Journal of Information Processing Vol.21 No.1 53–66 (Jan. 2013)

[DOI: 10.2197/ipsjjip.21.53]

Regular Paper

Online Kernel Log Analysis for Robotics Application

Midori Sugaya[1,a)] Hiroki Takamura[2,b)] Yoichi Ishiwata[3,c)]
Satoshi Kagami[3,d)] Kimio Kuramitsu[1,e)]

Received: March 5, 2012, Accepted: September 10, 2012

Abstract: Humanoid robot systems are composed of an assortment of hardware and software components, and they have complex embedded systems and real-time properties. These features make it difficult to isolate or to identify a fault in a short period of time even though such systems are expected to recover quickly in order to avoid any harmful behaviors that may cause harm to the users. This paper presents a new technological method for detecting errors in real-time applications online through the technique of online kernel log monitoring and analysis method. 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 to the kernel logs to detect anomaly behavior of real-time tasks. In order to reduce the analysis overhead of huge volumes of data, we propose a new system that places the kernel log analysis engine on a separate core from the one that runs the kernel log monitoring process. Based on this system, we provide a 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 applied the proposed system to actual robotics systems and successfully detected several deviated errors and faults that include a serious priority inversion that was not detected in over 10 years of operation in the actual operating system.

Keywords: online log analysis, multi-core system, dependability, robotics application, diagnosis

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 provided 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 or components. Subsequently, it is difficult to achieve dependability in these complex, networked systems [17], [27], [31]. These advanced robotic systems are generally supported by real-time operating systems that provide precise periodic execution of real-time applications. To support accurate and precise periodic execution, they provide special APIs for delivering timer, real-time scheduler and priority lock mechanisms in a predictable manner. Moreover, lots of complex human interaction software works in conjunction on the system. This makes it difficult to detect errors and their causes. Detecting errors and faults for quick system recovery is important for providing a stable robotics service for users. The requirements for fault detection in humanoid robotics are discussed, and summarized as follows.

Firstly, our system must be support its interaction with humans so that any faults or errors should be quickly detected and therefore avoiding any dangerous threats to the user. Secondly, our system should detect faults in real-time systems without interfering with the execution of these tasks. Thirdly, any modifications to the applications are likely avoided by our system, since robotics applications can be implemented with a variety of languages. And finally, the robotics applications can be easily added/removed from the system during development to adjust the movement of robotics. To implement these changes, the system should also be supportable in that it needs to be able to adapt to a changing environment.

There are lots of existent approaches that focus on detecting problems such as timing and concurrency, in complex real-time systems. There are also a variety of related techniques for uncovering software faults using static program analysis and providing runtime detections [21]. In these approaches, static approaches [11], such as source code analysis will not satisfy the requirement of detecting timing problems and we could not predict all of the feasible paths statistically. Compiler-based approaches [24] are strong at detecting faults, however, they have a high cost when applied in runtime and depend on the application implemented languages. Runtime verifications check whether the program violates a programmer-specified safety property [26]. They generally focus on creating fault and error models to detect the targets. In these approaches, developers generally write assertions in programs for detecting runtime errors [9]. However, the programmer-written assertions are not very effective in providing

[1] Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[2] Japan Science and Technology Agency, Dependable Embedded OS R&D Center, Tsukuba, Ibaraki 305–8568, Japan
[3] National Institute of Advanced Industrial Science and Technology, Koto, Tokyo 135–0064, Japan
[a)] doly@ynu.ac.jp
[b)] takamura@dependable-os.net
[c)] you1@ni.aist.go.jp
[d)] s.kagami@aist.go.jp
[e)] kimio@ynu.ac.jp

high coverage of runtime errors [25]. Pattabiraman et al. present an automated procedure runtime error detection mechanism [22] outside the applications to avoid writing assertions. This idea is effective since our system also believe that the detection mechanism should be outside of application.

Based on this idea, and also satisfy the robotics' requirements that we presented previous paragraph, we will propose a online kernel log analysis system.

The purpose of the presented system is both to support debugging for application developers, and for the actual operation of the robotic system. To achieve this purpose, we consider using kernel log as our general approach, and apply it in online kernel log analysis framework. Online support allows for faster detection of faults, while kernel log supports no modification of application. Our current architecture and tools support log analysis without disturb real-time processing. At the kernel level, behavior of applications is abstracted as a process, and then it can detect error behavior without modification of application. However, in general, kernel level detailed monitoring costs more. Therefore, we applied a cost effective tool that generates kernel logs with a constant low overhead [6], and separate the analysis cost to reduce overhead in the target system with proposed architecture.

In this paper, we describe the details of our contributions by showing actual examples of our prototype architecture. We successfully implemented a prototype system on ART-Linux, which provides a hard real-time extension on Linux kernel and multicore support. By using our proposed system, a non-experienced engineer was able to find a serious priority inversion's fault that was not detected for over 10 years in the actual operating robotics system by using our proposed system and associated tools.

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 analyze kernel logs, in Section 5, we will present a design and actual log volumes of the system, in Section 6, we will show the log analysis framework that analyzes kernel logs. In Section 7, we will show experimental studies based on actual cases using simple robotic systems. Section 8 will show our evaluation, then Section 9 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 work in the domain of on-line monitoring in distributed systems [3], [7] used resource parameters, system calls to detect performance bottleneck or overheads in some component 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 a fault. These approaches are effective, however, it might impose to use specific languages and methods that coincide with the designing of the applications.

To detect more specific software faults, such as concurrent bug and deadlocks, in general and concurrent programs, there are various approaches used [8], [14]. In distributed domain, both of static analysis [19] and runtime analysis [20] are presented. Runtime analysis [20] approach, generally, low level debugging information is needed for detecting faults. To reduce the runtime overhead, Ref. [20]'s Java-based tools that automate to collect debugging logs from each server and send it to debugger servers. This makes it possible to reduce the debugging cost using low-bandwidth line. Although our approach may be similar to this approach, the differences are that our focus is placed on real-time software.

In real-time systems, statically scheduled systems [32] are generally considered. In it, task synchronization was statically decided before the program runs that task parameters and constraints. If these parameters and constraints are known previously, the scheduling is guaranteed to satisfy safety and liveness properties. However, these days, real-time systems are constructed by general-purpose operating systems that cannot give these parameters ahead of time. These real-time systems need to treat thread priorities in a more flexible manner by using protocols like priority inheritance [28], used to prevent priority inversion. These threads work for a variety of purposes and it is difficult to predict their lifetime beforehand.

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 [10]. 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. It focuses mainly on detecting the performance of timing behavior such as worst/average/best execution cases. The cause of a failure will not be detected with this method. RESCH focuses on a more general operating system such as real-time Linux [2], it focuses on the real-time scheduler and it's scheduling method. Our system can treat more general errors and faults that can be detected from logs. No other approach that provides whole framework for online log analysis with satisfying performance and generosity.

A variety of software testing and simulation methods are presented and developed for improving the quality of software. These methods are generally used to detect faults before the systems or products are shipped out, because their approaches produce an extra load on the system that will be a burden during the operational phase. Dynamic program analysis also provides functions to check the runtime behavior of software in the developmental phase. However, the method is different from the testing and simulations in that they need to consider how to minimize the effect that instrumentation has on the execution of the target program because they need to consider the operational condition of the target program that will be running in the operational phase. Our approach will be the kind of dynamic program analysis.

All three of these approaches are generally used in the development phase. However, as a dynamic program analysis, our approaches focus more on using in the operational phase, and have a purpose not to modify source code of applications. As described in the introduction, the robotics applications are developed by a variety of programing languages. To check the real-time attributes of application, we did not select the approach that insert debugging code for application, but used a kernel logging that is effective for detecting faults from behavior's of applications.

As a dynamic program analysis, we need also consider to reduce the runtime cost, we presented the way in this paper in following sections.

3. Online Kernel Log Monitoring and Analysis

3.1 Proposal

Nowadays, 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 to failures are required since these systems have to interact with humans.

As discussed in the previous section, there are four requirements for the design of fault detection systems. To satisfy the requirements, we present online analysis system for reducing time for detecting faults. To achieve online analysis, we propose an example architecture of log-transferred system. The information of application is collected from kernel for generality, then analyzed in online. Online analysis can reduce the total logs that contains the unnecessary information because it can analysis in online and store only the results, and time to write data to storage. We provide these systems on a log analysis framework.

3.2 Design Purpose

Based on the previously mentioned idea, we developed online analysis system with the considerations based on the following three design purposes:

- *Generality:* Log analysis is a popular 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 applications is abstracted as transition states of a task from the viewpoint of the kernel. It is then possible to make generic application's behavior model without modifying the application. We compare the patterns of transition states of abstracted task model with transitions patterns recorded in the kernel log.
- *Performance:* To detect erroneous behavior of a process from the kernel log, detailed information is required. Usually, this increases the cost of monitoring. We separate monitoring and analysis functions in order to reduce analysis overhead from target. 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 a network, it was 40 times faster to transfer logs.
- *Log Analysis Support:* To adapt to the changing environment, we provide a log analysis support framework that can easily add analysis script as a user mode program. We support various log formats and libraries to reduce the overhead of developers. Our solution is simplified by focusing on log file analysis.

The complete architecture which we present is illustrated in **Fig. 1**. We assume multi-core system because of humanoid robot needs powerful machine in limited space. Moreover, they require both of real-time and non real-time application that works hand in hand to provides services. These applications sometimes needs close cooperation, so that the multi-operating system on multi-core is expected to provide efficient environment for their applications. In the figure, you can see on the left target system, real-time applications are running on real-time operating system on a core. In this operating system, kernel-monitoring mechanisms that collect logs of task scheduling and related parameters are running. These monitoring logs are then transferred to a separate core for the analysis process. On the other processor, a general-purpose operating system works to store the transferred log that was passed in from the monitoring core and checks the invariant of real-time task behaviors in kernel logs. If there are any anomaly behaviors on the task, it will report the path and errors.

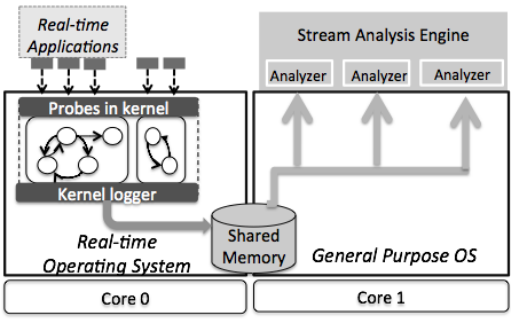


Fig. 1 Complete architecture.

In the following sections, we first present a kernel log analysis method and then we introduce the separation design architecture and log analysis support framework. Finally, we show the experimental studies of online kernel log analysis applied to actual robotic systems.

4. Analysis Method

To achieve the first design purpose, we develop a general model that is based on the transition state of a real-time task by using the information from the kernel. Once we build a model of tasks in the kernel, we analyze the logs so as to determine the normal or abnormal behavior of the task. In this section, we describe the kernel log analysis method.

4.1 Kernel Log Analysis

As we described previously, a behavior of an application can be considered a task, which is scheduled as an abstracted entity in the kernel. The state of the task will be changed according to the scheduling procedure functions that are called by kernel to switch the tasks for execution. We can consider that the task's scheduling sequences for functions can show an abstracted behavior of an application. Based on this, we developed the following two models.

First, we assume to use information of kernel, which is an operating system's view, and the behavior of tasks is modeled as transition states. Through functions, the state of tasks is changed. We define functions and transitions as the labeled transition system in Section 4.2.1. As 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 arose by the delaying of scheduling in Section 4.2.2.

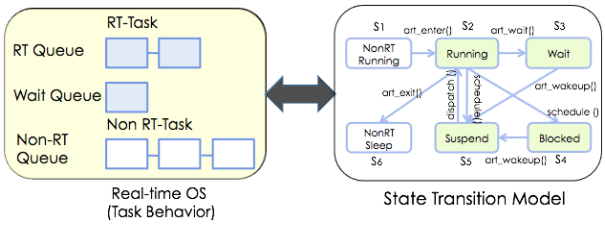


Fig. 2 Transition model for real-time system.

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

In **Fig. 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 a scheduling task, (2) Add a transition time to the above conditions, (3) Competitive resource for multiple tasks.

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 Fig. 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 $Running \times art\_wait() \rightarrow Wait$.

Our verification is to check the sequence of recorded states in the log data. The sequence of transition states in our labeled transition model is checked. If it corresponds to a valid sequence of transition states, the task did not take an improper state. If it dose not correspond to a valid sequence of transition states, 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.

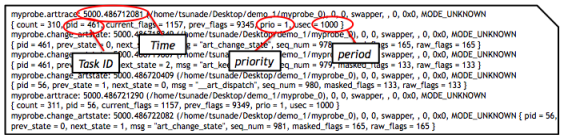


Fig. 3 Example of generated log.

5. System Architecture

5.1 Example Log Volume

To develop a log analysis system, first we need to consider how many logs are needed in order to find errors. To understand the volume required, we collected an actual log by assuming that a task will miss its deadline due to an API misuse. We used a tracer named LTTng (Linux Next Generation Trace Tool Kit) [6]. The code is inserted in to key places in the kernel. One is placed before *context_switch* of process, the second is inserted in holding *spin_lock* for avoiding resource contention, the last is in point *wake_up* in the kernel. We apply the patch to real-time OS, and set new probes according to the information received to specify the problem, such as the {`recording time, function, state (prev/next), id, priority`} from the kernel log insertion points in **Fig. 3**. We then developed a fault program and ran it with the collected logs from the kernel.

We collected around 25 MB of logs per second, which equates to around 245 bytes per entry. This means that about 150 MB per minute, 9 GB per hour. If we find more general problems in real-time tasks, priority inversion and context switch misses, we need to consider how to store and analyze these logs.

5.2 Prediction of Log Transfer

We collect logs from the application running on the target core and transfer it to the specified core that is exclusively assigned to log analysis by the log analysis method. With this architecture, we need to consider the log volumes that need to be transferred to the other core. If the output rate of logs exceeds the transfer rate of logs, output logs might be truncated by overwrites. We have developed a formula to calculate the log-generation rate, and apply it to the actual log generating measurement. We then compare the calculated result with the actual log measurement.

Generally, a real-time task will work periodically. We thought that if we set probes in a real-time task's periodic execution, we can estimate the log volumes which they will generate in the *while loop*. Using this idea we set up the prediction model for the log generation rate.

We assumed a fixed priority scheduling of periodic tasks, according to the periodic task model. A task is presumed to work in periodic execution in the *while loop*. In the periodic execution, the ith probe recorded logs. We define the amount of data [*bytes*] that is recorded per probe as d_i. It will be different for some probes, so we define the $\pm\Delta d_i$ for an additional parameter. To calculate the total number of logs within a periodic execution, we define c_i, which is the number of probes that are inserted in a periodic execution. We think that the actual number depends on the probability p_i that the named probe set is totally dependent on their behavior of the task in the periodic execution. We define the total number of the probes as n. The total number of gener-

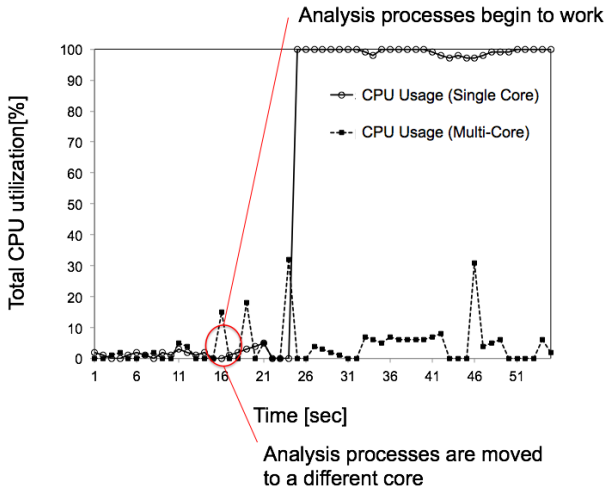


Fig. 4 Comparison of CPU utilizations.

ated logs is calculated using the formula $\sum_{i=1}^{n} p_i \cdot c_i \cdot (d_i \pm \Delta d_i)$. By applying the formula, we can calculate the total number of generated logs per periodic execution of a task. We assume that real-time tasks are worked more than once in this system, so we define the total number of real-time tasks as m, each of them are presumed to work in periodic execution. We define the period t_i as representing the periodic time in which the task is released for execution. We can calculate the rate of the generated log using the following equation (3).

$$\sum_{i=1}^{m} \sum_{j=1}^{n} (p_{ij} \cdot c_{ij}) \cdot (d_{ij} \pm \Delta d_{ij})/t_i \,[\text{Byte/sec}] \tag{3}$$

To understand the effectiveness of this formula, we calculated it with the actual measured parameters. There are seven problems in this case (*change_state*, *func_trace*, *syscal_exit*, *syscall_enter*, *mm.page_free*, *mm.add_to_page_cache*, *mm.remove_from_page_cache*).
The probability will be set to one because we want to know the worst-case volume of logs. The period of all the tasks are 1 *ms*. Then, we calculate the results using the formula as follows. $62.4*241/1+19.8*208/1+19*131/1+19*170/1+11.3*136/1+0.6*146/1+0.6*152/1 = 26.59$ [MB/sec]. The result was very close to the actual measured parameter (25 MB/sec), therefore we can confidently use the formula to judge whether or not it will run within the transmission capacity of the system.

5.3 Low-cost Monitoring Architecture

To achieve performance by low-cost monitoring, we insert a minimum number of probes into kernel at only a few points. From these points, a kernel monitoring module collect logs. This monitoring overhead is around 5% in an average 1 GHz robotics backend machine, and constant without big jitters. It will be acceptable in real-time systems, since real-time systems generally dislike unpredictable overheads. The CPU consume time of analyses will not constantly consume CPU resources because it depends on the type of information and analysis algorithms and patterns. To reduce the jittered overhead from target real-time system, we assume the analysis part will be moved from the target CPU core to other core.

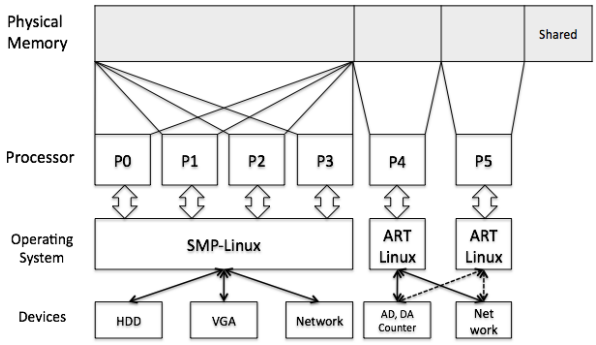


Fig. 5 ART-Linux for multi-core architecture.

In **Fig. 4**, the simple evaluation that compares the CPU utilization in the analysis between using single core and multi-core architecture. When the analysis component began to analyze kernel logs, the CPU utilization began to increase. In the case of a single-core, the analysis cost directly affected the target CPU, while in case of multi-core, the analysis costs were removed from the target CPU. We placed the lowest priority task to take care of kernel monitoring logs and send it to the next core through the shared memories. Actually, our system can send data to the other host through the network, however, it is 40 times slower than using shared memory. Shown in Section 8.3. Based on this evaluation, we propose this system architecture that will be constructed with multi-core or distributed architecture for that purpose.

5.4 Multi-OS on Multi-Core Architecture Basement System

Since our expected log volumes are so huge, it is not practical to store the logs in other hosts and use huge amounts of network bandwidth, 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 a multi-core architecture with one of the cores assigned exclusively to the log analysis using ART-Linux with Asymmetric multi-processing (AMP). The user can install a different operating system on each processor. To achieve the hard real-time performance, ART-Linux will apply real-time scheduling of the local scheduler; this means each core has a different operating system. We install ART-Linux on the Bootstrap Processor (BSP) as the first operating system to boot up, and install Linux as a general-purpose operating system on the Application Processor (AP).

These architectures not only support the real-time task's precise periodic execution, but also separate the overhead of log analysis from the level of hardware. To facilitate the implementation of a multi-core based operating system, we chose AMP. This means that, with few exceptions, each kernel will not share the physical memory, processor or devices. Our proposed architecture is shown in **Fig. 5**. Physical memory is divided, without overlapping areas, into different operating systems. Part of the physical memory is used to share memory between the operating systems. As with physical memory, processors are assigned to each kernel. These assignments are done in a static manner; it will not be changed during execution time to achieve basement

dependability. However, part of the device assignment can be changed dynamically during execution time because the kernels will not access the same device concurrently.

The reasons for the static assignment of multi-core based multi-operating systems are as follows: Firstly, we need to consider the exact hard real-time performance that is required in robotics systems. Because robotics systems are based on the machine controlling systems for each part, to control these parts with a realistic speed, at least microsecond control resolution is required for stopping and starting the arm and wheels correctly.

Secondly, in recent advanced robotics systems, the demand for high-performance computing has increased. Calculations such as image and sound analysis require high processor speed and band bandwidth for the system. Machine leaning and intelligent control also strengthen its requirements for the high-performance computing.

Thirdly, safety and accountability requirements in this area are increasing these days. When an accident happens, the system should store some evidence to explain the problem to their customers. To achieve this, we need to assign a core to store the log and evidence in the system. To meet the demand of these requirements, we present the AMP type of multi-core operating system and log analysis architecture.

The target real-time OS will only be allowed to consider the constant overhead of the log transfer task. Our proposed system architecture is demonstrated in Fig. 1. There are several multi-core architectures available for real-time systems, both for business use and for free. QNX [23], SPUMONE [15] 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, the message-passing overhead will not be neglected. There are other approaches that directly map operating systems on each of the cores [29]. 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 [12]. We apply ART-Linux to our online log analysis architecture for these reasons.

6. Log Analysis Support

6.1 Problems Log Analysis

To support log analysis, we present a stream analysis engine framework that support to analyze a log easily. The application programmer should not bother the diversity of log format and design of analyzers and it's management. This framework provides the basic facilities to analyze logs. This framework is applied for not only the kernel log, but also user level logs are easily analyzed.

As we described, we used a log analysis technique for automatic error or fault detection. However, we need to consider several problems in order to achieve quick adaptation to the faults. Log analysis is a very popular technique; however, there are still problems. To achieve quick adaptation with log analysis, the following problems must be considered: 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. Log management is needed for dealing with large volumes of computer-generated log messages for a standardized text format. However, these logs need to be integrated with the unified format. The second is lack of supporting libraries for improving 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 support framework.

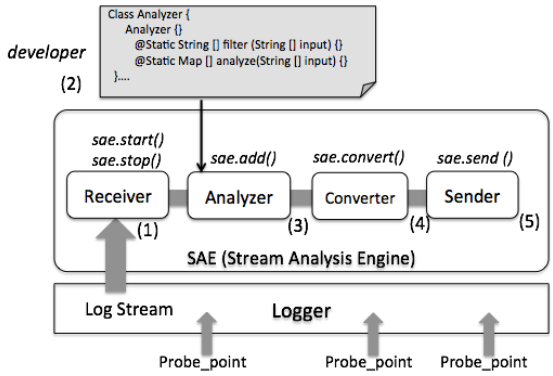


Fig. 6 Stream analysis engine overview.

6.2 Analysis Framework

To solve the problems, we propose the Stream Analysis Engine (SAE) which aims to focus on adding the log analysis code. To achieve this, SAE will provide supports for the analysis with the following processes. First, it supports to receive logs as an input stream from probe points in underneath system. Second, the appropriate analyzers that was written by developer analyze the sequence of events and collect their results. Third, the converter converts the results from the analyzer for the suitable format for visualizers. Finally, the sender sends the result for the other tools such as visualizer or storage host. We illustrated the processing functions in **Fig. 6**. With these log processing support libraries, a developer can focus on the development of *analyzer*.

To write an analysis and allow the developer to analyze code more easily, DTrace [4] and System Tap [13] provides the C-like scripting language. The purpose of these tools is to provide a scripting language where the syntax is familiar. For the same reason, we apply the Konoha scripting language [16]. 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 dose not give 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 our script, so that log is treated transparently with the output streams.

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);
}
```

6.3 Implementation of Stream Analysis Engine Components

We developed SAE, and it 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.
- *Analyzer*: It receives logs from *Receiver* and applies the analyzer algorithm that was written by the developer, then it receives the result. The detailed procedures are as follows: Develop *filter method* that extracts required events from the log streams, *analyze method* that applies to the filtered data and then *analyze the class* that contains these two methods. 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 viewer. The amount would be 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 the analysis to other protocol formats in order to apply the result stream that will be decoded by the other host easily. *convert* method will encode 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.

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. In the main function, an instance of SAE is created and added to the *filter* and *analyzer* method. To define the *filter* 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, the SAE will take into account the faults and errors that are reported by the log analysis.

7. Case Studies

Based on those transition models, we detected actual faults in the experimental system. In the following case studies, firstly we show the actual problem in the robotics system and then show the sequence based on the finite-state machine.

7.1 Target System

As a base operating system, we choose ART-Linux. ART-Linux is a real-time supported operating system based on Linux. It provides special APIs for achieving hard real-time for Robotics [12]. A real-time task will be scheduled according to the general model of the task's transition state. However, they always schedule before the non real-time task using a static priority scheduler with a simple priority driven algorithm [18]. To classify the algorithms, some of the real-time supported operating system will prepare the special scheduler for real-time tasks. In our case, we used a general real-time operating system model that has three queues such as **Real-time Task Queue**, **Non Real-time Task Queue** and **Wait Queue**. Generally, real-time tasks are transferred between these queues. At that time, the model of the transition states are shown in the right-hand square in Fig. 2. The model is a projection of the actual real-time system behavior. Based on the model, the logs are analyzed automatically.

7.2 Experimental Environment

The system environments are the following: In the main robot system, the logging real-time operating system is installed, it is a multi-core and multi-operating system as discussed above. The system can support hard real-time tasks for controlling the servo controller. On the robot, there are three main types of applications at work. One is the servo controller that controls the servo task to move around the floor. The second is a robot camera, which will detect obstacles in front of it by image histogram analysis. If the camera finds an obstacle, it will look around. The third is the other tasks that work on the system.

Normally, during the servo provides periodic operation, the robot will go straight. If the camera finds an obstacle, it looks around. Conversely, in anomalous behavior, a delay will occur because the servo will not provide the correct periodic operation. Also, the camera will not find the obstacle correctly so the results will show a conflict.

We will present the following typical examples of fault, all have different causes of faults:

- Misuse of API: Real-time OS provides the specific API for utilizing the real-time support. It is required to check that the invocation of APIs are correct, based on the specification.

- Response time, scheduling delay: Real-time OS should support precise and accurate execution of periodic tasks. It is required to check for errors and variance of delays.
- Priority inversion: Treatment of a common resource between high priority and low priority tasks is required for a real-time OS with PI_mutex or other support. It is required to check that it works by using the popular problem of priority inversion.

7.2.1 Fault 1: API Misuse

In our robot, a servo control application (real-time task) needs to work as a real-time task. In our model, the API for hard real-time scheduling is art_wait. It provides the periodic execution with a precision timer that can control the task under 1 ms. In our case, a developer used a non-real-time API in Linux (library API, usleep()) to control the servo application. It makes the servo perform a periodic execution every 4 ms because of the general periodic timer control. Consequently, the servo task worked at 1 / 4 of its potential.

- **Case 1: General API Uses**: To avoid this type of problem, we need to check that the correct sequence of API is invoked by the real-time task. To achieve this purpose, we firstly define the correct sequence of the API and transition state based on Eq. (1),
  $C_1 = \lceil(NonRTRunning) \times art\_enter() \rightarrow (Running)\rceil$ & $\lceil(Running) \times art\_exit() \rightarrow (NonRTSleep)\rceil$
  This is the general API sequence. We can check the sequence of the state in log. If we find the counter example as an improper sequence pattern, it means that the task did not exit normally and we can further investigate its cause.
- **Case 2: Periodic task Scheduling (Highest Priority)**: Compared to Case 1 and Case 2, we can verify a more detailed sequence of events for the task. In *Case1*, it only checks the *art_enter*() and *art_exit*() transition of the task, however, we need to focus in more if we want to be sure that the behavior of the periodic task was not correct. To find Fault 1, we define the additional sequence as Case 2 for detecting the problem. We define the transition sequence as follows:
  $C_2 = \lceil(Running) \times art\_wait() \rightarrow (Wait)\rceil$ & $\lceil(Wait) \times art\_wakeup() \rightarrow (Suspend)\rceil$& $\lceil(Suspend) \times dispatch() \rightarrow (Running)\rceil$.
  We use this sequence to find the pattern from the log. If the task has the highest priority, this sequence is correct because if it does not, we need to consider the blocking sequence in the definition.
- **Case 3: Periodic Task Scheduling** If the task does not have the highest priority, we need to consider the blocking sequence between the periodic executions. In this case, we need to define an additional transition sequence C_3 as follows:
  $C_3 = \lceil(Running) \times schedule() \rightarrow (Blocked)\rceil$ & $\lceil(Blocked) \times art\_wakeup() \rightarrow (Suspend)\rceil$ & $\lceil(Suspend) \times dispatch() \rightarrow (Running)\rceil$.
  In this case, the sequence needs to satisfy the condition defined above.

7.2.2 Fault 2: Response Time, Scheduling Delay

Task scheduling is feasible according to the rate monotonic algorithm, which provides the predictable, guaranteed scheduling for the real-time task set. If there are no variables in the scheduling, the tasks will not miss their deadline. In this case, three tasks are working on the system so that each task has their computation time and periods defined as C,T. The highest priority task 1 ms, 10 ms, the middle priority task 1 ms, 5 ms and the low priority task 4 ms, 20 ms. The results of the utilization are summarized as 1/10 + 1/5 + 4/20 = 0.5.

The utilization is feasible for rate monotonic algorithms. In this case, we want to know these tasks work without missing deadlines. First of all, we can check the periodic task's response time delay through use of an extended model of the basic transaction Eq. (2).

- **Case 4: Response Time Accuracy**: Periodic task response time is induced from the transition states
  $C_5 = \lceil(Running, t_1) \times art\_wait() \rightarrow (Wait, t_2)\rceil$ & $\lceil(Wait, t_2) \times art\_wakeup() \rightarrow (Suspend, t_3)\rceil$ & $\lceil(Suspend, t_3) \times dispatch() \rightarrow (Running, t_4)\rceil$
  If these transition sequences are verified, and time $\lceil(t_4 - t_1) - T < 10\mu\rceil$also is verified, the sequence and time are judged as correct.
- **Case 5-1: Scheduling Delay (Suspend to Running)**: We observe two types of scheduling delay in this case: one is the delay from $Suspend \rightarrow Running$. If there are some jitters happened in this case, it should not be allowed because it implicitly shows the long queue or that something happened in the real-time queuing task. On the other hand, a fixed and small delay can be allowed. So we need to pick up the disallowed case by the following transition model:
  $C_{5-1} = \lceil(Suspend, t_1) \times dispatch() \rightarrow (Running, t_2)\rceil$
- **Case 5-2: Scheduling Delay (Blocked to Suspend)**: The latter delay shows a different problem. The delay of the $Block \rightarrow Suspend$ means that the blocked task takes time to be released. If the time is long, we need to suspect the application waited too long for the execution.
  $C_{5-2} = \lceil(Blocked, t_1) \times art\_wakeup() \rightarrow (Suspend, t_2)\rceil$.
  In log analysis, we detect the delay between $\lceil(t_1) - (t_2)\rceil$.

7.2.3 Fault 3: Priority Inversion

In our system, we assume three tasks: the highest priority task is the servo controller, the next is a simple calculation task and the lowest priority is the joystick controller that writes a controller instruction from the joystick device to a shared resource. The servo controller will read the bit, which is written by the joystick in the shared resource. If the joystick did not write data for it, it will sleep until the next period. Priority inversion should be avoided by priority inheritance. ART-Linux provides priority inheritance by PI_mutex_lock. The joystick controller has low priority whereas the servo controller has high priority. If the joystick writes a bit for the shared resource with *PI_mutex_lock*, priority inheritance will be occur to finish the task faster.

- **Case 6: Lock Holding and Priority Inheritance Check**: Periodic task P_1 blocked by competitive task's P_2 resource access in transition, we check the priority and then if the priority is higher than the counter task, priority inheritance

Fig. 7 Pen2 robot machine.

worked. If not, wait until the lock is released. Both of the resource and priority parameters are checked at the each transition state for confirming successfully inherit priority or not. $C_6 = \lceil(Running) \times schedule() \rightarrow (Blocked)\rceil$ & $\lceil PI\_mutex\_lock\&P_1 > P_2\rceil \vee \lceil PI\_mutex\_lock\&P_1 < P_2\rceil$ & $\lceil(Blocked) \times art\_wakeup() \rightarrow (Suspend)\rceil$ & $\lceil(Suspend) \times dispatch() \rightarrow (Running)\rceil$

8. Evaluation

In this section, first we evaluate the effectiveness of our proposed method for detecting faults with kernel log analysis then we show the performance results of log transfer and analysis. Based on these results, we evaluate the approximate reaction time for robotics systems, and conclude with the planning log generation speed in our prototype system.

8.1 Environment

For evaluation we use an actual robotics system called Pen2 that is presented in **Fig. 7**. It has an Intel Core2 E7600 3.06 GHz CPU, 4 GB memory, and SSD (64 Gb). It was originally developed as an autonomous robot with an omni-directional microphone by Digital Human Research Center in AIST. We added a camera module to cope with the autonomous controller and to detect and avoid collisions with obstacles. We set up our prototype system that was installed on the one-core for the real-time operating system (ART-Linux-2.6.32-amp version), and the normal AP kernel (ART-Linux also provides non real-time SMP kernel) on the other core.

8.1.1 Composition of System

Within the three applications, we set static real-time priorities for servo controller tasks that provide control functions to achieve the feedback control of the Pen2 machine. It controls the trajectory of the two wheels with a 1 ms period. The input is the rotation speed of the motor and advancement of the vehicle. Motor speed is controlled by hardware PWM (period 25 μs) whose input is the rotational speed of the motor. If there is an input that comes from a fault detector, a safety control mechanism will work that is running with a period of 1 ms.

In order to perform the execution period with such high precision, the developer should use real-time API for a hard real-time task that is provided by ART-Linux. We show a simple example that calls on the API for this purpose in List 8.1.1.

8.1.2 Kernel Trace-point and Data Logging

In order to collect kernel logs from ART-Linux, we set up a hook inside in the kernel with a calling interface of LTTng [6].

LTTng is a type of tracer that directly inserts a hook point in the target kernel. Compared with a tracer that uses exceptions, this type can minimize the overhead. However, even if the overhead is small, this method also will produce a constant overhead from those hook points. To minimize the constant overhead, we set up the minimum hook point that was embedded only in the __dispach() function in front of the context switch of the priority scheduler. We show the place that we set up the hook point in List 8.1.2. This function is called by several upper functions that need to change the transition state of real-time tasks.

We assume that all of the state transitions of real-time tasks we showed in Section 4.2.1 change through the context_switch in ART-Linux. That was why only one hook point is enough for our analysis method. We show the sample logs in **Fig. 8** that were generated by kernel hook point. The log includes the number of current and next state transitions and event (function) name, process id, priority, and time. We can check whether the sequences of real-time tasks are correct or not by using these kernel logs with the proposed method in Section 7.

As we described in Section 5.2, the speed of generation of kernel would be very high if we set several hook points for each system call in kernel such as 25 MB/sec. In such a time, it is difficult to detect an error event from these large logs with a manual approach. In this case, our proposed automatic methods of analysis are effective in detecting errors. In the following sections, we will describe the results of our evaluation of this environment.

8.2 Detecting Faults

In our case studies, we detect faults by using the system described above sections. API misuse in Case 1, and scheduling

List 8.1.1 : Example of servo controller task [servo_control.k]

```
1  void artsv_servo(void)
2  {
3    if (art_enter (ARTSV_HIGH_PRIORITY,
4          ART_TASK_PERIODIC,
5          ARTSV_PERIOD_SERVO) == -1) {
6      perror(''art_enter error'');
7      exit(1);
8    }
9    while (artsv_enable_flag) {
10     art_wait();
11     /* do servo work */
12   }
13 }
```

List 8.1.2 ART-Linux dispatch function in kernel [linux/kernel/art_task.c]

```
1  int __art_dispatch(struct thread_info *prev_thread
       ,
2  art_task_t *prev)
3  {
4  /* select highest priority task from art_run_queue
       */
5  switching:
6    if (next_thread != prev_thread) {
7      next_thread->flags|=prev_thread->flags
8                  TIFNEED_RESCHED;
9      prev_thread->flags&=~TIF_NEED_RESCHED;
10     lttng_trace (1);
11     art_context_swithch(prev_thread->task,
12                 next_thread->task);
13   return 1;
14 }
```

```
change_artstate: 948.376423541 (/tmp/trace_inherit/myprobe_0), 0, 0, /usr/bin/sudo, , 0, 0x0,
SYSCALL { pid = 1208, prev_state = 1, next_state = 0, msg = "__art_dispatch", seq_num = 393 }
arttrace: 948.376424999 (/tmp/trace_inherit/myprobe_0), 0, 0, /usr/bin/sudo, , 0, 0x0, SYSCALL
{ count = 843, pid = 1208, current_flags = 1157, prev_flags = 0, prio = 1 }
change_artstate: 948.376431439 (/tmp/trace_inherit/myprobe_0), 0, 0, /usr/bin/sudo, , 0, 0x0,
SYSCALL { pid = 1208, prev_state = 0, next_state = 1, msg = "art_change_state", seq_num = 394 }
change_artstate: 948.376432098 (/tmp/trace_inherit/myprobe_0), 0, 0, /usr/bin/sudo, , 0, 0x0,
SYSCALL { pid = 1208, prev_state = 1, next_state = 2, msg = "art_keep_waiting", seq_num = 395 }
change_artstate: 948.376432687 (/tmp/trace_inherit/myprobe_0), 0, 0, /usr/bin/sudo, , 0, 0x0,
SYSCALL { pid = 28, prev_state = 1, next_state = 0, msg = "__art_dispatch", seq_num = 396 }
arttrace: 948.376433447 (/tmp/trace_inherit/myprobe_0), 0, 0, /usr/bin/sudo, , 0, 0x0, SYSCALL
{ count = 844, pid = 28, current_flags = 1157, prev_flags = 0, prio = 1 }
change_artstate: 948.376434235 (/tmp/trace_inherit/myprobe_0), 0, 0, /usr/bin/sudo, , 0, 0x0,
SYSCALL { pid = 28, prev_state = 0, next_state = 1, msg = "art_change_state", seq_num = 397 }
change_artstate: 948.376434789 (/tmp/trace_inherit/myprobe_0), 0, 0, /usr/bin/sudo, , 0, 0x0,
SYSCALL { pid = 28, prev_state = 1, next_state = 2, msg = "art_keep_waiting", seq_num = 398 }
change_artstate: 948.376435362 (/tmp/trace_inherit/myprobe_0), 0, 0, /usr/bin/sudo, , 0, 0x0,
```

Fig. 8 Example log for analysis.

delay with feasibility study misses and over-interrupted periodic task delay in Case 4.

We applied each pattern checking modules to kernel logs and detected the deviated patterns from the patterns defined in the previous Section 7. In our system, if the analyzer finds an error in the system, it shows a warning. Each analyzer will judge an incorrect behavior or pattern as an anomaly.

In the Section 7 we described the type of fault in the subsections. For example, fault 1 is a type of API misuse in Section 7.2.1, and fault 2 is a type of fault that included a response time and scheduling delay in Section 7.2.2. Fault 3 is a type of fault that comes from priority inversion. We described that in Section 7.2.3.

Fault 1 is actually detected by static analysis as opposed to online monitoring. However, it is difficult to detect fault 2, timing faults, fault 3, and the combination of faulty conditions. For example, we discussed the priority inversion problem. The two types of analyzers detect this problem. Even if the Case 6 analyzer detects that priority inherit has been successfully done after lock was released, the Case 4 analyzer detects a long response time delay before priority inheritance. Based on the results of both analyzers, we can find the root cause of the problem.

Generally, if the priority inheritance is missed, then long latencies appear in the high priority task. In this case, we can detect priority inversion. Upon detailed inspection, we can detect that the root cause is the wrong implementation of the low priority joystick task invoke *art_wait*()(sleep) function soon after holding the PI_mutex_lock for the shared resource. It induced the problem that ART-Linux kernel did not inherit the higher priority to the low priority task, since the kernel cannot inherit the priority to the task that slept in the local queue. For encountering these task, even if a high priority task, it should wait until the low priority task that slept in the local queue will wake up and release its lock. During the sleep time, the priority inheritance has not occurred. As a result, priority inversion was aroused when the middle priority task worked. It was the specification matter that the developer did not write that the task should not sleep after holding the lock, and the application developer that did not understand the specification.

This type of problem is difficult to identify in the actual system, however, the automatic support for log analysis will help the developer, who needs to identify the problem as soon as possible. In our system, each analyzer will report the result of a log analysis.

8.3 Performance of Log Transfer

In this section, we compare the two methods that make communication possible in the OS. One is shared memory, and the other is the socket. The experimental machine is a MacBook Pro, CPU is 2.13 GHz Intel Core 2 Duo, and memory is 2 GB 1067 MHz DDR3. The reason for using this hardware is easily simulate the Pen2 machine, since this machine provides Intel Core2 and high speed shared memory, and portable for AIST laboratories. For this machine, we installed ART-Linux-2.6.32-amp version on the one-core for the real-time operating system, and the normal AP kernel (ART-Linux also provides non real-time SMP kernel) on the other core.

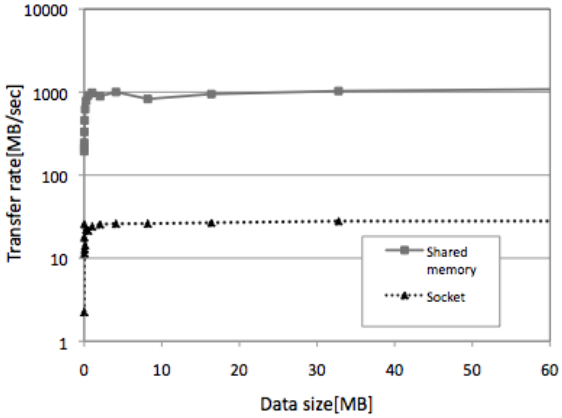


Fig. 9 Log transfer rate.

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 **Fig. 9**. 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 showed 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 μs period is permitted for a real-time task's logging.

8.3.1 Overhead

The average cost of an analyzer includes the work of processing and analyzing logs is, on average, 5%. This overhead would lineraly increase accordance to the increase of analyzer. Our system separated the analyzer overhead from the target core, succeessed to limit the overhead of the target machine to less than 5% in our evaluation.

A task that transfers the kernel logs to the log analysis core did not disturb the execution of real-time task, since the task was worked as a background task with lower priority than the real-time tasks.

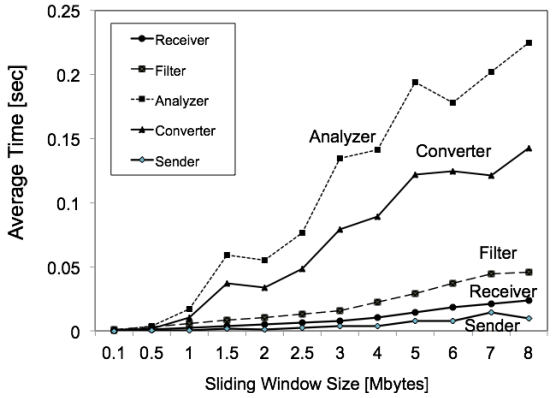


Fig. 10 Average time of different window size.

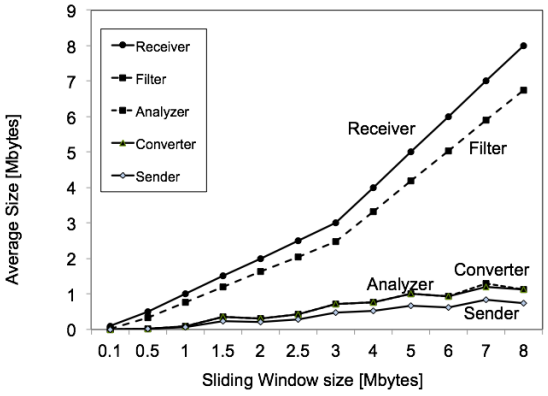


Fig. 11 Average size of different window size.

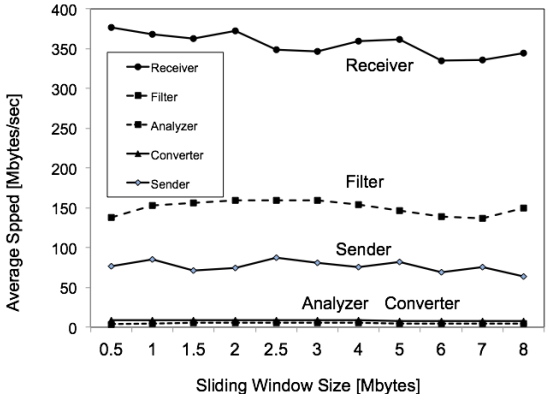


Fig. 12 Average speed of different window size.

8.4 Performance of Log Analysis

Using the same machine and environment described in Section 8.3, we evaluate the performance of log analysis.

We added the bug code to the robot program (1) based on the model, developed four faults for both of kernel and user application (2) added 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 wrote an analyzer using our framework [30] to detect bugs that will not call the API correctly. This means they will be caused by the abnormal sequence call in the transition. Then we set up the framework system and started the logger.

Figure 10 shows average processing time of each SAE components, such as *Receiver*, *Filter*, *Analyzer*, *Converter*, *Sender*. In this figure, *x*-axis indicates window size (Mbytes) and the *y*-axis indicates the average processing time of transferred kernel monitoring log. Obviously, in these components, the *Analyzer* takes the longest time. It consumes the time for judgment whether the normal pattern or not from function calls and event sequences in the log. *Analyzer* result shows that it does not consistently increase with increase of window size. There are variations in each size. We consider it comes from the differences in contained data type, not comes from the volume of logs. Since next **Fig. 11** shows the result of the processing size increasing almost linearly with the increasing of window size. The results differ depending on the contained data type of log that lots of pattern matches patterns or not.

Similar variation is seen in *Converter* in 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 11 shows average processing size (Mbytes) for different window size. *X*-axis indicates window size (Mbytes), and *y*-axis indicates average log processing size (Mbytes)in each SAE component.

Compared to the average time, which we showed as Fig. 10, the average processing size has linearly increased along with the window size. It means that the size of the window did not affect the analysis. On the other hand, there are differences of 7 Mbytes between the result of *Receiver* and *Sender* at most. This means that *Filter* could have reduced the data. It also means that the filter can reduce the volume of data by implementing suitable filters.

Figure 12 shows the average speed (Mbytes) in different window size. The *x*-axis indicates the window size (Mbytes) and *y*-axis indicates the average speed (Mbytes) of log processing in each SAE component. The result shows that regardless of the window size, these speeds are nearly constant. It indicates that we can successfully separate the component interface as an appropriate unit. In the results, the speed of *Receiver* is fastest in these components and it processes data at an average of 350 MB/sec. The process of the *Receiver* is consist with just accepting data from other hosts, while *Analyzer* and *Converter* parse and check data according with internal models.

8.5 Reaction Time

The assumption of reaction time for our proposed system is fundamental for the design requirements involving log generation, transfer, and analysis speed. In this section, we will consider this time. As we described in Section 8.1.1, our robot system has a control mechanism to stop the motor within 1 ms after it accepts any signal from safety control.

To understand the reaction time using our proposed system, we can briefly calculate the time that it takes to stop with our log transfer and analysis system is induced by calculating with the parameters such as the rate of amount of transferred logs (Mbytes), and Transfer speed of shared memory (Mbytes/sec) plus the rate of amount of log to be processed (Mbytes) and processing speed of SAE (Mbytes/sec), then plus execution time of task that doing this work. The generalized formula is as follows.

$$React\_Time = Trn\_Log/Trn\_Spd + Pros\_Log/Pros\_Spd + \alpha$$

Table 1 Parameters.

Name	Means	Units
React_Time	Reaction Time	sec
Trn_Log	Amount of transferred logs	Mbytes
Trn_Spd	Transfer speed of shared memory	Mbytes/sec
Pros_Log	Amount of log to be processed	Mbytes
Pros_Spd	Procesing speed of SAE	Mbytes/sec

The assumed the parameters are described in **Table 1**. There are two points we need consider to use this formula. The first, in our case, we need to not only consider the transfer time to analysis core, but also consider the return time to the target core, since the reaction program worked on the target core. However, the amount of data returned back from the analysis core will be very small compared to the transfer logs from the target core such as an error detect signal. Therefore, we neglect this parameter from the formula. The second, we need to consider the method of log processing and the analysis mechanism, such as a single processing or a parallel one. Currently, the formula will only consider the single processing. In the future, we need consider the parallel or threaded processing of log analysis to improve the efficiency.

To calculate the approximate reaction time, we apply the results of our previous experiences in Sections 8.3, 8.4. The transfer speed of shared memory resulted from Section 8.3 is about 1,000 Mbytes/sec on average. The average processing speed of SAE in Section 8.4 is the sum of the five processing functions: Receiver, Filter, Analyzer, Converter, Sender - 588.3 Mbytes/sec. The average execution time of the program is too small to be neglected in our case, because the program just called APIs to write logs to shared memory. The calculation results in a reaction time of 1 Mbytes of data and by substituting these values into the formula is approximately 1 Mbytes / 1,000 Mbytes + 1 Mbytes / 588.3 Mbytes = 0.0026998 sec (about 2.69 ms). Even if these calculation results are taken on the lower power machine (Intel Core2Duo 2.3 GHz) than the Pen2 machine, it would be the applicable for approximate calculation of worst case for higher power machine.

In our experiment, the average speed of kernel log generation is from 10 to 20 Mbytes/sec (on average 17.5 Mbytes/sec). If we re-calculate the result with this value for the formula, the reaction time is about 27–67 ms (on average 47 ms). In order to judge that we need about 50 ms for processing 20 Mbytes/sec kernel log, we need to consider the safety control requirements of the robot. In our experimental study, we use the Pen2 robot machine that had a feedback control mechanism within the hardware described in Section 8.1.1.

The safety mechanism in the servo controller of Pen2 that can stop the motors of the wheels within a 1 ms period will come into play after any emergency signals are received. The maximum speed of the Pen2 robot is 2 m/sec. This means that the robot will move a distance of 2 mm in 1 ms, and of 20 mm in 10 ms. The distance required to stop will depend on the friction between the wheels and the ground. In our laboratory, the floor is covered by linoleum so the robot stops after about 100 mm (taking into account the slip that will occur, the error is +/− 30 mm). Therefore, it will be difficult to stop after encountering an obstacle, however, it is acceptable to stop after sensing the distance between the obstacles in 50 ms. Since the Pen2 machine can move at a maximum speed of 2 m/sec, it needs 20–40 ms to stop and therefore will require a maximum of 130 mm distance. If we can send a stop control message before this time, it will stop safely.

Considering these requirements, our calculated result of about 50 ms with 20 Mbytes/sec logs, 25 ms with 10 Mbytes/sec logs would be acceptable for the Pen2 robot for safety control. For example, if you assume a car going at 100 km/hour, the speed of this car is 27.8 m/sec. It means it has moved 27.8 mm forward in 1 ms, 278–1,390 mm in 10–50 ms. From this point, a car is required to stop 40–50 m after accepting this reaction command. That means that it is necessary to consider whether further tuning should be considered for practical usage. On the other hand, it is known that for use in industry it would be required to stop within 1 ms of detecting an obstacle with sensors. If we reduce our logs from 20 Mbytes/sec to 1 Mbytes/sec (1/20) taking efforts to reduce hook point or events, our proposed system might have a use in industrial robotics.

8.5.1 Planning Log Volume and Analysis

As we mentioned in Section 5.2, in order to avoid the performance degradation caused by buffer overflow of the log, it is necessary to adjust the amount of log so as to satisfy the processing and generation speed of logs as *Generation Speed* ≤ *Processing Speed*.

In our case, the transfer speed of logs is very high: 1 Gbytes/sec (theoretically 12.8 GB/sec in DDR2), so we need to focus on the processing speed of processing logs in the analysis core. In this experiment, our log generation speed is 10–25 Mbytes/sec (on average 17.5 Mbytes/sec). This was less than the processing speed of the SAE system that provides 588.3 Mbytes/sec on average. Therefore, it is possible to avoid buffer overflow between the cores.

The average speed of SAE, as shown in Fig. 12, was almost constant and did not depend on the window size. In our experiment, we need to assume the typical *Analyzer* that includes pattern match code, on average abount eight to ten, except for the priority inversion detection. We assume the number of patterns, on average, these numbers in our proposed method for the *Analyzer* to detect a wrong sequence of logs. We needed to check both of events and transitions of tasks that average four or five transitions par case. However, if the system would be complex, and more patterns should be included in the *Analyzer*, or the number of the *Analyzer* should be increased to check for wrong sequences in the system, the total speed of the processing logs will be decreased. Before using our system, this type of planning is necessary for avoiding buffer overflow of log processing. As a next step, we plan to develop a simulator that adjusts the parameter of log generation and processing speed with necessary parameters such as hook point and number of events that should be checked their patterns.

9. Conclusion

In this paper, we proposed an online kernel monitoring and log analysis method and framework. Based on it, we developed a prototype system to satisfy the requirements of the advanced embedded system. The contributions of our approach are the following:

Develop (1) a log analysis method based on transition modeling of real-time tasks, and (2) an online logging and analysis system where the monitored application and systems are analyzed concurrently without disturbing the real-time execution of monitored applications. These tools will provide support for the engineer to perform root cause analysis. In the evaluation, we could show the effectiveness of our system that can detect faults including a serious one, which was not detected for ten years in ART-Linux. Also the performances results are considered to be acceptable in our approximate calculation based on the experiment. However, it is still difficult to detect the root case of failure within the theoretical time; this will be an issue for future work.

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

References

[1] Andrews, J.H. and Zhang, Y.: General test result checking with log file analysis, *IEEE Trans. Softw. Eng.*, Vol.29, No.7, pp.634–648 (2003).

[2] Asberg, M., Kraft, J., Nolte, T. and Kato, S.: A loadable task execution recorder for linux, *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems* (*WATERS 2010*), pp.31–36 (2010).

[3] Barham, P., Isaacs, R., Mortier, R. and Narayanan, D.: Magpie: Real-time. modelling and performance-aware systems, *9th Workshop on Hot Topics in Operating Systems* (*HotOS IX*), New York, NY, USA (2003).

[4] Cantrill, B.M., Shapiro, M.W. and Leventhal, A.H.: Dynamic instrumentation of production systems, *Proc. Annual Conference on USENIX Annual Technical Conference*, *ATEC '04*, p.2, Berkeley, CA, USA, USENIX Association (2004).

[5] Delgado, N., Gates, A.Q. and Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools, *IEEE Trans. Softw. Eng.*, Vol.30, No.12, pp.859–872 (2004).

[6] Desnoyers, M. and Dagenais, M.R.: The lttng tracer: A low impact performance and behavior monitor for gnu/linux, *Proc. Linux Symposium*, Vol.1, pp.209–224 (2006).

[7] Fickas, S. and Feather, M.S.: Requirements monitoring in dynamic environments, *Proc. Second IEEE International Symposium on Requirements Engineering*, *RE '95*, pp.140–147, Washington, DC, USA, IEEE Computer Society (1995).

[8] Fonseca, P., Li, C., Singhal, V. and Rodrigues, R.: A study of the internal and external effects of concurrency bugs, *DSN*, pp.221–230 (2010).

[9] Hiller, M.: Executable assertions for detecting data errors in embedded control systems, *Proc. 2000 International Conference on Dependable Systems and Networks* (*formerly FTCS-30 and DCCA-8*), *DSN '00*, pp.24–33, Washington, DC, USA, IEEE Computer Society (2000).

[10] M. Holenderski, van den Heuvel, M.M.H.P., Bril, R.J. and Lukkien, J.J.: Grasp: Tracing, visualizeing and measuring the behavior of real-time systems, *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems* (*WATERS 2010*), pp.37–42 (2010).

[11] Holzmann, G.J.: The model checker SPIN, *Software Engineering*, Vol.23, No.5, pp.279–295 (1997).

[12] Ishiwata, Y., Kagami, S., Nishiwaki, K. and Matsui, T.: Art-linux 2.6 for single cpu: Design and implementation, *Journal of Robotics Society of Japan*, Vol.26, No.6, pp.77–84 (2008).

[13] Jacob, B., Larson, P., Leitao, B.H. and da Silva S.A.M.M.: Systemtap: Instrumenting the linux kernel for analyzing perfomance and functional problems, *Proc. Annual Conference on USENIX Annual Technical Conference*, REDP-4469-00, IBM, International Technical Support Organization (2009).

[14] Jula, H., Tralamazza, D., Zamfir, C. and Candea, G.: Deadlock immunity: enabling systems to defend against deadlocks, *Proc. 8th USENIX Conference on Operating Systems Design and Implementation*, *OSDI'08*, pp.295–308, Berkeley, CA, USA, USENIX Association (2008).

[15] Kanda, W., Yumura, Y., Kinebuchi, Y., Makijima, K. and Nakajima, T.: Spumone: Lightweight cpu virtualization layer for embedded systems, *International Conference on Embedded and Ubiquitous Computing, EUC '08, IEEE/IFIP*, Vol.1, No.12, pp.144–151 (2008).

[16] Kuramitsu, K.: Konoha - Implementing a static scripting language with dynamic behaviors, *Workshop on Self-sustaining Systems* (*S3*) *ACM*, ACM Press (2010).

[17] Laprie, J.-C. and Randell, B.: Basic concepts and taxonomy of dependable and secure computing, *IEEE Trans. Dependable Secur. Comput.*, Vol.1, No.1, pp.11–33 (2004).

[18] Liu, J.W.S.W.: *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA (2000).

[19] Mcdowell, C.E. and Helmbold, D.P.: Debugging concurrent programs, *ACM Computing Surveys*, Vol.21, pp.593–622 (1989).

[20] Neophytou, N. and Evripidou, P.: Net-dbx: A web-based debugger of mpi programs over low-bandwidth lines, *IEEE Trans. Parallel and Distributed Systems*, Vol.12, pp.986–995 (2001).

[21] Pattabiraman, K., Kalbarczyk, Z. and Iyer, R.K.: Automated derivation of application-aware error detectors using static analysis, *Proc. 13th IEEE International On-Line Testing Symposium*, pp.211–216 (2007).

[22] Pattabiraman, K., Saggese, G.P., Chen, D., Kalbarczyk, Z. and Iyer, R.: Automated derivation of application-specific error detectors using dynamic analysis, *IEEE Trans. Dependable Secur. Comput.*, Vol.8, No.12, pp.640–655 (2011).

[23] QNX, available from ⟨http://www.qnx.com/⟩.

[24] Reis, G.A., Chang, J., Vachharajani, N., Rangan, R. and August, D.I.: Swift: Software implemented fault tolerance, *Proc. 3rd International Symposium on Code Generation and Optimization*, pp.243–254 (2005).

[25] Rela, M.Z., Madeira, H. and Silva, J.G.: Experimental evaluation of the fail-silent behaviour in programs with consistency checks, *Proc. The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing* (*FTCS '96*), pp.394–403, Washington, DC, USA, IEEE Computer Society (1996).

[26] Rosu, G., Schulte, W. and Serbanuta, T.F.: Runtime verification of C memory safety, Saddek Bensalem and Doron A. Peled, editors, *Runtime Verification* (*RV'09*), *Lecture Notes in Computer Science*, Vol.5779, pp.132–152 (2009).

[27] Japan Science and CREST Project Technology: Dependable embedded operating system project white paper version 3.0 (Oct. 2009), available from ⟨http://www.dependable-os.net/en/topics/file/White_Paper_V3.0E.pdf⟩.

[28] Sha, L., Rajkumar, R. and Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization, *IEEE Trans. Comput.*, Vol.39, No.9, pp.1175–1185 (1990).

[29] Sugai, N., Kondo, H. and Ochiai, S.: A software platform for multiple os extension on embedded chip-multiprocessor, *IPSJ SIG Technical Report*, Vol.3 (2007).

[30] Sugaya, M., Igarashi, K., Goshima, M., Nakata, S. and Kuramitsu, K.: Extensible online log analysis system, *Proc. 13th European Workshop on Dependable Computing*, *EWDC '11*, pp.79–84, New York, NY, USA, ACM (2011).

[31] Tokoro, M.: Open systems science: Solving problems of complex and time-varying systems, *Workshop on Agents and Multi-agent systems for Enterprise Integration*, Salamanca (Apr. 2011).

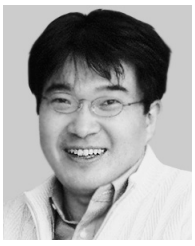
[32] Xu, J. and Parnas, D.: Scheduling processes with release times, deadlines, precedence and exclusion relations, *IEEE Trans. Softw. Eng.*, Vol.16, No.3, pp.360–369 (1990).

Midori Sugaya is a lecture professor of Yokohama National University, and previously worked in Dependable Embedded OS R&D Center, Japan Science and Technology Agency (JST). She has several years of work experience in the software industry. She received her Ph.D. in Computer Science from Waseda University in 2010. Her research interests include real-time operating and dependable systems and failure detection in fault tolerant systems. She is a member of IPSJ, IEICE, IEEE-CS, and ACM.

Hiroki Takamura is a researcher of Dependable Embedded OS R&D Center, Japan Science and Technology Agency (JST). He received his Ph.D. from the School of Information Science, Japan Advanced Institute of Science and Technology in 2004. His research interests are mathematical logic (especially, constructive mathematics and substructural logic), formal methods and dependability. ¿From 2009, He is a member of the IEC TC56 (Dependability).

Yoichi Ishiwata is a researcher of Digital Human Research Center of the National Institute of Advanced Industrial Science and Technology (AIST). He is also the developer of ART-Linux, which is a real-time extension of Linux kernel.

Satoshi Kagami is the deputy director of Digital Human Research Center of the National Institute of Advanced Industrial Science and Technology (AIST). He received a Ph.D. from the Graduate School of Information Engineering, the University of Tokyo in 1997. He has been a research associate at the University of Tokyo from 1997 to 2001. Since 2001, he joined to Digital Human Research Lab., AIST as a senior research scientist. His research interest includes, robot vision & audition, motion planning, bipedal control and system integration with real-time operating system. He is a member of the Robotics Society of Japan, and the IEEE Robotics & Automation Society.



Kimio Kuramitsu is an associate professor of Department of Electronic and Computer Engineering, Yokohama National University. He received his Ph.D. from the University of Tokyo in 2003 and has been working on scripting language design for dependable and distributed systems. He is also a director of Konoha open source project and his research group is developing dependable software include Konoha for future dependable systems.