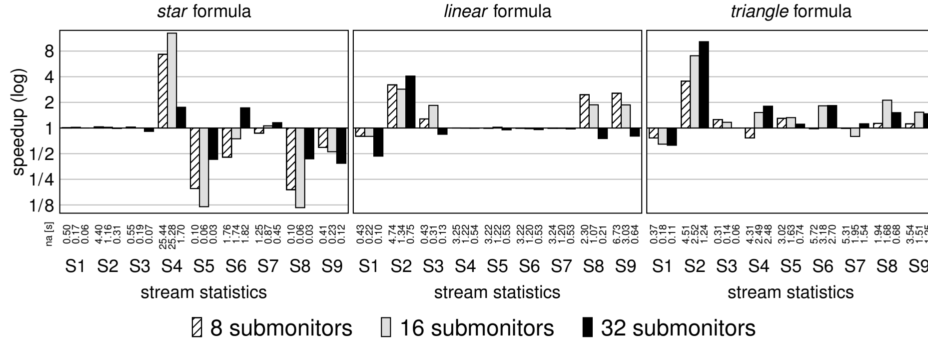

Fig. 7: Observed speedup (the ratio of the non-adaptive and the corresponding adaptive monitoring times) for different stream statistics, number of submonitors, and formulas

event	*star* formula				*linear* formula				*triangle* formula			
rate	na [s]	ad [s]	na/ad	ohd	na [s]	ad [s]	na/ad	ohd	na [s]	ad [s]	na/ad	ohd
1000	0.75	0.72	**1.04**	1.5%	2.86	0.89	**3.22**	5.4%	2.77	0.73	**3.78**	6.3%
2000	2.82	2.79	**1.01**	2.1%	11.05	3.38	**3.27**	3.9%	10.63	2.83	**3.76**	2.2%
4000	10.80	10.85	0.99	1.9%	44.80	14.04	**3.19**	1.8%	41.97	11.13	**3.77**	2.0%
6000	24.23	24.39	0.99	1.0%	111.90	30.75	**3.64**	1.1%	94.83	26.45	**3.59**	1.5%

Fig. 8: Observed **speedup** or slowdown (na/ad) and overhead (ohd) for different event rates and formulas when monitoring streams with statistics S2 using 8 submonitors

repeated three times and the measurements are averaged. Our experiments were executed on a machine with an Intel Core i5-7200U CPU running at 2.5 GHz, with 8 GB RAM. We monitored all slices sequentially, such that only one thread was active at any time.

Figure 7 summarizes the results of our evaluation using the parameters from Figure 6. We compare the execution times between the two types of monitoring runs (non-adaptive and adaptive) on the last part (part 2) of the event stream, where the slicing strategies differ. We consider the maximum time across all slices for each run. The bars show the observed speedup, i.e., the ratio of the time taken by the non-adaptive and the corresponding adaptive monitoring run. The non-adaptive time in seconds is given below each bar. To answer RQ1, note that monitoring the *star* formula does not benefit from the adaptation in most cases. This is due to its particular structure: The common variable a is the most efficient choice for slicing, independently of the relation rates (S1–S3). However, if any of a's valuations becomes a heavy hitter, the slices are no longer balanced and adaptivity helps (S4). In our non-adaptive runs with S4, the increased monitoring time when using 8 and 16 submonitors is due to a single slice accidentally receiving both the heavy hitter value and its shifted counterpart for R events. If a heavy hitter value is removed (S5, S6, S8), all slices in the adaptive run are monitored efficiently except for one that receives the first part's state associated with the heavy hitter. As this information is still relevant up to 10 seconds after the statistics change (due to the temporal subformulas' intervals), the corresponding monitor has a significantly larger workload, which causes the slowdown. This could be avoided by taking the formula's intervals into account to delay the adaptation by an appropriate amount of time.

In general, adaptation helps when monitoring the *linear* and *triangle* formulas. We obtained the largest consistent speedups for S2 (between 3.5 and 10.3 times for *triangle*). Here, the reduction of the slices' event rates due to the adaptation is reflected by the reduced execution time. Adapting to heavy hitters is often beneficial for those formulas, too. The slowdown observed for some numbers of submonitors is due to the impossibility to decompose these numbers into optimal factors, e.g., 8 into an integer square root [24].

Regarding RQ2, Figure 8 shows that higher event rates increase the benefit of adaptation if the stream statistics allow for a better strategy to be used in the first place. The columns *na* and *ad* show the maximum monitoring time (in seconds) of part 2 across all slices for the non-adaptive and adaptive monitoring runs, respectively. The measurement in the *ad* column includes the time taken to split and merge the state. The *na/ad* column shows the speedup, while the *ohd* column shows the overhead. The overhead is the ratio between the time spent performing non-monitoring tasks and the adaptive monitoring time, each summed over all slices. Non-monitoring tasks include state splitting and merging, as well as the time the submonitors would need to wait before all state fragments are available to be merged. Since we monitor the slices sequentially in our experiments, we estimate the wait time as the time difference until the last submonitor has finished splitting its state. The adaptation overhead ranges from 1% to 6% in our experiments with S2, and it decreases with the event rate (RQ3). However, the overhead can be as large as 700% in some of the other experiments from Figure 7 (*star* formula, S5 and S8). This is generally the result of imbalanced substreams. Thus, some submonitors in a parallel implementation would be forced to wait, for which we account in the overhead calculation.

6 Conclusion

We have laid the foundations of adaptive online monitoring by demonstrating how to implement the core functionality required: the state exchange between the parallel monitors. The state exchange consists of two operations, split and merge, which we prove to interact correctly with a simplified MFOTL monitor. We also implement them in a realistic monitor and demonstrate empirically that adapting to changing statistics is beneficial.

As ongoing work, we are extending our operations to support MonPoly's empty time-point filtering, which significantly improves performance, especially in combination with slicing. Because the monitors for the different slices may skip events at different time-points, their state structures may diverge, which complicates merging.

We have also performed initial experiments using our adaptive version of MonPoly within our Apache Flink-based parallel monitor [24]. While this setup works in principle, its performance is suboptimal, due to limitations of Flink. For example, Flink only allows exchanging parts of the state by sending all states to all monitors and only then performing the split operations locally. This incurs a large latency, and another stream-processing framework might be better suited for our needs. Timely Dataflow [21], with its recent extension to low-latency state migrations [16] is a promising candidate.

Finally, important questions that we have not studied in this paper are how to collect the necessary statistics and at which points to trigger adaptivity. While classic sketching algorithms [13] offer partial answers to the first question, answering the second one requires a realistic cost model to precisely calculate when adaptivity pays off.

Acknowledgment. Christian Fania helped us implement and evaluate our adaptive monitoring framework. The anonymous reviewers gave numerous helpful comments on earlier drafts of this paper. Joshua Schneider is supported by the US Air Force grant "Monitoring at Any Cost" (FA9550-17-1-0306). Srđan Krstić is supported by the Swiss National Science Foundation grant "Big Data Monitoring" (167162).

References

1. Afrati, F.N., Ullman, J.D.: Optimizing multiway joins in a map-reduce environment. IEEE Trans. Knowl. Data Eng. **23**(9), 1282–1298 (2011)
2. 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)
3. Alur, R., Mamouras, K., Stanford, C.: Modular quantitative monitoring. PACMPL **3**(POPL), 50:1–50:31 (2019)
4. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in Storm. In: DEBS 2013. pp. 207–218. ACM (2013)
5. Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.A., Hallé, S.: MapReduce for parallel trace validation of LTL properties. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 184–198. Springer (2012)
6. 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)
7. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
8. Basin, D., 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)
9. Basin, D., Klaedtke, F., Zălinescu, E.: Greedily computing associative aggregations on sliding windows. Information Processing Letters **115**(2), 186–192 (2015)
10. Beame, P., Koutris, P., Suciu, D.: Communication steps for parallel query processing. J. ACM **64**(6), 40:1–40:58 (2017)
11. Bersani, M.M., Bianculli, D., Ghezzi, C., Krstić, S., Pietro, P.S.: Efficient large-scale trace checking using MapReduce. In: Dillon, L.K., Visser, W., Williams, L. (eds.) ICSE 2016. pp. 888–898. ACM (2016)
12. Bianculli, D., Ghezzi, C., Krstić, S.: Trace checking of metric temporal logic with aggregating modalities using MapReduce. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 144–158. Springer (2014)
13. Cormode, G., Garofalakis, M.N., Haas, P.J., Jermaine, C.: Synopses for massive data: Samples, histograms, wavelets, sketches. Foundations and Trends in Databases **4**(1-3), 1–294 (2012)
14. Elseidy, M., Elguindy, A., Vitorovic, A., Koch, C.: Scalable and adaptive online joins. PVLDB **7**(6), 441–452 (2014)
15. Hallé, S., Soucy-Boivin, M.: MapReduce for parallel trace validation of LTL properties. Journal of Cloud Computing **4**(1), 8 (2015)
16. Hoffmann, M., Lattuada, A., McSherry, F., Kalavri, V., Liagouris, J., Roscoe, T.: Megaphone: Latency-conscious state migration for distributed streaming dataflows. PVLDB **12**(9), 1002–1015 (2019)

17. Kalavri, V., Liagouris, J., Hoffmann, M., Dimitrova, D.C., Forshaw, M., Roscoe, T.: Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In: Arpaci-Dusseau, A.C., Voelker, G. (eds.) OSDI 2018. pp. 783–798. USENIX Association (2018)
18. Kolchinsky, I., Schuster, A.: Efficient adaptive detection of complex event patterns. PVLDB **11**(11), 1346–1359 (2018)
19. Mamouras, K., Raghothaman, M., Alur, R., Ives, Z.G., Khanna, S.: StreamQRE: modular specification and efficient evaluation of quantitative queries over streaming data. In: Cohen, A., Vechev, M.T. (eds.) PLDI 2017. pp. 693–708. ACM (2017)
20. Mayer, R., Tariq, M.A., Rothermel, K.: Minimizing communication overhead in window-based parallel complex event processing. In: DEBS 2017. pp. 54–65. ACM (2017)
21. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: Kaminsky, M., Dahlin, M. (eds.) SOSP 2013. pp. 439–455. ACM (2013)
22. 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)
23. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Log. Methods Comput. Sci. **8**(1) (2012)
24. 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)
25. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Artifact associated with this paper. `https://bitbucket.org/jshs/monpoly/downloads/aom_atva2019.zip` (2019)
26. 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, Springer (2019), `http://people.inf.ethz.ch/trayteld/papers/rv19-verimon/verimon.pdf`, to appear.
27. Shah, M.A., Hellerstein, J.M., Chandrasekaran, S., Franklin, M.J.: Flux: An adaptive partitioning operator for continuous query systems. In: Dayal, U., Ramamritham, K., Vijayaraman, T.M. (eds.) ICDE 2003. pp. 25–36. IEEE (2003)
28. Vitorovic, A., Elseidy, M., Guliyev, K., Minh, K.V., Espino, D., Dashti, M., Klonatos, Y., Koch, C.: Squall: Scalable real-time analytics. PVLDB **9**(13), 1553–1556 (2016)