Vol.2012-SE-176 No.3
Vol.2012-EMB-25 No.3
2012/5/21

Online Kernel Logging and Analysis for Real-time Robotics Applications

Midori Sugaya,[†1] Hiroki Takamura,[†2] Youichi Ishiwata,[*1]
Satoshi Kagami[*1] and Kimio Kuramitsu[†1]

This paper presents a technologies for detecting errors of real-time application in online, through the technique of extensible online kernel log monitoring and analysis for robotics systems. The contributions of approaches are that we present a method for kernel log analysis based on a state transition model of scheduling tasks, and apply it for kernel logs to detect anomaly behavior of real-time task. To reduce the analysis overhead of huge volumes of data, we propose a separation of overhead that monitors and analyzes the kernel logs in different cores. Based on the system, we provide an extensible framework for writing analyzers to detect errors incrementally. In our system, these components work together to solve the problems highlighted by root cause analysis in robotic systems.

We apply the proposed system to actual robotics system, and report that it could find several deviated errors and faults that include a serious priority inversion that could not detect over 10 years in the actual operating robotics system.

1. Introduction

Today's diverse and sophisticated embedded systems such as the humanoid robot are connected to networks where they collect a massive amount of information for use in providing advanced services. These services are achieved through interaction with physical environments such as human communications and access to remote databases through network connections. They are updated frequently due to the addition or removal of devices so subsequently, it is difficult to achieve dependability in these complex, networked systems [13]. These advanced robotic systems are generally supported by real-time operat ing systems that will provide precise periodic execution of real-time applications. To support accurate and precise periodic execution, it provides special APIs for delivering timer, real-time scheduler and priority lock mechanisms in a predictable manner. Moreover, lots of human interaction complex softwares works together on the system. This aspect makes it difficult to detect errors and the cause of the failure. To reduce the time of detections of errors and fault for quick recovery are expected for ensure a stable service for their users. We discussed the requirements of fault detections in humanoid robotics, and summarized as follows.

The first, to interact with humans without serious accident or improve development cycles, the error or fault detection as soon as possible. The second, detect general real-time problems such as concurrency and scheduling without disturbing the execution of these tasks. The third, there are applications which were implemented with a variety of languages. The modification of applications are likely avoided. The last, the robotics applications are easily add/remove from the system in development time to adjust the movement of robotics. To adapt these changings, the supporting system should also extensible for adapting changing environment.

To satisfy the robotics's requirements, we consider online kernel log monitoring and analysis system with extensible framework. We present the following three design purposes of our system to show how we achieve this:

- *Generality :* Log analysis is a populer technique used to find evidence of attacks and patterns of performance of the system [1]. We propose a kernel log analysis method which can find application behavior errors. The behavior of application is abstracted as transition states of a task from view point of kernel. It makes possible to make application's behavior model without modifying the applications. We compare the patterns of transition states of abstracted task model with transitions patterns written in kernel log.
- *Performance :* To detect error behavior of a process from kernel log, it needs detailed information. Usually, it increases cost of monitoring. We separate monitoring and analysis functions to reduce analysis overhead from target system. To achieve this, we present a multi-core architecture to reduce the cost of transferred logs. Compared with sending it to the other host through network, it was 40 times faster to transfere logs.
- *Extensibility :* To adapt to the changing environment, we provide extensible log analysis framework. Compared to writing probes and analysis algorithm in the applications and kernels, developer easily decides on the monitoring domain and log-parsing algorithm. Our solution is simplified by focusing on log file analysis.

In kernel level, behavior of application is abstracted as a process, then it can detect

†1 Yokohama National University
∗1 Presently with National Institute of Advanced Industrial Science and Technology
†2 Dependable Embedded OS R&D Center

error behavior without modification of application. However, in general, kernel level detailed monitoring seems takes more cost. For this point, we took cost effective tool that just generate kernel logs with constant low overhead [6], and separate analysis cost to the other host to reduce overhead in target system with proposed architecture. This mechanism is general and extensible as a user level script.

In this paper, we describe the detail of our contributions by showing actual examples of our prototype architecture. We implement a prototype system on ART-Linux, which provides a hard real-time extension on Linux kernel. We detect actual faults by using our proposed system and associated tools. Non-experienced engineer can find a serious priority inversion that could not detect over 10 years in the actual operating robotics system by using the automatic log analysis support.

The paper will be constructed as follows: In section 2, we will introduce the related works for this area. In section 3, we will describe the proposed system architecture, and in section 4, we will describe the method to analysze kernel logs, in section 5, we will present a desing and actual log volumes of the system, and show the extensible framework that analyze logs. 7 will conclude the paper.

2. Related Work

Runtime software monitoring has been used for profiling, performance analysis, software optimization as well as software fault-detection, diagnosis, and recovery [5]. There has been a lot of wok in the domain of on-line monitoring in distributed systems [3] use resource or system calls to detect performance bottleneck or overheads in some component path with low overhead. Combined with the component-based approach, and provided it's framework, these approaches are effective to specify the component or path that contains fault. In other words, it might impose to use specific languages and method of designing applications.

For detecting faults in complex real-time systems a variety of tools are presented: GRASP is an integrated tool that can trace, visualize and measure the behavior of real-time systems [8]. It provides plug-in infrastructure for the μC/OS-II real-time Operating System, however, it does not take into account the performance of the real-time system or log volume. Its method focuses mainly on detecting the performance of timing behavior such as worst/average/best execution cases. The cause of failure will not be noticed with this method. RESCH focuses on a more general operating system such as real-time

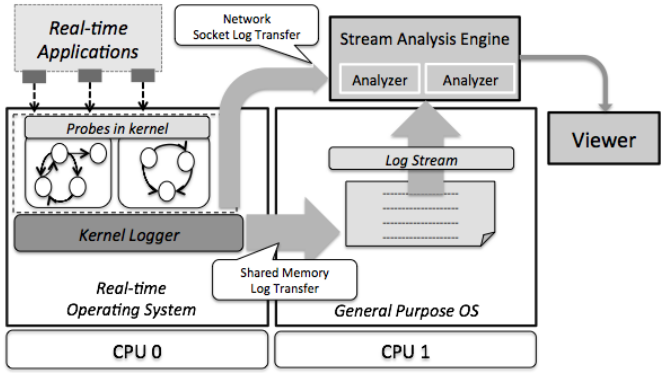

Fig. 1 Overall Architecture

Linux [2]. This tool only focuses on the real-time scheduler and debugging task.

3. Online Kernel Log Monitoring and Analysis

3.1 Requirements

Currently, humanoid robots that require high-performance embedded systems are expected to be flexible to allow for human instructions and be responsive to environmental changes. For these systems, proactive avoidance and quick reactions of failures are required since these systems have to interact with humans.

We presented in previous section in detail, there are requirements for the design. The first, to interact with humans without serious accident or improve development cycles, the error or fault detection as soon as possible. The second, detect general real-time problems such as concurrency and scheduling without disturbing the execution of these tasks. The third, there are applications which were implemented with a variety of languages. The modification of applications are likely avoided. The last, the robotics applications are easily add/remove from the system in development time to adjust the movement of robotics. To adapt these changings, the supporting system should also extensible for adapting changing environment.

Generally, failures are caused by unexpected faults where a developer forgot to write the exception handling or error code. In these cases, it is not easy to detect the cause

of the failure online because it did not produce an error message. Traditionally, experienced engineers analyze these complex failures. Offline analysis is now done using integrated development environment tools such as debugger and visualizer [7], or tracers and profilers [4, 6]. However, there are few successful online systems because most failures are not expected until they occur.

3.2 Online Kernel Monitoring and Analysis

We present online monitoring analysis system with low-overhead for considering performnance of first and second requirement. Then, the analysis data should be collected from kernel level to satisfy the requrement of third, generosity. It makes no modifications for their applications. For the last requirement, we provide these system on extensible framework. The whole system architecture which we present is illustrated in Figure1. In the Figure, you can see on the left target system, real-time applications are running on real-time operating system on a processor. In this operating system, kernel monitoring mechanism that collect logs of task schedulings and related parameters are running. These monitoring logs are transfered to the next core for the analysis. On the other processor, general-purpose operating system works to store the transfered log from monitoring core and checking whether the invariant of real-time task behaviors in kernel logs. If there are any anomaly behavior on the task, it will report the path and errors.

3.3 Low-cost Monitoring Architecture

To achieve performance by low-cost monitoring, we insert a minimum number of probes for kernel that only a few points. From these points, a kernel monitoring module collect logs. Actually, these monitoring overhead is around 5% in 1GHz average robotics backend machine, and constant without big jitters. It will be acceptable in real-time system, since real-time system dislike unpredictable overheads rather than predictable overhead for the system. The CPU consume time of analysises will not constantly consume CPU resouce because of is depending on the type of information and analysis algorithms and patterns. To reduce the jittered overhead from target real-time system, we consider the analysis part will be moved from the target CPU core to other core.

In the following sections, we firstly present kernel log analysis method, then we introduce the separation design of architecture and extensible framework. Finally, we show the experimental studies of online kernel log analysis applying to an actual robotic systems.

4. Analysis Method

To achieve generality by reducing the development cost that is needed to develop assertion for each monitoring point in application, we develop a general model that is based on the transition state of a real-time task by using the information from kernel. From the kernel, it makes possible to make application's behavior without depending on specific implementations and modifications of applications. Once we build the informarmodel of tasks in kernel, we analyze the logs that aim to determine the normal or abnormal behavior of the task in . In this section, we describe the kernel log analysis method.

4.1 Kernel Log Analysis

As we describe previously, a behavior of an application can be considered as a task which is scheduled as an abstracted entity in kernel. A state of the task will be changed according with the scheduling procedural functions which are called by kernel to switch the tasks for executions. We can consider that the task's scheduling sequences of functios can shows a abstracted behavior of an application as a task from kernel. Based on this consideration, we developed the following two models.

First, we assume to use information from kernel. From an operating system's view, and scheduling behavior of tasks are modeled as transition states. Through functions, state of tasks are changed. We define functions and transition are as the labeled transition system 4.2.1. Since this transition is finite for each task, we can apply this model for detecting an incorrect transition compared with the correct transition state. Then, we add the time element for the transition state. Based on the difference between the time of scheduling tasks, we can detect a type of faults which arouse by the delaying of scheduling 4.2.2.

Finally, we consider multiple tasks transition state. However, multiple tasks transition state itself was too complex to express the problem of competitive shared resource problems. Therefore, we omit the transition, and use only the order of tasks (priority), and shared resource information (lock), and time.

In Figure 2, we define a finite-state machine mechanism of a real-time task on an operating system. Based on the actual operating system scheduling procedure, we developed the following models. (1) State transition model of scheduling task, (2) Add a transition time of the above conditions, (3) Competitive resource of multiple tasks.

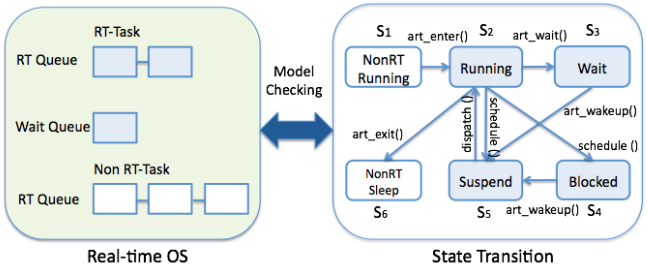

Fig. 2 Model Checking for Real-time System

4.2 Task Models

4.2.1 Transition of Single Task

In our model, a labeled transition system is a tuple $(S, L, \rightarrow)$ where S is a set of states. L is a set of labels. L is a trigger function that invokes the next state of the transition.

$$\rightarrow \subseteq S \times L \times S \tag{1}$$

is a ternary relation. In Figure 2, the right-hand picture shows the transitions. If $p, q \in S$ and $a \in L$, then $(p, a, q) \in \rightarrow$ is usually written as $p \times a \rightarrow q$. In our model, S and L are set of

- $S = \{NonRTRunning, NonRTSleep, Running, Wait, Suspend, Blocked\}$
- $L = \{art\_enter(), art\_wait(), art\_wakeup(), schedule(), dispatch(), art\_exit()\}$.

For example, if a running real-time task in state *Running* invokes function *art_wait*(), we can write it as a ternary relation $Runing \times art\_wait() \rightarrow Wait$. Our verification is to check the sequence of recorded states in log data. It is checking the sequence of transition states in our labeled transition model. If it corresponds to a valid sequence of transition states in our model, the task did not take an improper state. If it dose not correspond to a the valid sequence of transition state in our model, the task possibly took an improper state.

4.2.2 Timing Conditions

Based on the transition state of scheduling tasks, we propose a checking method with a time property. We consider that each state has time information. In such a case our labeled transition system is a quadruplet $(S, T, L, \rightarrow)$ where S is a set of states, T is a set of times, L is a set of labels and

$$\rightarrow \subseteq (S \times T) \times L \times (S \times T) \tag{2}$$

is a relation. $((p, t_1) \times a \rightarrow (q, t_2)) \in \rightarrow$ is usually written as $(p, t_1) \rightarrow_a (q, t_2)$. Using time information, we can verify the more concrete scheduling properties (in 3.5.2).

5. System Architecture

5.1 Basement System

Since our expected log volumes are so huge, it is not practical to store the logs in other hosts and use huge bandwidth of networks, especially for the embedded systems. Instead, we need to consider the overhead of the analysis because a real-time system is sensitive to the scheduling of the overhead. We consider that once the cost is predictable, it will be accepted.

Based on this idea, we propose to employ multi-OS on multi-core architecture with one of the cores exclusively assigned to the log analysis. The target real-time OS on the other core will only be allowed to consider the constant overhead of the log transfer task. Our proposed system architecture is demonstrated in Figure 1. There are several multi-core architectures available for real-time systems, both for business use and for free. QNX [15], SPUMONE [11] provides a microkernel architecture that supports both real-time and non real-time operating systems and applications. These architectures will be flexible and used without modification of the operating systems; however, message-passing overhead will not be neglected. There are other approaches that directly map operating systems on each of the cores [16]. This approach will benefit performance without the VMM layer. However, it depends on the core processor architecture. Currently, we work with ART-Linux that will support hard real-time applications, work on the general x86 processor and have facilities to use shared memory through file systems [9]. We apply ART-Linux to our online log analysis architecture for these reasons.

5.2 Extensible Log Analysis Framework

To achieve extensibility, we present stream analysis engine framework that support to write analyzer easily. Application programmer should not bother the diversity of log format and design of analyzers and it's management. This framework provide the basic facilities to analyze logs. This framework will be apply for not only kernel log, but also user level logs are easily analyzed.

Log analysis is a very popular technique; however, there are still problems. To achieve

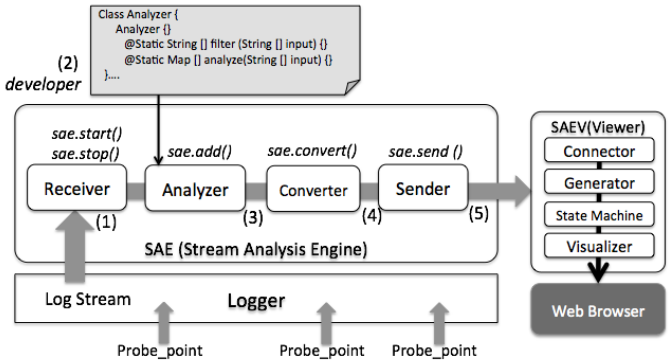

Fig. 3 Stream Analysis Engine Overview

the quick adaptation with log analysis, we consider the following problems to extend the system: One is the variety of log formats that developers need to treat during the input process. Actually, in the web area, log formats are defined by the RFC 1413, 2326 etc.

The second is lack of abstraction to write log analysis codes to improve the productivity. When a developer develops an analyzer, they need to understand what type of errors or anomalies should be detected. It depends on the target problem. Most of the codes are for detecting errors such as writing text filters and formatting the text for reporting but sometimes they need to write the network connections to send the result to the other host. We consider that these problems come from the lack of log analysis extensible framework.

5.2.1 Monitoring in kernel log

In order to comprehensively monitor the behavior of an application in a system, probes or checking codes need to be embedded to monitor its behavior. However, it obviously causes overheads for their function calls. Moreover, the cost of inserting probes in each application must not be neglected.

5.2.2 Extensible Framework

We propose Stream Analysis Engine (SAE) and its viewer (SAEV) which aim to add the log analysis code and visualize the result more clearly. To achieve this, SAE will provide support using the following three processes: One is input stream processing, the second is analyzer development, and the third is output support for the SAEV. We will develop the script interfaces and its libraries. In the following sections, we will introduce their detail. We need to develop a framework to receive logs from loggers and analyze them with an algorithm. We also have a requirement to re-write the code online to support the quick restart of the robot components. To write an analysis and allow the developer to analyze code more easily, DTrace [4] and System Tap [10] provides the C-like scripting language. The purpose of these tools is to provide a scripting language where the syntax is familiar to them. Scripting language.

For the same reason, we apply the Konoha scripting language [12]. The syntax is similar to C and Java where developers use the object oriented interfaces and methods for the analysis. Konoha provides type information for scripting and it will be safer than the other scripting languages that will be attacked without the knowledge of type information. Our team also extended the Konoha library to provide API that accesses a log stream transparently without dependency on format, methods for the analyzer, and binding the network socket interfaces for the languages to be treated transparently with the output streams.

5.2.3 Stream Analysis Engine Components

We develop SAE as the framework for developing analyzers. SAE provides the simple API for the developer who will develop or extend their analyzer such as *add*(), *start*(), *convert*(), *stop*(). SAE also provides the filters for extracting information from logs with Regex, and macros that they can personally define. The SAE consists of four components:

- *Receiver*: It treats the input data from the logs. In our framework, the SAE objects contain all of the metadata from the log stream that they read, the analyzer that it starts with, and the socket with which it connects to the viewer. SAE provides the *start*() and *stop*() mechanism that controls the reading and exit from the logs ((1) in Figure 3).
- *Analyzer*: It receives logs from *Receiver*, and applies the analyzer algorithm that was written by the developer, then it receives the result of it. The detailed procedures are as follows: Develop `filter method` that extracts required events from the log streams, `analyze method` that applies for the filtered data, and then `analyze the class` that contains these two methods ((2) in Figure 3). After creating an example of SAE, the `filter` and `analyze method` should be added to the SAE object.
- *Sender*: It sends log analysis results to other hosts such as some databases or a

Sample source code [sample_analyzer.k]

```
include "sae.k";
LOGFILE = "/path/to/sample.log";
class Analyzer {
  Analyzer() {}
  @Static String[] filter(String input) {
    /* filter method */
  }
  @Static Map[] analyze(String[] input) {
    /* analyzer method */
  }
void main(String[] args)
{
  StreamAnalysisEngine sae =
            new StreamAnalysisEngine();
  Analyzer a = new Analyzer();
  Func<String=>String[]> sample_filter =
            delegate(a, filter);
  Func<String[]=>Map[]> sample_analyzer =
            delegate(a, analyze);
  sae.add("sample", sample_filter);
  sae.add("sample", sample_analyzer);
  sae.start("sample", "LOG:" + LOGFILE);
}
```

viewer. The amount of the result would be very much smaller than the input log stream. The cost of sending the result to the other host is not high. We will show the cost of sending the result in the evaluation.

- *Converter*: It converts the result of analysis to other protocol formats in order to apply the result stream that will be decoded by the other host easily. *convert* method will encored the data according to the given protocol.

We will show the Analyzer Class in the sample code. In the code, the path to the log file is defined as LOGFILE. Then, the developer uses the *filter* method to extract the necessary data from the log stream. String type `all_lines` contains the results of the input strings with parsing the split method. The developer can use this typical method to extract the necessary data. The *analyzer* method can be applied to check the state according to the transitional state of a task. If the *analyzer* method finds a faulty condition, it will return the fault. Finally, the results and codes from the *analyzer* method are passed to SAE. Konoha provides the main function from which the entire program like C starts. In the main function, an instance of SAE is created and added to the *filter* and *analyzer* method. To define the *fileter* and *analyzer* as delegator, these instances can be called together. In the sample code, there is no method for *converter* because it assumes that you use the default protocol.

SAE will detect errors and faults. To shorten the debugging time, SAE viewer will take into account the faults and errors that are reported by the log analysis. To support the structural understanding of the problem, we simply map the result of the analyzer in a case tree. In this paper, we could not explain the detail of the tree but it was described in the paper [14]. To show the result to the viewer, we developed four components: (1) Connector, (2) Generator, (3) Viewer, and (4) State Manager. The complete architecture is shown in the Figure 3.

6. Evaluation

6.1 Detecting Faults

In our case studies, we detect fault by using this system. API missuse, task scheduling delay with feasibility study misses and over-interrupted periodic task delay. We applied each pattern checking modules for kernel logs, and detect the deviated patterns from defined pattern in previous section. In our system, if the analyzer finds an error in the system, it shows a warning. Each analyzer will judge an incorrect behavior or rule as an anomaly. An analyzer will detect the counter examples that response time variance above the threshold. Generally, if the priority inheritance is missed, then the long latencies appear in the high priority task. In this case, we can detect priority inversion. Investigating the problem with a detailed inspection, we can detect that the root cause is the wrong implementation that the low priority task invoke *art_wait()* function soon after holding the lock for the shared resource. It induced the kernel and did not inherit the higher priority from the high priority task that accessed the common resource after the low priority task kept hold of its lock. It was the specification matter and the developer that did not understand the specification.

6.2 Performance of Log Transfer

In this section, we compare the two methods that make communication possible in the OS. One is shared memory, the other is the socket. The experiment machine is MacOSX-10.5.8 CPU(2.13 GHz Intel Core 2 Duo) Memory(2 GB 1067 MHz DDR3). We set up multi-OS architecture and developed the program that writes logs to the shared memory and reads shared memory from the other operating system. We set a high-resolution time stamp count RDTSC to check the time of the application. The result is shown in Figure

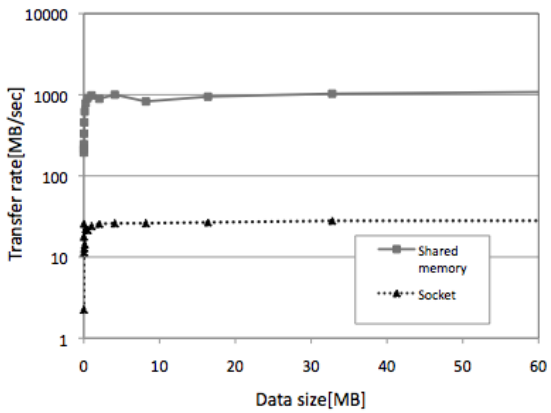

Fig. 4 Log Transfer Rate

4. Shared memory data transfer shows 1 GB per second, while socket communication transfers the data at 25 MB per second. Shared memory is implemented on RAM, the speed that the memory is written at is very high, compared to the socket that copies the buffer from user to kernel and kernel to user through the Ethernet. In section 5, we will show the experimental result of collecting logs. The generated log speed is 25 MB/sec, so, if we transfer the log to the other host, we find we should use shared memory.

We also evaluate the average cost to transfer the log. It was, on average, 7%. Compared to the average cost, which increases with the number of analyzers of around 5% per analyzer, it will keep down the cost in the target system. If we use the maximum transfer rate of shared memory, we can approximate, based on the generated log speed and time period of a task, that a 33.2*us* period is permitted for a real-time task's logging.

6.3 Peformance of Log Analysis

For the experience, we added the bug code to the robot program (1) based on the commitment model, develop four faults for both of kernel and user application (2) add the developed faults to the servo program (3) run the robot system with our logging system also works (4) the results will be stored to the evidence engine which moves the servo by sensing the result. Then, we write an analyzer by using the our framework [17] to detect bugs that will not call the API correctly. This means it will be caused by the abnormal sequence call in the transition. Then we setup the framework system and start

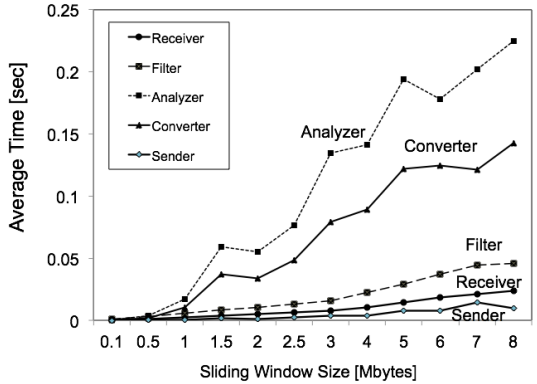

Fig. 5 Average Time of Different Sliding Window Size

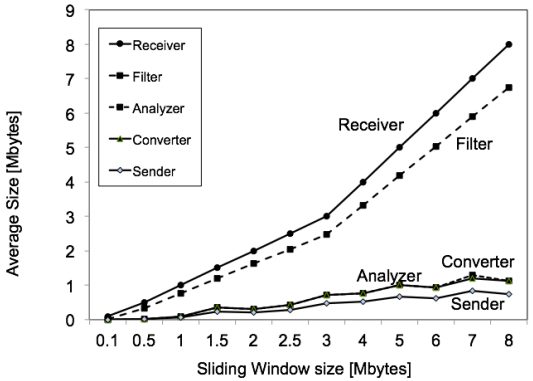

Fig. 6 Average Size of Different Sliding Window Size

the logger as well.

Figure 5 shows average processing time of each SAE components, such as *Receiver*, *Filer*, *Analyzer*, *Converter*, *Sender*. In this figure, *x*-axis indicates sliding window size (Mbytes), and *y*-axis indicates average processing processing time of transfered kernel monitoring log. Obviously, in these components, *Analyzer* takes the longest time. It consumes the time for judgement for sequential pattern of function calls and events in the log. *Analyzer* result shows that it dose not consistently increase with increasing of

sliding window size. There are variation in each size. We consider it comes from the differences of contained data type in log not comes from the volume of logs, since next Figure 6 shows the result of the processing size increasing linearly to the increasing of sliding window size. The results differ depending on contained data type of log that lots of pattern matches patterns or not.

Similer variation is seen in *Converter* that it's processing time depends on the result of *Analyzer*. *Sender* and *Receiver* time are small in this result. Both values occupy 7% in the SAE. Figure 6 shows average processing size (Mbytes) for different window size. *x*-axis indicates sliding window size (MByte), and *y*-axis indicates average log processing size (Mbytes) in each SAE compoonent.

Compared to the average time, which we showed as Figure 5, the average processing size is linearly increased along with the sliding window size. It means that the size of the sliding window did not affect the analysis. On the other hand, there are a differences in 7 Mbyte between the result of *Receiver* and *Sender* at most. It means that *Filter* could reduced the data. It also means filter can reduce the volume of data by implementing suitable filters.

7. Conclusion

In this paper, we proposed online kernel monitoring and log analysis method and system. The aim of our prototype system is to satisfy the requirements of the advanced embedded system. The contributions of our approach are the following: Develop a log analysis method based on model checking for transition modeling of real-time tasks. Develop an online logging and analysis system where the monitored application and systems are analyzed concurrently without disturbing the real-time execution of monitored applications. The result of the analysis will be sent to map the fault tree, where the engineer will execute the root cause analysis with fault tree viewer. It provides the procedure for logically structuring of the problem. In the future, we will evaluate our system to show its effectiveness in finding of the root causes of problems. Moreover, we will try to develop a more general technique to define and analyze problems.

Acknowledgment

This works was supported by the following projects: JST-CREST: Dependable Operating Systems for Embedded Systems at Aiming at Practical Application Project.

References

1) J.H. Andrews and Y.Zhang. General test result checking with log file analysis. *IEEE Trans. Softw. Eng.*, 29:634–648, July 2003.
2) M.Asberg, J.Kraft, T.Nolte, and S.Kato. A loadable task execution recorder for linux. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 31–36, 2010.
3) P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: real-time. modelling and performance-aware systems. In *9th. Workshop on Hot Topics in Operating Systems (HotOS IX)*, New York, NY, USA, 2003.
4) B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.
5) N.Delgado, A.Q. Gates, and S.Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30:859–872, December 2004.
6) M.Desnoyers and M.R. Dagenais. The lttng tracer: a low impact performance and behavior monitor for gnu/linux. In *Proceedings of the Linux Symposium*, volume 1, pages 209–224, 2006.
7) Eclips. http://www.eclipse.org/.
8) M. Holenderski, M. M.H.P, van den Heuvel, R. J. Bril, and J. J.Lukkien. Grasp: Tracing, visualizeing and measuring the behavior of real-time systems. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 37–42, 2010.
9) Y.Ishiwata, S.Kagami, K.Nishiwaki, and T.Matsui. Art-linux 2.6 for single cpu: Design and implementation. *Journal of Robotics Society of Japan*, 26(6):77–84, 9 2008.
10) B.Jacob, P.Larson, B.H. Leitao, and S.A. M.M. daSilva. Systemtap: Instrumenting the linux kernel for analyzing perfomance and functional problems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, REDP-4469-00, IBM, 2009. International Technical Support Organization.
11) W.Kanda, Y.Yumura, Y.Kinebuchi, K.Makijima, and T.Nakajima. Spumone: Lightweight cpu virtualization layer for embedded systems. In *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on*, volume 1, pages 144–151, Dec. 2008.
12) K.Kuramitsu. Konoha - implementing a static scripting language with dynamic behaviors. In *Workshop on Self-sustaining Systems (S3) ACM*. ACM Press, 2010.
13) J.-C. Laprie and B.Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.
14) Y.Matsuno, J.Nakazawa, M.Takeyama, M.Sugaya, and Y.Ishikawa. Toward a language for communication among stakeholders. In *Proc. of the 16th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'10)*, pages 93–100, 2010.
15) QNX. http://www.qnx.com/.
16) N.Sugai, H.Kondo, and S.Ochiai. A software platform for multiple os extension on embedded chip-multiprocessor. In *IPSJ SIG Technical Report*, volume3, 2007.
17) M. Sugaya, K. Igarashi, M. Goshima, S. Nakata, and K. Kuramitsu. Extensible online log analysis system. In *Proceedings of the 13th European Workshop on Dependable Computing*, EWDC '11, pages 79–84, New York, NY, USA, 2011. ACM.