

Design of Fault Tolerant System based on Runtime Behavior Tracing

Sumin Park*, Kwangyong Lee**

**Dept. of Computer Software and Engineering, University of Science of Technology(UST), Korea*

***Embedded SW Platform Research Team, Electronics and Telecommunications Research Institute(ETRI), Daejeon, Korea*
{spritos, kylee}@etri.re.kr

Abstract— The current researches to improve the reliability of operating systems have been focusing on the evolution of kernel architecture or protecting against device driver errors. In particular, the device driver errors are critical to the most of the complementary operating systems that have a kernel level device driver. Especially on special purpose embedded system, because of its limited resources and variety of devices, more serious problems are induced. Preventing data corruption or blocking the arrogation of operational level is not enough to cover the entire problems. For examples, when using device drivers, the violation of function's call sequence can cause a malfunction. Also a violation of behavior rules on system level involves the same problem. This type of errors is difficult to be detected by the previous methods. Accordingly, we designed a system that traces system behavior at runtime and recovers optimally when errors are detected. We experimented in Linux 2.6.24 kernel operating on GP2X-WIZ mobile game player.

Keywords— Fault tolerance, Reliability, System behavior, Runtime trace, Petri Net

I. INTRODUCTION

The reliability is becoming one of the most critical issues for operating systems. The current researches have been focusing on the evolution of kernel architecture and protecting against device driver errors. In Windows NT, it adopts Micro-kernel architecture but the device driver accesses hardware directly due to the performance. And 85% of errors are caused by device driver on Windows XP [4]. So, it looks sufficient to do efforts on device driver issue. For example, Nooks lets a device driver live in a separate memory domain from the kernel [4]. And the others have similar mechanisms.

However, there exists another type of error that is impossible to be detected by isolation of memory domain or checking the transfer data. Sometimes devices offer a rule about commands to be controlled. A misuse of this rule could make the device freeze. Also some device drivers have an own sequence of function calls. If the sequence is violated, the operating application would be crashed or unexpected errors might happen. These errors often occur when the device driver

is developed and released in short period without an enough validation. Furthermore, in embedded system with a limited resources and variety of devices, it often has some types of logical patterns on system behavior level. Therefore we designed and implemented fault tolerant system based on runtime behavior tracing.

The first step is to make a model that includes a target system's behavior information. A static analysis helps to improve the reliability of Fault Tolerant system. The second step is to construct a monitor module and a recovery module. They are attached to kernel. The third step is to locate the trace points in kernel. If possible, it could be in application with a simple system call. The method of static analysis for modeling can be varied. We just used a simple Petri-Net model. Also we adopted Petri-Net method to checking in the trace module.

We experimented in GP2X-WIZ mobile game player and experimented to solve a frame buffer problem when playing a game emulator. Our system detected the problem effectively and recovered without any remarkable victim.

II. RELATED WORKS

Nooks offers detection, isolation and recovery function for device driver errors. It is implemented on Linux. Key point is to let the device driver work on light-weight protection domain. Device driver can't access a kernel's memory space except its own memory space. Also it prevents a resource leak using object-tracking [4].

Shadow driver is designed to recovery the device driver and the application without a corruption of current status. Shadow driver has a same interface as like as an original driver but doesn't have the contents. On passive mode, it just monitors the transfer data. If any error happens, it changes to active mode and work instead of original driver. So while the shadow driver is working, the original driver is recovered with the status saved previously [5].

CuriOS is a micro-kernel OS. It recovers application transparently as like as Shadow driver. Its deference is to keep the status data in independent memory space for preventing a corruption. And the memory space for status data is connected dynamically when it is needed only [3].

EROS makes a snapshot every time intervals. If an error occur, it just reloads all system status and data from the last

This work was supported by MKE, "Development of Embedded SW Common Platform for Information and Industrial Devices" project (KEIT 2009-S-029-01). The corresponding author is K.Y.Lee.

snapshot. A heavy overhead is accompanied. And the status after the last snapping is lost [7].

III.DESIGN OF FAULT TOLERANT SYSTEM

A. System behavior modeling

We adopted Petri Net as a system modeling method. It is sufficient to model system behavior. And it offers the simple method to calculate whether any event is fault to its model information for tracing. Figure 1 shows a example of system modeling using Petri Net. $p\{1-3\}$ are places and $t\{1-4\}$ are transitions.

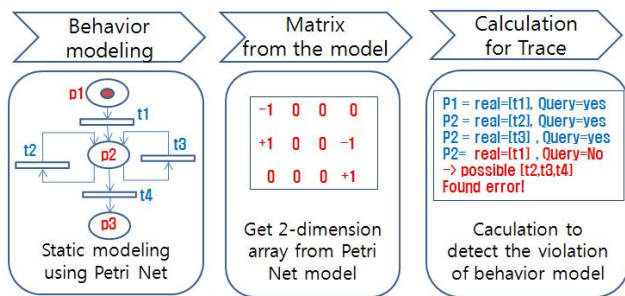


Figure 1. Pseudo code which shows the decision process of event.

We extracted a 2 dimension array from the model. The array is used to calculate on the trace monitor. And such Petri Net data obtained from system could be classified by the level of risk for the recovery policy. Several modeling are needed according to the system characteristics.

The depth of modeling can be differentiated according to the requirement of the system reliability. The deeper level modeling provides more reliable service with overhead increases on trace module.

B. Structure

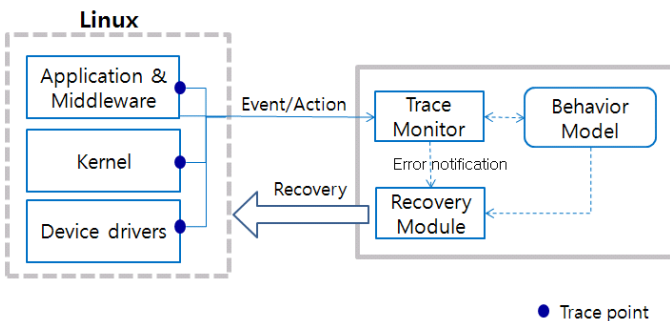


Figure 2. A system block diagram which shows the structure of Fault Tolerant System based trace point

Figure 2 shows the whole system block diagram. The system consists of operating system part and fault tolerant system modules. Operating system includes applications, middleware, a kernel and device drivers. FTS includes a monitor module, system behavior model and recovery module. If an event occurs on any trace point in operating system, the event data is delivered to the trace module. The trace module checks whether the event violates the system

These systems explained above don't care about the errors caused by a violation of system behavior.

model. If a violation occurs, it notifies to a recovery module. Then the recovery module operates according to the system model and the recovery policy. Also the trace point is possible to be located any place needed in operating system optionally. So, developers can control the overhead of performance caused by the tracing process.

C. Trace monitor

If an action or an event occurs from any trace point, the monitor module starts operating. At first, it collects possible actions at the moment. And it searches whether the occurred event exists in the possible action set. If the event exists in the possible action set, trace monitor treats the event as an error. If the event doesn't exist in the possible action set, move to next state and continues operation. Figure 3 shows the pseudo code of the decision process.

```

if ( EVENT ∈ {possible actions} )
    do next operation
else
    do recovery routine

```

Figure 3. Pseudo code of the decision process for error.

D. Recovery policy

Recovery module receives an error event from trace module. At first, the recovery module checks whether the error is a known error or not. If the error is a known error, the related recovery routine performed. If the error is an unknown error, the recovery module decides the level of risk and performed the recovery routine selected. Internal error need to restart the target component only. If the error has a possibility of propagation, restart the related components optionally. When high risk and unrecoverable error occurs, reboot the whole system. Figure 4 shows a pseudo code of the recovery policy.

```

if ( ERROR ∈ {known error set} )
    do related recovery routine
else
    if ( ERROR = High risk )
        rebooting
    else if ( ERROR = Internal error )
        recover target component
    else if ( ERROR = propagated error )
        recover related components

```

Figure 4. Pseudo code which shows the recovery policy.

IV.IMPLEMENTATION AND EXPERIMENTS

A. Implementation

We implemented in Linux 2.6.24 kernel and experimented in GP2X-WIZ mobile game player which uses a Pollux SoC chip. Table 1 shows the system specification. We developed on Eclipse-esto 3.3 and cygwin.

GP2X-WIZ spec.			
Processor core	Memory	Flash	Peripherals
ARM926EJ , 533MHz	SDRAM 64MB	NAND 1GB	USB 2.0, SD Card, 2.8 inch AMOLED

Table 1. Specification of GP2X-WIZ mobile game player.

We implemented a linked list queue for the events from trace point. To reduce the overhead for processing events at trace point, an event queue and a deferred checking are used in trace module. To reduce the polling overhead, the polling thread sleeps until a new data is delivered to event queue. Trace module polls the queue to pick up events. The events are serialized. Also *fts_enqueue* and *fts_dequeue* functions are atomic.

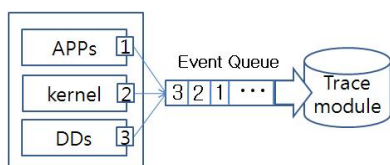


Figure 5. Event queue

The structure of element includes the target information (process or kernel components) and variables of an event. The *target_id* and *level* are needed for recovery. *target_id* is usually the target process that made the event or action. And *group_id* is used for propagated errors. The *flag* includes policy information for management. *EVENT* structure includes the Petri Net model information associated with the current event.

```

struct fts_event
{
    int model_id;           //Petri Net id
    int target_id;          //process id
    int action_id;          //transition in Petri Net
    unsigned char flag;     //management policy
    signed char level;      //depth of trace
    struct fts_event *next; //single linked list
}

```

Figure 6. Event structure

Trace module keeps a current state in an array. The length of current state array is the same with the number of places in Petri Net model. If the model has eight places and the state is P2 currently, the current state array is "01000000". To calculate whether the event is error, use a 2-dimension array(state-transition-engine) obtained from Petri Net model. First step is to get a transition which is saved *event* variable in *struct fts_event*. Second step is to sum each element of current state array with the column elements which matches on state-transition-engine with the transition number obtained on step 1. If one of the sum values is less than 0 once, it means an error occurred. Petri Net makes the process of trace easy and supports parallel process well. So we expect that it is useful for multi-threaded behavior modeling.

Recovery Module is implemented as a kernel thread because it needs to control all components of system for recovery. And we just use a switch-case statement and used the parameter in a given events as a condition for recovery policy. And it implemented as a loadable Linux module. So it could be loaded according to the demand. To decide the recovery method, use an *EVENT* structure and *flag* in *struct fts_event*. The *EVENT* structure includes whether target components are independent and restartable. And *flag* variable specifies whether the trace module continues tracing of the current corrupted model. If tracing is blocked during recovery, it will be resumed after completion of the recovery process.

B. Experiments

GP2X-WIZ, the mobile game player based on Linux, has a problem about frame buffer. After playing *mame* game emulator, its resolution is changed to wrong values. So its x-window manager(*gp2xmenu*) becomes unusable. It could be a bug in *gp2xmenu*, *mame* or frame buffer driver. Even if the bugs are fixed, some wrong application might produce the same error again. Figure 7 shows the x-y resolution corruption of frame buffer.

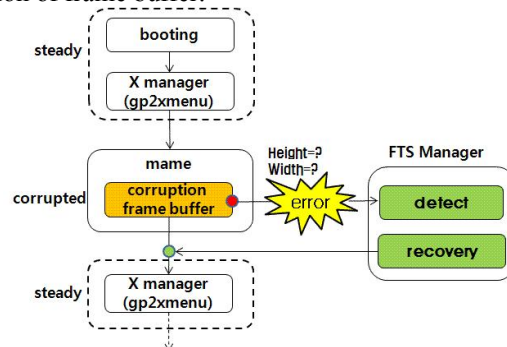


Figure 7. x-y resolution corruption of frame buffer

We made a model for this problem using Petri Net. And we gained the 2-dimension array for tracing and applied it to Trace Monitor. Because this error is included into the known error set, we implemented the recovery routine into Recovery Module and set the property of the error in the check function as a known error. Finally we located the trace point at the position where *gp2xmenu* and *mame* are provoked. Figure 8 shows the Petri Net model with matrix. And figure 9 shows how to locate the trace points and the process from detection to the recovery.

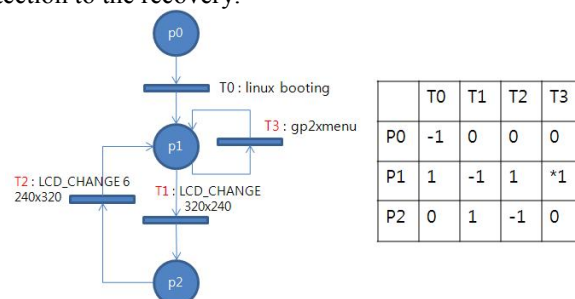


Figure 8. Frame buffer model

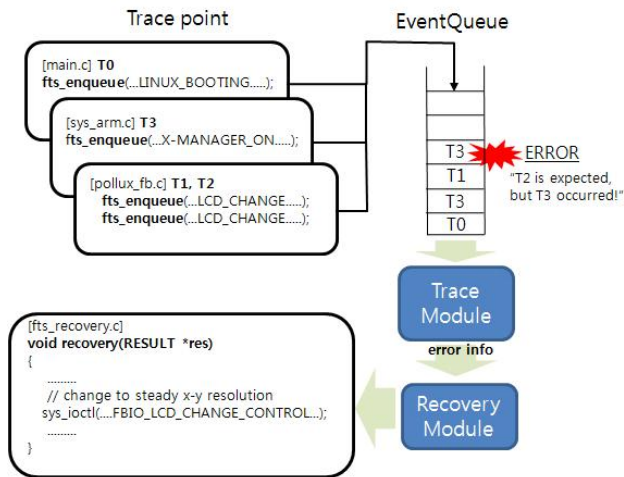


Figure 9. Tracing and recovery of the frame buffer corruption

In the experiment, we operated some applications at first. And we checked both the event queuing from the trace points and the calculation for tracing in Trace Module. After that, we operated *mame* for testing. Trace Module detected the error correctly as a known error and performed the recovery routine that we inserted. Figure 10 shows the result from our experiment. Figure (10-a) is when the frame buffer is corrupted without our system. The X-manager(gp2xmenu) looks abnormal and the users can't play without rebooting. Figure (10-b) shows after the recovering process with our system.



(a) Corrupted state

(b) Steady state after recovery

Figure 10. The result of experiment on GP2X-WIZ

V. CONCLUSIONS

In this paper, we designed fault tolerant system to cover system behavioral errors and we tested an example using this method. The structure showed that behavioral error can be dealt at runtime using tracing of events based on trace point effectively. However, the static analysis is difficult on some parts of system. And a generalization of method is required. We will try to model the Sound, Network, File system and the other parts of Linux. And we will try the measurements of overhead caused by runtime tracing. We expect that dealing with behavioral errors will be researched continuously in the future.

REFERENCES

- [1] CAMPBELL, R. H., JOHNSTON, G. M., AND RUSSO, V. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)". ACM Operating Systems Review 21, 3 (July 1987), 9.17.
- [2] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio File Cache: Surviving Operating System Crashes. In International Conference on Architectural Support for Programming Languages and Operating Systems (1996), pp. 74.83.
- [3] Francis M. David, Ellick Chan, Jeffrey C. Carlyle, Roy H. Campbell. CurIOS: Improving Reliability through Operating System Structure. USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008.
- [4] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. ACM Transactions on Computer Systems, 22(4), Nov. 2004
- [5] Michael Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. ACM Transactions on Computer Systems 24(4), November 2006
- [6] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. Tenth ACM SIGOPS European Workshop, Saint-Emilion, France, Sept. 2002
- [7] J. S. Shapiro, J. M. Smith and D. J. Farber1, EROS: A Capability System, Computer and Information Sciences Technical Report MS-CIS-97-03