École Polytechnique Fédérale de Lausanne

Scalable algorithms for pattern detection in out-of-order event streams

by Quentin Bouvet

Master Thesis

Examining Committee:

Dr. sc. Felix Klaedtke
Thesis Advisor

Prof. Dr. sc. Anastasia Ailamaki
Thesis Advisor

Dr. sc. Felix Klaedtke
External Expert

Dr. sc. Bikash Chandra
Thesis Supervisor

EPFL IC IINFCOM DIAS
BC 222 (Bâtiment BC)
Station 14
CH-1015 Lausanne

March 13, 2020

Abstract

In this thesis, we present an approach for processing event streams online, where events can carry data values. Events may also appear out of order, that is, the order in which events are received can be different from the order in which they are generated. Our approach aims at verifying events streams describing system behavior in real-time against specifications that are given as formulas in a temporal logic. The novelty of our approach is that we utilize multiple CPU cores to carry out the monitoring task concurrently and in parallel. To this end, we first present an architecture that is well suited for online processing event streams in parallel. In a nutshell, received events are assigned to substreams that are processed separately. Afterwards, we propose an algorithm based on stateful workers that efficiently aggregate these substreams. Finally, we report on the outcome of an experimental evaluation of a prototype implementation. Our evaluation shows that our prototype scales well with respect to the number of CPU cores. The evaluation also shows that our approach can be more efficient than prior approaches that process an event stream monolithically.

Contents

Chapter 1

Introduction

Ensuring the correct behavior of software systems is increasingly important in our modern world. Several techniques have been developed for this purpose, including theorem proving, model checking and testing [5]. Runtime verification is a model checking technique on which this work focuses. It consists in verifying certain properties of a system by analyzing execution logs generated by this system at runtime. Runtime verification distinguishes itself from other techniques by analyzing actual execution traces, as opposed to synthetic scenarios. It can be performed offline, on recorded execution traces, or online, on live data coming from the monitored system. In that case, another advantage of runtime verification is that violations of the monitored property can be reported immediately, instead of performing a post mortem analysis on log files. A desirable property for a runtime verification algorithm is the ability to handle out-of-order data streams. This property reduces latency due to delayed stream events, which contributes to reporting the violations promptly. Out-of-order streams can happen for a variety of reasons, especially when monitoring distributed systems that send event stream through a network. System components may have different network latency to the monitor, routing may cause delays and re-ordering of data packets... Basin et al. [3] propose an online algorithm for runtime verification of out-of-order streams. Our work aims to extend this algorithm process the stream in parallel, therefore increasing throughput, while retaining the desirable out-of-order and online processing properties.

We must solve two complementary challenges to achieve our goal. First, we must split our stream into substreams, in a way that allows submonitors processing the substreams independently to produces correct results. Intuitively, one result must depends on a subset of the event stream, so that we can process these subsets independently and still produce correct results. We select a setting where partitioning is possible, then define a partitioning scheme that allows us to split our stream into substreams that can be processed in parallel. Then, we must define an algorithm that handles the processing of these partitions in parallel. We will exploit data-parallelism of the events to partition the stream. However, since our algorithm is online, we do not know the partitions in advance, and since our stream is out-of-order, we need to give special attention to generating the correct substreams knowing that at any point, past events may still arrive late.

We illustrate this with an example. A stream contains $login(u, c)$, $logout(u, c)$ and $reset(c)$ events, where u, c are data parameters identifying user and computers. We want to verify that any user that logs in either logs out some time in the future, or the computer used to login, c, gets reset. The login system allows several users to login into a computer, as would be the case on a server. Suppose that we receive events $login(alice, c1)$, then $reset(c1)$. Clearly, the property is true for the first login of alice. However, since our stream is out of order, we can still receive a $login(bob)$ event happening prior to the reset. In this case, the property would be true for bob as well. Our algorithm needs to keep track of events that can still be relevant for potentially delayed messages, that may be in partitions that we do not have knowledge of yet, and also forget them once we are sure that no delayed events are missing.

This problem is only partially addressed by existing work, in the sense that existing solutions typically offer one or two of the desired properties: online processing, our-of-order processing, and parallelizable. Some existing work in the domain of runtime verification relies on finite state automata or temporal automata running in parallel as submonitors to achieve online, highly parallel stream processing [8], [7], [6]. However, these approaches do not handle out-of-order streams. Other approaches use temporal logic based specifications [2], [1], [9]. In these works, the monitor and submonitors verify the property with set theory operators. However, these works do not support out-of-order processing either

Our approach to solve the partitioning and parallellizing subproblems described above is as follow. We split the stream based on data parameters of the events. We determine the partition for some event based on the variable assignment given by its parameters. For instance, the event $login(alice, c1)$ belongs to the substream $[u \mapsto alice, c \mapsto c1]$. Events in different substream are processed independently. We define a partial order relation on partitions. Intuitively, we order substream by their mapping from variables to values. If the mapping of a partition is a "sub-mapping" of the other, then the partitions are partially ordered. For instance, partitions for $login(alice, c1)$ and $login(bob, c1)$ are incomparable, while the partition for $login(alice, c1)$ *extends* the partition for $reset(c1)$. Finally, we present the setting in which such partitioning works. Then we define an algorithm to process substreams in parallel, online, out-of-order. Our algorithm consists of a non-terminating procedure called dispatcher, and worker procedures. The dispatcher infers the necessary stream partitions for each event of the stream and forwards the event to the workers processing each of these partitions. If no worker exists for a partition, the dispatcher creates it by cloning a worker. The cloned worker is selected using the partial order on partitions, as to ensure that the new worker is aware of all past events that are relevant for him. Note that workers do not store events forever. In a way, they aggregate events over a sliding window as the stream progresses. The receive new events, update their state, potentially produce result, and forget events that they no longer need to produce new results. The aggregation operation is carried out by the polimon algorithm proposed and implemented by Basin et al. [3], which we introduce in section 2.3. Each of our worker relies on an instance of a polimon monitor to perform the stream.

We test our implementation on up to 6 CPU cores. The results show that for a fixed problem size, our algorithm scales well on multiple cores compared to the original polimon implementation, and that it can scales better with larger problem size, i.e. when increasing the stream event rate. Depending on the monitored property, our implementation can scale linearly with the event rate, whereas the original implementation's performance degrades faster at higher event rate.

The core contributions of this this thesis are the following:

- An approach to partitioning a parametric event stream for online, out-of-order monitoring
- An algorithm to efficiently process the stream partitions in parallel
- An implementation of this algorithm for multi core processors and an experimental evaluation

This thesis is organized as follow. In chapter 2, we provide background information on which our design relies. We introduce our solution in chapter 3. In chapter 4, we describe implementation aspects of our solution. In chapter 5, we present our experimental evaluation and its results. We discuss related work in chapter 6, and we conclude this thesis in chapter 7.

Chapter 2

Background

In this chapter, we introduce the runtime verification problem and related concepts. We present temporal logic based specification languages that are used in our solution. Finally, we describe the Polimon algorithm, which we use as submonitor.

2.1 Runtime verification

Runtime verification Runtime verification *is the discipline of computer science that deals with the study, development and application of those computer techniques that allow checking whether a run of a system under scrutiny satisfies or violate a given index property* [5]. In this section, we aim to provide the necessary intuition about runtime verification. For more details, we direct the reader to [5].

Concretely, a runtime verification problem includes the following elements. The *trace* represents an infinite sequence of *events* coming from the monitored system. The *specification* is the property that the system is expected to verify. The *monitor* is procedure that reads a finite prefix of the trace, our *event stream* and outputs a *verdict stream.* The specification is expressed using a specification language. Typical examples are finite state automata [8], [7] or different versions of temporal logic, like Linear Temporal Logic [6]. In our case, we express the specification using the MTL$_{prenex}$ temporal logic, which is defined in section 2.2.

In this thesis, we monitor a stream of events. An event consists of the following elements: a *timestamp* τ, a *registers valuation* ϱ and a predicate interpretation function σ. In order to define these elements, we need to define the following sets: let P be a finite set of *predicate symbols* (intuitively, the identifiers of our predicates). Let $\iota(p)$ denoting the arity of $p \in P$ Let D be the *data domain,* the set of possible data values, and R the set of registers identifier. Let V be the finite set of variable identifiers in our proposition. Then ϱ is a function $\varrho : R \mapsto D$ and σ a function over P

so that $\sigma(p) \in D^{\iota(p)}$. We note this event as $<\tau, \varrho, \sigma>$.

Combing back to the example introduced in the introduction, a possible event would be $<1.66, [u/"alice", c/"c1"], \{login : \{("alice", "c1")\}, logout : \{\}, reset : \{\}\}>$. At timepoint 1.66, the variable u is assigned the data value "alice" and the variable c the data value $c1$. the predicate with symbol $login$ evaluate to true for the arguments pair $("alice", "c1")$ and false with other argument pairs. The interpretations of $logout$ and $reset$ are empty, meaning that they evaluate to false for any possible argument. In future notation, we omit empty predicates interpretation.

We also define partial functions $f : A \operatorname{def} B$. We write $[a_1 \mapsto f(a_1), ...a_n \mapsto f(a_n)]$ for f with $dom(f) = a_1, ...a_n$. [] denotes the "undefined everywhere" partial function. We write $f[a \mapsto b]$ to note an update to the domain and codomain of f with value a and b. Finally, we write $f \leq g$ for f, g partial functions so that $dom(f) \in dom(g)$ and $f(a) = g(a) \forall a \in dom(f)$. We call partition functions from V to D *valuations.*

We call a *timepoint* a point in time if some event is received with that time as timestamp. This is opposed to *gaps,* time intervals in which we have not received any events so far, but where we may still receive some. For each timepoint, the monitor should eventually produce a *verdict,* a Boolean value denoting whether the specification holds at this timepoint.

2.2 Specification languages: MTL, MTL$^{\downarrow}$

In this work, we use Metric Temporal Logic (MTL), MTL$^{\downarrow}$ and MTL$_{prenex}$ as specification languages. In this section, we give a reminder of the MTL and MTL$^{\downarrow}$, then define MTL$_{prenex}$. For more details about MTL and MTL$^{\downarrow}$, we direct the reader to section 2 of [3].

MTL$^{\downarrow}$ The syntax of MTL$^{\downarrow}$ is defined as follows:

$$\varphi ::= t \mid p(x_1, ..., x_n) \mid \downarrow^{r} x.\, \varphi \mid \neg\, \varphi \mid \varphi \vee \varphi \mid \mathsf{U}_I\, \varphi \mid \mathsf{S}_I\, \varphi \mid \bigcirc_I \varphi \mid \bullet_I\, \varphi \tag{2.1}$$

We recall the semantics of the MTL$^{\downarrow}$ logic in figure 2.1. The semantics are defined recursively using a *satisfaction relation* $\models$ between a MTL$^{\downarrow}$ specification and a tuple composed of the indexed event stream S, its index i at which the specification is evaluated, and a partial function $v : V \mapsto D$ assigning variables from V to data values from D. The stream is the indexed sequence $<\tau_0, \varrho_0, \sigma_0>$, $<\tau_1, rho_1, sigma_1>$, ..., $<\tau_j, rho_j, sigma_j>$, ...

We use the standard syntactic sugar to define the constant and operators false $\mathsf{f} = \neg \mathsf{t}$, or $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, eventually $\Diamond_I\, \varphi = \mathsf{t}\ \mathsf{U}_I\ \varphi$, always $\Box_I\, \varphi = \neg \Diamond \neg\varphi$, and their past counterparts $\blacklozenge_I\, \varphi = \mathsf{t}\ \mathsf{S}_I\ \varphi$, historically $\blacksquare_I\, \varphi = \neg \blacklozenge \neg\varphi$.

Operator	**Satisfaction relation**	**Truth value**
True constant	$S, i, v \vert\!\approx \varphi$	t
Predicate symbol	$S, i, v \vert\!\approx pred(x_1, ...x_m)$	if $v(x_1, ...x_m) \in \sigma_i(pred)$
Freeze quantifier	$S, i, v \vert\!\approx \downarrow^{\overline{reg}} \overline{var}.\, \varphi$	if $w, i, v[\overline{var} \mapsto \varrho_i(\overline{reg})] \vert\!\approx \varphi$
Not	$S, i, v \vert\!\approx \not\varphi$	if $\neg(S, i, v \vert\!\approx phi)$
Or	$S, i, v \vert\!\approx \varphi \vee \psi$	if $(S, i, v \vert\!\approx \varphi) \vee (S, i, v \vert\!\approx \varphi)$
Since	$S, i, v \vert\!\approx \varphi \,\mathsf{S}\, \psi$	if $\bigvee_{j \in \mathbb{N}, j \leq j} (\tau_i - \tau_j \in I \wedge (S, j, v \vert\!\approx \psi) \wedge \bigvee_{j < k \leq i} (S, k, v \vert\!\approx \varphi))$
Until	$S, i, v \vert\!\approx \varphi \,\mathsf{U}\, \psi$	if $\bigvee_{j \in \mathbb{N}, j \geq j} (\tau_j - \tau_i \in I \wedge (S, j, v \vert\!\approx \psi) \wedge \bigvee_{j \leq k < i} (S, k, v \vert\!\approx \varphi))$
Previous	$S, i, v \vert\!\approx \bullet\, phi$	if $i > 0 \wedge \tau_i - \tau_{i-1} \in I \wedge (S, i-1, v \vert\!\approx \varphi)$
Next	$S, i, v \vert\!\approx$	if $\tau_{i+1} - \tau i \in I \wedge (S, i+1, v \vert\!\approx \varphi)$

Figure 2.1 – Semantics of MTL$^\downarrow$

For instance, the property expressed in the introduction would be written as $\Box \downarrow^{r_u, r_c} user, computer.\, login(user, computer) \rightarrow \Diamond(logout(user, computer) \vee reset(computer))$.

MTL We can define MTL as the fragment of MTL$^\downarrow$ where formulas have no freeze quantifier and all predicate symbols have arity 0. This is also called the *propositional setting*, because all predicates are propositions, as opposed to the *parametric setting*, when predicates are parameterized with variables and interpreted with data values.

2.3 The Polimon algorithm

Polimon is the runtime verification tool developed in [3]. Its distinguishing feature is its ability to process online, out-of-order streams. Polimon operates over MTL and MTL$^\downarrow$ specifications domains. It processes stream messages similar to those described in section 2.1.

In order to support out-of-order processing, polimon relies on a directed acyclic graph structure where it aggregates events to produce truth values. The monitored formula is encoded as a tree-like computation graph The top node is the outermost operator of the formula, the intermediary nodes and vertices follow the structure of the syntax tree of the formula. The leaf nodes represent atomic subformulas : predicates (including propositions) and constants. Such a graph is created for each timepoint known to the monitor. Upon reception of an event, the graph update state is as follows:

1. A new root node and the matching tree structure are added to the graph, ordered by time.
2. If the registers contain data values, these data values are frozen at the freeze quantifier nodes, and propagated downwards to predicate nodes, where they can be used to evaluate predicates
3. The predicate interpretations contained in the message are used to evaluate the leaf nodes

at this timepoint. Truth values are propagated upwards, and nodes that propagated their truth values and are no longer needed can be deleted.

4. values propagated to root nodes are outputted as verdicts.

This structure allows Polimon to efficiently process events out of order, adding new nodes and deleting old nodes online. This brief description leaves out how parts of the graph are duplicated to handle multiple data values at once, and how knowledge gaps are represented in this graph. These details are found in [3].

A similar structure is used to process MTL and MTL$^{\downarrow}$ streams. However, because they do not have data values, processing MTL streams is significantly faster than processing MTL$^{\downarrow}$ streams of similar characteristics (formula and stream structure, propositional messages instead of messages with data values). For this reason, we use polimon MTL monitors as our submonitors.

Chapter 3

Design

In this chapter, we describe our design decisions. We give the reader a very high level view of our architecture to develop intuition. Then we dive into the details and concepts driving our system. We finish with a bigger picture showing how the system fits together.

3.1 Overview

High-level view Our system is pictured in figure 3.1. Our system receives as input events from an out-of-order events stream. Two types of components are needed to process these events. The dispatcher runs as a unique, non-terminating instance. it receives stream events, and sends them to the workers for processing. The goal is to process the stream in parallel. To that end the dispatcher determines for each event on or more substreams which it should be part of. the goal of substreams is to group together events that should be processed together.

Substreams are processed by workers, each worker being dedicated to a particular substream. The figure shows in the workers' box the key identifying the substream they process. Workers are started only when needed. The dispatcher decides when to start a new worker, depending on which events are received and on which substreams are need the event to compute correct verdicts. Workers are started by cloning a worker with a smaller compatible key. The cloned worker inherits the state of its ancestor, which provides it with the information it needs about events received earlier, that may be needed for its verdicts computation.

The dispatcher and the workers communicate over communication channels that we assume reliable. They exchange either events, or control messages. The workers can receive event with data values from the dispatcher, partialize them, and input them to an underlying Polimon MTL monitor. They produce verdicts that are output on the verdict stream. Workers can also receive control messages. 'clone' control messages instruct them to clone their start and start a

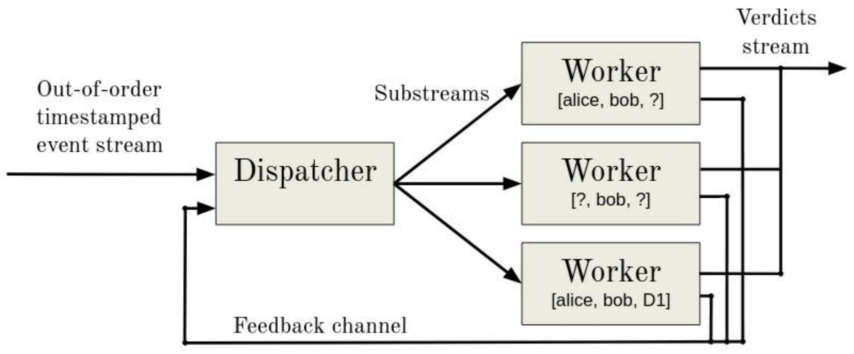


Figure 3.1 – High-level system view

new worker with it. 'terminate' messages instructs the worker to finish processing the events it received and exit. The workers can also request termination from the dispatcher by sending a 'termination request' control message on the feedback channel. However, no stopping criteria is currently implemented so the workers never request termination.

3.2 Setting

MTLprenex We introduce MTL $_{prenex}$ as a fragment of MTL $^{\downarrow}$. Formulas in MTL $_{prenex}$ use the same syntax as MTL $^{\downarrow}$, but have additional constraints. They contain no Next $\bigcirc$ or Previous $\bullet$ temporal operator. They have a single top-level freeze quantifier. Finally all their temporal operators are *guarded.*

next, previous Additionally, the $next$, $\bigcirc$ and $previous$, $\bullet$ operators are not supported by our monitor. Supporting them leads to partitioning the stream into a single substream equal to the stream itself, which defeats the purpose of partitioning. We present a quick illustration.

As an example, we evaluate the formula $\bigcirc(\varphi)$ at timepoint τ_1 with timestamp t, where φ is a non-trivial subformula (not $true$ or $false$). Suppose φ is in the earliest case true at timepoint τ_2 with timestamp $t+\epsilon$. The stream is partitioned so that a partition that contains τ_1 also contains τ_2. The existence of a timepoint τ_x between τ_1 and τ_2 determines the truth value of $\bigcirc\psi$ at τ_1. If there exists such a timepoint, then the formula evaluates to false, and if there exists none, the formula evaluates to true. Note that the existence of any stream event with timestamp between t and $t+\epsilon$ creates such a timepoint. As a consequence, to evaluate the formula, a submonitor should be aware of all stream events. This makes partitioning pointless, as one would obtain a single partition containing all the events. A similar argument can be made for the $\bullet$ operator.

Top-level freeze Currently, our monitor supports only formulas of the form $\downarrow^{regs} vars.\, \varphi$, where phi is free of freeze quantifier. It is unclear whether nested freeze quantifiers could be supported, in particular due to event partializing (defined in section 3.4). For simplicity, we decide restrict freeze quantifier to a single, top-level freeze quantifier that can freeze an arbitrary number of values.

guards We define guarded for the Until U and Since S operators. The definition follows for always $\square$, eventually $\diamondsuit$, historically $\blacksquare$ and once $\blacklozenge$, as these operators are syntactic sugar.

The Until temporal operator is guarded in a formula of the form $(\alpha_1 \to \varphi_1) \,\mathsf{U}\, (\alpha_2 \wedge \varphi_2))$, where α_1 and α_2 are predicates (that can be of arity 0, propositions). α_1 and α_2 are called the *guards*. The same form and conditions on α_1, α_2 are necessary to guard a Since operator: $(\alpha_1 \to \varphi_1)\mathsf{S}(\alpha_2 \wedge \varphi_2))$. We also derive the following forms for other temporal operators using syntactic sugar. guarded always, $\square\, \alpha \to \varphi$, guarded eventually, $\diamondsuit\, \alpha \wedge \varphi$, guarded historically, $\blacksquare\, \alpha \to \varphi$, guarded once, $\blacklozenge\, \alpha \wedge \varphi$, where α is a predicate.

The guards guarantee that the events a submonitor needs to produce a verdict belong to its substream. We illustrate the problem with the non-guarded formula $\downarrow^{regs} vars.\, \diamondsuit\, \neg p(vars)$. Suppose that a submonitor runs for some substream with valuation v. Any event event where p does not hold results in a violation. The problem is, p can hold or not hold independently of events substream. In this case, the submonitor may miss crucial events.

Guarded formulas avoid this problem. The guard is a predicate interpretation. For values for which the guard holds, the data values guarantee that matching partitions receive the event. If the guard is false, then the formula is trivially true, which solves the problem

3.3 Substreams and keys

Our design split the input stream into substreams that can be processed in parallel by workers. In this subsection, we introduce the concept of *key*, which is central to our design: keys identify substreams, workers, are used to determine how to clone workers, clone workers, translate data-valued messages into propositional messages... We define the properties of the keys, and explain their purpose in our design

Definition. A key is a partial data valuation. That is, a partial function $V \to D$ from the set V of variables in the monitored formula to the set D of possible data values.

Notation. In addition to the partial function notation defined in the background section, we define a notation for the keys. For a given MTLfreeze formula, the variables are indexed by the leading freeze operator in some order. We write the valuation between square brackets, and write comma-separated data values for assigned variables, or '?' for variables with no assigned value, in the order given by the FREEZE operator. For example, for the formula $\downarrow^{r_x, r_y} x, y.\, P(x) \rightarrow \Diamond Q(y)$ and the valuation $[x \mapsto "alice"]$, we would note the key as $[alice, ?]$.

Properties. We define several properties on the keys.

Compatibility. Keys k_1 and k_2 are compatible if and only if $\forall v \in dom(k_1) \cap dom(k_2).\ k_1(v) = k_2(v)$. Example: $[alice, ?]$, $[alice, bob]$, $[?, bob]$ are mutually compatible, but only the last one is compatible with $[bob, ?]$.

Partial order. We define a partial order $b \leq a$ ("a refines b") on keys. $k_1 \leq k_2 \iff dom(k_1) \subseteq dom(k_2) \land \forall v \in dom(k_1).\ k_1(v) = k_2(v)$. Example: $[alice, ?] \leq [alice, bob]$

Comparability. If neither $k_1 \leq k_2$ nor $k_2 \leq k_1$, we say that k_1 and k_2 are incomparable. ($[alice, ?]$, $[?, bob]$), ($[alice, ?]$, $[bob, ?]$) are incomparable.

Combination. We define the combination of two keys as follows:

$$combination(k_1, k_2) = \begin{cases} k_1 \cup k_2 & \text{if compatible}(k_1, k_2) \\ \perp & \text{otherwise} \end{cases}$$

where $\cup$ denotes the union for partial functions and $\perp$ denotes an error.

keys, substreams and workers Our goal is to split the stream based on data values. Substreams are processed in parallel by workers. Keys are a link between the two. A substream is a sequence of events taken from the stream, which are processed according to some data valuation. This data valuation is the key. In that sense, substreams and workers are identified and matched by keys. Substreams are inferred from data valuation carried by stream events. Keys identify the substream in the sense that keys are extracted from events based on their data valuations, and are used to group events together in substreams. Keys identify the workers in the sense that one worker processes one substream and that they process this substream according to the data valuation provided by their key (workers are actually parameterized buy a key at start).

Creating keys. Our design splits the data stream on the data values. To build some intuition, consider a simple in-order setting with no past temporal operators. Events arrive ordered by time, and temporal operators only refer to events in the future.

Upon receiving an event, we want to decide to which partitions it belongs. We generate a first key by taking the valuation of the registers of the event. Then, we consider the predicate interpretations.

We first consider the content of the registers. If the registers hold a new valuation, then this is a new partition and we start a worker for it. Then, we consider the predicate interpretations. If the valuation formed by some predicate evaluation is equal to some existing key, clearly there exists a worker that needs to know about this event. We also generate a key for the valuation of the predicate interpretation.

As an example, consider the formula $\varphi = \downarrow^{r_x, r_y} x, y.\ \ P(x) \rightarrow \Diamond Q(y)$, and the event $e = <\tau, [r_x/alice],\ P = \{(bob)\},\ Q = \{\}>$. At time τ, register r_x holds the value 'alice', predicate P evaluates to true for arguments {('bob')} and Q is always false. This event is relevant to workers processing the partitions $[alice, ?]$, due to the frozen data values, and $[bob, ?]$, due to the predicate interpretation.

Figure 3.2 ExtractKeys

```
1: procedure ExtractKeys($\varphi$, $e$)
2:     $<\tau, \varrho, \sigma> := e$
3:     if $\varphi = \downarrow^{\overline{regs}} \overline{vars}.\ \psi$ then
4:         return $\{r\} \cup ExtractKeys(\psi)$
5:     if $\varphi = predicate(x_1, ...x_j)$ then
6:         return $\{[x_1 \mapsto v_1^i, ...x_j \mapsto v_j^i] \mid \forall (v_1^i, ...v_j^i) \in p(predicate)\}$
7:     if $\varphi = not\ \psi$ then
8:         return $ExtractKeys(\psi)$
9:     if $\varphi = \psi \vee \gamma \mid\mid \varphi = \varphi \mathsf{U} \gamma \mid\mid \varphi = \varphi \mathsf{S} \gamma$ then
10:        $keys_1 := ExtractKeys(\psi)$
11:        $keys_2 := ExtractKeys(\gamma)$
12:        return $keys_1 \cup keys_2 \cup$
13:            $\{Combine(k_1, k_2) \mid \forall (k_1, k_2) \in keys_1 \times keys_2 : Compatible(k1, k_2)\}$
       return $\{\}$ // Else, constant symbol
```

ExtractKeys procedure. The procedure $ExtractKeys$ formalizes the key extraction from an event. The pseudo code is given in figure 3.2. It differs from the simple case seen in the previous paragraph in two things. In more complex formulas, a predicate symbol can appear several times and index different variables. This is why we need to follow the formula structure and generate all possible key combinations. Additionally, in a more general case, we do not know if we receive the message containing the registers valuations or the predicate interpretations first (due to out-of-orderness or presence of past temporal operators). This is why we generate both the keys from the predicate interpretations valuations and the register valuation.

3.4 Partializing

One key element of our design is that workers do not process events in the same setting as a sequential algorithm would. Our design features a translation step, carried out by the worker before feeding events to the wrapped monitor. In our implementation, the worker translates the event from the MTLfreeze setting to the MTL setting. This is beneficial for performance, as the polimon MTL monitor is significantly faster than the MTLdata monitor, as shown in the evaluation section of [3]

Formula rewriting. The system monitors a MTL$^{\downarrow}$ formula. The workers monitor a MTL formula derived from the MTLfreeze formula. The rewriting consists of two steps: every predicate must be replaced by a proposition; and the leading freeze operator must be removed or replaced. We replace freeze operators $\downarrow^{regs} vars.\ \psi$ by a leading 'freeze implies' $freeze \to \psi$ in the MTL formula, where $freeze$ is a fresh proposition.

Additionally, a predicate symbol may appear several times with different arguments. In this case, each instance of the predicate may have a different interpretation, so a different truth value. To reflect this scenario in our MTL formula, we number predicate instances so that predicate symbols indexing different variables are transformed into different propositions that can have different truth values.

To illustrate this with an example:

$$\psi_{MTLfreeze} = \downarrow^{r_{u1}, r_{u2}, r_D} u_1, u_2, d.\ transfer(u_1, u_2, d) \to (\blacklozenge\ login(u_1) \lor \blacklozenge\ login(u_2)) \tag{3.1}$$

is rewritten as:

$$\psi_{MTL} = freeze \to (transfer_1 \to (\blacklozenge\, login_2 \lor \blacklozenge\, login_3)) \tag{3.2}$$

There are no predicates or variables left in ψ_{MTL}, only propositions. The $login$ predicate symbol is rewritten into different propositions. For simplicity, the predicates symbols are numbered by their order of appearance.

Figure 3.3 Partialize

```
1: procedure PARTIALIZE($e$, $k$)
2:     $<\tau, \varrho, \sigma> := e$
3:     let $\sigma_2 : P \mapsto B$
4:     with $\sigma_2(p) = \begin{cases} \varrho = k & if p = freeze \\ true & if \varrho(p) \in k \\ false & otherwise \end{cases}$
5:
6:     return $<\tau, [/], \sigma_2>$
```

Events partializing. The workers wrap a MTL monitor, but receive MTLdata events from the dispatcher. A translation step is required before the underlying monitors can process the events. We call it *partializing*, because it reduces the problem to a simpler domain.

When receiving an event, a worker must generate a truth value for each predicate. For this, the worker compares each predicate interpretation p_j contained in the event with its own key k. Both are partial functions $V \mapsto D$. If $p_j \le k$, then the matching proposition is true, else it is false.

This causes a problem: suppose there exist two workers with different keys. Upon receiving an event, they might partialize it differently, leading to two different verdicts. Clearly, at most one of them can be correct. The $freeze$ proposition solves this problem by imitating the behavior of the $FREEZE$ operator in the MTLdata setting. There exists a single worker that interprets the predicates in the same way as they would be interpreted in the MTLfreeze setting: the worker with a key equal to the registers valuation. This single worker sets its $freeze$ proposition to true. Other workers, whose key is not equal to the registers valuation, set it to false, and their verdicts become "trivially true", as $false \rightarrow \psi$ is true regardless of ψ.

This leads to a final problem: for a given event, we may produce several trivially true verdicts, along with the correct verdict. To avoid this, we suppress output to the workers whose verdicts is a "trivially true" verdict. As a result, at a given timepoint, we only consider the verdict produced by the single worker whose $freeze$ proposition is true.

The procedure to partialize events is shown in figure 3.3

We argue that this way to partialize events is correct. When monitoring a MTL $_{prenex}$ formula on non-partialized events, a verdict if produced only if the freeze operator evaluates. For this, it needs to freeze data values and propagate the values to the operators below. So, all operators being the top-level freeze operator receive their data values from it, in particular the predicates, which evaluate by comparing the received data values and predicate interpretations. In comparison, our setting consists of multiple workers, one for each combination of data values frozen. In our setting, the predicates get their values from the worker's key, not from a freeze gate. However, Since we start a worker for each data valuation that is frozen or that can be frozen in the future, there exists a worker in which the valuations is identical to the one that is in the event's registers. Note also that because of the $freeze$ proposition, only the single worker with key identical to the content of the event's registers can produce output. If this worker output, then the predicates evaluated to produce this output have been evaluated with the same data values as would have been provided by a freeze operator. Thus, when a worker outputs, its verdict is identical to the one that would have been produced by a non-partialized monitor.

3.5 Cloning workers

Our design relies on cloning workers. We slightly abuse the language: cloning means copying the underlying polimon graph structure of a given worker (the *ancestor*), and starting a new worker (the *clone*) with the copy of the graph structure as its underlying monitor, and with a different key.

In this section, we describe under which conditions workers can be cloned, and argue why it is correct for the clone to use the state of the ancestor at start, even though they would partialize events differently due to their different keys.

Cloning rules A worker with some key k_w processes the stream partition matching that key. It receives all events e so that $\exists k_j \in ExtractKeys(e).\ k_j \leq k_w$. There exists at most one worker for a given partition. We define cloning rules to match the partial order on partitions. A worker with key k_{w_2} can be cloned from another worker with key k_{w1} if and only if $k_{w1} \leq k_{w2}$. We make a few statements about this.

First, If w_1 with k_1 is cloned from w_2 with k_2, $k_2 \leq k_1$, w_2, then w_1 would have received the same set of messages as k_2 received before the moment of cloning if it had been running since the beginning. Note that the dispatcher issues the 'clone' message at the first event with key k_1 that it receives. This implies that before the cloning, no message with key k_1 is received. Note that according to our substreams rules, w_1 receives all messages with keys $\leq k_1$. This means that it receives all messages that w_2 receives, since $k_2 \leq k_1$. And since w_1 gets cloned from w_2, this means that there is no other worker with key k so that $k_2 \leq k \leq k_1$ otherwise w_1 would clone from the other worker. Consequently no key extending k_2 are received, and w_1 would have received the same set of messages as k_2 received before the moment of cloning if it had been running since all along.

Additionally, When cloning w_1 with k_1 from w_2 with k_2, any message received and partialized by w_2 would have been partialized exactly the same by w_1 if it had been running all along. Since w_1 is cloned from w_2, we use the same argument to argue that no messages with key extending k_2 have been received until cloning. By definition $k_2 \leq k_1$, the assignment in k_1 and k_2 are the same over the domain of k_2. Since no message holds data values that extend k_2, all messages would have been partialized in the same manner.

As a result. When cloning w_1 with k_1 from w_2 with k_2, w_1 is an identical state as if it had been running all along, receiving messages and partializing them according to its own key. This is obviously not true after receiving the first message with key extending k_2.

3.6 Architecture details

After exposing the necessary concepts, we come back to a high-level overview of our system, in more details this time. Here, we will state some hypotheses necessary for the system to work properly, and, and give pseudocode for the dispatcher and workers

We provided in figure 3.1 a high-level view of our system is provided in Figure 3.1. We make a few hypotheses on these channels and how the components communicate. The channels provide reliable in-order delivery. The messages arrive in order, are not modified, nor dropped. Additionally, it must be possible to exchange information in both directions, as shown by the feedback channel. This requirement is necessary to allow workers to request termination to the dispatcher.

Note that these requirements are not necessarily tied to a particular implementation.

Figure 3.4 Dispatcher algorithm

```
1: procedure DISPATCHER($\varphi$)
2:     $w0 \leftarrow Worker(\varphi, [\,])$
3:     $K_{workers} \leftarrow \{[\,]\}$
4:     loop
5:         $e \leftarrow receive\_event$
6:         $K_e \leftarrow ExtractKeys(e)$
7:         $K_{new} \leftarrow \{(k_e, k_w) \in K_e \times W_{workers}.\ Compatible(k_e, k_w) \wedge Combine(k_e, k_w) \notin K_{workers}\}$
8:         $K_{start} \leftarrow \{(k_e, k_w) \in K_{new}.\ \forall (k_1, k_2) \in K_{new}.$
9:                 $Combine(k_e, k_w) = Combine(k_1, k_2) \rightarrow !(k_w \leq k_2)\}$
10:        for $(k_{ei}, k_{wi}) \in K_{new}$ do
11:            $k_{new} \leftarrow Combine(k_{ei}, k_{wi})$
12:            Send($k_{wi}$, $<$'clone'$, k_{new} >$)
13:            $K_{workers} \leftarrow K_{workers} \cup \{k_{new}\}$
14:        for $k \in K_{workers}$ do
15:            if $\exists k_e \in K_e.\ k_e \leq k_w$ then
16:                Send($k_w, e$)
```

Dispatcher algorithm The dispatcher is described by algorithm 3.4. The keyword $Send(k, e)$ denotes the sending of the message/event e to the worker with key k, through a channel with the properties described at the beginning of section 3.6. Single uppercase letter variable names denote sets. Not pictured in this pseudocode: the dispatcher implements a termination protocol to allow workers to request termination without losing information. The dispatcher maintains an incremental sequence number for each worker. When a worker wants to terminate, it sends a 'terminate' control message on the feedback channel. along with the sequence number of the message it last received. The dispatcher compares the two sequence numbers. If they are equal. there is no message in flight for this worker, so the dispatcher allows the worker to terminate.

We provide some intuition on K_e, K_{new} and K_{start}. Then, we provide an example to argue why combining keys from $K_{workers}$ and K_e is necessary to start the necessary workers.

We have two concerns : start all the necessary workers, and start each worker by cloning the existing worker with the biggest possible compatible key. We need to start workers for each new key in K_e and each new key that can be obtained by combining keys from $K_{workers}$ and K_e. Note that the first set is a subset of the second, since $k_{workers}$ contains the *undefined everywhere* key. K_{new} corresponds to that second set. Note that combining pairs from K_{new} may produce several times the same key. Then, we create K_{start} by filtering out elements from k_{new}. For each key obtained by combining pairs from K_{new}, we retain only the pair where k_w is the biggest.

Now, why is it necessary to combine keys from $K_{workers}$ and K_e ? We provide an example for this. Assume a dispatcher monitoring our running example formula. The dispatcher receives events $< 0, login(alice) >$ and $1, login(bob)$. Without combining the keys, the following workers are started: $\{[alice, ?, ?], [?, alice, ?], [bob, ?, ?], [?, bob, ?]\}$. Note that the two $alice$ workers have no knowledge of the bob event, since their keys are incomparable. Then, the dispatcher receives a $< 2, transfer(alice, bob, D_1) >$ event. Clearly, the dispatcher should start a $[alice, bob, D_1]$ worker, which should produce a $true$ verdict . However, none of the existing workers contain the necessary timepoints: none received both the $alice$ and bob events. With combining, when receiving the bob event, workers $\{[alice, bob, ?], [bob, alice, ?]\}$ are cloned from workers $[alice, ?, ?]$ and $[?, alice, ?]$ respectively. As a result, their state accounts for the $alice$ event. Then, workers $[alice, bob, ?], [bob, alice, ?], [bob, ?, ?], [?, bob, ?]$ receive the bob event. Upon receiving the $< 2, transfer(alice, bob, D_1) >$ event, worker $[alice, bob, ?]$ clones its state to start worker $[alice, bob, D_1]$. Which then processes the event and produces a $true$ verdict.

Figure 3.5 Worker algorithm

```
1: procedure WORKER($\varphi, k_m, monitor$)
2:     $\psi \leftarrow ToMtl(\varphi)$
3:     if $monitor = nil$ then
4:         $monitor \leftarrow MtlMonitor(\psi)$
5:     loop
6:         $e \leftarrow receive\_event$
7:         if $e = < \tau, R, P >$ then
8:             $e_{MTL} \leftarrow Partialize(psi, k_m, e)$
9:             $monitor(e_{MTL})$
10:        if $e = <$ 'clone'$, k_{new} >$ then
11:            $clone \leftarrow Clone(monitor)$
12:            Start $Worker(\varphi, k_{new}, clone)$
13:        if $e = <$ 'terminate' $>$ then return
```

worker algorithm The worker is described by algorithm 3.5. The $Start$ keyword denote the action of asynchronously starting a procedure. We abuse the notation $monitor$ to denote both the state of the MTL monitor and its functionality. $monitor$ is the state of the monitor, and $monitor(e)$ is a function that accepts a MTL event, updates the state, and outputs some verdicts.

A worker wraps an instance of a Polimon MTL monitor. The worker handles the control messages created by the dispatcher, and performs transformation step on the events before the MTL monitor.

Upon reception of a stream event, the worker translates the event to MTL according to its key, using the procedure $Partialize()$ described in section 3.4. The new event is processed by the MTL monitor.

The dispatcher and workers exchange control messages to request cloning and request/demand termination. Note that the cloning procedure is carried out in the worker themselves, not the dispatcher. The worker receives termination messages in two scenarios. Either it sent a termination demand to the dispatcher previously, or the dispatcher is terminating and sends termination requests to all its workers. In both cases, the worker terminates immediately.

Finally, we make a few statements to argue the correctness of our system.

If a worker is needed to produce some output, then it has been created. Remark that in order to produce an output in the general setting (without substreams and submonitors), the top-level freeze operator needs to produce an output. The freeze operator only produces an output when it freezes variables to some data values, which it finds in the registers of an event. In our algorithm, upon receiving an event, a key is created for the valuation of the registers. As a result, the worker needed to output said verdict is started by the dispatcher if it did not already exist.

To complete the statements made in section 3.3, we can now argue that whenever a worker must be started for key k, there exists a single worker with key k_c so that $\forall k_o.\ k_o \preceq k \implies k_o \prec k_c$. I.E. there exists a single "best candidate", and not multiple candidate with mutually incomparable keys. First, there always exists at least the worker with the empty key, as any key extends the empty key and this worker is always started. Additionally, suppose by contradiction that there are two workers with keys k_1, k_2 so that $k_1 \preceq k \wedge k_2 \preceq k$ and $k1$, $k2$ are incomparable. Clearly, since k extends them both, they are compatible. Since they are incomparable, it means that each of them has a value of their domain that the other does not have. As a result, merging these two keys would result in a key strictly bigger than both of them, which k extends. No matter how k_1 and k_2 have been constructed, be it at once, or by successive cloning of existing runners, that key would have been created in the merge step of our algorithm, which is a contradiction. As a result, there exists a single "best candidate" to clone any new runner.

Chapter 4

Implementation

In this chapter, we cover implementation details of the project. We motivate our choice of programming language for implementation. We describe the existing codebase and provide an overview of our implementation work. We describe interesting optimizations. Finally, we suggest possible extensions.

4.1 Golang

We implement our design using the golang programming language. In this section, we motivate the choice of golang for this project. Golang offers built-in concurrency primitives for scalability, and profiling tools for optimization. In this section, we give more details about the concurrency primitives.

Goroutines are a central primitive for concurrency in golang. A goroutine is "a lightweight thread managed by the golang runtime" (https://tour.golang.org/concurrency/1). Goroutines have several advantages over OS threads. Starting a goroutine requires less memory than an OS thread, thanks to goroutines' dynamically-sized stack. A newly started goroutine allocates by default 4Kb of memory. For comparison, a POSIX thread allocates by default 2MB on the x86_64 architecture, 500x more; or 1MB for a JVM thread.

Additionally, the golang runtime schedules these goroutines, not the OS scheduler. This has two benefits. The runtime knows "more" about that the goroutines have useful work to do than the OS know about threads, which allows for better scheduling decisions. Then, when making a scheduling decision, the context-switching cost is smaller for a goroutine than for a thread (the golang runtime actually multiplexes goroutines to run a single OS thread).

Finally, the runtime provides channels for communication between goroutines. Channels allow for typed, buffered, non-blocking communication between goroutines. In particular, system calls are not needed to use channels. If a channel operation is blocking, the runtime can schedule another goroutine to run in the current thread, instead of a costly context switch (that could for example happened with UNIX pipes).

These advantages make golang a very compelling option for writing scalable concurrent programs.

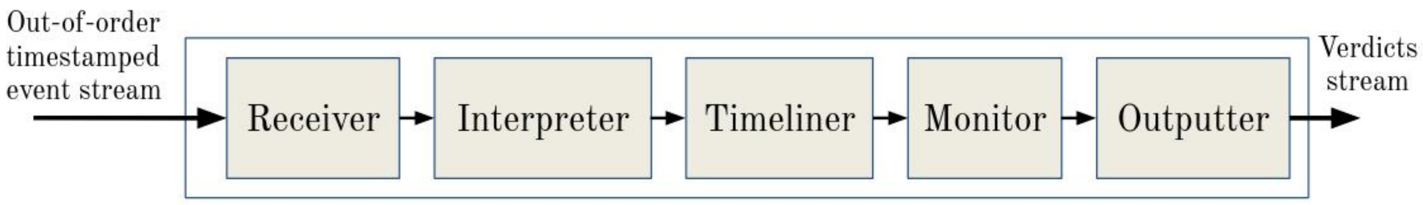
4.2 Overview

The polimon tool. Our monitor is implemented as a component of the polimon monitoring tool. We give a quick overview of the architecture of polimon to define where our implementation fits.

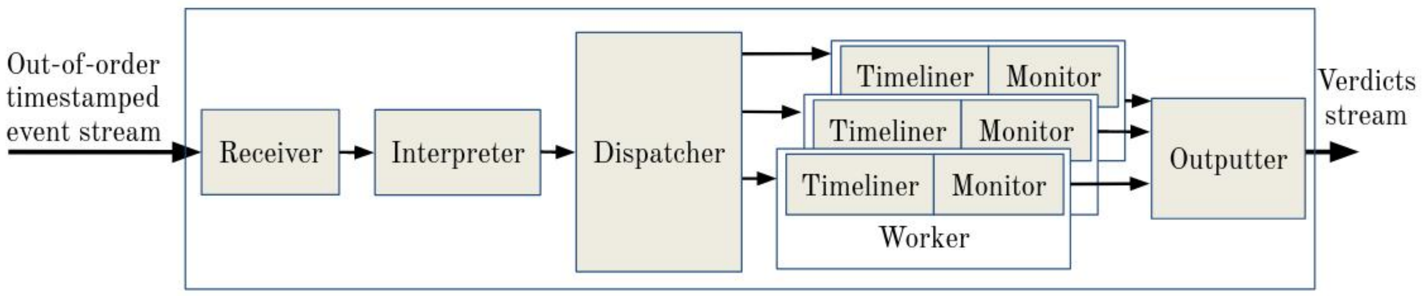
The polimon tool is parameterized by a specification containing a formula in MTL or MTL$^{\downarrow}$, and instructions for events parsing. It takes a sequence of events as input, either through a UDP port or from a local file, and produces a verdict $true$ or $false$ for each timepoint in the stream of events. It is divided in components that carry out different processing steps, and that are laid out in a pipeline as shown in figure 4.1. The receiver is the source of the pipeline. It implements the reception of messages from various sources. The interpreter parses and preprocesses events. The timeliner keeps track of knowledge gaps using sequence numbers, and signals the monitor when knowledge gaps are closed. The monitor maintains a data structure to compute verdicts efficiently. The outputter produces the verdicts stream. The heavy lifting is done by the monitor, and to some extent by the timeliner. Thanks to its pipelined design and to some extent some parallelism in the monitor stage, the original tool also benefits to some degree from parallelism.

Scope of our implementation. Our implementation is a replacement for the timeliner and the monitor stages. It implements the same interfaces as these components and can directly be plugged in place. Figure 4.1 compares the original pipeline and the pipeline with our new component. The dispatcher receives messages directly from the interpreter and directs them to the appropriate workers. The workers output then directly to the outputter. Each worker is a polimon MTL monitor, composed of the timeliner and monitor stage.

Implementation overview We implement the dispatcher and workers as goroutines. They form one pipeline stage in the tool, and interface with the previous and following stages using channels. The dispatcher receives events through its input channel, and the workers writes verdicts into the output channel of the pipeline stage.



(a) Original pipeline



(b) Pipeline with our new component

Figure 4.1 – Polimon pipeline comparison

4.3 Optimizations

We discuss here implementation choices for efficiently carrying out operations described in chapter 3. In particular, operations that would otherwise be costly if implemented naively. We justify why these implementation choices are particularly important for scalability.

Generating K_{start}. In our implementation, the dispatcher performs the costliest sequential operations. It is therefore extremely important to optimize it. The prime target for optimization is the generation of the set K_{start}. A naive implementation would loop over $K_e \times K_{workers}$ to generate K_{start}, then loop again over $K_{start} \times K_{start}$ to filter out pairs with small k_w. Both loops take $O(n \times m)$ time.

We implement the computation of K_{start} efficiently using a prefix tree data structure. Concretely, we directly store $K_{workers}$ in a prefix tree. We present how the keys are represented, how the prefix tree data structure operate, and the benefits it provides.

A key is represented as a slice of slice of bytes. Slices are dynamic array in go. Our representation store each image of the codomain of the partial function (*values*) as a slice of bytes. Partial functions do not sort their preimage-images or images. In our representation, we order the images according to the order or apparition of their preimage-image in the $freeze$ quantifier of the monitored formula. For elements of the domain that have no image, we store a special value called *wildcard*.

A prefix tree [4] is a tree-like data structure that typically store string keys. Each node of the tree stores a substring present in some key. Traversing the tree from root to leaf and concatenating the substrings stored in each traversed nodes reconstructs one stored key. Our prefix tree implementation differs from usual implementations, in that it has a special *wildcard* value which we define to match any existing string. Additionally, for each key we store in the trie, we want to keep the different images separate. As a result, we use as substring the byte slice representing each image at each node. Although it is not the most space efficient way, we still get the main benefit of using a trie, which is efficient lookup traversal.

Prefix tree allow us to efficiently lookup the set of current worker keys for keys that are compatible with some given k_e given as argument. We proceed in a depth-first manner, filtering out nodes with incompatible prefixes. Concretely, we create an empty stack of nodes and enqueue the root of the tree. While the stack is not empty, we pop a node from the stack, and enqueue on the stack all of its children that match the values of k_e. That is, if they exist, the child for the wildcard value and the child for the value matching exactly the current slice of byte in k_e. When we reach a leaf, we return it, else we loop until the stack is empty. Each node stores pointers to its children in a map for constant time access. As a result, the entire operation requires $O(r \times d)$ byte slice comparisons operations, is the length of the key (also the depth of the tree), and r is the number of compatible keys returned.

Cloning Another optimization relies on go primitives to hide the latency due to cloning to the dispatcher. This means, when the dispatcher sends a clone message, it can immediately assume the worker is cloned without delay, and start sending messages to that worker immediately.

Go channels make this possible. A go channel is a first-in-first-out, buffered (in our case) in memory data structure. As a result, a channel can be opened and written to without a recipient. This allows us to implement the cloning process as follows:

1. The dispatcher opens a new channel
2. The dispatcher sends a clone message containing a pointer to the channel
    (a) A worker receives the clone message and starts cloning the state of its monitor
    (b) The worker finishes its cloning its state and starts the new worker in a new goroutine
    (c) The new worker begins reading from the channel
3. The dispatcher sends events on the channel

Thanks to the go channels doing the buffering for us, points 3. can be interleaved at any moment before, between or after points (a, b, c), effectively hiding any cloning latency from the dispatcher.

That being said, a protocol for cloning could also be implemented without go relying on go primitives.

1. The dispatcher starts a stateless worker for the new key, and the matching
2. The stateless worker buffers any received event until it is initialized with a state

3. The dispatcher send a clone message to the ancestor worker and remembers the "ancestor key $\rightarrow$ stateless worker key" mapping
   (a) the dispatcher sends events to the new worker
4. The worker receives the clone message and serializes its state
5. The worker sends the serialized state along with its key via the feedback channel
6. The dispatcher reads the feedback channel, looks up the ancestor key, and sends the serialized state to the stateless worker
7. The stateless worker initializes itself, then process any event it had buffered in the meantime

Here again, point (a) can interleave before, between or after any of points (4, 5, 6, 7). This protocol would also allow the Dispatcher to ignore the cloning latency, leaving most of task to both workers.

Genericity Finally, the system has does not rely on knowledge specific to the problem at hand. Concretely, the dispatcher and the workers algorithms do not use directly any task-specific runtime monitoring concept. Instead, the are parameterized by functions that encapsulate that knowledge. Namely, the dispatcher takes as argument the function $GetKeys()$ to extract keys from an application message, and the workers a function $Partialize()$ to convert messages from the outer setting to the setting suitable for the monitoring program they interface with.

4.4 Extensions

In this section, we discuss possible extensions to this work.

Multi-stage dispatching Reducing the fraction of single-threaded work in a design is beneficial for scalability. Here, the dispatcher carries out most of it. One could think of alternative designs to reduce the amount of work carried out by the dispatcher. In this alternative design, a single entity would have the role of both worker and dispatcher. Instances of this entity are created under the same conditions as the workers. Each instance contains an underlying monitor, and carries out the same task as a worker : handling control messages, partializing events and feeding the underlying monitor. Additionally, each instance also responsible for forwarding events and messages to others. Instances are arranged in a tree-like structure, similarly to the nodes of the trie. Each instance would decide of the routing for a particular index of the key. For instance, these exists a first instance at depth 0 in the tree, which forwards events to the workers at depth 1 based on the data valuation at index 0 in the event. It also partializes and feeds events to its underlying workers if the events have a key.

There are trade-offs in this design. Indeed, the work of the dispatcher is split across more instances. And in particular, no single instance needs to compare the entire key like the dispatcher does, only a single value at a given index. So while there still exists a single instance at the root of the tree that processes the entire event stream, it has less work to do than the current dispatcher. Ideally, the nodes closer to the leaves of the tree would spend more time cloning and running Te underlying monitor, while nodes closer to the root would spend more time forwarding messages, leading to a relatively balanced load. However, this design would significantly increase the message passing cost. A single amasses could be forwarded as many times as the tree is deep, whereas it could only have been passed once in our current design. Quantifying this trade-off, less single threaded work versus more message passing, could be interesting.

DSPF Additionally, the design does not constraint the implementation to remain single-node. This design could be implemented in a distributed stream processing framework as long as the hypotheses listed in section 3.6 are respected. In particular, a DSPF should support stateful computation as our worker have state. It should allow workers to be initialized with an arbitrary state, this is essentially what we are doing with cloning. It should allow the implementation of the feedback channel, from workers to the dispatcher. Storm, Flink, spark or kafka would be good candidates for this extension.

Chapter 5

Evaluation

Evaluation setup Our experimental setup consists of a desktop computer running the Linux operating system Ubuntu 18.04, with 6 physical CPU cores running at 3.3 GHz and 32 GB of RAM. The tool is compiled using the go compiler version 1.14. Hyperthreading is disabled, as it has shown to lead to underwhelming results. We suspect that this problem is due to synchronization cost in our design increasing when some goroutines are schedules on secondary threads, as a faster thread and a slower thread may need to synchronize.

We evaluate our design on temporal formulas P1, P2 and P3 listed in figure 5.1. The event stream and formulas are inspired by compliance policies from the banking domain, and inspired from formulas used for evaluation of [3]. Our event stream describe banking transactions. It contains three kind of predicates. $transaction(cid,\ tid,\ amt)$ denotes the transaction made by customer cid, identified by transaction id tid, with the amount of amt. $report(cid,\ tid)$ denotes that the transaction made by customer cid identified by tid is picked up as a suspicious transaction and reported by the banking system. $unflag(cid)$ denotes that a customer with a previously reported transaction is cleared by the banking system.

We first provide intuition on the meaning of these formulas and explain what each of them aims to illustrate, before we comment on the log files. Formula P1 verifies that any transaction whose amount is superior to \$2000 is reported in a timely manner, at most 5 seconds after the execution of the transaction. It freezes two data values: the transaction ID, and the amount of the transaction. Formulas P2 and P3 verify the same property of the stream: whenever a transaction is considered suspicious, the customer who requested the transaction should not perform another suspicious transaction withing 5 seconds. The difference in formulation is that P2 uses $suspicious$ as a proposition, while P3 uses it as a predicate parameterized by the variable cid. P2 is extremely unfavorable to the dispatched monitor, as according to the dispatching rules, every worker needs to know about any $suspicious$ event. P3 expresses the same property in an optimized way. It is interesting to compare the difference.

We generate synthetic log files. Each log file spans over 60 seconds. We also control for the

$$\downarrow^{r_{tid},r_{amt}} tid, amt.\ (trans(tid, amt) \wedge a > 2000) \rightarrow \Diamond_{(0,5]}\, report(tid) \tag{P1}$$

$$\downarrow^{r_{cid}} cid.\ suspicious \rightarrow \neg(\Diamond_{(0,5]}\, trans(cid) \wedge suspicious) \tag{P2}$$

$$\downarrow^{r_{cid}} cid.\ suspicious(cid) \rightarrow \neg(\Diamond_{(0,5]}\, trans(cid) \wedge suspicious(cid)) \tag{P3}$$

Figure 5.1 – Formulas used for experimental evaluation

degree of out-of-orderness of the events, and the time window for related events (i.e. how far apart a $trans()$ and the related $report()$ or $unflag()$ can be).

We quickly comment on the relation between event rate, duration, out-of-orderness and the *problem size*. A log hard to verify when it contains more events. Therefore, if either the duration or the event rate is increased, the problem size increase. That being said, the problem size also increases with the out-of-orderness of the stream. Indeed, with higher out-of-orderness, events interleave more, so the size of intermediary data structures in the algorithm increase as well. Note that increasing the stream density for a fixed event rate also causes events to interleave more. This is due to how we shuffle our traces: we add a normally distributed "virtual delay" to each event timestamp, and sort the stream on the "virtually delayed timestamp". As a result, denser stream means more interleaving for a the same delay. We can increase the problem size by increasing any of these three quantities. We decide to change the problem size by varying the event rate, instead of varying the duration, then the out-of-orderness in separate experiments.

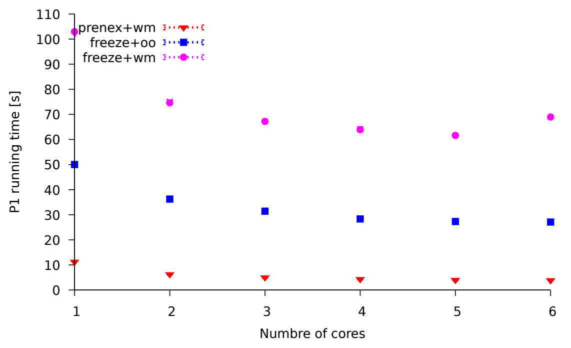
We measure scalability according to two different metrics :

- runtime as a function of core count, while keeping the problem size constant
- runtime as a function of event rate, while keeping the core count constant

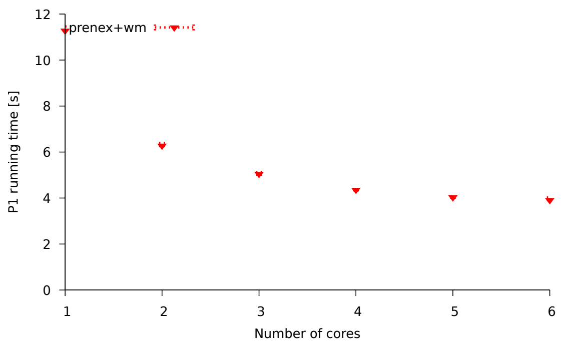
We measure the runtime, memory consumption, and CPU usage of monitors for a given specification and trace file using the 'time' command.

We compare our implementation to the existing monitors and timeliners combination. Namely, MTLfreeze and watermarks (freeze+wm), and MTLfreeze and sequence numbers(freeze+oo). Our implementation is MTLprenex and watermarks (prenex+wm). Note that in a non-distributed context, the sequence number timeliner is a straight upgrade to the watermark timeliner. As a result, we expect MTLfreeze and watermarks to perform consistently worse than MTLfreeze and sequence numbers. We will show this in the first experimental results, then remove the MTLfreeze with watermarks monitor from the results.

Additionally, the original Polimon implementation also support some degree of parallelism. First, the MTL and MTL$^{\downarrow}$ monitors benefit from being arranged in a pipeline design where stages may run in parallel and communicate via buffered channels. Second, the MTLfreeze monitor also also carries out some operation on its graph in parallel, benefiting to some extent from



(a) All monitors



(b) MTLprenex monitor

Figure 5.2 – P1 Scaling as a function of thread count

multiple CPU cores as well

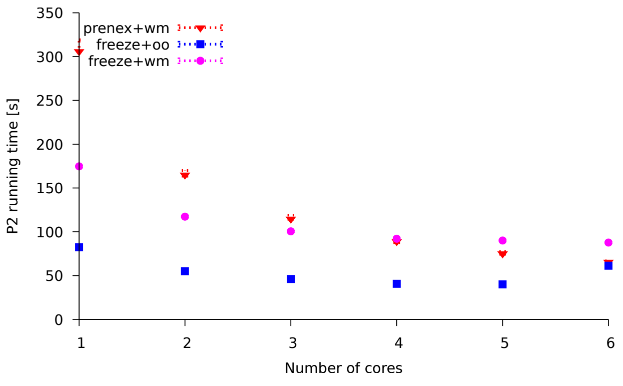
5.1 Core-count scaling

We First consider the scalability of our design with respect to CPU core count.

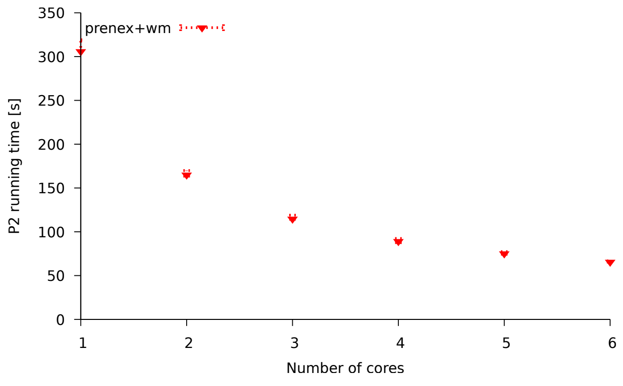
The left-hand side plot of figure 5.2 shows the comparative performance of our implementation (in red) and the previous monitors scaling from 1 to 6 threads. The log file spans 60s, the event rate is 1000 events per second, and the out-of-orderness parameter of the stream is 1. The first observation is that our implementation runs consistently faster on P1 than both other monitors. Our implementation runs approximately 5 times faster than MTLfreeze with sequence numbers and 10 times faster than MTLfreeze and watermarks. This advantage even at low thread count is due to the MTL monitors that our workers use. These monitors are one order of magnitude faster that a MTLfreeze monitor monitoring a comparable formula [3]. Additionally, we see that the MTLfreeze monitors scale poorly past 4 threads, while our implementation improves even with a larger core count, as shown on the right-hand side plot of figure 5.2

P2 illustrate the limit of our implementation. Figure 5.3 shows the performance results. Due to the presence of the proposition in the formula, every worker must process every $suspicious$ event. This not only increases the communication costs, but more importantly cause each worker to be triggered at each iteration, causing huge CPU utilization Our implementation still scales well, overtaking the MTLfreeze with watermarks after 3 threads. Nevertheless, the proposition in P2 generates a lot of additional work, so our monitor is not ideal for this formula.

P6 shows how we can optimize the property expressed by P2 for our monitor. Transforming the proposition into a predicate with a data value solves the problem encountered in P2. We make several observations. First, our implementation scales well here. It is faster than on P2, but keeps the good scaling. We explain this by the choice of variable: the cid variable has relatively few different values. As a result, the prefix tree in the dispatcher is traversed faster, speeding up

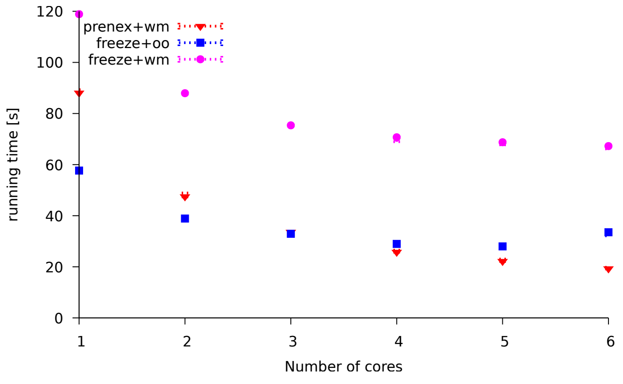


(a) All monitors

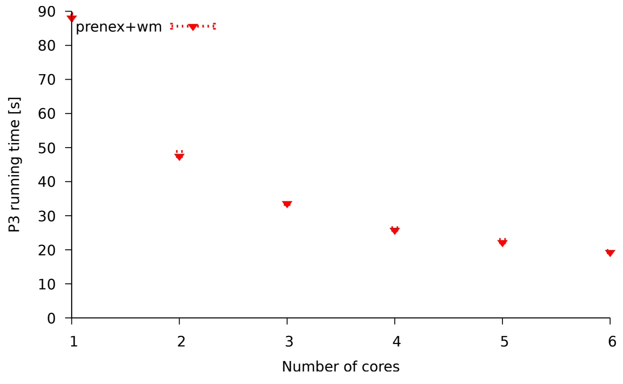


(b) MTLprenex monitor

Figure 5.3 – P2 Scaling as a function of thread count



(a) All monitors



(b) MTLprenex monitor

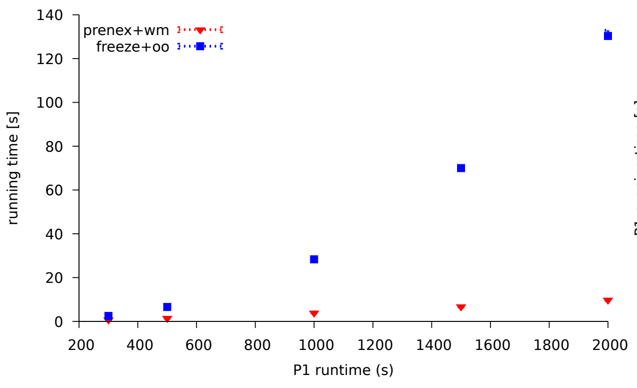
Figure 5.4 – P3 Scaling as a function of thread count

the sequential part of the program, the dispatcher. This leads to better scaling.

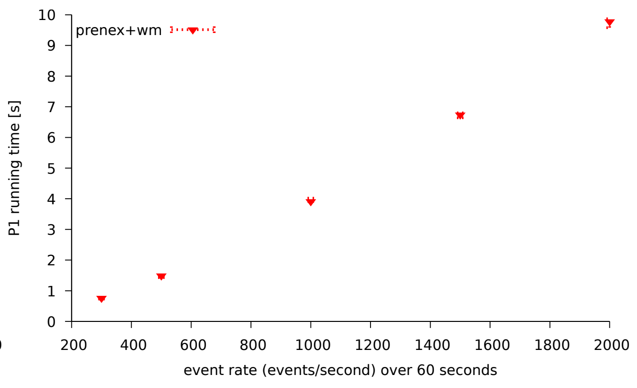
5.2 Event rate scaling

To represent scaling, we also take a look at how our implementation can handle larger problem size compared to the existing monitors. Here, we measure the runtime of each monitor at different event rates. We use the optimal core count for each monitor, that is 4 for the MTLfreeze monitor and 6 for the MTLprenex monitor. We see that our implementation is generally better at handling large event rates.

Our implementation scales well on P1 compared to the previous implementations. On the left, we see that our runtime increases much slower than the other monitors. On the right, we see that on P1, our monitor increases almost linearly as a function of the event rate. P1 is an ideal case for our monitor: we have many workers that can be scheduled efficiently by the go

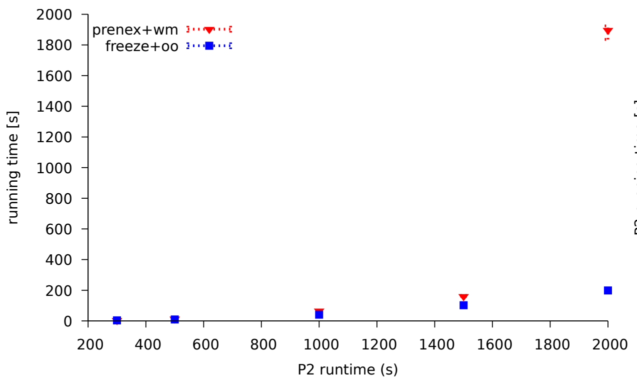


(a) All monitors

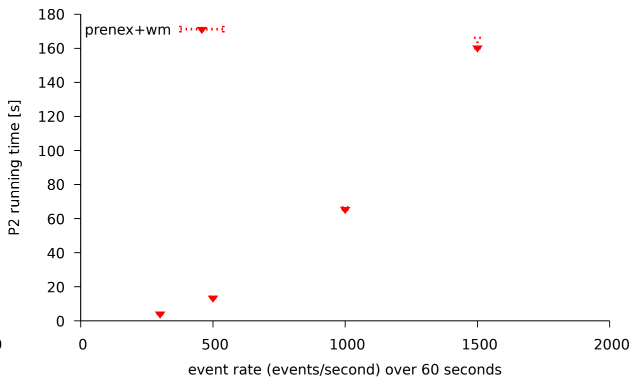


(b) MTLprenex monitor

Figure 5.5 – P1 Scaling as a function of event rate



(a) All monitors



(b) MTLprenex monitor

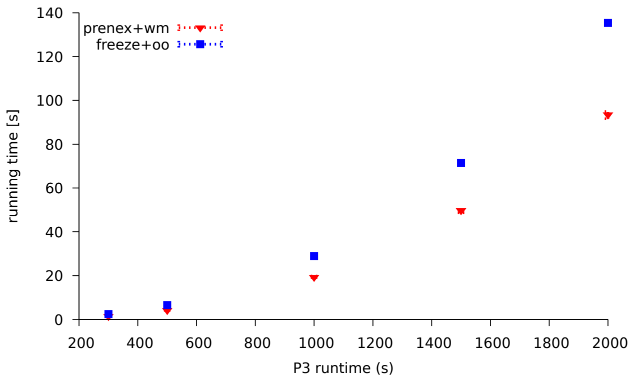
Figure 5.6 – P2 Scaling as a function of event rate

runtime, and at most iterations, events are not broadcasted to all workers as in P2. This allows our monitor to run efficiently.

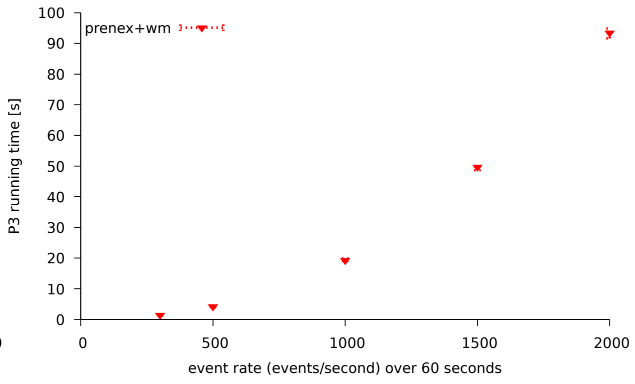
In contrast to P1, our monitor does not scale well on P2. It performs and scales worse than MTLfreeze with sequence numbers. The high runtime at event rate 2000 is due to lack of system memory and swapping to disk . We look at memory consumption in the next section.

We remove this data point on the right axis to observe scaling. Our monitor no longer scales linearily. This behavior is explained by the combination of two factors. First, workers don't have a termination condition. As a result, they never terminate. That is not the root cause of this behaviour though, as this happens with P! as well without causing any issue problems. The second, more problematic factor in that in this formula, each event must be broadcasted to every worker. In this case the communication costs explode, expeciall in terms of memory as we will see later

Finally, We see that P3, although optimized, is still not an ideal case for our monitor. Scaling is

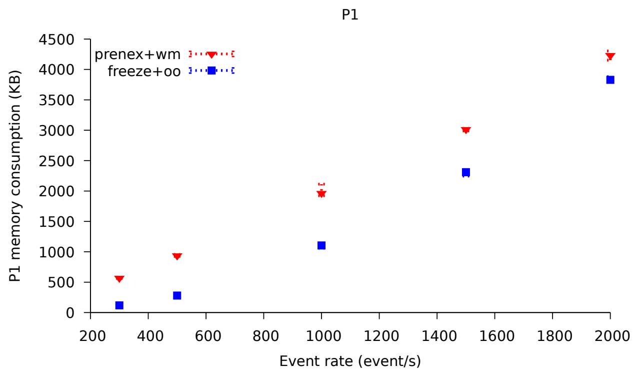


(a) All monitors



(b) MTLprenex monitor

Figure 5.7 – P3 Scaling as a function of event rate



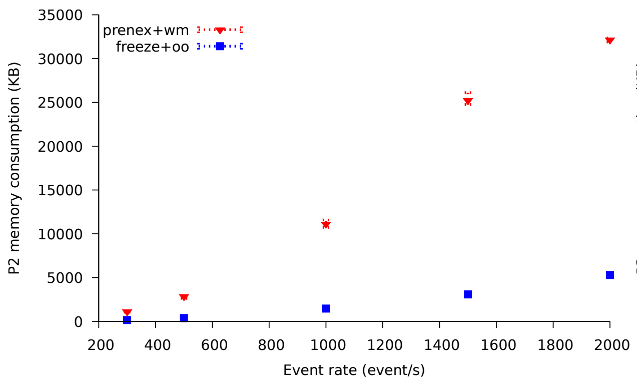
(a) P1 memory consumption as a function of event rate

not linear. This can be explained as follows. In this case again, this is explained by communicatin costs. In the stream are *report* events. these events are forwarded to all workers, because the formula doesn't partion on them. That being said, our monitor scales better than the existing monitors, that also scale non-linearly.

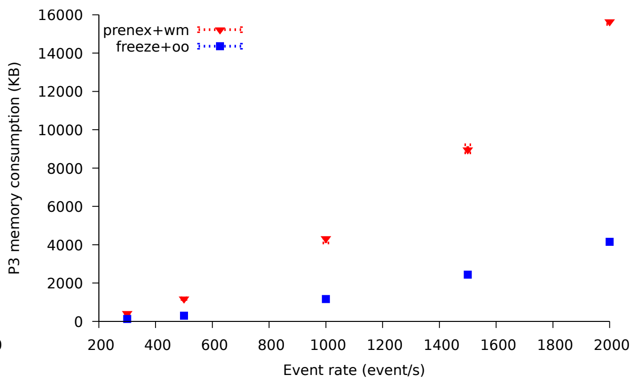
5.3 Memory usage

Finally, we compare the memory usage of the two implementations. We see that memory usage is comparable when monitoring the favorable P1 proposition, with an advantage for the original monitor at low event rate, which diminishes as the event rate increases.

However, in the unfavorable cases, our implementation is at a clear disadvantage in terms of memory. Consuming up to 4 times as much for P3, and 6 times as much for P2. In the case of P2, the Operating System runs out of RAM and starts swapping to disk, causing the performance deficit observed previously.



(a) P2 Memory consumption as a function of event rate



(b) P3 Memory consumption as a function of event rate

The increased communication costs put our implementation at a great disadvantage in terms of memory efficiency. It is not the absolute number of workers that causes this problem. Based on the selected data values, P1 has more workers than P2. However, the fact that P2 must broadcast every event to every worker via buffered channels causes a steep increase in memory usage.

Implementing a termination condition for the workers could help greatly reduce memory usage, as this would reduce the number of workers to broadcast to. While reducing memory usage would be a priority in future work,

Chapter 6

Related Work

Our contribution is based the work by Basin et al. [3] that introduces an online, out-of-order monitor for specifications expressed in the MTLfreeze logic. We extend this work with a stream partitioning technique and parallel processing algorithm to design a system capable of processing larger streams.

Similar to our work, Basin et al. [1] split a stream in slices and run the parallel monitoring task on each of them. The stream can be sliced on both on time and data parameters. This contribution is comparable to ours in the sense that the specification languages are comparable and aim at monitoring the same kind of properties. We do not slice on time, but slicing on data parameters to produce data-parallel subsets of the stream that can be processed by independent monitors is a common goal. A first difference is that this work relies on the MapReduce framework, and as a result does not support online processing. This is addressed in an extension of this work by Schneider et al. [9] where the Flink framework is used to implement online processing. However, the main difference between our design and these two solutions is that both they do not support out-of-order streams, while our portioning strategy and our parallel algorithm are designed with out-of-order streams in mind.

Roşu and Chen [8] and Reger et al. [7] use a partitioning technique close to ours. In their contribution, the data stream is sliced on parameter instances, and a propositional slices are generated for each parameter instance or combination of parameter instances. Each propositional slice is then monitored in parallel. However, their specification language is quite different than ours. They use *quantified event automata*, essentially finite state automatons parameterized by a data instance. A QEA encode sequences of states transitions and accepting states, as opposed to temporal logic specification languages which define temporal properties, i.e. properties with time constraints. Additionally, finite state automaton-based monitors do not support out-of-order stream processing, and the partitioning technique, although close to ours, does not seem to be designed to support out-of-order processing.

Finally, the runtime monitoring problem, more broadly considered, shows many similarities with more general stream processing tasks. For instance, there are many similarities between a general sliding window aggregation problem as described in the work of Tangwonsan et al. [10] and our runtime monitoring problem, be it in terms of design, goals, and technical solution. Both problems operate on timed, out-of-order streams. Specification expressed as of temporal logic operators can be seen as an operator that aggregates multiple stream events into a single verdict. The sliding window, while not explicit, also exists in runtime monitoring problem based on temporal logic. Whenever the temporal operators have a metric constraint, their output is a function of only some time-slice of the stream, defining a time window for each result of our operator. Additionally, the goals of our contribution and Tangwonsan et al's work on more general stream processing problems are also similar. Both aim to process a stream of elements at high throughput and with low latency. As a result, online and out-of-order processing capabilities are intermediary design goals shared by both solutions, and in both cases comparable techniques are developed to achieve these intermediary design goals. The graph structure used in our submonitors is very similar to the graph structure used in Tangwonsan et al's work. Both structures aim to eagerly process incoming events. In order to do so, both arrange event based on their timestamps at the leaves, compute intermediary results that propagate up and output at the root. That being said, even though similar technical solutions can be found in monitoring tasks and in stream tasks, these two problems are indeed different. The solution proposed by Tangwonsan et al. is applicable for monoid operators, which temporal operators are not. Additionally, the time model in our solution is more generic, supporting unix-timestamped events for timekeeping and sequence numbers for ordering, versus events with timestamps in $\mathbb{N}$, where the timestamp covers both the role of timekeeping and ordering.

Chapter 7

Conclusion

In this thesis, we presented an approach to process out-of-order streams online and concurrently. Our core contributions consist in a partitioning strategy for splitting the stream into substreams that can be processed independently, an algorithm to efficiently process these substreams, and an implementation of this algorithm.

We evaluated this solution on favorable and unfavorable monitoring specifications. The experimental evaluation shows that the algorithm scales well on favorable cases, and that unfavorable cases can be optimized and scale fairly well too.

Future work could address the weaknesses of this solution. In particular, finding a good termination criteria for the workers could result in significant improvements in unfavorable cases.

Bibliography

[1] David A. Basin, Germano Caronni, Sarah Ereth, Matús Harvan, Felix Klaedtke, and Heiko Mantel. “Scalable offline monitoring of temporal specifications.” In: *Formal Methods in System Design* 49.1-2 (2016), pp. 75–108. URL: `http://dblp.uni-trier.de/db/journals/fmsd/fmsd49.html#BasinCEHKM16`.

[2] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zundefinedlinescu. “Monitoring Metric First-Order Temporal Properties”. In: *J. ACM* 62.2 (May 2015). ISSN: 0004-5411. DOI: `10.1145/2699444`. URL: `https://doi.org/10.1145/2699444`.

[3] David Basin, Felix Klaedtke, and Eugen Zălinescu. “Runtime Verification over Out-of-order Streams”. In: *ACM Trans. Comput. Log.* 21.1 (2020).

[4] Peter Brass. *Advanced Data Structures*. 1st ed. USA: Cambridge University Press, 2008. ISBN: 0521880378.

[5] Martin Leucker and Christian Schallhart. “A Brief Account of Runtime Verification”. In: *Journal of Logic and Algebraic Programming* 78 (May 2009), pp. 293–303. DOI: `10.1016/j.jlap.2008.08.004`.

[6] Ramy Medhat, Borzoo Bonakdarpour, Sebastian Fischmeister, and Yogi Joshi. “Accelerated Runtime Verification of LTL Specifications with Counting Semantics”. In: *Runtime Verification*. Ed. by Yliès Falcone and César Sánchez. Cham: Springer International Publishing, 2016, pp. 251–267. ISBN: 978-3-319-46982-9.

[7] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. “MarQ: Monitoring at Runtime with QEA”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 596–610. ISBN: 978-3-662-46681-0.

[8] Grigore Roşu and Feng Chen. “Semantics and Algorithms for Parametric Monitoring”. In: *Log. Methods Comput. Sci.* 8.1 (2012).

[9] Joshua Schneider, David Basin, Frederik Brix, Srđan Krstić, and Dmitriy Traytel. “Scalable Online First-Order Monitoring”. In: *Runtime Verification*. Ed. by Christian Colombo and Martin Leucker. Cham: Springer International Publishing, 2018, pp. 353–371. ISBN: 978-3-030-03769-7.

[10] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. “Optimal and General Out-of-Order Sliding-Window Aggregation”. In: *Proc. VLDB Endow.* 12.10 (June 2019), pp. 1167–1180. ISSN: 2150-8097. DOI: `10.14778/3339490.3339499`. URL: `https://doi.org/10.14778/3339490.3339499`.