Received June 3, 2020, accepted June 10, 2020, date of publication June 17, 2020, date of current version June 30, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.3003087*

# Runtime Monitoring of Software Execution Trace: Method and Tools

**SHIYI KONG[1,2], MINYAN LU[1,2], (Member, IEEE), LUYI LI[3], AND LIHUA GAO[4]**

[1]Key Laboratory on Reliability and Environmental Engineering Technology, Beihang University, Beijing 100191, China
[2]School of Reliability and System Engineering, Beihang University, Beijing 100191, China
[3]China Academy of Electronics and Information Technology, China Electronics Technology Group Corporation, Beijing 100041, China
[4]China North Vehicle Research Institute, Beijing 100072, China

Corresponding author: Minyan Lu (lmy@buaa.edu.cn)

This work was supported by the Advanced Research Domains Funding of China under Grant JZX7Y20190242013901.

**ABSTRACT** As with the extensive use of complex software in many fields, such as finance, transportation, aeronautics, and astronautics, software plays an increasingly more important role in society. Software reliability becomes a critical bottleneck for system reliability and draws increasingly greater attention from software engineers. Many researchers find that traditional software verification and validation techniques are not sufficient to ensure complex software reliability, especially after being deployed. In this paper, therefore, we propose a novel software runtime monitoring method that can help with software runtime verification and software failure prediction. This method aims to monitor software execution trace that reveal the software runtime status. First, the framework of this method is proposed to introduce the entire process and two key problems. Second, an instrumentation method based on srcML, an open source tool used to extract the software abstract syntax tree, is presented. By this instrumentation technique, monitoring code can be inserted into source code and runtime data can be exported during the software execution process. Then, a runtime data-collection technique is proposed to collect runtime data exported by monitoring code. On one hand, the file stream mechanism is used to export data files that can be used to support offline analysis, while on the other hand a client-server structure is proposed to support online analysis. Finally, a case study on Nginx is used to show the feasibility of the proposed method.

**INDEX TERMS** Runtime monitoring, software execution trace, failure mechanism, software reliability.

## I. INTRODUCTION

With the development of computer science and software engineering, complex software systems have assumed an increasingly important role in human life. How to ensure the reliability of such software systems has attracted increasingly more attention from researchers.

Traditional software reliability assurance technologies consist of software testing, reliability testing, formal verification, reliability prediction and estimation, and standard compliance. However, recent research shows that even in the case of traditional software systems these methods cannot ensure software reliability after deployment. The adequacy of software testing cannot be guaranteed [1]. Formal verification methods are faced with complexity issues [2]. Reliability prediction and estimation methods usually work in early periods, which lack sufficient real data, so the result may not be accurate enough [3]. The design of highly reliable software systems is driven by the functionality it must deliver as well as the faults it must survive [4]. Thus, more researchers have turned their attention to software run-time reliability assurance technologies, such as failure prediction, failure management, and software health management. These technologies aim at observing system behaviors [5]–[7], detecting abnormal behaviors [8]–[10], and manipulating detected failures [11], [12].

We follow the definition of runtime verification in [13]: "Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property." Runtime verification should 1) monitor software runtime states, and 2) check if the current states satisfies or violates a given correctness property.

Run-time software reliability assurance techniques consist of three main parts, 1) monitor software runtime states,

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Pietrantuono.

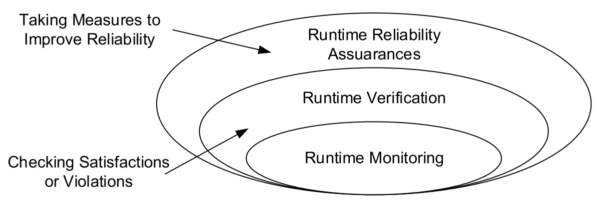

Taking Measures to Improve Reliability
Runtime Reliability Assuarances
Runtime Verification
Runtime Monitoring
Checking Satisfactions or Violations

**FIGURE 1.** **Software runtime reliability assurances,runtime verification, and rumtime monitoring.**

2) check if there are violations against a given correctness property, and 3) take operations used to improve runtime reliability.

The procedure of software run-time monitoring is defined as follows: A monitor observes the behaviors of the monitored system and detects if they are consistent with the specifications [4]. Run-time monitoring technologies observe software running status, monitor software behaviors, record run-time data, and support future analysis both online and offline. Software run-time monitoring provides run-time reliability improvement operations with necessary run-time data.

The relationships of Software runtime reliability assurances,runtime verification, and rumtime monitoring are shown in Figure 1.

Software run-time monitoring technologies are proposed as methods aimed at improving the reliability of safety-critical software systems at beginning [4], as a part of runtime verification. The health of general complex software systems are the next priority. These technologies observe software behaviors and obtain software run-time behavior information during the software execution process, determine whether the software behavior complies with its expected behavior, and estimate the credibility of current software status. Software behavior information can be used to perform fault localization and failure prediction. Currently, software run-time monitoring technologies have been applied to several domains, such as performance analysis, cost-effect analysis, software optimization, software fault detection, diagnosis, and recovery.

Although software run-time monitoring technologies have been studied for 30 years, the recent emergence of complex software systems brings new challenges. These systems have many new features, e.g., multi-threading and concurrency. Increasingly more software failures are related to the run-time environment and user profile [14]. Traditional reliability assurance technologies cannot deal with these features very well, as stated above. Software run-time monitoring therefore assumes a more important role in these situations.

We propose a novel approach for software system run-time monitoring, Software Runtime Monitoring Tool (SRMT) with a GUI interface. SRMT can monitor software execution trace during software execution process selectively. There are two main parts of this approach: 1) instrumenting software source code based on srcML, and 2) collecting runtime data via the SRMT framework proposed in this paper. The method is aiming at C/C++ programs that work on Windows XP SP3 or later Windows operating systems. Selective monitoring are also supporting in SRMT framework. SRMT also uses multi-thread design to mitigate the overheads brought by the instrumentation.

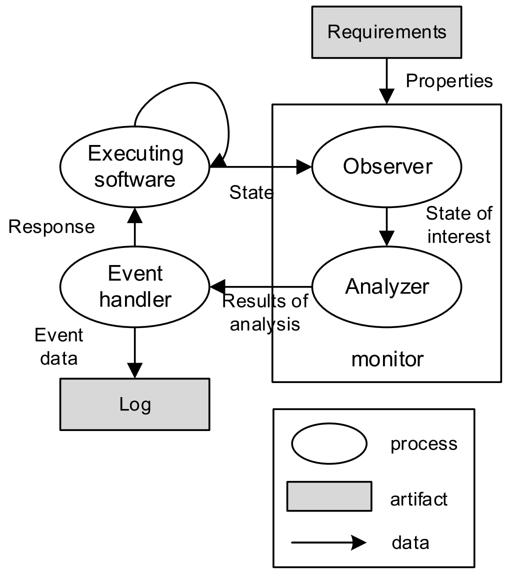


**FIGURE 2.** **High-level view of runtime monitor [15]. The circles represent process, the rectangle represents the artifact, and the arrows represents data flow.**

This paper can be divided into seven parts. Section II introduces the related work of this paper. Section III provides a framework for how our method and tools work. Instrumentation techniques are introduced in Section IV. Section V introduces data-collection mechanisms for online and offline analysis. Section VII presents the case study on Nginx. A summary of this approach is given in Section VIII.

## II. STATE OF THE ART

In this section, a brief review of existing software run-time monitoring technologies are given from several aspects.

### A. BASIC RUN-TIME MONITORING FRAMEWORK

Delgado *et al.* [15] gives a typical high-level view of a run-time monitor, which is shown in Figure2.

The entire monitoring system takes an executable software system and the specifications of software properties as inputs. The specifications came from software requirements are a series of descriptions of software external behaviors. The specifications contains a sequence of states and states transformation relationships. They will show which inner states can lead to normal external behaviors [16], [17].

The *monitor* can be divided into two parts in this model, *observer* and *analyzer*. The *observer* observes actual transitions between software states, extracts interesting states related to its monitoring goal, and passes the data to the *analyzer*. The *analyzer* verifies the compatibility between actual states of interest and software properties. The latter are extracted from software requirements and passed to the *analyzer* by the *monitor*.

When a violation of a property is detected by the *analyzer*, the results are passed to an *event handler*, and a series of operations must be executed to provide a response, such as halting the program, entering a recovery routine, or sending data to a log.

### B. REVIEW OF RUNTIME MONITORING TECHNOLOGIES

There are two ways to classify runtime monitoring technologies from different aspects.

First, the following classification is based on different performing periods:

- Offline Runtime Monitoring:
  This kind of method collects software runtime information together with the software execution process and analyzes the data offline. Research work described in [18], [19] use this method.
- Online Runtime Monitoring:
  All data collection and analysis work in online methods run parallel with the execution process, as in the methods used in [20], [21].

Second, the following classification is based on the way runtime data are collected:

- Invasive Runtime Monitoring:
  Invasive methods insert monitoring code into object systems to collect runtime data and analyze it. All of the above work is completed in object systems. Typical methods are Luckham's [22] and Chodrow's [23].
- Non-Invasive Runtime Monitoring:
  Non-invasive methods monitor object systems in another parallel program outside object systems, and neither insert code nor change any source code of object systems. Bro System [20] and BusMOP frameworks [24], [25] use this method to do the monitoring work.
- Combination of Above Two:
  In this type, the data analysis work is noninvasive while the behavior observation work usually is invasive. References [26], [27] are examples of this type.

Above all is just a simple classification based on two viewpoints, which is the concern in this paper. For more details on the taxonomy and catalog of software runtime monitoring, see Nelly's survey in [15]. In the following sections, more details of these methods will be introduced.

#### 1) OFFLINE RUNTIME MONITORING

Most software monitoring methods proposed historically were offline methods. The term *monitor* especially refers to collecting program traces in earlier periods of development. Offline software runtime monitoring is defined as collecting runtime data online while analyzing the data offline.

Offline methods usually work in debugging work. Tracking program execution process can help perform fault localization work and replay bugs [18], [28]. This kind of method usually uses sensors to capture system calls, interrupts, context switches, and variable timestamps. These methods try their best to minimize the interference to the system being monitored and do not add much runtime overhead [19]. By doing this, one can replay bugs and analyze program execution processes while doing the analysis work offline.

In offline methods, data-analysis work is done separately from data-collection work and the program execution process [28]. This situation has some advantages and limitations.

First, because little work must be done at the same time with the program execution process, relatively little overhead is added to the original software system. Usually, the monitor outputs little data at several specific breakpoints or uses external sensors to collect CPU and memory usage information [29].

Second, separating data-analysis work from data-collection work can avoid some time-related problems since there is no need to do analysis online. Less time consumption allows monitored systems to continue their processes as soon as possible after the break caused by invasive monitoring techniques [15].

#### 2) ONLINE RUNTIME MONITORING

Online runtime monitoring methods observe software runtime status dynamically, collect runtime information, and judge the consistency between software behaviors and specifications. Different from offline methods, online methods analyze data at the same time as the software execution process and give an immediate response most of the time.

*Bro* is a typical independent system for real-time detection of network intruders by passive monitoring of network traffic packets [30]. *Bro* uses an event engine to reduce the stream of network packets into network events and sends these events to a security analyzer. The security analyzer checks the event stream with the site's security policy and provides real-time notification to users. More details on how *Bro* captures these network packets will be introduced in the section covering non-invasive runtime monitoring methods.

Luckham *et al.* [22] proposed a framework for online runtime monitoring. A series of programs are used to convert formal comments that are converted from formal specifications into runtime check code in this framework. Monitoring code is inserted into the underlying program. When the resulting program is executed, check code can find any inconsistency in the program with respect to the formal specification. Check code can trigger an external response tool for abnormal events.

The two on-line runtime monitoring methods mentioned above check software behaviors during the program execution process, which may block the working process of the original program. Chodrow proposed two methods, synchronous and asynchronous, to mitigate the effects of this interruption [23]. In the synchronous method, only critical points will be checked, while in the asynchronous method a separate task thread is built to analyze the consistency with respect to the formal specification. In this way, Chodrow expanded runtime monitoring methods into real-time systems. Since then, increasingly more researchers have tried to mitigate the impact of instrumentation and make the overhead more affordable [31], [32].

### 3) INVASIVE RUNTIME MONITORING METHODS

Invasive runtime monitoring methods simply insert monitoring code into the source code of the monitored software system. Thus, there is no need for additional external modules. There are two kinds of such methods, depending on the type of monitoring code inserted into the source code.

An *annotation* is a form of metadata added in the source code. Almost all levels of software parts can be annotated, e.g., classes, methods, variables, parameters, and packages. Annotation usually does not have any direct effect on program execution, although some annotations can achieve that goal.

In Luckham's work [22] on runtime consistency checking, which we introduced in section II-B2, annotations are converted from formal specifications and inserted into source code. This work is mainly for the ***Ada*** programming language. Luckham extends Ada with annotations, called ***Anna*** (ANNotated Ada). Annotations act like checking rules that an analyzer can process automatically to check if current execution is consistent with specifications. The analyzer can choose whether to enable or disable the specific annotation. In this way, programmers can customize their own annotation sets depending on their separate requirements.

As described above, Luckham's method monitors program execution and analyzes consistency in the same thread with the monitored program, which may cause unaffordable time consumption and additional system overhead. Therefore, this method merely applies to debugging and testing work. Many researchers have made efforts to mitigate overheads. A synchronous and asynchronous method proposed in [23] have been described in Section II-B2. To solve the problem of too much time consumption, Jahanian introduced an environment for distributed real-time system runtime monitoring [33]. This method solves several critical issues in the distributed real-time monitoring domain. To make the monitoring more useful, this method detects violations as early as possible. This method also minimizes the number of messages that must be exchanged to reduce time consumption. Jahanian also determines a proper granularity of timestamping to strike a balance between decision accuracy and system overheads.

Recent research based on annotations includes the following. Grigoropoulos used macros and hook function mechanisms in C/C++ programming languages [31], [34]. In this work, a separate process is opened up for the monitoring module, and it communicates with other working processes via inter-process communication mechanisms provided by ***Contiki OS*** [35]. Bodden proposed ***Clara***, which can use a sequence of static analyses to automatically convert a monitoring aspect into a residual runtime monitor [36]. Clara uses ***JavaMOP*** [37] to generate annotations and uses a dependency state machine to weave the monitoring aspects into a program. The dependency state machine also provides Clara with enough domain specific knowledge to analyze the woven program.

*Assertion* is a statement that a predicate (Boolean-valued function) is always true at that point in normal code execution, which can be applied for automated fault detection during debugging, testing, and production-use periods [38].

Assertion-based runtime monitoring methods usually consist of the following features [39]:

- A well-defined high-level language is used to describe Boolean-valued expressions to characterize valid program behaviors;
- A specific, well-defined syntax is used to map those logical expressions to pre-defined program states;
- An approach is given for automatic transformation from logical expressions to executable code, in which the latter can be used to check the states of the program;
- A response mechanism is applied if there is any violation of assertions found during the runtime execution process. This part is optional.

There are several typical runtime monitoring methods based on assertions. Gan used assertions to help monitor web-service conversations in [40]. Assertions are added into *Sequence Diagrams* for modeling behavioral scenarios. Then, the sequence diagrams are converted to automation used to monitor messages online. Kosmatov proposed a solution [41] for memory monitoring of C programs based on ***FRAMA-C*** [42], a platform for analysis of C programs. The proposed solutions can be applied in two ways, *passive* and *active* monitoring. More details on the history of assertions used in runtime monitoring work can be found in Clarke's review [39].

### 4) NON-INVASIVE RUNTIME MONITORING METHODS

Non-invasive methods use an external monitor to monitor program execution process. This monitor usually runs independently with the program being monitored, without consuming any resources of the monitored software.

Existing non-invasive methods focus more on performance measures or temperature control [37]. Spanoudakis and Mahbub [43] proposed an approach for web-services runtime monitoring. The monitoring process works in parallel with the normal operation of the object web application without interrupting it. Intercepting events are used to exchange information between the object and the monitoring process. Since there is no need to instrument the source code, there is limited additional overhead added to object systems.

There is also a kind of monitoring method that uses hardware as the monitor that has seen less active research recently. Lu and Forin [44] used ***eMIPS***, a dynamically self-extensible processor, to monitor program execution without interrupting its temporal behavior. The *PSL2Verilog* compiler is used to transform a set of assertions into hardware that runs in parallel with the object program. ***BusMOP***, a framework realized by Pellizzoni *et al.* [25], was plugged into a peripheral bus. It monitors the COTS peripheral behaviors by examining read and write transactions on the bus. The monitor is executed on FPGAs, resulting in zero runtime overhead on the system CPU.

#### 5) COMBINATION METHODS

Combination methods usually make the analysis work non-invasive, in turn making the influence on the system being monitored as light as possible.

***MaC*** framework, as an example, has invasive observation parts and non-invasive consistency analysis parts [45]. All of these parts work at the same time with monitored software. There are three main components in *MaC*. *Filter* is used to extract low-level information (namely values of variables or time when variables changed) from running code. *Event recognizer* is used to transform low-level information obtained by *filter* into high-level information that *runtime checker* can understand. Finally, *runtime checker* checks the consistency of program execution with pre-defined specifications. A toolset named ***Java-MaC*** [26], implemented by Lee, can monitor temporal properties [28] and dynamic properties [46] in real-time systems [7].

## III. FRAMEWORK FOR COMPLEX SOFTWARE EXECUTION TRACE RUNTIME MONITORING

Software execution trace as used in this paper refers to the information consisting of software call relationships, the time when the invocation occurs, the duration of each invocation, and calling frequency, which are similar to the concepts in existing execution-trace-extracting research.

Software execution trace can be applied to many domains. Moe and Carr [47] extracted execution trace to help understand and modify distributed systems. Taniguchi used trace help with JAVA software to understand the evolution process [48]. Liu *et al.* [49] used execution trace to locate software functions in source code.

As far as we are concerned, execution trace reveal software runtime behaviors and statuses. They also can help understand, change, and improve software. More specifically, that information can be applied to failure prediction domains [50]. Therefore, it is important to obtain that information first.

A framework for software runtime monitoring of software execution trace is proposed in this section, which aims at supporting software failure prediction work in the future. Both online (such as online consistency analysis) and offline (such as logging information for future offline analysis) application purposes can be supported in this framework.

As we introduced in Section II, most existing non-invasive monitoring methods focus on performance measures, because performance usually relates to the external resource, which can be monitored outside of the monitored program. However, regarding correctness verification or specification consistency analysis, researchers choose invasive monitoring methods to observe program execution, because certain properties, usually related to internal information, such as variables, messages, and execution path, cannot be easily observed outside of the monitored program [50].

The framework is shown in Figure 3. The three main parts of the framework are detailed below.

### A. EXECUTION PROCESS

Source code is instrumented with monitoring code in this process. We propose a novel instrumentation technique with the support of srcML, an infrastructure for the exploration, analysis, and manipulation of source code [51]. After instrumentation, the source code is compiled and outputs an executable program. Then, we run this program, and monitoring code inserted in the object software can automatically monitor program execution and record specific information.

### B. MONITORING WORK

This section can be divided into two parts, before and after the execution part. In the former, monitor specifications are defined according to monitor goals. These specifications help do the instrumentation work. In the latter, there are two kinds of scenarios. In the first, monitoring data are in the form of messages that are sent to the online analyzer via a well-designed multi-threaded message-sending mechanism. In the second, monitoring data are written to log files; to mitigate time consumption as much as possible, we design this part carefully by using multi-threaded and file-stream techniques supporting C++.

### C. ANALYZING WORK

This framework supports two kinds of analyzers, both online and offline. In the offline part, log files obtained in previous steps are processed to rebuild the program execution process. We extract execution trace from primitive data and build a program trace with timestamps using those trace. Taking failure prediction as an example, failure indicators can be extracted from the program trace, as Li described in [50]. Regarding the online part, messages received from the monitor are pre-processed according to the requirements of the module that takes the responsibility of correctness verification. The latter module will check data from pre-processing work and call a response if there is any violation of the specifications in the monitoring module. For example, failure can be converted into failure symptoms and then to specifications that the checking module can use for prediction. If the data match the rules, a recovery mechanism will be called to ensure that the system will work normally (after recovery).

## IV. INSTRUMENTATION TECHNIQUES

Instrumentation techniques are used to form a set of mechanisms that can export runtime trace during the program execution process. There are three steps to do this work. First, determine what is to be monitored. Then, determine where the monitoring should be done. Finally, weave the monitor specifications into the source code and generate executable code. This process is illusrated in Figure 4.

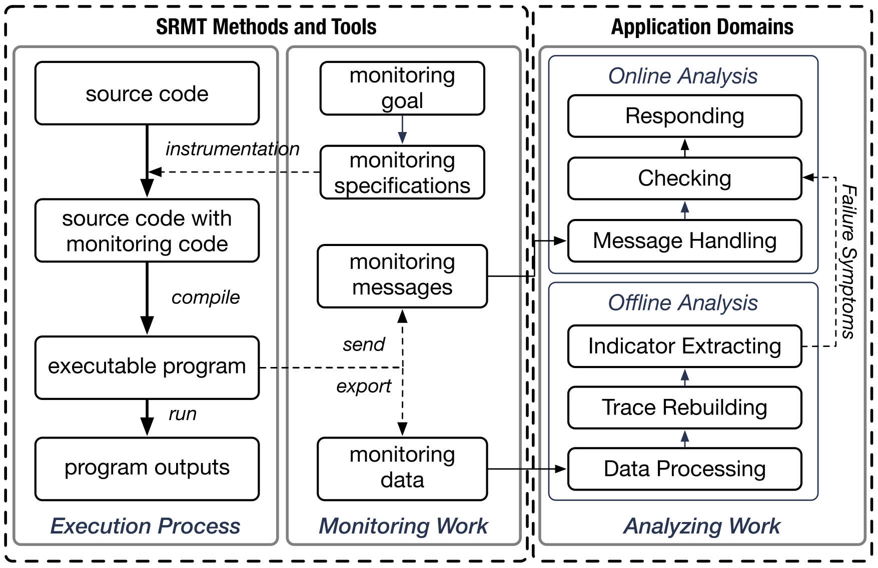


**FIGURE 3.** Framework of complex software runtime monitoring for software verification.

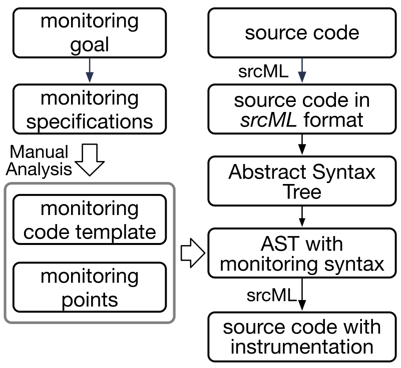
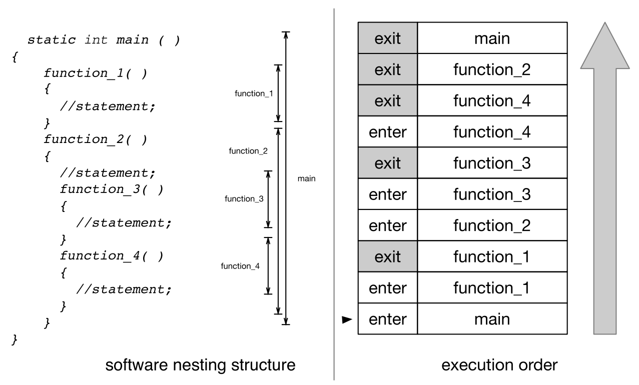


**FIGURE 4.** Instrumentation process.



```
    static int main ( )
{
    function_1( )
    {
        //statement;
    }
    function_2( )
    {
        //statement;
        function_3( )
        {
            //statement;
        }
        function_4( )
        {
            //statement;
        }
    }
}
```

| | |
|---|---|
| exit | main |
| exit | function_2 |
| exit | function_4 |
| enter | function_4 |
| exit | function_3 |
| enter | function_3 |
| enter | function_2 |
| exit | function_1 |
| enter | function_1 |
| enter | main |

**FIGURE 5.** Software invocation process.

## *A. DETERMINE MONITORING SPECIFICATIONS ACCORDING TO MONITORING GOALS*

As we introduced in Section 3, software execution trace consist of software call sequences, the time when those invocations occur, the duration times of those invocations, and calling frequency. The monitoring goal used in this paper is software execution trace, but not all of them can be obtained directly during the execution process.

Bryant and O'Hallaron, in their well-known work *"Computer Systems: A Programmer's Perspective,"* described the software invocation process in detail [52]. C/C++ applications use a stack-like structure to manage memory, so the software execution process becomes a series of functions nested inside another series of functions, as Figure 5 shows [53].

### 1) INVOCATION DURATION

To obtain the invocation duration, we can record the current time at each function's entry and exit points. Thus, the durationt can be calculated by Equation (1), in which $T_{Duration}$ represents the duration of the current function. $t_{Enter}$ is the time at function enter point and $t_{Exit}$ the time at function exit. $T^i_{CalleeDuration}$ denotes the duration time of function $i$ called by the current function:

$$T_{Duration} = t_{Enter} - t_{Exit} - \sum_{i=1}^{m} T^i_{CalleeDuration} \qquad (1)$$

### 2) CALLING SEQUENCE AND FREQUENCY

Owing to the special structure of invocation, we can rebuild the calling sequence from the nested structure using algorithm 1.

The process program first reads execution records line by line. If it is noted that the current line was collected at the entrance of a function, then a new node is added to the tree model. Entry time will be recorded, and the new node will be marked as the current node. If the current line is collected at the exit of a function, the exit time will be recorded and the parent node will be marked as the current node.

Calling sequence is stored in the form of a tree model, and the calling times of each invocation pair $N_{i \to j}$ can be obtained

**Algorithm 1** Building Calling Sequence (in Tree model)

```
Input: $L \geq 2$ {L denotes execution order list}
Output: $T$ {T denotes calling sequence tree}
1:  $length \leftarrow length\ of\ L$
2:  $current\_node \leftarrow L_0$
3:  for $i = 1$ to $length$ do
4:      if $L_i$ is $enter$ line then
5:          $current\_node$ add a new node named $L_i.function\_name$
6:          $child\_node.enter\_time = L_i.enter\_time$
7:          $current\_node \leftarrow child\_node$
8:          update $T$
9:      else {$L_i$ is $exit$ line}
10:         $current\_node.exit\_time \leftarrow L_i.exit\_time$
11:         $current\_node \leftarrow current\_node.parent\_node$
12:         update $T$
13:     end if
14: end for
```

**TABLE 1.** Example of runtime data that must be collected during the execution process.

| No | Flag | Function Name | Enter Time | Exit Time |
|---|---|---|---|---|
| 0 | enter | main | 12300 | - |
| 1 | enter | func1 | 12345 | - |
| 2 | enter | func2 | 12355 | - |
| 3 | exit | func2 | - | 12435 |
| 4 | exit | func1 | - | 12540 |
| 5 | exit | main | - | 12550 |

from that model. $N_{i\to j}$ denotes the occurring times of the events that function $i$ calls function $j$.

Calling frequency $P_{i\to j}$ can be calculated using Equation (2):

$$P_{i\to j} = \frac{N_{i\to j}}{\Delta t} \tag{2}$$

While $\Delta t$ denotes the execution time of total programs for calculating. For example, the calling frequency of $P_{i,j}$ can represent the calling times of function $i$ calling function $j$ per seconds.

After the above analysis, we determine the data that need recording directly during the execution process as in Table 1. These data can be collected at the entrance and exit of all or specific (depending on the monitoring goals) functions in the program.

## B. EXTRACT ABSTRACT SYNTAX TREE OF OBJECT SOFTWARE AND MARK MONITORING POINTS

### 1) PARSE SOURCE CODE USING srcML

***srcML*** is an infrastructure for the exploration, analysis, and manipulation of source code [51]. srcML uses XML to represent source code without loss of information. srcML extracts the abstract syntax tree (AST) of source code and adds semantic labels to the XML files converted by srcML tools. The

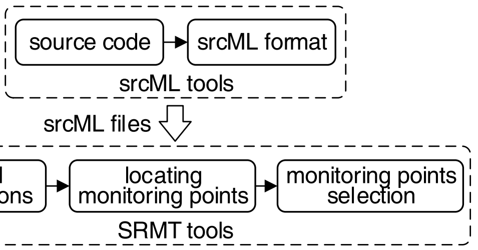


**FIGURE 6.** Parsing process for source code and determination of monitoring points.

conversion between srcML and source code is reversible, so it is possible to make some changes to the XML automatically and transform it and return it to the source code. The parsing process is shown in Figure 6.

First, srcML tools are used to convert source code into AST structures in srcML (XML) format. Then, **Software Run-time Monitoring Tools** (SRMT), our tools proposed in this paper, can be used to understand the program structure. SRMT use a traversal algorithm (Algorithm 2) to analyze srcML files one by one. Each function and function invocation are marked, and the SRMT can return XPath strings to help locate them easily.

Properties of functions such as function name, parameter list, and returns (the values of return expressions) are also marked and stored in SRMT, as in algorithm 3. In particular, each function will be attached with a label that shows if this function must be monitored or not. The value of this label depends on users when they make a choice in the SRMT GUI interface. The SRMT provide a GUI interface for monitoring-point selection. Users can choose to monitor all functions (usually for offline analysis) or just monitor several specific functions (usually for online analysis).

### 2) CREATE MONITORING TEMPLATES

Monitoring templates are used to generate real code in the process in which srcML files are converted to source code. The monitoring module is independent of the monitored system. These two communicate with each other by invocation processes. Therefore, monitoring templates designed in this section aim at providing the monitored system a general structure to interact with the monitoring module. Monitoring templates are different at different monitoring points. The following is an example of three common templates. Following three functions are implemented in `monitor.cpp` written by SRMT. This file `monitor.h` must be included in source code while compiling.

1) `monitorAtFunctionBegin (funcName)` is used at the entrances of functions.
2) `monitorAtFunctionEnd (funcName)` is used at the exit of functions.
3) `monitorBeforeFunctionCall(callerName, calleeName)` is used at the monitoring points at which an invocation process starts.

srcML can also transform code snippets into srcML nodes. The above three templates can be converted to srcML nodes

**Algorithm 2** srcML File Traversal Algorithm

```
Input: srcML file
Output: each function's locations (using XPath) and properties
1: node ← xmlDocument.SelectSingleNode("/", mgr) {root node of srcML file}
2: FUNCTION ITERATION ( )
3: BEGIN
4: if node.HasChildNode and node.FirstChildNode.Name == "xml" then {srcML contains several source files}
5:     if node.SelectNodes("/unit/unit", mgr).Count != 0 then {find the start node of single source file}
6:         node ← node.ChildNodes[1] {root node in this source file's AST structure}
7:         for all node1 in node.ChildNodes do
8:             node ← node1
9:             call ITERATION ( )
10:        end for
11:    end if
12: else
13:    if node != null and !node.Name.StartsWith("#") then
14:        call RECORD ( ) {Definition in Algorithm 3}
15:        for all node1 in node.ChildNodes do
16:            node ← node1
17:            call ITERATION ( )
18:        end for
19:    end if
20: end if
21: END
```

**Algorithm 3** Record Function and Invocation Properties

```
1: Class Function {funcName, retType, ParamList, xPath, isMonitored}
2: Class FuncCall {callerName, calleeName, xPath, isMonitored}
3: FUNCTION RECORD ( )
4: BEGIN
5: switch (node.name:)
6: case function:
7:     funcName ← node.ChildNodes[ "name" ].InnerText
8:     retType ← node.ChildNodes[ "type" ].InnerText
9:     paramList ← node.ChildNodes[ "parameter_list" ].InnerText
10:    xPath ← current xPath
11:    funcList. Add( new Function( funcName, retType, paramList, xPath, false ))
12: case call:
13:    calleeName ← node.ChildNodes[ "name" ].InnerText
14:    callerName = node.FindAncestor["function"].ChildNodes["name"].InnerText
15:    xPath ← current xPath
16:    funcCall = new FuncCall(callerName, calleeName, xPath, false)
17: if not funcCallList.Contains( funcCall) then
18:    funcCallList.Add( funcCall )
19: end if
20: END
```

that can be easily added to the srcML file of the original software.

### C. GENERATE SOURCE CODE WITH MONITORING CODE

This section introduces how SRMT weaves monitoring code into monitored programs. Users can choose which functions are to be monitored, and the SRMTs will insert monitoring nodes into srcML files converted from the monitored program. The insertion process entails the following steps. After that, the srcML tool can transform the srcML file's return to the executable source code.

There are six steps to insert a monitoring node:

1) Determine the monitoring types of the current monitoring points. Different types need different templates and parameters.
2) Choose the proper monitoring template from those given in Section IV-B2.
3) Copy the monitoring node in the srcML file converted from the templates.
4) Find dynamic tags in the monitoring code. (Usually refer to parameters such as `funcName`, `callerName`, and `calleeName`.)
5) Replace dynamic tags with real function names, caller names, and callee names.
6) Insert this node into the srcML file at the proper location.

Monitoring modules in other separate files also must be included in monitored software. These files must be copied into the same folder with the source code and included using the command `#include monitor.h`. SRMT will do this work automatically. More details on monitoring modules are introduced in section V.

## V. DATA-COLLECTION MECHANISMS

Another important part of this method comprises data-collection mechanisms. Two kinds of data-collection methods are proposed in this paper for two different kinds of analysis purposes (offline and online analysis). Data-collection mechanisms are shown in Figure 7.

The monitoring module supplies several monitoring functions to the monitored program. Once the instrumentation is weaved in during execution of the monitored program, these monitoring routines get called. In the example shown in Section IV-B2, there are three monitoring functions: monitorAtFunctionBegin, monitorAtFunctionEnd, and monitorBeforeFunctionCall.

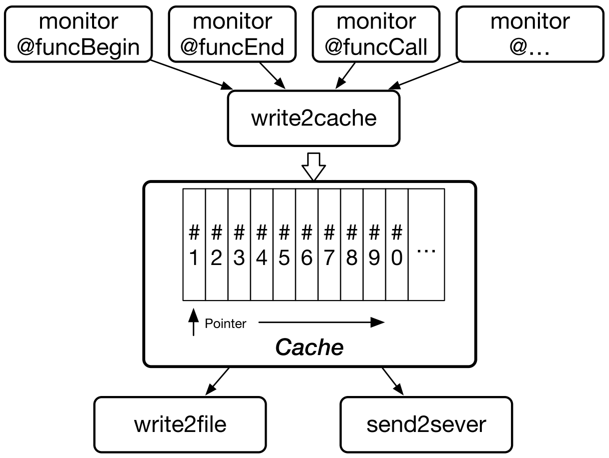


**FIGURE 7.** Data-collection mechanisms.

Each of the above three functions must obtain the time when the program executes at the current monitoring point. Two functions that are integrated in Win32 API, `QueryPerformanceFrequency` and `QueryPerformanceCounter`, are used to calculate the current time using (3). Accuracy in these methods can be microseconds ($\mu s$).

$$t = \frac{CPUPerformanceCounts}{CPU\ frequency} \times MILLION \quad (3)$$

These three functions obtain execution trace as shown in Table 1. For the sake of reducing memory usage, the SRMTs use a data structure name `union` to store this information. The union structure is shown as following.

```
struct FuncExecBegin_LLN{
char funcName [20];
Int64 start_t;
}
struct FuncExecEnd_LLN{
char funcName [20];
Int64 end_t;
}
struct FuncCallBefore_LLN{
char callerName [20];
char calleeName [20];
Int64 t;
}
union {
struct FuncExecBegin_LLN;
struct FuncExecEnd_LLN;
struct FuncCallBefore_LLN;
}
```

A submodule named `write2cache` exists in the monitoring module. `write2cache` receives data from monitor functions, sorts data by time, and writes these data to a piece of memory space allocated to variable named `cache`, which is used like a cache in the CPU. `cache` has several elements with a length of $n$, and the module writes data to `cache` according to the rule that the piece of data ordered $x$ (the $x^{th}$ elements of the data records) must be written into `cache[i]`. $i$ can be calculated using Eq. 4:

$$i = x \bmod n. \quad (4)$$

To reduce time overhead caused by monitoring operations, only data-collection parts (monitoring functions) work on the same thread with the monitored system. Operations such as storing data to the cache, writing data to the file, or sending data to the server work on other separate threads.

Each element in `cache` is controlled by the semaphore mechanism to avoid data-racing problems as Algorithm 4 shows. There is only one writing thread that can operate one element at a time, and the element must be empty at this time. The cache size $n$ is a variable parameter. For example, with the cache size $n = 100$, cache element 1 will be assigned to process the run-time data numbered $\{ 1^{st}, 101^{th}, \ldots \}$. Since the semaphore mechanism will block the monitored object, the cache size $n$ must be set to a big enough size to give the writing thread enough time to empty the specific element. The cache structure can shorten the block time brought by the recording process. As the overheads caused by semaphore mechanism will be discussed in Section VI.

**Algorithm 4** Semaphore Mechanism
```
1: int getTimes ← 0
2: currentGetTimes ← getTimes++
3: index = currentGetTimes % size_of_cache
4: waitForSingleObject (h_Semaphore_isnull[index])
5:     write data into element numbered index
6: Release (h_Semaphore_notnull[index])
```

The data stored in `cache` are processed in two ways, offline and online analysis. More details are introduced in the following sections.

## A. OFFLINE ANALYSIS

Offline analysis does not have too many limits on time consumption. Users can monitor all functions in the monitored program as desired. Offline methods need data to be written into files, so they can be analyzed at another period. The SRMTs use the *file stream* mechanism supported in C++ Standard to create a data file to store runtime execution trace.

As Figure 7 shows, there is a pointer used to over the elements in the cache. If the element is not empty, data are fetched out and this element is set empty so that `write2cache` can write new data in this element. After being fetched out from the element, the data are stored in memory and written to file periodically.

## B. ONLINE ANALYSIS

The combination methods introduced in Section II-B5 put data-collection mechanisms into the monitored program, which refer to the monitoring module that we introduce here. The data-analysis work must be done on separate computers

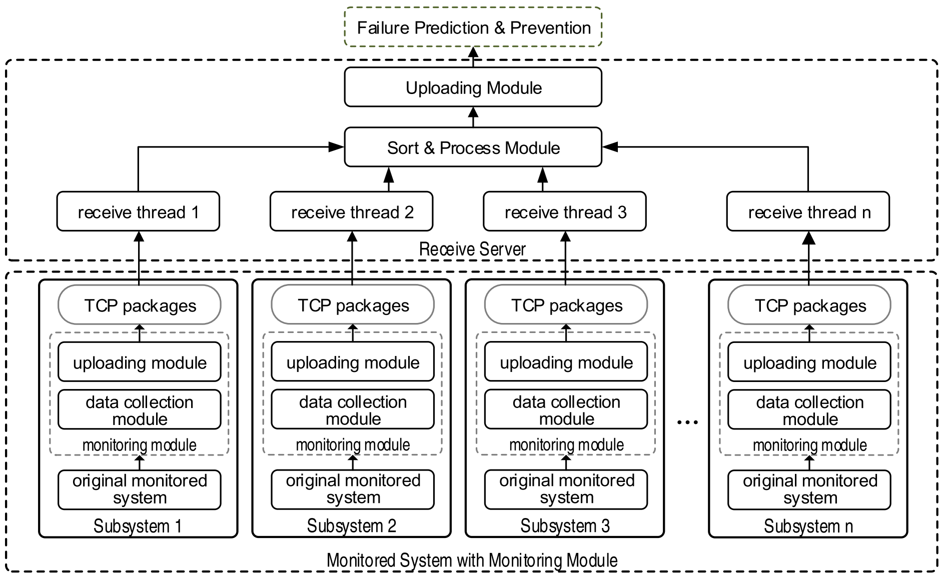


**FIGURE 8. Data-collection mechanisms for online analysis.**

at the same time as monitored program execution. Thus, the raw data that are collected from the monitored program should be uploaded to an independent data-processing system, to reduce the effects that monitoring mechanisms cause. Data-collection mechanisms for online analysis used in this paper are shown in Figure 8. The SRMTs provide users the ability to deal with multi-threaded software or distributed systems.

The monitored system contains the monitoring module described in Section V. Different from that described in Section V-A, using the file-stream mechanism, the SRMTs use an uploading module to send data to a receiving server instead of writing it to file. Instead of writing the data into files, SRMT writes the data into networking sockets. The data is sent to receiving server via different threads, as Figure 8 shows.

The receiving server consists of several threads that map the uploading modules in monitored systems one by one. The server gathers run-time data from different threads, sorts them according to the timestamp stored in each line of data, and marks each line with the type of function entrances, function exits, and function invocations. After initial processing, the receiving server can upload data to a prediction server for online prediction.

## VI. EVALUATION OF THE SRMT FRAMEWORK

### A. FUNCTIONAL COMPARISON WITH EXISTING FRAMEWORK

As reviewed in Section I, a large quantity of monitoring tools are made for JAVA programming languages. SRMT is proposed for those C/C++ Programs running on Window XP SP3 or greater operating systems.

We have made a comparison focused on monitoring type (invasive, non-invasive, or combination methods), monitoring level, monitoring goals, probe code generation methods, and supporting systems/languages among existing monitoring frameworks in Table 2.

Most existing tools are made for software runtime violation checking or validation, as claimed in [59]. SRMT, proposed in this work, are focused on monitoring software (developed using C/C++ on Windows platform) runtime execution trace information. Whicn like [56], [57] done for Java programs.

The probes are inserted with the help of srcML. Unlike Aspect-oriented Programming insert probes by byte code instrumentation [56], [57], [60], srcML make the insert process much more easy and visible. SRMT also provides a GUI interface to perform this.

SRMT also use multi-thread design to mitigate the overheads brought by the probes inserted in monitored programs. Performance evaluation is done in VI-C.

### B. EMPIRICAL EVALUATION USING "HelloWorld!"

We build a "HelloWorld!" application for performance evaluation in this section. Function `printHello` call `cout` from `<iostream>` and print "HelloWorld!" on the console. The function is called by `main` in a loop which loop times are adjustable. The frame of the experiment program is as in Figure 9.

The experiments are done on a Windows laptop (Thinkpad X1c 7th Gen), with an i5-10210U CPU and 8GB memories.

Two issues are considered in the evaluation.

- **How much overheads does the monitoring process bring to the monitored applications?**
- **Does the multi thread cache mechanism really can mitigate the overheads?**

We evaluate the performance of SRMT by add monitoring probes, measure the execution duration, compare it with the

**TABLE 2.** Theoretical comparison woth related works.

| Monitoring Types | Level | Typical Work | Monitoring Goals | Probe (Code) Generation | Supporting Object |
|---|---|---|---|---|---|
| Invasive | Function/Methods | [22] | Formal Specification | Ada Annotation-Based | Ada |
| | | [23] | Event History (Timing Properties) | Annotation-Based | C |
| | | [54] | Specication Compliance | Design by Contract Assertion-Based | Java |
| | | [55] | Contract Specification Violation | Bytecode Instrumentation | Java |
| | Subsystem | [33] | Timing Constraints | Timing Assertion-Based | C |
| Non-Invasive | Component | [20] | Network Intruder Detection | Network packet monitoring | TCP/UDP Protocols |
| | | [21] | Network Events | Broadcast LAN Ports Monitoring | Network Protocols |
| | Subsystem | [24], [25] | Read and Write Transactions on the Bus | Hardware Monitors | Java |
| Combination Methods | Function/Methods | [45] [26] | low-level information (namely values of variables or time when variables changed) | Manually Added | Java |
| | | [56], [57] | Execution Trace | Bytecode Instrumentation | Java |
| | | [58] | Runtime State machine | Aspect-oriented Languages | C |
| | | SRMT (in this paper) | Execution trace | Source code instrumentation | C/C++ |
| | Component | [59] | Parametric Properties | AspectJ Compiler | Java |
| | | [60] | Parametric Properties | Aspect-oriented Programming | C |
| | Subsystem | [61] | Safety Properties | LTL-based component generation | - |
| | | [62] | Component Trace | LTL-based component generation | Cloud Systems |

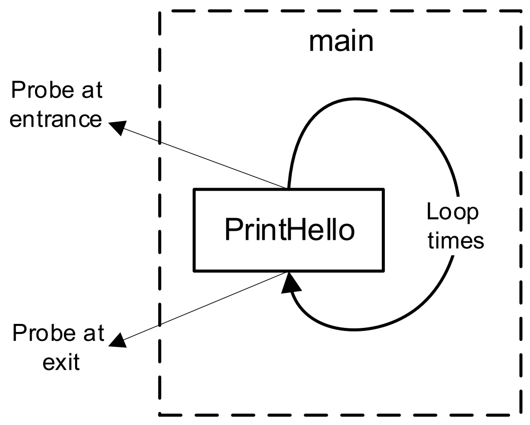


**FIGURE 9.** Frame of the "HelloWorld!" application.

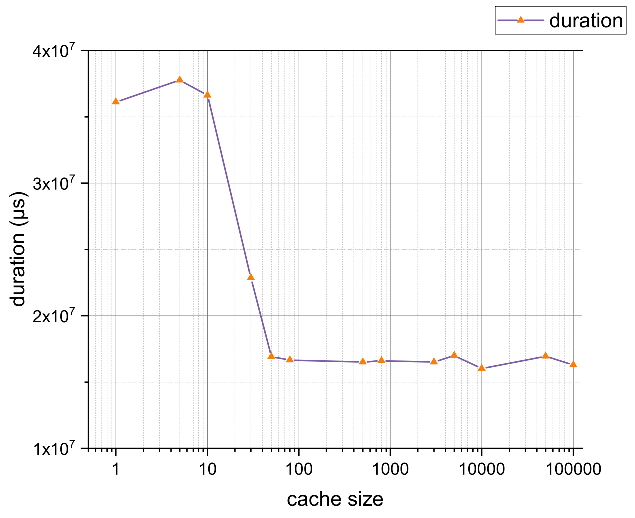


**FIGURE 10.** Duration changes with cache size of "HelloWorld!".

duration without monitoring. We set the loop times from 1 to 1,000,000. The cache size is set to 100 in this experiment. The results are as in Table 3. The duration in Table 3 with a unit $\mu$s.

The overheads rate $r$ can be calculated using Equation 5. $D$ denotes the duration of a specific execution.

$$r = \frac{D_{with\_probes} - D_{without\_monitoring}}{D_{without\_monitoring}} \quad (5)$$

We have noticed that the duration of an application may affected by many factors, such as Operating System functioning, cache misses, other programs. We use the simple "HelloWorld!" application with one loop which may mitigate the cache misses. The application is running on a clean machine just with SRMT deployed. Each time we run the executable file with three instances and take the average of three durations from different instances. We also find that more than three instances will cause a considerable delay to each applications.

A study has show that cache mechanism we proposed in SRMT can reduce the overheads to some extent. The execution duration changes with the cache size are shown in Figure 10.

## C. EMPIRICAL EVALUATION USING TERIS

We also evaluate the SRMT framework with an open source game application named "Teris". We also answer the above two problems in former evaluation study. The evaluation work is conducted from two aspects, efficiency improvements brought by multi-thread mechanisms and overheads caused by the invasive probes.

TABLE 3. Evaluation of the overheads to "HelloWorld!" applications.

| Loop times | Duration without monitoring | Duration with monitoring | Overheads rate |
|---|---|---|---|
| 1 | 2,353 | 2,822 | 0.19 |
| 10 | 4,475 | 5,156 | 0.15 |
| 100 | 95,029 | 113,561 | 0.19 |
| 1,000 | 691,606 | 758,642 | 0.10 |
| 10,000 | 16,642,097 | 18,609,864 | 0.12 |
| 100,000 | 34,468,187 | 37,404,578 | 0.09 |
| 1000,000 | 510,847,404 | 583,210,049 | 0.19 |

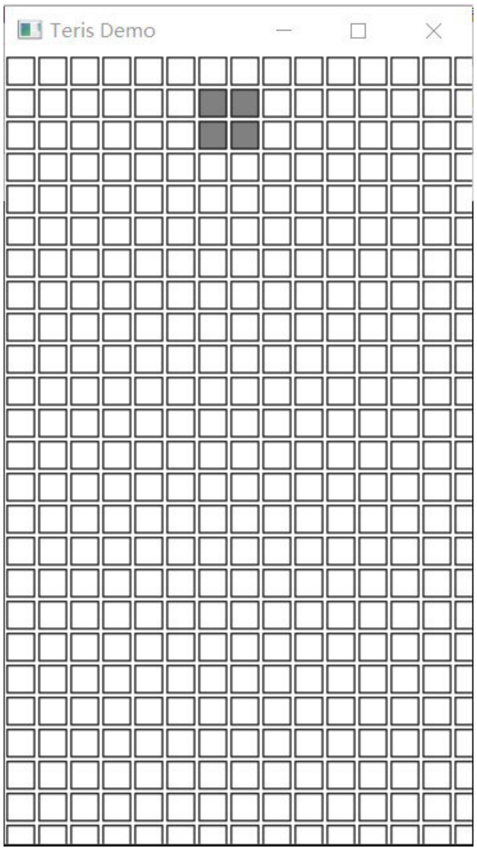


FIGURE 11. GUI of Teris.

TABLE 4. Execution Duration vs. Cache Size of Teris.

| Cache Size | Execution Duration |
|---|---|
| 1 | 33216858 |
| 10 | 32745861 |
| 100 | 32392364 |
| 1,000 | 31646319 |
| 10,000 | 31323131 |
| 100,000 | 31349664 |
| 1,000,000 | 31324309 |

### 1) EXPERIMENT SETTINGS

An open source game application named "Teris" is chosen as the experiment object. "Teris" is programmed using

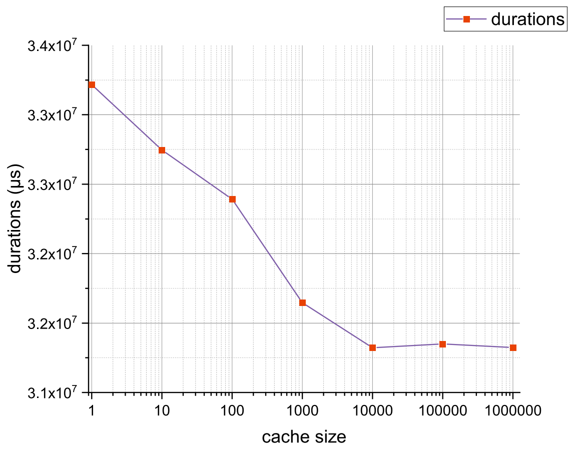


FIGURE 12. Duration changes with cache size of Teris.

C++ and has 11 functions. The GUI of "Teris" is shown in Figure 11.

To mitigate the effect of human operation time, we use the time from the first block born to the last one landed as the benchmark. The random shapes are also set to specific square block as Figure 11 shows. So there don't need any human operations during each experiment. The base time interval of panel refresh frequency is set to zero so the block can drop as fast as possible.

Three functions are related with the test most. `Export Block` will create a new block when the former one landed. `IsTouchBottom` will check if the current block touch the bottom line (landed). `refreshPanel` will refresh the panel at a specific frequency so the user can see the block dropping process. Equation 3 is also used to obtain duration in this section.

### 2) EFFICIENCY IMPROVEMENTS BROUGHT BY MULTI-THREAD MECHANISMS

As introduced in Section V, multi-thread mechanisms have been used to improve the efficiency of runtime monitoring of SRMT. Probe insertion process have been separated from data recording process using different threads, shown in Figure 7.

A key variable parameter in this mechanism is the size of the `cache`. We set cache size as different values from 1 to

**TABLE 5.** Execution Duration vs. Cache Size.

| Probes Point | None | ExportBlock | isTouchBottom | refreshPanel | All Three Functions |
|---|---|---|---|---|---|
| **duration** | 29,403,020 | 32,377,680 | 36,127,458 | 36,562,367 | 36,923,224 |
| **overheads rate** | - | 0.101 | 0.228 | 0.243 | 0.255 |

**TABLE 6.** 10 functions chosen for monitoring.

| No. | Function | CalleeFunction |
|---|---|---|
| 1 | ngx_conf_set_bitmask_slot | ngx_conf_log_error<br>ngx_strcasecmp |
| 2 | ngx_radix128tree_find | - |
| 3 | ngx_strcasestrn | ngx_strncasecmp |
| 4 | ngx_http_charset_map_block | ngx_conf_parse<br>ngx_palloc<br>ngx_pcalloc<br>ngx_strcasecmp<br>ngx_array_push<br>ngx_conf_log_error<br>ngx_http_add_charset |
| 5 | ngx_http_log_variable_getlen | ngx_http_log_escape<br>ngx_http_get_indexed_variable |
| 6 | ngx_http_proxy_rewrite_complex_handler | ngx_http_proxy_rewrite<br>ngx_rstrncmp<br>ngx_http_complex_value |
| 7 | ngx_http_upstream_zone_copy_peer | ngx_slab_free_locked<br>ngx_slab_alloc_locked<br>ngx_memcpy<br>ngx_slab_calloc_locked |
| 8 | ngx_http_script_compile | ngx_http_script_done<br>ngx_http_script_add_copy_code<br>ngx_http_script_add_args_code<br>ngx_http_script_add_var_code<br>ngx_conf_log_error<br>ngx_http_script_add_capture_code<br>ngx_http_script_init_arrays |
| 9 | ngx_http_v2_spdy_deprecated | ngx_conf_log_error |
| 10 | ngx_stream_proxy_set_ssl | ngx_ssl_crl<br>ngx_ssl_trusted_certificate<br>ngx_ssl_ciphers<br>ngx_ssl_certificate<br>ngx_log_error<br>ngx_pool_cleanup_add<br>ngx_ssl_create<br>ngx_pcalloc |

1,000,000 (decided by the limit of the max number of an array in C++).

The execution duration changes with the cache size are shown in Figure 12 and Table 4.

The results show that the cache mechanism do can improve the monitoring performance. The cache size can mitigate the recording delay but when the size is big enough, cache is not a limit to the performance.

#### 3) OVERHEADS CAUSED BY INVASIVE PROBES

We use "Teris" to evaluate the performance of SRMT. The three key functions are monitored, respectively and wholly. In this process, the cache size are set to 100.

The results are shown in Table 5. The overheads are depending on the calling frequency of the monitored functions. `exportBlock` is called when creating a new block, while the other two are called at each step dropping and each time refreshing.

## VII. CASE STUDY

*Nginx* is a HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server, originally written by Sysoev [63]. Nginx is famous for its high performance, stability, and low resource overhead, so it is widely applied in many areas. Nginx has high reliability requirements, and meets the characteristics of complex software. Nginx is also open-source software and is totally free. Thus, we chose

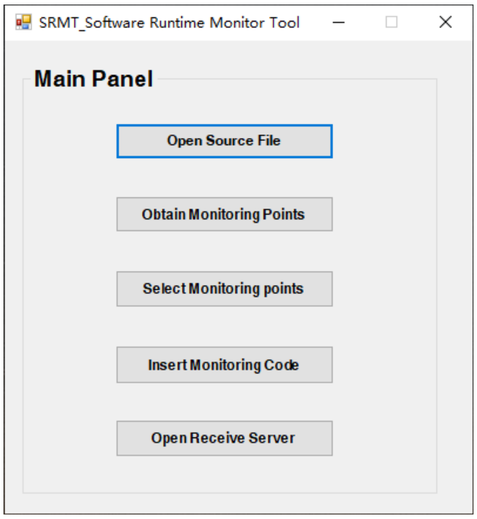
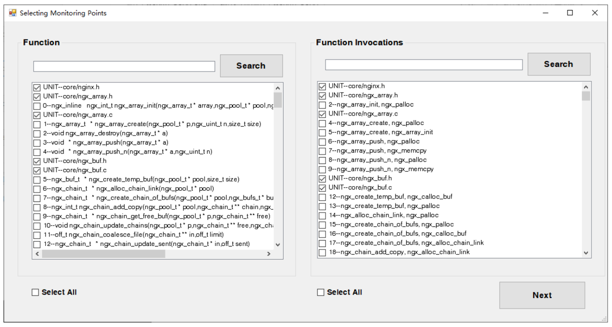


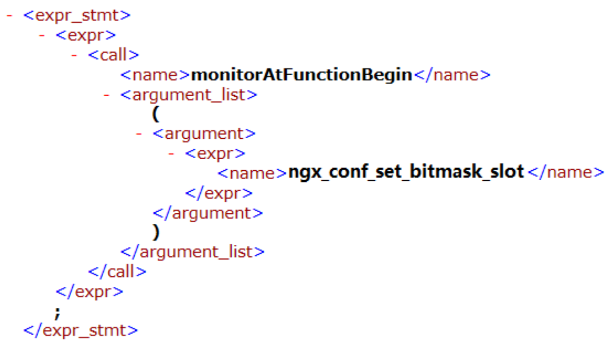
**FIGURE 13.** Main panel of SRMT.



```
Function
[✓] UNIT--core/nginx.h
[✓] UNIT--core/ngx_array.h
[ ] 0--ngx_inline   ngx_int_t ngx_array_init(ngx_array_t * array,ngx_pool_t * pool,ng
[✓] UNIT--core/ngx_array.c
[ ] 1--ngx_array_t  * ngx_array_create(ngx_pool_t * p,ngx_uint_t n,size_t size)
[ ] 2--void ngx_array_destroy(ngx_array_t * a)
[ ] 3--void  * ngx_array_push(ngx_array_t * a)
[ ] 4--void  * ngx_array_push_n(ngx_array_t * a,ngx_uint_t n)
[✓] UNIT--core/ngx_buf.h
[✓] UNIT--core/ngx_buf.c
[ ] 5--ngx_buf_t  * ngx_create_temp_buf(ngx_pool_t * pool,size_t size)
[ ] 6--ngx_chain_t  * ngx_alloc_chain_link(ngx_pool_t * pool)
[ ] 7--ngx_chain_t  * ngx_create_chain_of_bufs(ngx_pool_t * pool,ngx_bufs_t * bu
[ ] 8--ngx_int_t ngx_chain_add_copy(ngx_pool_t * pool,ngx_chain_t ** chain,ngx_
[ ] 9--ngx_chain_t  * ngx_chain_get_free_buf(ngx_pool_t * p,ngx_chain_t ** free)
[ ] 10--void ngx_chain_update_chains(ngx_pool_t * p,ngx_chain_t ** free,ngx_ch
[ ] 11--off_t ngx_chain_coalesce_file(ngx_chain_t ** in,off_t limit)
[ ] 12--ngx_chain_t  * ngx_chain_update_sent(ngx_chain_t * in,off_t sent)

Function Invocations
[✓] UNIT--core/nginx.h
[✓] UNIT--core/ngx_array.h
[ ] 2--ngx_array_init, ngx_palloc
[✓] UNIT--core/ngx_array.c
[ ] 4--ngx_array_create, ngx_palloc
[ ] 5--ngx_array_create, ngx_array_init
[ ] 6--ngx_array_push, ngx_palloc
[ ] 7--ngx_array_push, ngx_memcpy
[ ] 8--ngx_array_push_n, ngx_palloc
[ ] 9--ngx_array_push_n, ngx_memcpy
[✓] UNIT--core/ngx_buf.h
[✓] UNIT--core/ngx_buf.c
[ ] 12--ngx_create_temp_buf, ngx_calloc_buf
[ ] 13--ngx_create_temp_buf, ngx_palloc
[ ] 14--ngx_alloc_chain_link, ngx_palloc
[ ] 15--ngx_create_chain_of_bufs, ngx_palloc
[ ] 16--ngx_create_chain_of_bufs, ngx_calloc_buf
[ ] 17--ngx_create_chain_of_bufs, ngx_alloc_chain_link
[ ] 18--ngx_chain_add_copy, ngx_alloc_chain_link
```

**FIGURE 14.** SRMT monitoring-point selection GUI.



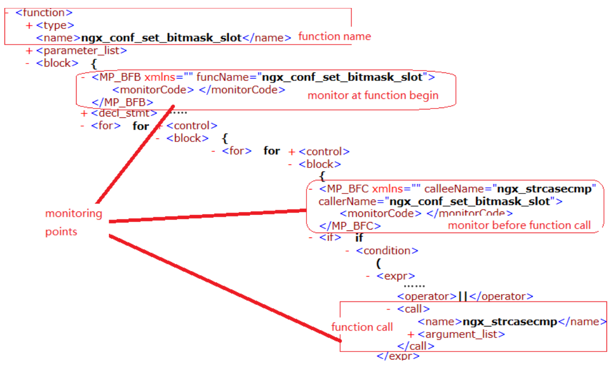
```
- <expr_stmt>
  - <expr>
    - <call>
        <name>monitorAtFunctionBegin</name>
      - <argument_list>
          (
        - <argument>
          - <expr>
              <name>ngx_conf_set_bitmask_slot</name>
            </expr>
          </argument>
          )
        </argument_list>
      </call>
    </expr>
    ;
  </expr_stmt>
```

**FIGURE 15.** Example node of monitoring function.

Nginx (version 1.12.2) as an experimental object system and used the SRMTs to monitor its execution process.

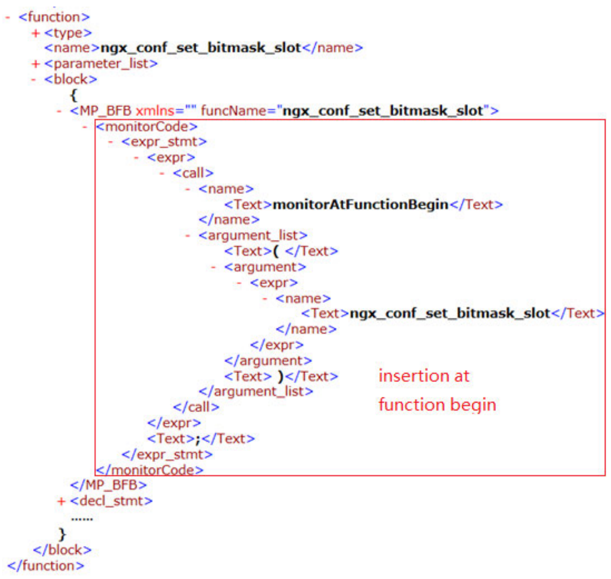
## A. OBTAIN MONITORING POINTS AND INSERT MONITORING CODE

First, we created a compressed document of Nginx source code. Using the command "`srcML src.zip -o src.xml`", the source code can be parsed and transformed to srcML format. Using SRMTs to analyze the source code in srcML format, 2490 functions and 11059 function invocations were extracted. Duplicate results were removed



```
- <function>
    + <type>
      <name>ngx_conf_set_bitmask_slot</name>
    + <parameter_list>
    - <block>  {
        - <MP_BFB xmlns="" funcName="ngx_conf_set_bitmask_slot">
              <monitorCode> </monitorCode>
          </MP_BFB>
        + <decl_stmt> ......
        - <for>  for  + <control>
                      - <block>  {
                          - <for>  for  + <control>
                                        - <block>
                                            {
                                          - <MP_BFC xmlns="" calleeName="ngx_strcasecmp"
                                            callerName="ngx_conf_set_bitmask_slot">
                                                <monitorCode> </monitorCode>
                                            </MP_BFC>
                                          - <if>  if
                                              - <condition>
                                                  (
                                                - <expr>
                                                     ......
                                                    <operator>||</operator>
                                                  - <call>
                                                      <name>ngx_strcasecmp</name>
                                                    + <argument_list>
                                                    </call>
                                                  </expr>
```

**FIGURE 16.** Monitoring points in srcML file.


insertion at
function begin

```
- <function>
  + <type>
    <name>ngx_conf_set_bitmask_slot</name>
  + <parameter_list>
  - <block>
      {
    - <MP_BFB xmlns="" funcName="ngx_conf_set_bitmask_slot">
      - <monitorCode>
        - <expr_stmt>
          - <expr>
            - <call>
              - <name>
                  <Text>monitorAtFunctionBegin</Text>
                </name>
              - <argument_list>
                  <Text>( </Text>
                - <argument>
                  - <expr>
                    - <name>
                        <Text>ngx_conf_set_bitmask_slot</Text>
                      </name>
                    </expr>
                  </argument>
                  <Text> )</Text>
                </argument_list>
              </call>
            </expr>
            <Text>;</Text>
          </expr_stmt>
        </monitorCode>
      </MP_BFB>
    + <decl_stmt>
      ......
      }
    </block>
</function>
```

**FIGURE 17.** srcML file after insertion.

in this process. The main panel of the SRMTs is shown in Figure 13.

Ten functions and 26 function invocations were chosen to be monitored in this case study, and are listed in Tables 6 and 7. The functions and invocations should be selected according to the failure mechanisms obtained from previous failure analysis work as described in [50].

the failure mechanisms defined in [50] are as in Equation 6. It means that when the duration of function A larger than 300 ms the software will failure, or when the duration of function B larger than 200 and call times between A and B larger than 10 the software will failure.

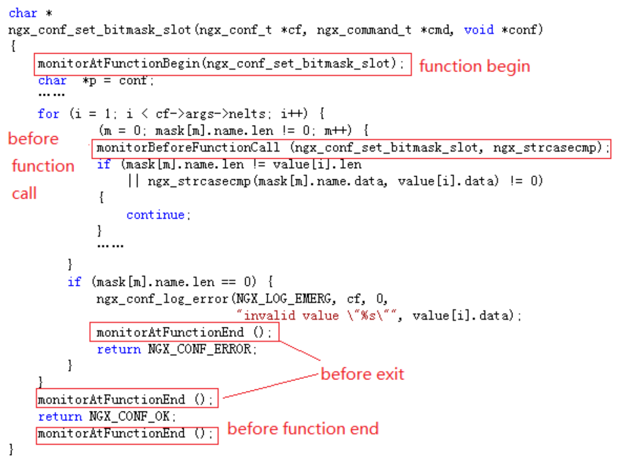
$$fm = \begin{cases} Duration_{funcA} > 300, \\ Duration_{funcB} > 200 \cap Calltimes_{A \to B} > 10. \end{cases} \quad (6)$$

These monitoring points can be chosen in the SRMTs' monitoring-point selection GUI, as Figure 14 shows.

After the monitoring-point selection process, the monitoring code is inserted. The SRMTs insert monitoring code and transform the srcML file back to source code. An example

**TABLE 7.** 26 pairs of function invocations chosen for monitoring.

| No. | CallerFunction | CalleeFunction |
|---|---|---|
| 1 | ngx_core_module_init_conf | ngx_pstrdup |
| 2 | ngx_conf_parse | ngx_conf_read_token |
| 3 | ngx_conf_set_bitmask_slot | ngx_strcasecmp |
| 4 | ngx_write_chain_to_temp_file | ngx_thread_write_chain_to_file |
| 5 | ngx_hash_find_combined | ngx_hash_find_wc_head |
| 6 | ngx_init_cycle | ngx_open_file |
| 7 | ngx_pmemalign | ngx_memalign |
| 8 | ngx_output_chain_get_buf | ngx_palloc |
| 9 | ngx_strcasestrn | ngx_strncasecmp |
| 10 | ngx_event_acceptex | ngx_atomic_fetch_add |
| 11 | ngx_ssl_stapling_update | ngx_ssl_ocsp_start |
| 12 | ngx_ssl_stapling_update | ngx_ssl_ocsp_request |
| 13 | ngx_http_chunked_create_trailers | ngx_palloc |
| 14 | ngx_http_charset_map_block | ngx_http_add_charset |
| 15 | ngx_http_charset_map_block | ngx_strcasecmp |
| 16 | ngx_http_charset_map_block | ngx_palloc |
| 17 | ngx_http_log_variable_getlen | ngx_http_get_indexed_variable |
| 18 | ngx_http_proxy_rewrite_complex_handler | ngx_rstrncmp |
| 19 | ngx_http_upstream_zone_copy_peer | ngx_memcpy |
| 20 | ngx_http_script_compile | ngx_http_script_init_arrays |
| 21 | ngx_http_script_compile | ngx_http_script_add_capture_code |
| 22 | ngx_http_script_compile | ngx_http_script_add_var_code |
| 23 | ngx_http_script_compile | ngx_http_script_add_copy_code |
| 24 | ngx_http_v2_spdy_deprecated | ngx_conf_log_error |
| 25 | ngx_stream_proxy_set_ssl | ngx_ssl_ciphers |
| 26 | ngx_stream_proxy_set_ssl | ngx_ssl_crl |



```
char *
ngx_conf_set_bitmask_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    monitorAtFunctionBegin(ngx_conf_set_bitmask_slot);
    char  *p = conf;
    ……
    for (i = 1; i < cf->args->nelts; i++) {
        (m = 0; mask[m].name.len != 0; m++) {
        monitorBeforeFunctionCall (ngx_conf_set_bitmask_slot, ngx_strcasecmp);
        if (mask[m].name.len != value[i].len
            || ngx_strcasecmp(mask[m].name.data, value[i].data) != 0)
        {
            continue;
        }
        ……
    }
    if (mask[m].name.len == 0) {
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                           "invalid value \"%s\"", value[i].data);
        monitorAtFunctionEnd ();
        return NGX_CONF_ERROR;
    }
}
monitorAtFunctionEnd ();
return NGX_CONF_OK;
monitorAtFunctionEnd ();
}
```

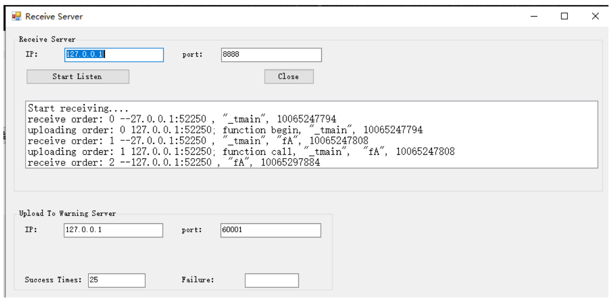
**FIGURE 18.** Source code after insertion.

node of a monitoring function waiting to be inserted is shown in Figure 15. Monitoring-point examples in srcML are shown in Figure 16.

An example of the srcML file after insertion is shown in Figure17.

### B. GENERATE SOURCE CODE

The SRMTs will automatically generate source code after inserting monitoring code using the command defined in the srcML tools. The file structure will also be reconstructed as it used to be. Two files of the monitoring module, `m_monitor.cpp` and `m_monitor.h`, will be automatically copied to the same folder with source code.

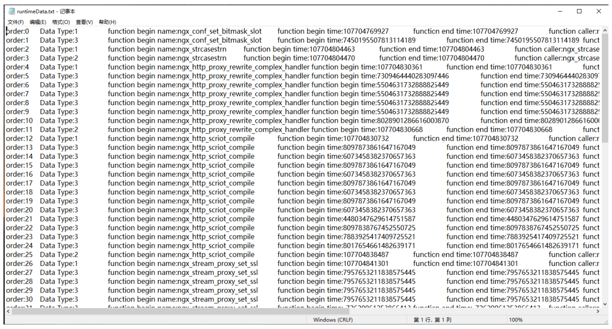


Receive Server
IP: 127.0.0.1    port: 8888

```
Start receiving....
receive order: 0 --27.0.0.1:52250 , "_tmain", 10065247794
uploading order: 0 127.0.0.1:52250; function begin, "_tmain", 10065247794
receive order: 1 --27.0.0.1:52250 , "_tmain", "fA", 10065247808
uploading order: 1 127.0.0.1:52250; function call, "_tmain", "fA", 10065247808
receive order: 2 --127.0.0.1:52250 , "fA", 10065297884
```

Upload To Warning Server
IP: 127.0.0.1    port: 60001
Success Times: 25    Failure:

**FIGURE 19.** Receiving server.



```
order:0   Data Type:1   function begin name:ngx_conf_set_bitmask_slot   function begin time:107704769927   function end time:107704769927   function caller:r
order:1   Data Type:3   function begin name:ngx_conf_set_bitmask_slot   function begin time:7450195507813114189   function end time:7450195507813114189  funct
order:2   Data Type:1   function begin name:ngx_strcasestrn   function begin time:107704804463   function end time:107704804463   function caller:ngx_strcase
order:3   Data Type:2   function begin name:ngx_strcasestrn   function begin time:107704804470   function end time:107704804470   function caller:ngx_strcase
order:4   Data Type:1   function begin name:ngx_http_proxy_rewrite_complex_handler function begin time:107704830361   function end time:107704830361   funct
order:5   Data Type:3   function begin name:ngx_http_proxy_rewrite_complex_handler function begin time:7309464440283097446   function end time:73094644402830
order:6   Data Type:3   function begin name:ngx_http_proxy_rewrite_complex_handler function begin time:5504631732888825449   function end time:55046317328888
order:7   Data Type:3   function begin name:ngx_http_proxy_rewrite_complex_handler function begin time:5504631732888825449   function end time:55046317328888
order:8   Data Type:3   function begin name:ngx_http_proxy_rewrite_complex_handler function begin time:5504631732888825449   function end time:55046317328888
order:9   Data Type:3   function begin name:ngx_http_proxy_rewrite_complex_handler function begin time:5504631732888825449   function end time:55046317328888
order:10  Data Type:3   function begin name:ngx_http_proxy_rewrite_complex_handler function begin time:8028901286616000870   function end time:80289012866160
order:11  Data Type:2   function begin name:ngx_http_proxy_rewrite_complex_handler function begin time:107704830668   function end time:107704830668   funct
order:12  Data Type:1   function begin name:ngx_http_scriot_compile   function begin time:107704830732   function end time:107704830732   function caller:r
order:13  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:8097873861647167049   function end time:8097873861647167049  funct
order:14  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:6073458382370657363   function end time:6073458382370657363  funct
order:15  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:8097873861647167049   function end time:8097873861647167049  funct
order:16  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:6073458382370657363   function end time:6073458382370657363  funct
order:17  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:8097873861647167049   function end time:8097873861647167049  funct
order:18  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:6073458382370657363   function end time:6073458382370657363  funct
order:19  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:8097873861647167049   function end time:8097873861647167049  funct
order:20  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:6073458382370657363   function end time:6073458382370657363  funct
order:21  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:4480347629614751587   function end time:4480347629614751587  funct
order:22  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:8097838767452550725   function end time:8097838767452550725  funct
order:23  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:7883925417409725521   function end time:7883925417409725521  funct
order:24  Data Type:3   function begin name:ngx_http_scriot_compile   function begin time:8017654661482639171   function end time:8017654661482639171  funct
order:25  Data Type:2   function begin name:ngx_http_scriot_compile   function begin time:107704838487   function end time:107704838487   function caller:r
order:26  Data Type:1   function begin name:ngx_stream_proxy_set_ssl   function begin time:107704841301   function end time:107704841301   function caller:r
order:27  Data Type:3   function begin name:ngx_stream_proxy_set_ssl   function begin time:7957653211838575445   function end time:7957653211838575445  funct
order:28  Data Type:3   function begin name:ngx_stream_proxy_set_ssl   function begin time:7957653211838575445   function end time:7957653211838575445  funct
order:29  Data Type:3   function begin name:ngx_stream_proxy_set_ssl   function begin time:7957653211838575445   function end time:7957653211838575445  funct
order:30  Data Type:3   function begin name:ngx_stream_proxy_set_ssl   function begin time:7957653211838575445   function end time:7957653211838575445  funct
```

**FIGURE 20.** Data file.

The source code after insertion is shown in Figure 18.

### C. COLLECT RUNTIME DATA

The receiving server collects runtime data as Figure 19 shows. An example of the data files created by the file stream is shown in Figure 20.

## VIII. CONCLUSIONS

This study focuses on software runtime monitoring, which is an important issue in research domains such as failure prediction and runtime verification. A general framework for runtime monitoring has been proposed to introduce the total process described in this work. Two key problems, instrumentation and data collection, are solved herein. We also provide a feature that allows users to choose those functions to be monitored. In this way, overhead can be reduced further. A software prototype named SRMT is built and used in a case study to show the feasibility of this work.

Since invasive data-collection and export mechanisms will inevitably cause time delay and then affect system performance, more non-invasive methods must be developed in the future. In addition, the effect on system performance also must be analyzed.

## REFERENCES

[1] R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Trans. Softw. Eng.*, vol. 19, no. 1, pp. 3–12, Jan. 1993.

[2] S. D. Johnson, "Formal methods in embedded design," *Computer*, vol. 36, no. 11, pp. 104–106, Nov. 2003.

[3] S. Kong, M. Lu, and L. Li, "Fault propagation analysis in software intensive systems: A survey," in *Proc. 2nd Int. Conf. Rel. Syst. Eng. (ICRSE)*, Jul. 2017, pp. 1–9.

[4] A. E. Goodloe and L. Pike, "Monitoring distributed real-time systems: A survey and future directions," Nat. Aeronaut. Space Admin., Langley Res. Center, Hampton, VA, USA, Tech. Rep. NASA/CR-2010-216724, 2010.

[5] D. Heffernan and C. MacNamee, "Runtime observation of functional safety properties in an automotive control network," *J. Syst. Archit.*, vol. 68, pp. 38–50, Aug. 2016.

[6] M. Vierhauser, R. Rabiser, P. Grünbacher, K. Seyerlehner, S. Wallner, and H. Zeisel, "ReMinds: A flexible runtime monitoring framework for systems of systems," *J. Syst. Softw.*, vol. 112, pp. 123–136, Feb. 2016.

[7] R. Medhat, B. Bonakdarpour, D. Kumar, and S. Fischmeister, "Runtime monitoring of cyber-physical systems under timing and memory constraints," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 4, pp. 1–29, Dec. 2015.

[8] K. Vaidyanathan and K. S. Trivedi, "A measurement-based model for estimation of resource exhaustion in operational software systems," in *Proc. 10th Int. Symp. Softw. Rel. Eng.*, 1999, pp. 84–93.

[9] G. A. Hoffmann, K. S. Trivedi, and M. Malek, "A best practice guide to resource forecasting for computing systems," *IEEE Trans. Rel.*, vol. 56, no. 4, pp. 615–628, Dec. 2007.

[10] A. Pellegrini, P. D. Sanzo, and D. R. Avresky, "A machine learning-based framework for building application failure prediction models," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, May 2015, pp. 1072–1081.

[11] N. Mahadevan, A. Dubey, and G. Karsai, "Application of software health management techniques," in *Proc. 6th Int. Symp. Softw. Eng. Adapt. Self-Managing Syst. (SEAMS)*, 2011, pp. 1–10.

[12] J. Schumann, T. Mbaya, O. Mengshoel, K. Pipatsrisawat, A. Srivastava, A. Choi, and A. Darwiche, "Software health management with Bayesian networks," *Innov. Syst. Softw. Eng.*, vol. 9, no. 4, pp. 271–292, Dec. 2013.

[13] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, May 2009.

[14] R. Rabiser, S. Guinea, M. Vierhauser, L. Baresi, and P. Grünbacher, "A comparison framework for runtime monitoring approaches," *J. Syst. Softw.*, vol. 125, pp. 309–321, Mar. 2017.

[15] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Trans. Softw. Eng.*, vol. 30, no. 12, pp. 859–872, Dec. 2004.

[16] B. Alpern and F. B. Schneider, "Verifying temporal properties without temporal logic," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 11, no. 1, pp. 147–167, Jan. 1989.

[17] Z. Han, "Research on runtime monitoring for composite Web services," M.S. thesis, Graduate School, Dept. Comput. Sci. Technol., Nat. Univ. Defense Technol., Changsha, China, 2011.

[18] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi, "A noninterference monitoring and replay mechanism for real-time software testing and debugging," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 897–916, Aug. 1990.

[19] P. S. Dodd and C. V. Ravishankar, "Monitoring and debugging distributed realtime programs," *Software, Pract. Exper.*, vol. 22, no. 10, pp. 863–877, Oct. 1992.

[20] P. V. Bro, "A system for detecting network intruders in real-time," in *Proc. 7th USENIX Secur. Symp.*, 1998. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/sec98/paxson.html

[21] K. Bhargavan and C. A. Gunter, "Requirements for a practical network event recognition language," *Electron. Notes Theor. Comput. Sci.*, vol. 70, no. 4, pp. 1–20, Dec. 2002.

[22] D. Luckham, S. Sankar, and S. Takahashi, *Two Dimensional Pinpointing: An Application of Formal Specification to Debugging Packages*. Stanford, CA, USA: Stanford Univ., 1989.

[23] S. E. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *Proc. 12th Real-Time Syst. Symp.*, 1991, pp. 74–75.

[24] K. Havelund and G. Roşu, "Synthesizing monitors for safety properties," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2002, pp. 342–356.

[25] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu, "Hardware runtime monitoring for dependable COTS-based real-time embedded systems," in *Proc. Real-Time Syst. Symp.*, Nov. 2008, pp. 481–491.

[26] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime assurance based on formal specifications," in *Proc. Int. Conf. Parallel Distrib. Process. Techn. Appl. (PDPTA)*, Jul. 1999.

[27] S. Sankar and M. Mandal, "Concurrent runtime monitoring of formally specified programs," *Computer*, vol. 26, no. 3, pp. 32–41, Mar. 1993.

[28] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *Proc. 11th Euromicro Conf. Real-Time Syst. Euromicro (RTS)*, 1999, pp. 114–122.

[29] J. J. P. Tsai, Y. Bi, S. J. H. Yang, and R. A. W. Smith, *Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis*. New York, NY, USA: Wiley, 1996.

[30] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, nos. 23–24, pp. 2435–2463, Dec. 1999.

[31] N. Grigoropoulos, S. Lalis, and M. Koutsoubelias, "An event-based framework for the specification and runtime checking of timing constraints in wireless sensor and actuator networks," in *Proc. IEEE Int. Conf. Data Sci. Data Intensive Syst.*, Dec. 2015, pp. 123–130.

[32] O. Baldellon, J.-C. Fabre, and M. Roy, "Minotor: Monitoring timing and behavioral properties for dependable distributed systems," in *Proc. IEEE 19th Pacific Rim Int. Symp. Dependable Comput.*, Dec. 2013, pp. 206–215.

[33] F. Jahanian, R. Rajkumar, and S. C. V. Raju, "Runtime monitoring of timing constraints in distributed real-time systems," *Real-time Syst.*, vol. 7, no. 3, pp. 247–273, Nov. 1994.

[34] N. Grigoropoulos, "Specification and runtime checking of timing constraints in distributed event-based applications," Ph.D. dissertation, Dept. Elect. Comput. Eng., Univ. Thessaly, Volos, Greece, 2017.

[35] A. Dunkels, O. Schmidt, N. Finne, J. Eriksson, F. Österlind, and N. T. M. Durvy. (2011). *The Contiki OS: The Operating System for The Internet of Things*. [Online]. Available: http://www.contikios.org

[36] E. Bodden, P. Lam, and L. Hendren, "Clara: A framework for partially evaluating finite-state runtime monitors ahead of time," in *Runtime Verification*. Berlin, Germany: Springer, 2010, pp. 3–197.

[37] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *Int. J. Softw. Tools Technol. Transf.*, vol. 14, no. 3, pp. 249–289, Jun. 2012.

[38] C. A. R. Hoare, "Assertions: A personal perspective," *IEEE Ann. Hist. Comput.*, vol. 25, no. 2, pp. 14–25, Apr. 2003.

[39] L. A. Clarke and D. S. Rosenblum, "A historical perspective on runtime assertion checking in software development," *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 25–37, May 2006.

[40] Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O'Farrell, and J. Waterhouse, "Runtime monitoring of Web service conversations," in *Proc. Conf. Center Adv. Stud. Collaborative Res. (CASCON)*, 2007, pp. 2–17.

[41] N. Kosmatov, G. Petiot, and J. Signoles, "An optimized memory monitoring for runtime assertion checking of C programs," in *Proc. Int. Conf. Runtime Verification*. Berlin, Germany: Springer, 2013, pp. 167–182.

[42] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C," in *Proc. Int. Conf. Softw. Eng. Formal Methods*. Springer, 2012, pp. 233–247.

[43] K. Mahbub and G. Spanoudakis, "Run-time monitoring of requirements for systems composed of Web-services: Initial implementation and evaluation experience," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jul. 2005, pp. 257–265.

[44] H. Lu and A. Forin, "The design and implementation of P2V, an architecture for zero-overhead online verification of software programs," Microsoft Res., Microsoft Corp., Redmond, WA, USA, Tech. Rep. MSR-TR-2007-99, Aug. 2007.

[45] I. Lee, H. Ben-Abdallah, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "A monitoring and checking framework for run-time correctness assurance," in *Proc. Korea-U.S. Tech. Conf. Strategic Technol.*, Vienna, VA, USA, Oct. 1998. [Online]. Available: https://repository.upenn.edu/cis_papers/296/?utm_source=repository.upenn.edu%2Fcis_papers%2F296&utm_medium=PDF&utm_campaign=PDFCoverPages

[46] O. Sokolsky, U. Sammapun, I. Lee, and J. Kim, "Run-time checking of dynamic properties," *Electron. Notes Theor. Comput. Sci.*, vol. 144, no. 4, pp. 91–108, 2006.

[47] J. Moe and D. A. Carr, "Using execution trace data to improve distributed systems," *Softw., Pract. Exper.*, vol. 32, no. 9, pp. 889–906, 2002.

[48] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting sequence diagram from execution trace of java program," in *Proc. 8th Int. Workshop Princ. Softw. Evol. (IWPSE)*, 2005, pp. 148–151.

[49] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2007, pp. 234–243.

[50] L. Li, M. Lu, and T. Gu, "Extracting interaction-related failure indicators for online detection and prediction of content failures," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Memphis, TN, USA, Oct. 2018, pp. 278–285.

[51] M. L. Collard, M. J. Decker, and J. I. Maletic, "SrcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 516–519.

[52] R. E. Bryant, O. D. Richard, and O. D. Richard, *Computer Systems: A Programmer's Perspective*, vol. 2. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.

[53] S. Kong, M. Lu, and L. Li, "Tracing error propagation in C/C++ applications," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2018, pp. 308–315.

[54] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass—Java with assertions," *Electron. Notes Theor. Comput. Sci.*, vol. 55, no. 2, pp. 103–117, 2001.

[55] M. Karaorman and P. Abercrombie, "JContractor: Introducing design-by-contract to java using reflective bytecode instrumentation," *Formal Methods Syst. Design*, vol. 27, no. 3, pp. 275–312, Nov. 2005.

[56] K. Havelund and G. Roşu, "Efficient monitoring of safety properties," *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 2, pp. 158–173, Aug. 2004. [Online]. Available: https://link-springer-com-s.vpn.buaa.edu.cn:8118/article/10.1007/s10009-003-0117-6

[57] K. Havelund and G. Roşu, "An overview of the runtime verification tool java PathExplorer," *Formal Methods Syst. Design*, vol. 24, no. 2, pp. 189–215, Mar. 2004.

[58] K. Havelund, "Runtime verification of C programs," in *Testing of Software and Communicating Systems*, K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, Eds. Berlin, Germany: Springer, 2008, pp. 7–22.

[59] D. Jin, P. O. Meredith, C. Lee, and G. Rosu, "JavaMOP: Efficient parametric runtime monitoring framework," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*. IEEE Press, Jun. 2012, pp. 1427–1430.

[60] Z. Chen, Z. Wang, Y. Zhu, H. Xi, and Z. Yang, "Parametric runtime verification of C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Germany: Springer, 2016, pp. 299–315.

[61] A. Bauer, M. Leucker, and C. Schallhart, "Model-based runtime analysis of distributed reactive systems," in *Proc. Austral. Softw. Eng. Conf. (ASWEC)*, 2006, p. 10.

[62] J. Zhou, Z. Chen, J. Wang, Z. Zheng, and W. Dong, "A runtime verification based trace-oriented monitoring framework for cloud systems," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops*, Naples, Italy, Nov. 2014, pp. 152–155.

[63] D. DeJonghe, *Compelte NGINX Cookbook*. Newton, MA, USA: O'Reilly Media, 2017.

**SHIYI KONG** received the B.S. degree from the School of Reliability and Systems Engineering, Beihang University, China, in 2015, where he is currently pursuing the Ph.D. degree. His main research interests include software reliability engineering, software failure prediction, and software health management.

**MINYAN LU** (Member, IEEE) has been a Professor and a Ph.D. Supervisor with Beihang University, China, since 2006. She is currently the Chief of the Software Reliability Unit, Key Laboratory on Reliability and Environmental Engineering Technology, School of Reliability and Systems Engineering, Beihang University. Her main research interests include software reliability engineering, software reliability testing, software safety, and software dependability.

**LUYI LI** received the B.S. degree in computer science and technology from the Beijing University of Technology, China, in 2010, and the Ph.D. degree in system engineering from Beihang University, China, in 2019. He is currently an Engineer with the China Academy of Electronics and Information Technology, China Electronics Technology Group Corporation, Beijing. His main research interests include software reliability engineering and software health management.

**LIHUA GAO** received the B.S. degree from the College of Information Technology, Hebei Normal University, China, in 2014, and the master's degree from the School of Reliability and Systems Engineering, Beihang University, China, in 2018. She is currently an Engineer with the China North Vehicle Research Institute, Beijing, China. Her research interests include software runtime monitoring techniques and management information systems.