Vamos: Middleware for Best-Effort Third-Party Monitoring

Marek Chalupa (✉), Fabian Muehlboeck, Stefanie Muroya Lei, and Thomas A. Henzinger

Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
marek.chalupa@ista.ac.at

Abstract. As the complexity and criticality of software increase every year, so does the importance of run-time monitoring. Third-party monitoring, with limited knowledge of the monitored software, and best-effort monitoring, which keeps pace with the monitored software, are especially valuable, yet underexplored areas of run-time monitoring. Most existing monitoring frameworks do not support their combination because they either require access to the monitored code for instrumentation purposes or the processing of all observed events, or both.

We present a middleware framework, Vamos, for the run-time monitoring of software which is explicitly designed to support third-party and best-effort scenarios. The design goals of Vamos are (i) efficiency (keeping pace at low overhead), (ii) flexibility (the ability to monitor black-box code through a variety of different event channels, and the connectability to monitors written in different specification languages), and (iii) ease-of-use. To achieve its goals, Vamos combines aspects of event broker and event recognition systems with aspects of stream processing systems.

We implemented a prototype toolchain for Vamos and conducted experiments including a case study of monitoring for data races. The results indicate that Vamos enables writing useful yet efficient monitors, is compatible with a variety of event sources and monitor specifications, and simplifies key aspects of setting up a monitoring system from scratch.

1 Introduction

Monitoring—the run-time checking of a formal specification—is a lightweight verification technique for deployed software. Writing monitors is especially challenging if it is *third-party* and *real-time*. In third-party monitoring, the monitored software and the monitoring software are written independently, in order to increase trust in the monitor. In the extreme case, the monitor has very limited knowledge of and access to the monitored software, as in black-box monitoring. In real-time monitoring, the monitor must not slow down the monitored software while also following its execution close in time. In the extreme case, the monitor may not be able to process all observed events and can check the specification only approximately, as in best-effort monitoring.

We present middleware—called Vamos ("Vigilant Algorithmic Monitoring of Software")—which facilitates the addition of best-effort third-party monitors to deployed software. The primary goals of our middleware are (i) performance

L. Lambers and S. Uchitel (Eds.): FASE 2023, LNCS 13991, pp. 260–281, 2023.
https://doi.org/10.1007/978-3-031-30826-0_15

(keeping pace at low overhead), (ii) flexibility (compatibility with a wide range of heterogeneous event sources that connect the monitor with the monitored software, and with a wide range of formal specification languages that can be compiled into VAMOS), and (iii) ease-of-use (the middleware relieves the designer of the monitor from system and code instrumentation concerns).

All of these goals are fairly standard, but VAMOS' particular design tradeoffs center around making it as easy as possible to create a best-effort third-party monitor of actual software without investing much time into low-level details of instrumentation or load management. In practice, instrumentation—enriching the monitored system with code that is gathering observations on whose basis the monitor generates verdicts—is a key part of writing a monitoring system and affects key performance characteristics of the monitoring setup [11]. These considerations become even more important in third-party monitoring, where the limited knowledge of and access to the monitored software may force the monitor to spend more computational effort to re-derive information that it could not observe, or combine it from smaller pieces obtained from more (and different) sources. By contrast, current implementations of monitor specification languages mostly offer either very targeted instrumentation support for particular systems or some general-purpose API to receive events, or both, but little to organize multiple heterogeneous event streams, or to help with the kinds of best-effort performance considerations that we are concerned with. Thus, VAMOS fills a gap left open by existing tools.

Our vision for VAMOS is that users writing a best-effort third-party monitor start by selecting configurable instrumentation tools from a rich collection. This collection includes tools that periodically query webservices, generate events for relevant system calls, observe the interactions of web servers with clients, and of course standard code instrumentation tools. The configuration effort for each such *event source* largely consists of specifying patterns to look for and what events to generate for them. VAMOS then offers a simple specification language for filtering and altering events coming from the event sources, and simple yet expressive event recognition rules that produce a single, global event stream by combining events from a (possibly dynamically changing) number of event sources. Lastly, monitoring code as it is more generally understood—which could be written directly or generated from existing tools for run-time verification like LTL formulae [47], or stream verification specifications [8] such as TeSSLa [41]—processes these events to generate verdicts about the monitored system.

VAMOS thus represents middleware between event sources that emit events and higher-level monitoring code, abstracting away many low-level details about the interaction between the two. Users can employ both semi-synchronous and completely asynchronous [11] interactions with any or all event sources. Between these two extremes, to decouple the higher-level monitoring code's performance from the overhead incurred by the instrumentation, while putting a bound on how far the monitoring code can lag behind the monitored system, we provide a simple load-shedding mechanism that we call *autodrop buffers*, which are buffers that drop events when the monitoring code cannot keep up with the rate of in-

coming events, while maintaining summarization data about the dropped events. This summarization data can later be used by our event recognition system when it is notified that events were dropped; some standard monitoring specification systems can handle such *holes* in their event streams automatically [32,42,54]. The rule-based event recognition system allows grouping and ordering buffers dynamically to prioritize or rotate within variable sets of similar event sources, and specifying patterns over multiple events and buffers, to extract and combine the necessary information for a single global event stream.

Data from event sources is transferred to the monitor using efficient lock-free buffers in shared memory inspired by Cache-Friendly Asymmetric Buffers [29]. These buffers can transfer over one million events per second per event source on a standard desktop computer. Together with autodrop buffers, this satisfies our performance goal while keeping the specification effort low. As such, VAMOS resembles a single-consumer version of an event broker [18,58,48,55,26,1] specialized to run-time monitoring.

The core features we built VAMOS around are not novel on their own, but to the best of our knowledge, their combination and application to simplify best-effort third-party monitoring setups is. Thus, we make the following contributions:

- We built middleware to connect higher-level monitors with event sources, addressing particular challenges of best-effort third-party monitoring (Section 2), using a mixture of efficient inter-process communication and easy-to-use facilities for load management (Section 3) on one hand, and *buffer groups* and other event recognition abstractions (Section 4) on the other hand.
- We implemented a compiler for VAMOS specifications, a number of event sources, and a connector to TeSSLa [41] monitors (Section 5).
- We conducted some stress-test experiments using our framework, as well as a case study in which we implemented a monitor looking for data races, providing evidence of the feasibility of low-overhead third-party monitoring with simple specifications (Section 6).

2 Architectural Overview

Writing a run-time monitor can be a complex task, but many tools to express logical reasoning over streams of run-time observations [19,34,16,49,24,27,41] exist. However, trying to actually obtain a concrete stream of observations from a real system introduces a very different set of concerns, which in turn have a huge effect on the performance properties of run-time monitoring [11].

The goal of VAMOS is to simplify this critical part of setting up a monitoring system, using the model shown in Figure 1. On the left side, we assume an arbitrary number of distinct event sources directly connected to the monitor. This is particularly important in third-party monitoring, as information may need to be collected from multiple different sources instead of just a single program, but can be also useful in other monitoring scenarios, e.g. for multithreaded programs.

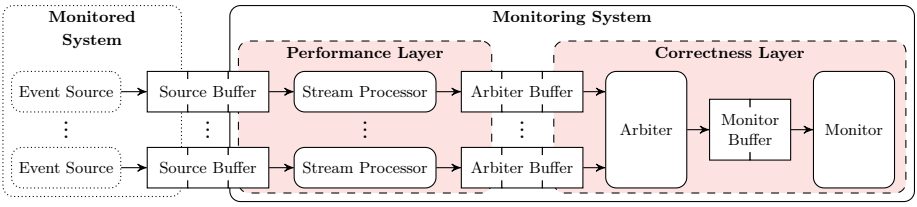

Fig. 1. The components of a VAMOS setup.

The right-most component is called the *monitor*, representing the part of the monitoring system that is typically generated by a monitoring specification tool, usually based on a single global event stream. As middleware, VAMOS connects the two, providing abstractions for common issues that monitor writers would otherwise have to address with boilerplate, but still complicated code.

Given that there are multiple event sources providing their own event streams, but only one global event stream consumed by the monitor, a key aspect is merging the incoming streams into one, which happens in the *arbiter*. Third-party monitoring often cannot rely on the source-code-based instrumentation that is otherwise common [21,4,14,16,25]; for example, TeSSLa[1] [41] comes with a basic way of instrumenting C programs by adding annotations into the specification that identify events with function calls or their arguments. Instead, it has to rely on things that can be reliably observed and whose meaning is clear, for example system calls, calls to certain standard library functions, or any other information one can gather from parts of the environment one controls, such as sensors or file system. These do not necessarily correspond in a straightforward way to the events one would like to feed into the higher-level monitor and thus need to be combined or split up in various ways. For example, when a program writes a line to the standard output, the data itself might be split into multiple system calls or just be part of a bigger one that contains multiple lines, and there are also multiple system calls that could be used. Therefore, the arbiter provides a way to specify a rule-based event recognition system to generate higher-level events from combinations of events on the different event sources.

Another common assumption in monitoring systems is some global notion of time that can be used to order events. This is not necessarily true for multiple, heterogeneous event sources, and even just observing the events of a multi-threaded program can cause events to arrive in an order that does not represent causality. VAMOS arbiter specifications are flexible enough to support many user-defined ways of expressing ways of merging events into a single global stream.

Doing this kind of sorting and merging and then potentially arbitrarily complex other computations in both the arbiter and the monitor may take longer than it takes the monitored system to generate events. Especially in third-party monitoring, a monitor may have to reconstruct information that is technically

[1] We keep referring to TeSSLa in the rest of the paper and also chose to use it in our implementation because it is one of the most easily available existing tools we could find. In general, the state of the field is that, while many papers describing similar tools exist, few are actually available [48].

```
1 stream type Observation { Op(arg : int, ret : int); }
2 event source Program : Observation to autodrop(16,4)
3 arbiter : Observation {
4     on Program: hole(n) | ;
5     on Program: Op(arg, ret) | yield;
6 }
7 monitor(2) { on Op(arg, reg) $$ CheckOp(arg, ret);  $$ }
```

Listing 1.1. A basic asynchronous best-effort monitor.

present in the monitored system but cannot be observed, or, worse, the monitor may have to consider multiple different possibilities if information cannot be reliably recomputed. However, as part of our performance goal, we want the monitor to not lag too far behind the monitored system. Therefore, our design splits the monitoring system into the *performance* and *correctness* layers. In between the two, events may be dropped as a simple load-shedding strategy.

The performance layer, on the other hand, sees all events and processes each event stream in parallel. *Stream processors* enable filtering and altering the events that come in, reducing pressure and computational load on the correctness layer. This reflects that in third-party monitoring, observing coarse-grained event types like system calls may yield many uninteresting events. For example, all calls to `read` may be instrumented, but only certain arguments make them interesting.

A Simple Example Listing 1.1 shows a full VAMOS specification (aside from the definition of custom monitoring code in a C function called `CheckOp`). *Stream types* describe the kinds of events and the memory layout of their data that can appear in a particular buffer; in this example, streams of type `Observation` contain only one possible event named `Op` with two fields of type **`int`**. For source buffers—created using event source descriptions as in line 2—these need to be based on the specification of the particular event source. Each event source is associated with a *stream processor*; if none is given (as in this example), a default one simply forwards all events to the corresponding arbiter buffer, here specified as an *autodrop* buffer that can hold up to 16 events and when full keeps dropping them until there is again space for at least four new events. Using an autodrop buffer means that in addition to the events of the stream type, the arbiter may see a special *hole* event notifying it that events were dropped. In this example, the arbiter simply ignores those events and forwards all others to the monitor, which runs in parallel to the arbiter with a blocking event queue of size two, and whose behavior we implemented directly in C code between `$$` escape characters.

3 Efficient Instrumentation

Our goals for the performance of the monitor are to not incur too much overhead on the monitored system, and for the monitor to be reasonably up-to-date in terms of the lag between when an event is generated and when it is processed. The

key features VAMOS offers to ensure these properties while keeping specifications simple are related to the performance layer, which we discuss here.

3.1 Source Buffers and Stream Processors

Even when instrumenting things like system calls, in order to extract information from them in a consistent state, the monitored system will have to be briefly interrupted while the instrumentation copies the relevant data. A common solution is to write this data to a log file that the monitor is incrementally processing. This approach has several downsides. First, in the presence of multiple threads, accesses to a single file require synchronization. Second, the common use of string encodings requires extra serialization and parsing steps. Third, file-based buffers are typically at least very large or unbounded in size, so slower monitors eventually exhaust system resources. Finally, writing to the log uses relatively costly system calls. Instead, VAMOS event sources transmit raw binary data via channels implemented as limited-size lock-free ring buffer in shared memory, limiting instrumentation overhead and optimizing throughput [29]. To avoid expensive synchronization of different threads in the instrumented program (or just to logically separate events), VAMOS also allows dynamically allocating new event sources, such that each thread can write to its own buffer(s). The total number of event sources may therefore vary across the run of the monitor.

For each event source, VAMOS allocates a new thread in the performance layer to process events from this source[2]. In this layer, event processors can filter and alter events before they are forwarded to the correctness layer, all in a highly parallel fashion. A default event processor simply forwards all events. The computations done here should be done at the speed at which events are generated on that particular source, otherwise the source buffer will fill up and eventually force the instrumentation to wait for space in the buffer.

3.2 Autodrop Buffers

As we already stated, not all computations of a monitor may be able to keep up with the monitored system. Our design separates these kinds of computations into the correctness layer, which is connected with the performance layer via *arbiter buffers*. The separation is achieved by using *autodrop buffers*. These buffers provide the most straightforward form of load management via load shedding [59]: if there is not enough space in the buffer, it gathers summarization information (like the count of events since the buffer became full) and otherwise drops the events forwarded to it. Once free space becomes available in the buffer, it automatically inserts a special *hole* event containing the summarization information. The summarization ensures that not all information about dropped

[2] When event sources can be dynamically added, the user may specify a limit to how many of them can exist concurrently to avoid accumulating buffers the monitor cannot process fast enough. When that limit is hit, new event sources are rejected and the instrumentation drops events that would be forwarded to them.

events is lost, which can help to reduce the impact of load shedding. At minimum, the existence of the *hole* event alone makes a difference in monitorability compared to not knowing whether any events have been lost [35], and is used as such in some monitoring systems [32,42,54].

In addition to autodrop buffers, arbiter buffers can also be finite-size buffers that block when space is not available, or ininite-size buffers. The former may slow down the stream processor and ultimately the event source, while the latter may accumulate data and exhaust available resources. For some event sources, this may not be a big risk, and it eliminates the need to deal with *hole* events.

4 Event Recognition, Ordering, and Prioritization

Vamos' arbiter specifications are a flexible, yet simple way to organize the information gathered from a—potentially variable—number of heterogeneous event sources. In this section, we discuss the key relevant parts of such specifications—a more complete specification can be found in the Technical Report [13].

4.1 Arbiter Rules

We already saw simple arbiter rules in Listing 1.1, but arbiter rules can be much more complex, specifying arbitrary sequences of events at the front of arbitrarily many buffers, as well as buffer properties such as a minimum number of available events and emptiness. Firing a rule can also be conditioned by an arbitrary boolean expression. For example, one rule in the *Bank* example we use in our evaluation in Section 6 looks as follows:

```
1 on Out : transfer(t2, src, tgt) transferSuccess(t4) |,
2    In  : numIn(t0, act) numIn(t1, acc) numIn(t3, amnt) |
3    where $$ t2 == t0 + 4 $$
4 $$ $yield SawTransfer(src, tgt, amnt); ... $$
```

This rule matches multiple events on two different buffers (`In` and `Out`), describing a series of user input and program output events that together form a single higher-level event SawTransfer, which is forwarded to the *monitor* component of the correctness layer. Rules do not necessarily consume the events they have looked at; some events may also just serve as a kind of lookahead. The "|" character in the events sequence pattern separates the consumed events (left) from the lookahead (right). Code between `$$` symbols can be arbitrary C code with some special constructs, such as the **`$yield`** statement (to forward events) above.

The rule above demonstrates the basic event-recognition capabilities of arbiters. By ordering the rules in a certain way, we can also prioritize processing events from some buffers over others. Rules can also be grouped into *rule sets* that a monitor can explicitly switch between in the style of an automaton.

4.2 Buffer Groups

The rules shown so far only refer to arbiter buffers associated with specific, named event sources. As we mentioned before, VAMOS also supports creating event sources dynamically during the run of the monitoring system. To be able to refer to these in arbiter rules, we use an abstraction we call *buffer groups*.

As the name suggests, buffer groups are collections of arbiter buffers whose membership can change at run time. They are the only way in which the arbiter can access dynamically created event sources, so to allow a user to distinguish between them and manage associated data, we extend stream types with *stream fields* that can be read and updated by arbiter code. Buffer groups are declared for a specific stream type, and their members have to have that stream type[3]. Therefore, each member offers the same stream fields, which we can use to compare buffers and order them for the purposes of iterating through the buffer group. Now the arbiter rules can also be `choice` blocks with more rules nested within them, as follows (`Both` is a buffer group and `pos` is a stream field):

```
1  choose F,S from Both {
2    on F : Prime(n,p) | where $$ $F.pos < $S.pos $$
3    $$ ... $$
4    on F : hole(n) |
5    $$ $F.pos = $F.pos + n; $$
6  }
```

This rule is a slightly simplified version of one in the *Primes* example in Section 6. This example does not use dynamically created buffers, but only has two event sources, and uses the ordering capabilities of buffer groups to prioritize between the buffers based on which one is currently "behind" (expressed in the stream field `pos`, which the buffer group `Both` is ordered by). The `choose` rule tries to instantiate its variables with distinct members from the buffer group, trying out permutations in the lexicographic extension of the order specified for the buffer group. If no nested rule matches for a particular instantiation, the next one in order is tried, and the `choose` rule itself fails if no instantiation finds a match.

To handle dynamically created event sources, corresponding stream processor rules specify a buffer group to which to add new event sources, upon which the arbiter can access them through `choose` rules. In most cases, we expect that `choose` blocks are used to instantiate a single buffer, in which case we only need to scan the buffer group in its specified order. Here, a round-robin order allows for fairness, while field-based orderings allow more detailed control over buffers prioritization, as it may be useful to focus on a few buffers at the expense of others, as in our above example.

Another potential option for ordering schemes for buffer groups could be based on events waiting in them, or even the values of those events' associated data. VAMOS currently does not support this because it makes sorting much more

[3] Note that stream processors may change the stream type between the source buffer and arbiter buffer, so event sources may use different types, but their arbiter buffers may be grouped together if processed accordingly.

expensive—essentially, all buffers may have to be checked in order to determine the order in which to try matching them against further rules. Some of our experiments could have made use of such a feature, but in different ways—future work may add mechanisms that capture some of these ways.

5 Implementation

In this section, we briefly review the key components of our implementation.

5.1 Source Buffers and Event Sources

The source buffer library allows low-overhead interprocess communication between a monitored system and the monitor. It implements lock-free asynchronous ring buffers in shared memory, inspired by Cache-Friendly Asymmetric Buffering [29], but extended to handle entries larger than 64 bits[4]. The library allows setting up an arbitrary number of source buffers with a unique name, which a monitor can connect to explicitly, and informing such connected monitors about dynamically created buffers. A user can also provide stream type information so connecting monitors can check for binary compatibility.

We have used the above library to implement an initial library of event sources: one that is used for detecting data races, and several which use either *DynamoRIO* [9] (a dynamic instrumentation framework) or the *eBPF* subsystem of the Linux Kernel [10,28,50] to intercept the `read` and `write` (or any other) system calls of an arbitrary program, or to read and parse data from file descriptors. The read/write related tools allow specifying an arbitrary number of regular expressions that are matched against the traced data, and associated event constructors that refer to parts of the regular expressions from which to extract the relevant data. Example uses of these tools are included in our artifact [12].

5.2 The Vamos Compiler and the TeSSLa Connector

The compiler takes a Vamos specification described in the previous sections and turns it into a C program. It does some minimal checking, for example whether events used in parts of the program correspond to the expected stream types, but otherwise defers type-checking to the C compiler. The generated program expects a command-line argument for each specified event source, providing the name of the source buffer created by whatever actual event source is used. Event sources signal when they are finished, and the monitor stops once all event sources are finished and all events have been processed.

The default way of using TeSSLa for online monitoring is to run an offline monitor incrementally on a log file of serialized event data from a single global

[4] Entries have the size of the largest event consisting of its fixed-size fields and identifiers for variable-sized data (strings) transported in separately managed memory.

event source. A recent version of TeSSLa [33] allows generating Rust code for the stream processing system with an interface to provide events and drive the stream processing directly. Our compiler can generate the necessary bridging code and replace the *monitor* component in VAMOS with a TeSSLa Rust monitor. We used TeSSLa as a representative of higher-level monitoring specification tools; in principle, one could similarly use other standard monitor specification languages, thus making it easier to connect them to arbitrary event sources.

6 Evaluation

Our stated design goals for VAMOS were (i) performance, (ii) flexibility, and (iii) ease-of-use. Of these, only the first is truly quantitative, and the majority of this section is devoted to various aspects of it. We present a number of benchmark programs, each of which used VAMOS to retrieve events from different event sources and organize them for a higher-level monitor in a different way, which provides some qualitative evidence for its flexibility. Finally, we present a case study to build a best-effort data-race monitor (Section 6.4), whose relative simplicity provides qualitative evidence for VAMOS' ease of use.

In evaluating performance, we focus on two critical metrics:

1. How much overhead does monitoring impose on the monitored system? We measure this as the difference of wall-clock running times.
2. How well can a best-effort third-party monitor cover the behavior of the monitored program? We measure this as the portion of errors a monitor can (not) find.

Our core claim is that VAMOS allows building useful best-effort third-party monitors for programs that generate hundreds of thousands of events per second without a significant slow down of the programs beyond the unavoidable cost of generating events themselves. We provide evidence that corroborates this claim based on three artificial benchmarks that vary various parameters and one case study implementation of a data race monitor that we test on 391 benchmarks taken from SV-COMP 2022 [7].

Experimental setup All experiments were run on a common personal computer with 16 *GB* of RAM and an *Intel(R) Core(TM) i7-8700* CPU with 6 physical cores running on 3.20 *GHz* frequency. Hyper-Threading was enabled and dynamic frequency scaling disabled. The operating system was Ubuntu 20.04. All provided numbers are based on at least 10 runs of the relevant experiments.

6.1 Scalability Tests

Our first experiment is meant to establish the basic capabilities of our arbiter implementation. An event source sends 10 million events carrying a single 64-bit number (plus 128 bits of metadata), waiting for some number of cycles between

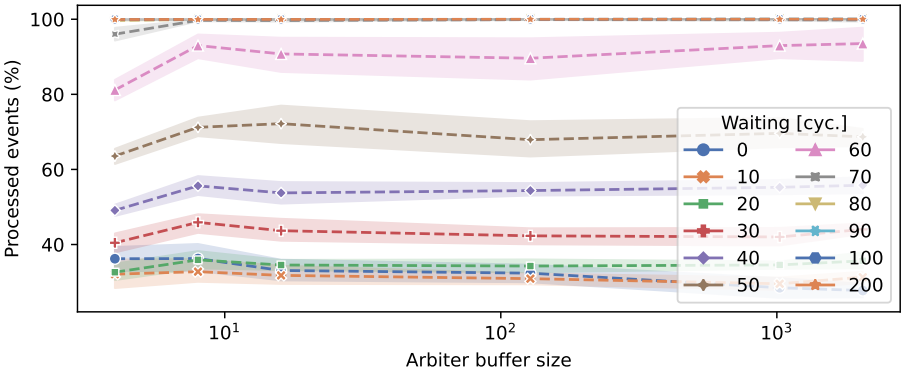

Fig. 2. The percentage of events that reached the final stage of the monitor in a stress test where the source sends events rapidly. Parameters are different arbiter buffer sizes (x-axis) and the delay (*Waiting*) of how many empty cycles the source waits between sending individual events. The shading around lines shows the 95 % confidence interval around the mean of the measured value. The source buffer was 8 pages large, which corresponds to a bit over 1 300 events.

each event. The performance layer simply forwards the events to autodrop buffers of a certain size, the arbiter retrieves the events, including holes, and forwards them to the monitor, which keeps track of how many events it saw and how many were dropped. We varied the number of cycles and the arbiter buffer sizes to see how many events get dropped because the arbiter cannot process them fast enough—the results can be seen in Figure 2.

At about 70 cycles of waiting time, almost all events could be processed even with very small arbiter buffer sizes (4 and up). In our test environment, this corresponds to a delay of roughly 700 *ns* between events, which means that Vamos is able to transmit approximately 1.4 million of events per second.

6.2 Primes

As a stress-test where the monitor actually has some work to do, this benchmark compares two parallel runs of a program that generates streams of primes and prints them to the standard output, simulating a form of differential monitoring [45]. The task of the monitor is to compare their output and alert the user whenever the two programs generate different data. Each output line is of the form `#n : p`, indicating that `p` is the `n`th prime. This is easy to parse using regular expressions, and our DynamoRIO-based instrumentation tool simply yields events with two 32-bit integer data fields (`n` and `p`).

While being started at roughly the same time, the programs as event sources run independently of each other, and scheduling differences can cause them to run out of sync quickly. To account for this, a Vamos specification needs to allocate large enough buffers to either keep enough events to make up for possible scheduling differences, or at least enough events to make it likely that there is

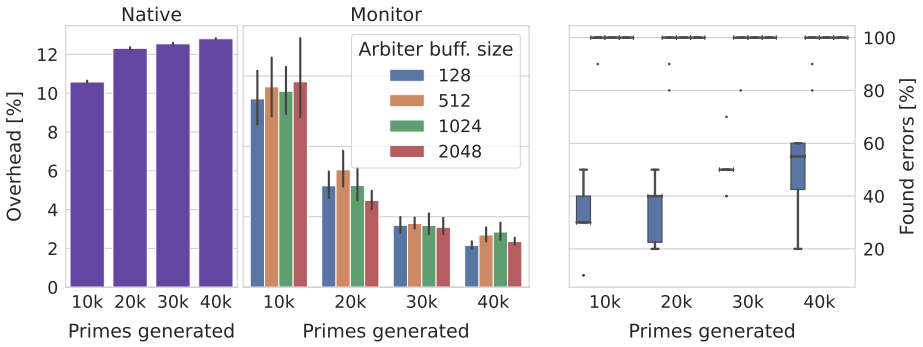

Fig. 3. Overheads (left) and percentage of found errors (right) in the primes benchmark for various numbers of primes and arbiter buffer sizes relative to DynamoRIO-optimized but not instrumented runs. DynamoRIO was able to optimize the program so much that the native binary runs slower than the instrumented one.

some overlap between the parts of the two event streams that are not automatically dropped. The arbiter uses the event field for the index variable n to line up events from both streams, exploiting the buffer group ordering functionality described in Section 4.2 to preferentially look at the buffer that is "behind", but also allowing the faster buffer to cache a limited number of events while waiting for events to show up on the other one. Once it has both results for the same index, the arbiter forwards a single pair event to the monitor to compare them.

Figure 3 shows results of running this benchmark in 16 versions, generating between 10 000 and 40 000 primes with arbiter buffer sizes ranging between 128 and 2024 events. The overheads of running the monitor are small, do not differ between different arbiter buffer sizes, and longer runs amortize the initial cost of dynamic instrumentation. We created a setting where one of the programs generates a faulty prime about once every 10 events and measured how many of these discrepancies the monitor can find (which depends on how many events are dropped). Unsurprisingly, larger buffer sizes are better at balancing out the scheduling differences that let the programs get out of sync. As long as the programs run at the same speed, there should be a finite arbiter buffer size that counters the desynchronization. In these experiments, this size is 512 elements.

Primes with TeSSLa We experimented with a variation of the benchmark that uses a very simple TeSSLa [17,41] specification receiving two streams for each prime generator (i.e., four streams in total): one stream of indexed primes as in the original experiment, and the other with hole events. The specification expects the streams to be perfectly lined up and checks that, whenever the last-seen pairs on both streams have the same index, they also contain the same prime (and ignores non-aligned pairs of primes). We wrote three variants of an arbiter to go in front of that TeSSLa monitor:

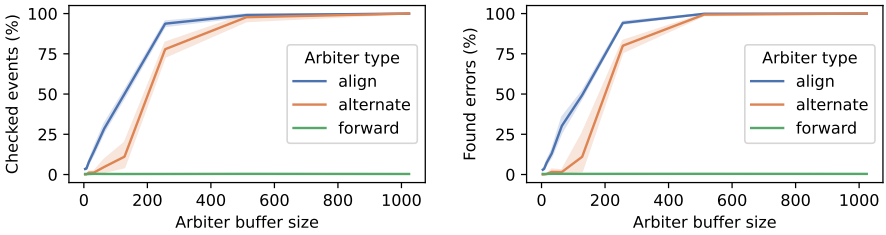

Fig. 4. Percentage of primes checked and errors found (of 40 000 events in total) by the TeSSLa monitor for different arbiter specifications and arbiter buffer sizes.

1. the *forward* arbiter just forwards events as they come; it is equivalent to writing a script that parses output of generators and (atomically) feeds events into a pipe from which TeSSLa reads events.
2. the *alternate* arbiter always forwards the event from the stream where we have seen fewer events so far; if streams happen to be aligned (that is, contain no or only equally-sized *hole* events), the events will perfectly alternate.
3. the *align* arbiter is the one we used in our original implementation to intelligently align both streams

Figure 4 shows the impact of these different arbiter designs on how well the monitor is able to do its task, and that indeed more active arbiters yield better results—without them, the streams are perfectly aligned less than 1% of the time. While one could write similar functionality to align different, unsynchronized streams in TeSSLa directly, the language does not easily support this. As such, a combination of TeSSLa and Vamos allows simpler specifications in a higher-level monitoring language, dealing with the correct ordering and preprocessing of events on the middleware level.

6.3 Bank

In this classic verification scenario, we wrote an interactive console application simulating a banking interface. Users can check bank account balances, and deposit, withdraw, or transfer money to and from various accounts. The condition we want to check is that no operations should be permitted that would allow an account balance to end up below 0.

We use an event source that employs DynamoRIO [9] to dynamically instrument the program to capture its inputs and outputs, which it parses to forward the relevant information to the monitor. The monitor in turn starts out with no knowledge about any of the account balances (and resets any gathered knowledge when hole events indicate that some information was lost), but discovers them through some of the observations it makes: the result of a check balance operation gives precise knowledge about an account's balance, while the success or failure of the deposit/withdraw/transfer operations provides lower and upper bounds on the potential balances. For example, if a withdrawal of some amount

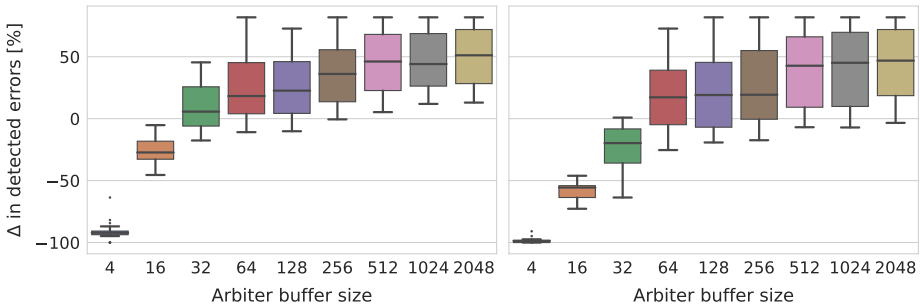

Fig. 5. Results of monitoring a simple banking simulator with VAMOS monitor (left) and TeSSLa monitor (right). Boxplots show the difference in the number of reported errors versus the number of errors the application made, in percent.

fails, this amount provides an upper bound on an account's balance, and any higher successive withdrawal attempt must surely fail too.

In the spirit of third-party monitoring, however, the stateful interface does not necessarily make it easy to derive these higher level events. For example, there is no individual confirmation that says that the withdrawal of some amount from some account was successful or not. Instead, the user selects an account, then the withdraw action, is then prompted which amount they would like to withdraw from said account, and after entering said amount, the system only displays a message that the withdrawal failed or was successful. The event source parses each individual step and provides them on two separate streams, one for the inputs and one for the outputs. This is where VAMOS' higher-level event recognition capabilities (see also the example in Section 4.1) allow the arbiter to recognize the higher-level events to forward to the monitor, which itself is therefore again much easier to specify.

To conduct measurements, we randomly generated 10 000 (well-formed) inputs and fed them to the banking application as fast as possible. We also let the application generate erroneous outputs (wrong balances, swapping success and failure messages) at random and measured how many those our best-effort third-party monitor was able to detect. The size of the source buffer was one page (128 events) and we varied the size of arbiter buffers from 4 to 2048.

The heavyweight instrumentation we used in this scenario caused the banking application to run through its script about 40% slower than without instrumentation for all sizes of the arbiter buffer, which is more than in our other benchmarks, but seems still plausible for interactive programs, and could be much more optimized. Our second metric is how many errors the monitor actually detects. Figure 5 shows this for both the monitor we described above and a TeSSLa variant that only considers exact knowledge about account balances (no upper or lower bounds) and thus finds fewer errors, demonstrating both an alternate monitor design and the use of our TeSSLa connector. The results vary quite a bit with arbiter buffer sizes and between runs, and the monitor may report more errors than were inserted into the run. This is because, first, especially

with smaller buffer sizes, the autodrop buffers may drop a significant portion (up to 60% at arbiter buffer size 4, 5% at size 256) of the events, but the monitor needs to see a contiguous chunk of inputs and outputs to be able to gather enough information to find inconsistencies. Second, some errors cause multiple inconsistencies: when a transfer between accounts is misreported as successful or failed when the opposite is true, the balances (or bounds) of two accounts are wrong. Overall, both versions of the monitor were able to find errors with even smaller sizes of arbiter buffers, and increasing numbers improved the results steadly, matching the expected properties of a best-effort third-party monitor.

6.4 Case Study: Data Race Detection

While our other benchmarks were written artificially, we also used Vamos to develop a best-effort data race monitor. Most tools for dynamic data race detection use some variation of the Eraser algorithm [51]: obtain a single global sequence of synchronization operations and memory accesses, and use the former to establish happens-before relationships whenever two threads access the same memory location in a potentially conflicting way. This entails keeping track of the last accessing threads for each location, as well as of the ways in which any two threads might have synchronized since those last accesses. Implemented naïvely, every memory access causes the monitor to pause the thread and atomically update the global synchronization state. Over a decade of engineering efforts directed at tools like ThreadSanitizer [52] and Helgrind [57] have reduced the resulting overhead, but it can still be substantial.

Vamos enabled us to develop a similar monitor at significantly reduced engineering effort in a key area: efficiently communicating events to a monitor running in parallel in its own process, and building the global sequence of events. To build our monitor, we used ThreadSanitizer's source-code-based approach[5] to instrument relevant code locations, and for each such location, we reduce the need for global synchronization to fetching a timestamp from an atomically increased counter. Based on our facilities for dynamically creating event sources, each thread forms its own event source to which it sends events. In the correctness layer, the arbiter builds the single global stream of events used by our implementation of a version of the Goldilocks [22] algorithm (a variant of Eraser [51]), using the timestamps to make sure events are processed in the right order. Autodrop buffers may drop some events to avoid overloading the monitor; when this happens to a thread, we only report data races that the algorithm finds if all involved events were generated after the last time that events were dropped. This means that our tool may not find some races, often those that can only be detected looking at longer traces. However, it still found many races in our experiments, and other approaches to detecting data races in best-effort ways have similar restrictions [56].

Our implementation (contained in our artifact [12]) consists of:

[5] This decision was entirely to reduce our development effort; a dynamic instrumentation source could be swapped in without other changes.

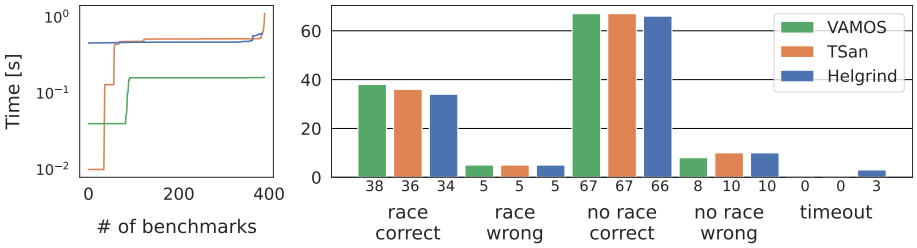

Fig. 6. Comparing running times of the three tools on all 391 benchmarks (left) and the correctness of their verdicts on the subset of 118 benchmarks for which it was possible to determine the ground truth (right). *Race* vs. *no race* indicates whether the tool found at least one data race, *correct* vs. *wrong* indicates whether that verdict matches the ground truth. For benchmarks with unknown ground truth, the three tools agreed on the existence of data races more than 99% of the time.

- a straightforward translation of the pseudocode in [22], using the C++ standard library `set` and `map` data structures, with extensions to handle holes;
- a small VAMOS specification to retrieve events from the variable number of event streams in order of their timestamps to forward to the monitor; the biggest complication here is deciding when to abandon looking for the next event in the sequence if it may have been dropped;
- an LLVM [40] instrumentation pass post-processing ThreadSanitizer's instrumentation to produce an event source compatible with VAMOS.

As such, we were able to use VAMOS to build a reasonable best-effort data-race monitor with relatively little effort, providing evidence that our ease-of-use design goal was achieved. To evaluate its performance, we tested it on 391 SV-COMP [7] concurrency test cases supported by our implementation, and compared it to two state-of-the-art dynamic data race detection tools, ThreadSanitizer [52] and Helgrind [57]. Figure 6 shows that the resulting monitor in most cases caused less overhead than both ThreadSanitizer and Helgrind in terms of time while producing largely the same (correct) verdicts.

7 Related Work

As mentioned before, VAMOS' design features a combination of ideas from works in run-time monitoring and related fields, which we review in this section.

Event Brokers/Event Recognition A large number of event broker systems with facilities for event recognition [18,58,55,26,1] already exist. These typically allow arbitrary event sources to connect and submit events, and arbitrarily many observers to subscribe to various event feeds. Mansouri-Samani and Sloman [44] outlined the features of such systems, including filtering and combining events, merging multiple monitoring traces into a global one, and using a database to

store (parts of) traces and additional information for the longer term. Modern industrial implementations of this concept, like Apache Flink [1], are built for massively parallel stream processing in distributed systems, supporting arbitrary applications but providing no special abstractions for monitoring, in contrast to more run-time-monitoring-focused implementations like ReMinds [58]. Complex event recognition systems also sometimes provide capabilities for load-shedding [59], of which autodrop buffers are the simplest version. Most event recognition systems provide more features than Vamos, but are also harder to set up for monitoring; in contrast, Vamos offers a simple specification language that is efficient and still flexible enough for many monitoring scenarios.

Stream Run-Time Verification LoLa [19,24], TeSSLa [41], and Striver [27] are stream run-time verification [8] systems that allow expressing a monitor as a series of mutually recursive data streams that compute their current values based on each other's values. This requires some global notion of time, as the streams are updated with new values at time ticks and refer to values in other streams relative to the current tick, which is not necessarily available in a heterogeneous setting. Stream run-time verification systems also do not commonly support handling variable numbers of event sources. Some systems allow for dynamically instantiating sub-monitors for parts of the event stream [3,6,49,24] in a technique called *parametric trace slicing* [15]. This is used for logically splitting the events on a given stream into separate streams, making them easier to reason about, and can sometimes be exploited for parallelizing the monitor's work. These additional streams are internal to the monitoring logic, in contrast, Vamos' ability to dynamically add new event sources affects the monitoring system's outside connections, while, internally, the arbiter still unifies the events coming in on all such connections into one global stream.

Instrumentation The two key questions in instrumentation revolve around the technical side of how a monitor accesses a monitored system as well as the behavioral side of what effects these accesses can have. On the technical side, static instrumentation can be either applied to source code [39,30,36,37,40,34] or compiled binaries [23,20], while dynamic instrumentation, like DyanmoRIO, is applied to running programs [43,46,9]. Alternatively, monitored systems or the platforms they run on may have specific interfaces for monitors already, such as PTrace and DTrace [10,28,50] in the Linux kernel. Any of these can be used to create an instrumentation tool for Vamos.

On the behavioral side, Cassar et al. surveyed various forms of instrumentation between completely synchronous and offline [11]. Many of the systems surveyed [21,4,14,16] use a form of static instrumentation that can either do the necessary monitoring work while interrupting the program's current thread whenever an event is generated, or offer the alternative of using the interruption to export the necessary data to a log to be processed asynchronously or offline. A mixed form called *Asynchronous Monitoring with Checkpoints* allows stopping the monitored system at certain points to let the monitor catch up [25]. Our au-

todrop buffers instead trade precision for avoiding this kind of overhead. Aside from the survey, some systems (like TeSSLa [41]) incrementalize their default offline behavior to provide a monitor that may eventually significantly lag behind the monitored system.

Executing monitoring code or even just writing event data to a file or sending it over the network is costly in terms of overhead, even more so if multiple threads need to synchronize on the relevant code. Ha et al. proposed Cache-Friendly Asymmetric Buffering [29] to run low-overhead run-time analyses on multicore platforms. They only transfer 64-bit values, which suffices for some analyses, but not for general-purpose event data. Our adapted implementation thus has to do some extra work, but shares the idea of using a lock-free single-producer-single-consumer ring buffer for low overhead and high throughput.

While we try to minimize it, we accept some overhead for instrumentation as given. Especially in real-time systems, some run-time monitoring solutions adjust the activation status of parts of the instrumentation according to some metrics of overhead, inserting hole events for phases when instrumentation is deactivated [5,31,2]. In contrast, the focus of load-shedding through autodrop buffers is on ensuring that the higher-level part of the monitor is working with reasonably up-to-date events while not forcing the monitored system to wait. For monitors that do not rely on extensive summarization of dropped events, the two approaches could easily be combined.

Monitorability and Missing Events Monitorability [38,47] studies the ability of a runtime monitor to produce reliable verdicts about the monitored system. The possiblity of missing arbitrary events on an event stream without knowing about it significantly reduces the number of monitorable properties [35]. The *autodrop* buffers of VAMOS instead insert *hole* information, which some LTL [32], TeSSLa [42], and Mealy machine [54] specifications can be patched to handle automatically. Run-time verification with state estimation [53] uses a Hidden Markov Model to estimate the data lost in missing events.

8 Conclusion

We have presented VAMOS, which we designed as middleware for best-effort third-party run-time monitoring. Its goal is to significantly simplify the instrumentation part of monitoring, broadly construed as the gathering of high-level observations that serve as the basis for traditional monitoring specifications, particularly for best-effort third-party run-time monitoring, which may often need some significant preprocessing of the gathered information, potentially collected from multiple heterogeneous sources. We have presented preliminary evidence that the way we built VAMOS can handle large numbers of events and lets us specify a variety of monitors with relative ease. In future work, we plan to apply VAMOS' to more diverse application scenarios, such as multithreaded webservers processing many requests in parallel, or embedded software, and to integrate our tools with other higher-level languages. If a system's behavior conforms to the

expectation of a third party, this is generally recognized as inspiring a higher level of trust than if that monitor was written by the system's developers. We hope that our design can help making best-effort third-party run-time monitoring more common.

Acknowledgements This work was supported in part by the ERC-2020-AdG 101020093. The authors would like to thank the anonymous FASE reviewers for their valuable feedback and suggestions.

References

1. Apache Software Foundation: Apache Flink (2023), https://flink.apache.org/
2. Arafa, P., Kashif, H., Fischmeister, S.: Dime: Time-aware dynamic binary instrumentation using rate-based resource allocation. In: EMSOFT 2013. pp. 1–10 (2013). https://doi.org/10.1109/EMSOFT.2013.6658603
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: FM 2012. pp. 68–84 (2012). https://doi.org/10.1007/978-3-642-32759-9_9
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: VMCAI 2004. pp. 44–57 (2004). https://doi.org/10.1007/978-3-540-24622-0_5
5. Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S.A., Stoller, S.D., Zadok, E., Seyster, J.: Adaptive runtime verification. In: RV 2012. pp. 168–182 (2012). https://doi.org/10.1007/978-3-642-35632-2_18
6. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. Journal of the ACM **62**(2) (May 2015). https://doi.org/10.1145/2699444
7. Beyer, D.: Progress on software verification: SV-COMP 2022. In: TACAS 2022. pp. 375–402 (2022). https://doi.org/10.1007/978-3-030-99527-0_20
8. Bozzelli, L., Sánchez, C.: Foundations of boolean stream runtime verification. Theoretial Computer Science **631**, 118–138 (June 2016). https://doi.org/10.1016/j.tcs.2016.04.019
9. Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. In: VEE 2012. p. 133–144 (2012). https://doi.org/10.1145/2151024.2151043
10. Cantrill, B., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: USENIX 2004. pp. 15–28 (2004), http://www.usenix.org/publications/library/proceedings/usenix04/tech/general/cantrill.html
11. Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A.: A survey of runtime monitoring instrumentation techniques. In: PrePost@iFM 2017. EPTCS, vol. 254, pp. 15–28 (2017). https://doi.org/10.4204/EPTCS.254.2
12. Chalupa, M., Muehlboeck, F., Muroya Lei, S., Henzinger, T.A.: VAMOS: Middleware for best-effort third-party monitoring, artifact (2023). https://doi.org/10.5281/zenodo.7574688
13. Chalupa, M., Muehlboeck, F., Muroya Lei, S., Henzinger, T.A.: VAMOS: Middleware for best-effort third-party monitoring, technical report. Tech. Rep. 12407, Institute of Science and Technology Austria (2023), https://research-explorer.ista.ac.at/record/12407

14. Chen, F., Roşu, G.: Java-MOP: A monitoring oriented programming environment for java. In: TACAS 2005. pp. 546–550 (2005). https://doi.org/10.1007/978-3-540-31980-1_36
15. Chen, F., Rosu, G.: Parametric trace slicing and monitoring. In: TACAS 2009. pp. 246–261 (2009). https://doi.org/10.1007/978-3-642-00768-2_23
16. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time java programs (tool paper). In: SEFM 2009. pp. 33–37 (2009). https://doi.org/10.1109/SEFM.2009.13
17. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: Temporal stream-based specification language. In: SBMF 2018. pp. 144–162 (2018). https://doi.org/10.1007/978-3-030-03044-5_10
18. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Computing Surveys **44**(3), 15:1–15:62 (2012). https://doi.org/10.1145/2187671.2187677
19. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: TIME 2005. pp. 166–174 (2005). https://doi.org/10.1109/TIME.2005.26
20. De Bus, B., Chanet, D., De Sutter, B., Van Put, L., De Bosschere, K.: The design and implementation of FIT: A flexible instrumentation toolkit. In: PASTE 2004. p. 29–34 (2004). https://doi.org/10.1145/996821.996833
21. Drusinsky, D.: Monitoring temporal rules combined with time series. In: CAV 2003. pp. 114–117 (2003). https://doi.org/10.1007/978-3-540-45069-6_11
22. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: A race and transaction-aware java runtime. In: PLDI 2007. p. 245–255 (2007). https://doi.org/10.1145/1250734.1250762
23. Eustace, A., Srivastava, A.: ATOM: A flexible interface for building high performance program analysis tools. In: USENIX 1995. pp. 303–314 (1995), https://www.usenix.org/conference/usenix-1995-technical-conference/atom-flexible-interface-building-high-performance
24. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A stream-based specification language for network monitoring. In: RV 2016. pp. 152–168 (2016). https://doi.org/10.1007/978-3-319-46982-9_10
25. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors. Formal Methods in System Design **46**(3), 226–261 (2015). https://doi.org/10.1007/s10703-014-0217-9
26. Giatrakos, N., Alevizos, E., Artikis, A., Deligiannakis, A., Garofalakis, M.: Complex event recognition in the big data era: A survey. The VLDB Journal **29**(1), 313–352 (July 2019). https://doi.org/10.1007/s00778-019-00557-w
27. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: RV 2018. pp. 282–298 (2018). https://doi.org/10.1007/978-3-030-03769-7_16
28. Gregg, B.: DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD. Prentice Hall (2011)
29. Ha, J., Arnold, M., Blackburn, S.M., McKinley, K.S.: A concurrent dynamic analysis framework for multicore hardware. In: OOPSLA 2009. pp. 155–174 (2009). https://doi.org/10.1145/1640089.1640101
30. Havelund, K., Rosu, G.: Monitoring Java programs with Java pathexplorer. In: RV 2001. pp. 200–217 (2001). https://doi.org/10.1016/S1571-0661(04)00253-1

31. Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S.A., Stoller, S.D., Zadok, E.: Software monitoring with controllable overhead. International Journal on Software Tools for Technology Transfer **14**(3), 327–347 (2012). https://doi.org/10.1007/s10009-010-0184-4
32. Joshi, Y., Tchamgoue, G.M., Fischmeister, S.: Runtime verification of LTL on lossy traces. In: SAC 2017. p. 1379–1386 (2017). https://doi.org/10.1145/3019612.3019827
33. Kallwies, H., Leucker, M., Schmitz, M., Schulz, A., Thoma, D., Weiss, A.: TeSSLa - an ecosystem for runtime verification. In: RV 2022. pp. 314–324 (2022). https://doi.org/10.1007/978-3-031-17196-3_20
34. Karaorman, M., Freeman, J.: jMonitor: Java runtime event specification and monitoring library. In: RV 2004. pp. 181–200 (2005). https://doi.org/10.1016/j.entcs.2004.01.027
35. Kauffman, S., Havelund, K., Fischmeister, S.: What can we monitor over unreliable channels? International Journal on Software Tools for Technology Transfer **23**(4), 579–600 (2021). https://doi.org/10.1007/s10009-021-00625-z
36. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP 2001. pp. 327–353 (2001). https://doi.org/10.1007/3-540-45337-7_{1}{8}
37. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: Java-MaC: A runtime assurance tool for Java programs. In: RV 2001. pp. 218–235 (2001). https://doi.org/10.1016/s1571-0661(04)00254-3
38. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: Computational analysis of run-time monitoring - fundamentals of java-mac. In: RV 2002. pp. 80–94 (2002). https://doi.org/10.1016/S1571-0661(04)80578-4
39. Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., Sokolsky, O.: Formally specified monitoring of temporal properties. In: ECRTS 1999. pp. 114–122 (1999). https://doi.org/10.1109/EMRTS.1999.777457
40. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004. pp. 75–88 (2004). https://doi.org/10.1109/CGO.2004.1281665
41. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: runtime verification of non-synchronized real-time streams. In: SAC 2018. pp. 1925–1933 (2018). https://doi.org/10.1145/3167132.3167338
42. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Thoma, D.: Runtime verification for timed event streams with partial information. In: RV 2019. pp. 273–291 (2019). https://doi.org/10.1007/978-3-030-32079-9_16
43. Luk, C., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI 2005. pp. 190–200 (2005). https://doi.org/10.1145/1065010.1065034
44. Mansouri-Samani, M., Sloman, M.: Monitoring distributed systems. IEEE Network **7**(6), 20–30 (1993). https://doi.org/10.1109/65.244791
45. Muehlboeck, F., Henzinger, T.A.: Differential monitoring. In: RV 2021. pp. 231–243 (2021). https://doi.org/10.1007/978-3-030-88494-9_12
46. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI 2007. pp. 89–100 (2007). https://doi.org/10.1145/1250734.1250746
47. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: FM 2006. pp. 573–586 (2006). https://doi.org/10.1007/11813040_38

48. Rabiser, R., Guinea, S., Vierhauser, M., Baresi, L., Grünbacher, P.: A comparison framework for runtime monitoring approaches. Journal of Systems and Software **125**, 309–321 (2017). https://doi.org/10.1016/j.jss.2016.12.034
49. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: Monitoring at runtime with QEA. In: TACAS 2015. pp. 596–610 (2015). https://doi.org/10.1007/978-3-662-46681-0_55
50. Rosenberg, C.M., Steffen, M., Stolz, V.: Leveraging DTrace for runtime verification. In: RV 2016. pp. 318–332 (2016). https://doi.org/10.1007/978-3-319-46982-9_20
51. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems **15**(4), 391–411 (November 1997). https://doi.org/10.1145/265924.265927
52. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: Data race detection in practice. In: WBIA 2009. p. 62–71 (2009). https://doi.org/10.1145/1791194.1791203
53. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: RV 2011. pp. 193–207 (2012). https://doi.org/10.1007/978-3-642-29860-8_15
54. Taleb, R., Khoury, R., Hallé, S.: Runtime verification under access restrictions. In: FormaliSE@ICSE 2021. pp. 31–41 (2021). https://doi.org/10.1109/FormaliSE52586.2021.00010
55. Tawsif, K., Hossen, J., Raja, J.E., Jesmeen, M.Z.H., Arif, E.M.H.: A review on complex event processing systems for big data. In: CAMP 2018. pp. 1–6 (2018). https://doi.org/10.1109/INFRKM.2018.8464787
56. Thokair, M.A., Zhang, M., Mathur, U., Viswanathan, M.: Dynamic race detection with O(1) samples. PACMPL **7**(POPL) (January 2023). https://doi.org/10.1145/3571238, https://doi.org/10.1145/3571238
57. Valgrind: Helgrind (2023), https://valgrind.org/docs/manual/hg-manual.html
58. Vierhauser, M., Rabiser, R., Grünbacher, P., Seyerlehner, K., Wallner, S., Zeisel, H.: ReMinds: A flexible runtime monitoring framework for systems of systems. Journal of Systems and Software **112**, 123–136 (2016). https://doi.org/10.1016/j.jss.2015.07.008
59. Zhao, B., Viet Hung, N.Q., Weidlich, M.: Load shedding for complex event processing: Input-based and state-based techniques. In: ICDE 2020. pp. 1093–1104 (2020). https://doi.org/10.1109/ICDE48307.2020.00099