10-2019

Overhead-Aware Deployment of Runtime Monitors

Teng Zhang
University of Pennsylvania, tengz@cis.upenn.edu

Greg Eakman

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Overhead-Aware Deployment of Runtime Monitors

Abstract

One important issue needed to be handled when applying runtime verification is the time overhead introduced by online monitors. According to how monitors are deployed with the system to be monitored, the overhead may come from the execution of monitoring logic or asynchronous communication. In this paper, we present a method for deciding how to deploy runtime monitors with awareness of minimizing the overhead. We first propose a parametric model to estimate the overhead given the prior knowledge on the distribution of incoming events and the time cost of sending a message and executing monitoring logic. Then, we will discuss how to statically decide the boundary of synchronous and asynchronous monitors such that the lowest overhead can be obtained.

Keywords

runtime verification, monitor deployment, overhead

Disciplines

Computer Engineering | Computer Sciences

Comments

The 19th International Conference on Runtime Verification (RV 2019), Porto, Portugal, 8-11 October 2019

Overhead-aware deployment of runtime monitors

Teng Zhang[1], Greg Eakman[2], Insup Lee[1], and Oleg Sokolsky[1]

[1] University of Pennsylvania, Philadelphia PA 19104, USA,
{tengz,lee,sokolsky}@cis.upenn.edu
[2] BAE Systems, Burlington MA 01803, USA,
gregory.eakman@baesystems.com

Abstract. One important issue needed to be handled when applying runtime verification is the time overhead introduced by online monitors. According to how monitors are deployed with the system to be monitored, the overhead may come from the execution of monitoring logic or asynchronous communication. In this paper, we present a method for deciding how to deploy runtime monitors with awareness of minimizing the overhead. We first propose a parametric model to estimate the overhead given the prior knowledge on the distribution of incoming events and the time cost of sending a message and executing monitoring logic. Then, we will discuss how to statically decide the boundary of synchronous and asynchronous monitors such that the lowest overhead can be obtained.

Keywords: Runtime verification · Monitor deployment · Overhead.

1 Introduction

Runtime verification (RV) has been widely used to check properties of software systems. The time overhead brought by online monitors may influence the performance of the system to be monitored (denoted as the target system). Multiple factors can influence the overhead such as event sampling rate [6] or monitoring algorithm [8]. Deployment of monitors may also have impact on the overhead [5]. According to how to interact with the target system, monitors can be deployed synchronously or asynchronously with the target system. If multiple monitors are involved, they can be deployed in a hybrid way. A generally accepted assumption is that synchronous monitoring can detect the violation timely while asynchronous monitoring can incur less overhead [4]. In the real world, however, finding the deployment to achieve the least overhead is undecidable. Nevertheless, if the termination of monitors to handle each event is guaranteed and prior knowledge about the distribution of incoming events is available, it is possible to estimate the time overhead statically.

This paper presents an initial study on deciding deployment of monitors to reduce the overhead. More specifically, we propose a model to estimate the time overhead of SMEDL [12] monitors, parameterized by the distribution of incoming events, the execution time of sending a message and making a transition. Then, by analyzing the structure of monitors given the knowledge about the incoming

event stream, we will present a way to decide the boundary of synchronous and asynchronous monitors to obtain the lowest overhead.

Related Work. There are multiple approaches to reduce monitoring overhead. Considerable number of studies focus on event sampling [6, 3, 9, 1, 7]. In [8], efficient monitoring algorithms are proposed to reduce the overhead. By contrast, we are concerned with the overhead of event propagation. The RV framework in [5] supports tuning of deployments but does not offer a quantitative method. In [4], a hybrid instrumentation technique to dynamically switch between synchronous and asynchronous monitoring. The goal is to reduce the overhead by minimizing the synchronous instrumentation while ensuring timely detections. In their approach, synchronous monitoring is built upon the asynchronous communication and always has higher overhead. By contrast, we decide the deployment statically based on quantitative overhead model.

2 Preliminaries

This section briefly introduces the syntax and semantics of SMEDL. A SMEDL specification contains a set of monitor specifications and an architecture description that captures patterns of communication between them. During execution, each monitor can be instantiated statically during system startup. Specified in the architecture description, monitors can be deployed synchronously or asynchronously with the target system.

Single monitor. A SMEDL monitor is a collection of EFSMs (Extended Finite State Machines) in which the transitions are performed by reacting to events sent from the environment, other monitors or raised within the monitor. EFSMs interact with each other using shared state variables or by triggering execution of other EFSMs through raised events. Each transition is triggered by an event and attached to a guard condition and a list of actions to be executed after the transition. Actions of transitions include raising events and updating state variables. Primitive data types, arithmetic and logical operations are supported in SMEDL. The reader can refer to [13] for detailed description and formal semantics.

Monitor network. The target system and monitors interact with each other using events. Communication pattern of events among monitors is specified in the architecture description. For instance, Fig.1(a) illustrates event connection between the target system and three monitors M_1, M_2 and M_3. During runtime, multiple instances are created with different identities as shown in Fig.1(b). The architecture description specifies how events raised by a monitor instance are sent to specific instances of another monitor. For instance, when e_4 is sent from M_1 to M_2 as e_5, the first identity of M_1(x) must be equal to the identity of M_2(z). In Fig.1(b), we can observe that instance $M_1(1, 1)$ and $M_1(1, 2)$ connect to $M_2(1)$ while $M_1(2, 1)$ connects to $M_2(2)$, which is complying with the static specification. SMEDL supports specifying deployment form of monitors. As shown in Fig.1(a), M_1 is deployed synchronously with the target system while M_2 and M_3 are asynchronous monitors. Event connection specification is independent of

deployment form but the communication is decided by how they are deployed. The synchronous monitors interact with the target system by direct API calls while asynchronous communication can be implemented using communication middleware such as RabbitMQ [10].

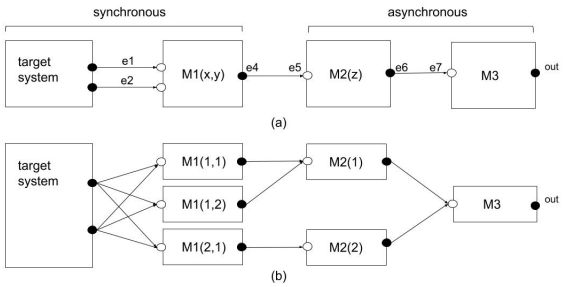


Fig. 1: An example of connections between monitors and the target system

3 Estimation and comparison of monitoring overhead

Notations. The types of events generated by the target system *sys* is a set $ES = \{e_1, e_2, ..., e_k\}$. The event stream raised by *sys* and its corresponding length are respectively denoted as S and n. *Mons* is the set of monitors in the monitor network. $Mons_{sync}$ and $Mons_{async}$ are respectively the subsets of monitors which are deployed synchronously and asynchronously with *sys*. Asynchronous monitors can receive events from synchronous monitors or the target system, but not vice versa. The accumulated overhead of the system is denoted as $OH(n)$, which is the sum of the overhead brought by the synchronous ($OH_{sync}(n)$) and the asynchronous part ($OH_{async}(n)$).

Assumptions. The time for sending an event asynchronously and making a transition, respectively denoted as t_m and t_s, can be accurately measured or estimated. Note that t_s includes time to make transition and executing actions in the transition. Actions are arithmetic/logical operations, raising and sending events to other synchronous monitors. For simplicity of analysis, we assume that transitions take approximately the same time to execute actions. It is straightforward to relax it by estimating execution time for each transition and aggregating them. The prior knowledge about the distribution of incoming events in S is also assumed to be available and simplified as the normalized frequency of appearance in S for all events in ES, denoted as $f_{e_1}, ...f_{e_2}, ..., f_{e_k}$ where $\Sigma_{e \in ES} f_e = 1$. This assumption is realistic in systems sending different types of events in a regular rate and enough data can be collected to estimate the distribution.

Overhead model. Overhead for synchronous monitors come from execution of transitions. The set of external events to be consumed by $Mons_{sync}$ is denoted as $ES_{sync} = \{e_{s_1}, e_{s_2}, ..., e_{s_i}\}$, which is a subset of ES. Each event e in ES_{sync}

may directly or indirectly trigger transitions in $Mons_{sync}$. The corresponding overhead is $t_s * tr_{(e,Mons_{sync})} * f_e$. The denotation $tr_{(e,MS)}$ represents the number of transitions triggered by e in the monitor set MS, which can be estimated by static analysis. However, the transitions triggered by an event depend on the dynamic state of the monitors and parameter values carried by the event so we do not know which transitions will be executed statically. If we choose the largest possible transition set, $OH_{sync}(n)$ may be overestimated while the smallest transition set leads to an underestimation of it. The accumulated overhead brought by S can then be computed using the following formulae:

$$OH_{sync}(n) = n * t_s * \Sigma_{e \in ES_{sync}}(tr_{(e,Mons_{sync})} * f_e)$$

$OH_{async}(n)$ includes sending events raised by sys and $Mons_{sync}$ to asynchronous monitors. Denote $ES_{async} = ES - ES_{sync}$ as the events raised from sys and sent to $Mons_{async}$. The set of events that are raised by $Mons_{sync}$ and sent to $Mons_{async}$ is denoted as ES_{raised}. Each event in ES_{raised} is directly or indirectly triggered by one or multiple events in ES_{sync}. We use $g_{(e',e)}$ to denote the number of instances of e generated by each instance of e'. $OH_{async}(n)$ can be computed using the following formula:

$$OH_{async}(n) = n * t_m * (\Sigma_{e \in ES_{async}} f_e + \Sigma_{e \in ES_{raised} \wedge e' \in ES_{sync}} f_{e'} * g_{(e',e)})$$

Note that to estimate the value of g and tr, we assume that all event instances of the same type are dispatched to the same monitor instances regardless of their parameter values. Furthermore, n will be ignored in the rest of the paper as both formula are the linear function of n.

Determine the deployment. The SMEDL monitor network can be modeled as a direct acyclic graph (DAG) where nodes are the target system and monitors and edges are event connections. All instances of the same monitor are treated as one node in DAG. M is a direct upstream monitor to M' when M sends events to M' and M' is the direct downstream monitor of M. In this paper, we consider a simpler case in which the monitor network is a chain of monitors, which means each monitor in $Mons$ only has one direct upstream and downstream monitor and only one monitor directly receives events from the target system. Algorithm 1 computes $Mons_{sync}$, the set of monitors to be deployed synchronously. While traversing the monitor chain ($MonsChain$) and the current monitor is mon, the overall overhead OH_{cur} including mon as the synchronous monitor is computed (Line 4 to Line 9). If it is smaller than the least overhead seen so far (denoted as OH_{min}), mon and all pending monitors in $Temp_{sync}$ are added to $Mons_{sync}$ (Line 11 to Line 14). Otherwise, add mon to $Temp_{sync}$. Note that the set of input events of mon is the set of output events of its direct upstream monitor. As a result, f_e can be computed for every e in the set of output events since values of all f'_e are already available.

To summarize, the method includes the following steps: 1) measure t_m and t_s on the actual platform for executing the target system and monitors; 2) estimate frequencies f_{e_i} of events raised by the system 3) for each monitor $m \in Mons$ with

Algorithm 1 Determination of synchronous monitors

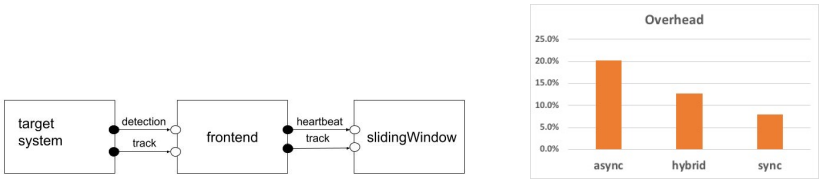
$Mons_{sync} \leftarrow \emptyset$, $Temp_{sync} \leftarrow \emptyset$, $OH_{min} \leftarrow t_m$, $OH_{async} \leftarrow t_m$, $OH_{cur} \leftarrow t_m$
while $MonsChain \neq \emptyset$ **do**
  $mon \leftarrow dequeue(MonsChain)$
  $Ev \leftarrow inputEvents(mon)$
  $tempOH_{sync} \leftarrow t_s * \Sigma_{e' \in Ev}(tr_{(e',\{mon\})} * f_{e'})$
  for $e \in outputEvents(mon)$ **do**
    $f_e \leftarrow \Sigma_{e' \in Ev} f'_e * g_{(e',e)}$
  $tempOH_{async} \leftarrow t_m * \Sigma_{e \in outputEvents(mon)} f_e$
  $OH_{cur} \leftarrow OH_{cur} + tempOH_{async} + tempOH_{sync} - OH_{async}$
  $OH_{async} \leftarrow tempOH_{async}$
  if $OH_{min} > OH_{cur}$ **then**
    $Mons_{sync} \leftarrow Mons_{sync} \cup \{mon\} \cup Temp_{sync}$
    $Temp_{sync} \leftarrow \emptyset$
    $OH_{min} \leftarrow OH_{cur}$
  else
    $Temp_{sync} \leftarrow Temp_{sync} \cup \{mon\}$
return $Mons_{sync}$

the set EI_m and EO_m of input and output events, compute $g_{(e',e)}$ and $tr_{(e',\{m\})}$ where $e' \in EI_m$ and $e \in EO_m$; 4) compute $Mons_{sync}$ using Algorithm 1.

4 Case study

We present two examples to illustrate the use of method presented above. Both examples use a tracking application which receives sensor data of tracks. The experiments were conducted on a virtual machine of Ubuntu 18.04 64-bit run on a laptop with 2.5GHz Intel i7 processor and 16GB RAM. The first case study is a single monitor *checkFormat* which takes the input messages collected from the sensor. For each input event, one transition is taken to check whether the format complies with certain protocol. Only fully asynchronous and synchronous deployments need to be considered. The synchronous deployment has less overhead if $t_m/t_s > \Sigma_{e \in ES}(tr_{(e,\{checkFormat\})} * f_e)$. In this example, the right-hand side is equal to 1 and t_m/t_s is around 16. The testing results validate the estimation: the overhead of synchronous monitor is less than 5% while the overhead of the asynchronous monitor is about 20%.

The second example is the track quality monitors [11]. The monitors check output track quality of the tracking application by computing average duration over a sliding window time interval. There are two types of events generated from the target system, *track* which forms the track and *detection* which is used to generate *heartbeat* event as the boundary of the sliding window. The structure of the monitor is illustrated in Fig. 2(a). We assume that both *frontend* and *slidingWindow* have one instance. This monitor has three possible deployments: fully synchronous, fully asynchronous, and hybrid, where only *frontend* is deployed synchronously.



(a) Monitor structure (b) Overhead

Fig. 2: Track quality monitor and the overhead for 10000 detection events

Suppose the size of sliding window is 1000ms and the time gap between each *detection* event is about 10ms, then $g_{(detection,heartbeat)}$ is 1/100. The input event stream has the identical number of *track* and *detection* events so $f_{detection}$ and f_{track} are equal to 1/2. The overhead of fully asynchronous monitoring is t_m. According to Algorithm 1, we first compute the overhead when *frontend* is synchronously deployed. Each *detection* and *track* event trigger one transition in the $frontend$ monitor so $tr_{(detection,\{frontend\})}$ and $tr_{(track,\{frontend\})}$ are equal to 1. Moreover, *frontend* immediately resends the *track* event. Consequently, the overhead is $t_s * (1/2 + 1/2) + t_m * (1/2 + 1/2 * 1/100) = t_s + 0.505 * t_m$. We can deduce that if $t_m/t_s > 200/99$, *frontend* should be deployed synchronously. Recall that t_m is 15 times greater than t_s. Fig. 2(b) illustrates that the overhead of hybrid deployment is less than asynchronous deployment, which is consistent with the model.

5 Future work

In this paper, we proposed a model to estimate the overhead of monitors statically given the prior knowledge of frequency among different type of events and the static structure of the monitor specification. We give an intuitive method to decide the deployment of chains of monitors. Although the model is specific to SMEDL monitors, it can also be used in other automata-based RV techniques. Moreover, the idea of trade-off between synchronous and asynchronous monitoring is not unique to specific formalisms and one future work would be the generalization of the model and algorithm to other formalisms for monitoring logics. For example, the model for rule-based monitors such as Eagle [2] can be expressed in terms of the number of rule firings rather than transitions.

Avenues of on-going work include: 1) more experiments on multiple applications to yield conclusive results for validation of the model; 2) monitor analysis to estimate the number of transitions triggered by each input event; and 3) for dynamic instantiation of monitors, we will extend the model to account for instantiation overhead. Finally, we will invest in a more accurate overhead measurement infrastructure. Currently, for computationally intensive systems, overhead calculation is often noisy, making it hard to validate predictions of our model when differences between deployments are small, as in the case with hybrid vs. synchronous deployment in the second example above.

References

1. Arnold, M., Vechev, M., Yahav, E.: Qvm: an efficient runtime for detecting defects in deployed systems. ACM Sigplan Notices **43**(10), 143–162 (2008)
2. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Program monitoring with LTL in EAGLE. In: 18th International Parallel and Distributed Processing Symposium (IPDPS) (Apr 2004). https://doi.org/10.1109/IPDPS.2004.1303336, https://doi.org/10.1109/IPDPS.2004.1303336
3. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Sampling-based runtime verification. In: International Symposium on Formal Methods. pp. 88–102. Springer (2011)
4. Cassar, I., Francalanza, A.: On synchronous and asynchronous monitor instrumentation for actor-based systems. arXiv preprint arXiv:1502.03514 (2015)
5. Colombo, C., Francalanza, A., Mizzi, R., Pace, G.J.: polylarva: runtime verification with configurable resource-aware monitoring boundaries. In: International Conference on Software Engineering and Formal Methods. pp. 218–232. Springer (2012)
6. Fei, L., Midkiff, S.P.: Artemis: Practical runtime monitoring of applications for execution anomalies. In: ACM SIGPLAN Notices. vol. 41, pp. 84–95. ACM (2006)
7. 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)
8. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Tools for Technology Transfer **14**(3), 249–289 (2012)
9. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: International conference on runtime verification. pp. 193–207. Springer (2011)
10. Videla, A., Williams, J.J.: RabbitMQ in action: distributed messaging for everyone. Manning (2012)
11. Zhang, T., Eakman, G., Lee, I., Sokolsky, O.: Flexible monitor deployment for runtime verification of large scale software. In: International Symposium on Leveraging Applications of Formal Methods. pp. 42–50. Springer (2018)
12. Zhang, T., Gebhard, P., Sokolsky, O.: SMEDL: Combining synchronous and asynchronous monitoring. In: International Conference on Runtime Verification. pp. 482–490. Springer (2016)
13. Zhang, T., Wiegley, J., Giannakopoulos, T., Eakman, G., Pit-Claudel, C., Lee, I., Sokolsky, O.: Correct-by-construction implementation of runtime monitors using stepwise refinement. In: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications. pp. 31–49. Springer (2018)