International Journal of Advanced Science and Technology
Vol. 19, June, 2010

KTAS: Analysis of Timer Latency for Embedded Linux Kernel

Ki-Duk Kwon
Department of Computer Science and Engineering, Waseda University, Japan
kwonkiduk@dcl.info.waseda.ac.jp

Midori Sugaya
Dependable Embedded OS Center, Japan Science and Technology Agency (JST)
doly@dependaable-os.net

Tatsuo Nakajima
Department of Computer Science and Engineering, University of Waseda, Japan
tatsuo@dcl.info.waseda.ac.jp

Abstract

These days embedded systems are making great strides with the development of hardware to satisfy user's varied demands. As one can see many kinds of systems operated in our daily life, software becomes dependent on real-time processing functions as well as various functions (for instance, multimedia system or network). These systems have the disadvantage of increasing complexity and are prone to problems when system engineer develop the system. Especially, if problems occur in the kernel layer, a developer needs to spend a lot of time and effort to solve them. In addition, as of now there are not enough tools for solving kernel timer latency problem effectively.

In this paper, we propose a system named Kernel Timer Analysis System (KTAS) that can detect timer problems in kernel. The KTAS find High Resolution Timer latency which is a serious problem for the real-time processing system in kernel layer. It can effectively detect the problem and help finding its cause. In the future, we want to generalize the system to detect other problems and their causes to support the developers.

Keywords: *Embedded Linux kernel, Event log, High resolution timer, System analysis, Timer latency*

1. Introduction

In recent years, the demand for embedded system has increased rapidly. In addition, to do progress in the development of hardware, operating systems (OS) for embedded system have become huge and as complicated as OSes for low efficiency personal computer. In embedded Linux, you can use not only any pre-existing development environment in Linux but also compiler, debugger [12] and library. This maximizes productivity. Moreover, it is possible to transplant Linux software to embedded system easily. This makes that some PC's functions, such as memory protection, security and network, are also available on embedded systems. Since developers for Linux systems are increasing, even for inexperienced developer, it is enough to try to develop the embedded applications. In spite of such a great advantage, there is a possibility that a critical unexpected problem occurs in embedded system because the system is developed in a special hardware environment. Moreover, it leads to negative publicity for the company and tremendous economic damage. For examples, in September

2004, Dime-Chrysler had to recall 1.3 million vehicles due to a shortcoming of embedded electronic braking system. [25]

Among many problems in an embedded kernel, the most important is time latency. In general, most kernel functions like device drivers and user applications are operating with time. Even a short time delay in kernel might cause a critical problem in the whole system. Therefore, it is very important for a developer to analyze time delay problems and solves them.

However, problems of timer latency occur in embedded systems because of two main reasons. First, there are not enough methods for enhancing reliability and accuracy. Second, tools for analysis are vastly insufficient. Because structure of embedded system is becoming more complex, a skilled engineer needs more knowledge for analyzing and fixing performance problems. However, limited skilled engineers' knowledge cannot be a fundamental solution for the problem.

We propose Kernel Timer Analysis System (KTAS) that can effectively find the timer problem and support to find the cause of the problem in the Linux kernel. In this paper, we focus on the methodology and tool development that can help to find the timer latency of real-time Linux. The problems that occur in the kernel can be analyzed more easily and effectively through the use of this proposed system.

The remainder of this paper is as follows. In Section 2, we introduce previous case studies. In Section 3, we discuss problem definition for High Resolution Timer. In Section 4, we describe KTAS. In Section 5, we present the result of experiment using KTAS. Finally, In Section 6, we present conclusions and future work.

2. Related Work

First of all, we begin by inquiring some kernel analysis tools used these days. The most widely used tools in open source are Dtrace [18], Ftrace and SystemTAP [19]. These are very similar to the LTTng event log analysis tool. Also, there are some other tools – LKST, and Mevalet developed by NEC Japan.

SystemTAP uses the time source primitives provided by the Linux kernel. Those are taking a sequence lock to protect non-atomic data structure accesses. This implies that instrumentation coverage cannot include non-maskable interrupt handles NMIs, because a NMI taking a sequence read lock nested over a sequence write lock would deadlock.

Ftrace [20] is a small utility that uses the frysk engine to trace system calls in a similar manner to strace. DTrace [18] is a comprehensive dynamic tracing framework created by Sun Microsystems for troubleshooting kernel and application problems on production systems in real time. Originally developed for Solaris, it has since been released under the free Common Development and Distribution License (CDDL) and has been ported to several other Unix-like systems.

There is LKST (Linux Kernel State Tracer) [11] among event trace tools occurring in the Linux kernel. LKST performs similarly performance to LTTng and it can record event occurring inside of the kernel such as process context switch, signal emission, exception, memory allocation, sending packets, etc. LKST can analyze the problems in the kernel mode [7] and users can expand the performance dynamically.

Another measuring and analysis tool is Mevalet [8]. Mevalet is developed by system software department of NEC. Mevalet inserts hook points inside kernel of embedded systems. Mevalet is also applicable to any computer languages. In addition, Mevalet can analyze every application by measuring the performance from several tens of ns to several tens of s without changing application and the overhead by several percentages.

It is well known that LTTng [9] is a new version of LTT. By using LTTng, we can copy and record the events occurring inside of the kernel such as thread, fork, interrupt, signal, and memory information, etc from the kernel space to user space quickly. In addition to using LTTV (Linux Trace Tool Viewer), we can record and review the event log visually, and the overhead is reduced from 1.54 to 2.28 [3].

Kernel Function Tracer (KFT) [13] and Kernel Function Instrumentation (KFI) [2] are also solving the same problems. Kernel Function Instrumentation has higher overhead than LTTng, but it can log event in more detail.

As mentioned above, existing research can be roughly divided between profiling tools and event-based performance monitoring tools. Profiling tools include gporf, porf, time, top, nmon [22], and oprofile, etc., and event-based performance monitoring tools comprehend LTT, SystemTAP [19], and LKST [11], etc. Such tools monitor and profile the system. But they do not analyze the reason why problems occurred.

Above all, there are various tools that can log events occurring in the kernel, however, the purpose of this paper is different from event tracing. The purpose of this paper is to analyze the kernel problem (timer) easily and effectively by using event tracing and to propose system architecture which can explore the reason of problem.

3. Problem definition for High Resolution Timer

3.1 Background

Recently, embedded Linux focused on real-time applications and time-sensitive applications, which are characterized by temporal constraints. Such applications may require period accuracy where, for example, the period is derived from the frame rate of an audio/video stream [1]. Timers should provide accurate and precise time for the system, but sometimes they cannot afford to cover it. In this case, it causes critical problem for the a whole system.

In a Linux implementation, there are some timers such as PIT, High Precision Event Timer (HPET) [17], and High Resolution Timer (HRTimer) [6] to provide a time service for the Linux applications.

PIT provides information about overtime through occurring timer interrupt when it passes a period set up by the kernel. PIT makes IRQ0 produce something in every 1ms (1000Hz), and send it to every CPU. The name of the timer block has been changed from Multimedia Timer to HPET (High Precision Event Timer). HPET [25] produces periodic interrupts at a much higher resolution than the RTC and is often used to synchronize multimedia streams, providing smooth playback and reducing the need to use other timestamp calculations such as an x86 CPU's RDTSC instruction.

General Linux resolution timer can support up to one millisecond unit; however, the HRTimer could support up for nanosecond unit. HRTimer provides the most highly accurate and precise function for time-sensitive applications. In the next section, we describe the problem of the HRTimer.

3.2 Problem of the High Resolution Timer

Generally, Linux uses periodic timer. Therefore if developers need a timer of high resolution then it needs to raise the clock frequency. On the other hand, HRTimer allows a system to execute callback functions directly from the next event interrupt handler in x86 [4] without increasing the frequency. The software interrupt (softirq) for running the HRTimer queues and executes the callbacks has been separated from the tick bound timer softirq to allow accurate delivery of HRTimer signals. The execution of this softirq can still be delayed by other softirqs. Although the resolutions are different depending on the system, currently it is possible to set up to 10μs in the kernel that supports HRTimer with less overhead. As compared with other timers, HRTimer also operates from interrupt occurrence.

We define the HRTimer latency model as follows:

- T^{lapic} - The period from occurring HRTimer hardware interrupt (hardirq) to occurring hardirq to be expired.
- $T^{softirq}$ - The processing period to latency softirq after the occurrence hardirq.
- $T^{\exp ired}$ - The period of HRTimer (softirq handler) which has expired.

Figure 1 shows the model of timer latency. The accuracy of timer in Linux depends on the accuracy of hardware and software interrupts. Timer interrupts are not occurring accurately when the system is overloaded. It would cause timer latency in kernel. This paper conducts an experiment on finding the HRTimer latency based on proposed KTAS. In this paper, we analyze the reason of the HRTimer latency by analyzing the event log.

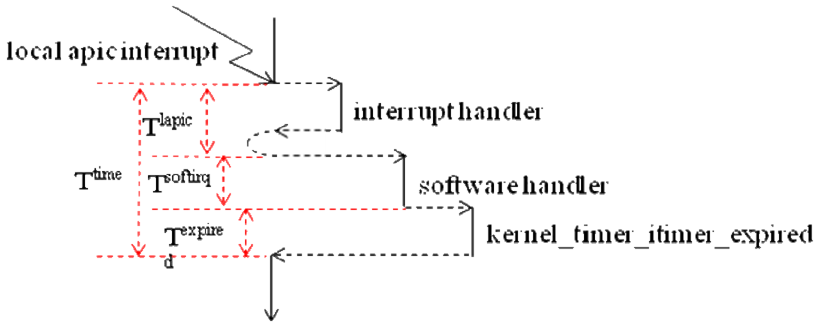


Fig.1 HRTimer latency model

Formula (1) T^{time} means a period of the HRTimer's execution which is the sum of time for event's execution and time latency.

In Formula (2), the HRTimer is checking whether time latency occurred or not by comparing HRT^{tick} to T^{time} (HRT^{tick} is time set by a programmer). If $HRT^{latency} > 0$, the HRTimer consider time latency happened.

$$T^{time} = T^{lapic} + T^{softirq} + T^{\exp ired} \qquad (1)$$

$$HRT^{latency} = T^{time} - HRT^{tick} \qquad (2)$$

4. Kernel Timer Analysis System

As we mention above, most kernel functions are operating with time. For that reason, even a short time delay in kernel might cause a critical problem in a whole system. In real-time system, there are soft real time systems and hard real time systems. The time latency is a significant factor to problems occurrence in hard real time systems compared to soft real-time systems. Therefore, it is important for a developer to analyze time delay problems and solve them.

In this paper, we suggest a system which can analyze kernel events, find out timer problems for kernel and propose an effective solution. Because developing in an embedded system is in cross development environment, it differs from developing in server or PC. Therefore, if a timer problem occurs, more time and effort is needed to fix up in an embedded system compared to in server or PC environment. For a system developer of an embedded system, the system we suggest would enhance the convenience in development and the stability in the system.

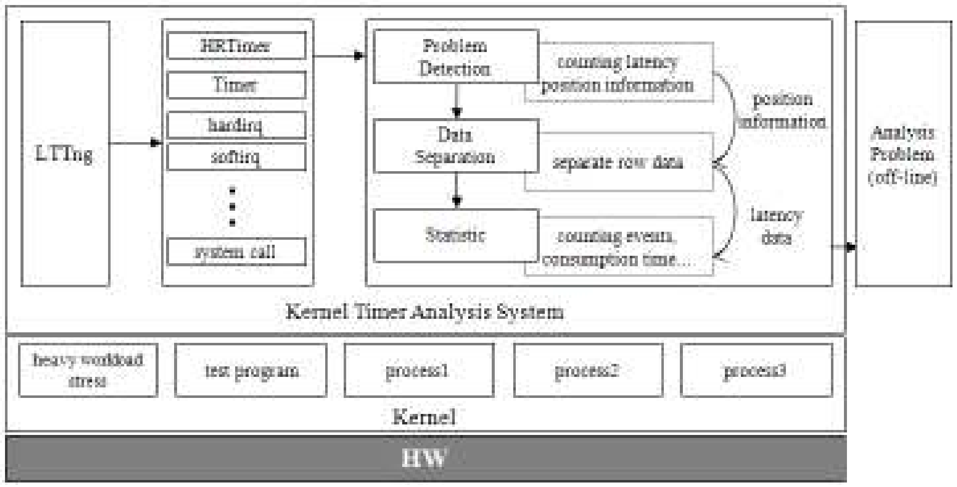


Fig.2 Proposed architecture

To analyze problems that occur in kernel, a solution can be found by analyzing event information. For example, in order to analyze the timer latency, not only timer event but also all the information regarding to events (for example, system call, interrupt, thread, memory etc.) that occurred in kernel must be analyzed. If we want to analyze the specific problem, we have to input the hook point into the kernel source for logging the event information.

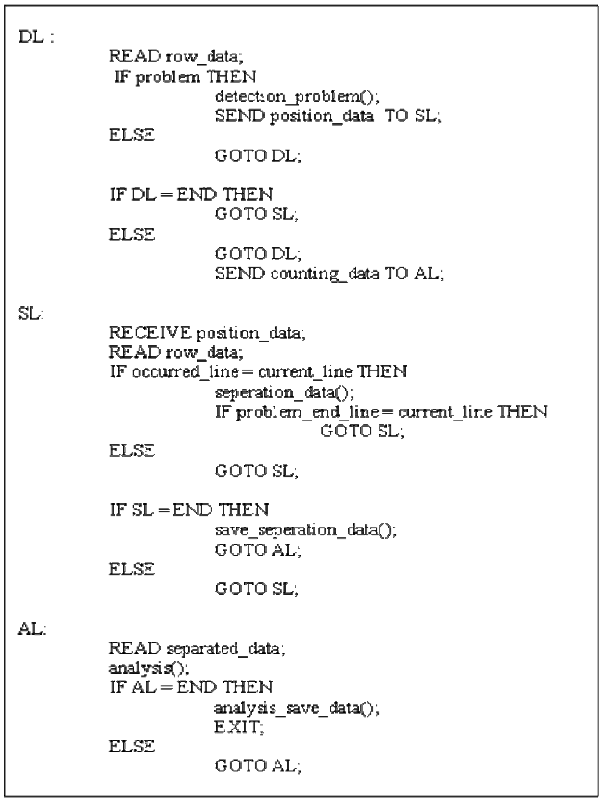
Figure 2 shows the structure of the proposed system. The logging of events occurred in kernel are using LTTng [5]. The logged data can be used to analyze the problems that occur in kernel by using Kernel Timer Analysis System (KTAS) proposed. The KTAS generates the analysis data according to each layer to analyze the source of a problem easily by analyzing an amount of event log.

KTAS can be separated into three major parts.

• Detection Layer (DL): Check whether a problem occurred and count the number of times problem occurs by using event log. Then, save the data's location information and send it to SL.

• Separation Layer (SL): Separate the events of the part problem occurred from the entire log using the data created in the previous step, Detection Layer.

• Analysis Layer (AL): Come up with the statistics for the problem of event execution, the time spent for each event, the overall accumulated time for the part problem occurred. Find the events cause the problem using the statistical information.



```
DL :
        READ row_data;
         IF problem THEN
                detection_problem();
                SEND position_data  TO SL;
        ELSE
                GOTO DL;

        IF DL = END THEN
                GOTO SL;
        ELSE
                GOTO DL;
                SEND counting_data TO AL;

SL:
        RECEIVE position_data;
        READ row_data;
        IF occurred_line = current_line THEN
                seperation_data();
                IF problem_end_line = current_line THEN
                        GOTO SL;
        ELSE
                GOTO SL;

        IF SL = END THEN
                save_seperation_data();
                GOTO AL;
        ELSE
                GOTO SL;

AL:
        READ separated_data;
        analysis();
        IF AL = END THEN
                analysis_save_data();
                EXIT;
        ELSE
                GOTO AL;
```

Fig.3 Pseudo code of KTAS.

In Figure 3, Pseudo code shows the reliance relation between each layer. First, in detection layer, the KTAS checks whether the time latency occurred or not. If it happened, *detection_problem()* function save information of the location and the number of times error occurred. Next, in separation layer, *separation_data()* function separates the events of the part problem occurred by using position_data. After that, *save_separation_data()* function saves the information. Finally, in analysis layer, *analysis()* function analyzes the information, and *analysis_save_data()* function unifies and save the data analyzed. A problem solution can be more easily and effectively found by analyzing the cause of the problem using the results from the three steps defined above.

5. Evaluation of HRTimer using KTAS

This section addresses the specification of experiments set up and evaluation of HRTimer latency. The system is with a 1.83GHz Intel Pentium 4 uni-processor and 1GB RAM, on which is running a Linux kernel 2.6.23.

5.1 Setup for Measurement

This paper attempts to solve the problem by analyzing event log and to explore the reason of problem. First of all, we apply the LTTng patch to the Linux kernel in order to collect event logs. We use *setitimer()* system call to send SIGALRM signal to processor when timer is finished, the function of *setitimer()* occurs interrupts in the process itself at certain future time. We set HRT^{tick} as 100μs and set the cycle of repetition as 10,000 with heavy background load.
The $HRT^{latency}$ analysis based on a loop that:

1. reads the hardirq for HRTimer T^{lapic}
2. reads softirq of HRTimer $T^{softirq}$
3. reads itimer_expired time $T^{\exp ired}$
4. computes formula (1) and formula (2)

5.2 Analysis of problem by using KTAS

The experiment of HRTimer latency is based on the proposed system architecture as shown in Figure 1. There are three layers to analyze the event log explained in Section 3.

Table 1 is a result from Detection Layer. Latency-time was expressed T^{time}- HRT^{tick} , and it means HRTimer latency. We defined equation $HRT^{latency}$ 3 100μs as latency, and record the number of latency where latency-count as Table 1. It records not only the latency-time but also the position information where the latency has occurred.

Table 1: Result of Detection Layer

Execution times (Period of Timer)	***Latency-time(ns)***	***Latency-count***
1	198,298	1
2	99,931	-
3	140,071	2
4	99,973	-
5	106,052	3
…	**…**	**…**
9,998	184,863	343
9,999	99,881	-
10,000	610	-

Table 2 is generated by Analysis layer when the latency has occurred. Analysis layer generated the event name, frequency of execution, and consumption time of each event. Table 2 shows result of analysis layer. In the table, the case (a) means execution times when the time latency did not occur, and the causes (b) means execution times when the time latency occurred by network stress and I/O stress program. By comparing the case (a) to the case (b), we can figure out what event cause the time latency. In the result, the events - *kernel_arch_syscall_entry*, *kernel_arch_syscall_exit*, *mm_page_alloc* - were executed most of the time. Especially, the *mm_page_alloc* event caused the biggest time latency.

Consequently, we can find the events when the timer latency is occurred. As a result, we can analyze HRTimer latency more easily and efficiently by using proposed KTAS.

Table 2: Result of Analysis Layer: (a) Execution times when the time latency did not occur, (b) Execution times when the time latency occurred

Event name	***(a) Execution times***	***(b) Execution times***
Kernel_arch_syscall_entry	37	92
Kernel_arch_syscall_exit	37	91
Net_socket_recvmsg	0	2
Net_socket_sendmsg	0	88
net_dev_xmit	0	9
mm_page_alloc	3	359
mm_page_free	3	20
…	…	…
kernel_softirq_entry	1	6
kernel_softirq_exit	1	6
Kernel_timer_itimer_expired	1	1

Figure 4 shows how HRTimer analyzes HRTimer latency. In the process of executing, between each softirq handler execution, HRTimer softirq (*HRTIMER_SOFTIRQ*) is executed. However, we can find that *run_hrtimer_softirq()* occurred because *net_tx_action* (*NET_TX_SOFTIRQ)* and *blk_done_softing (BLOCK_SOFTIRQ)*, high priority in softirq, have been executing.

Figure 5 indicates the result of experiment of HRTimer latency in Linux-2.6.23 with heavy network and I/O stress. We set the timer to 100,000ns and execute. The result is the gap execution time and 100,000ns. Among 10,000 executions, HRTimer latency happened 343 times. From the result of experiment of KTAS, we set up HRTimer softirq on a higher priority than network and softirq related I/O.

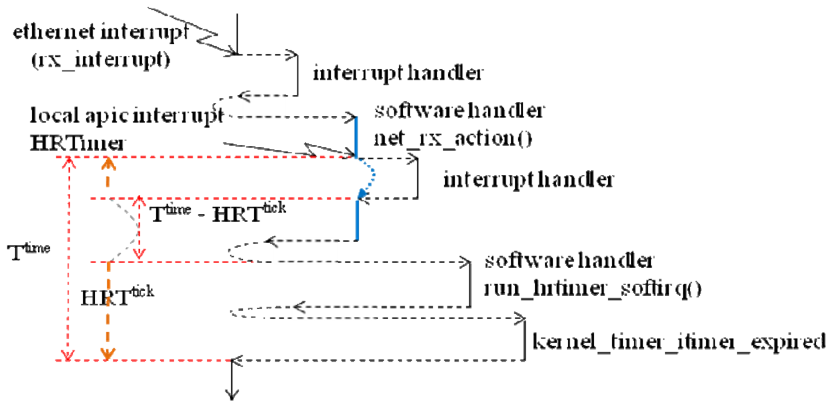


Fig. 4 One of the reasons of HRTimer latency

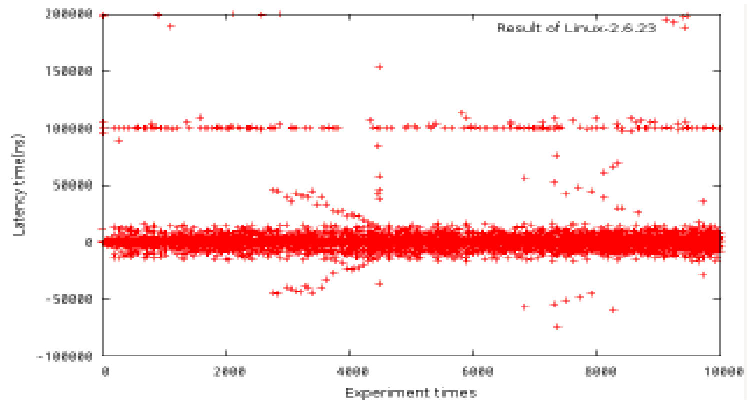


Fig.5 Result of experiment of HRTimer latency in Linux-2.6.23 with heavy background load

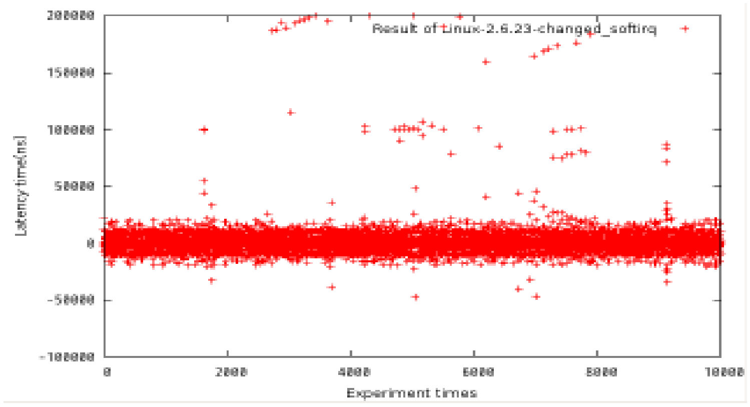


Fig.6 Result of experiment of HRTimer latency load in Linux-2.6.23-changed-softirq with heavy background load

Figure 6 shows the result after changing softirqs. Because *NET_TX_SOFTIRQ*, *BLOCK_SOFTIRQ* are the main reason of the time latency, we changed the priority of *NET_TX_SOFTIRQ*, *BLOCK_SOFTIRQ* and priority of HRTimer's softirq. Among 10,000 executions, HRTimer latency happened only 88 times. From Figure 6, there is not only one reason for latency in the kernel but also complex factors of latencies coming from the kernel; the reasons could be followed by the dependencies with hardirq, softirq, and other processes [10, 15, 16]. It is difficult to solve the problem using one perfect solution. Therefore, if one wants to get a more accurate solution, we can figure out some other cause of the latency by analyzing each process from KTAS more specifically. However, the system performance will be improved, when just one problem is solved and analyzed accurately from the cause of a lot of problems.

6. Conclusion and Future work

KTAS shows one of the solutions to analyze the timer latency occurring in the kernel. The main goal of the experiment of HRTimer latency is to analyze the cause of the latency, and to diminish it. As we can see the result of experiment of HRTimer latency, we cannot solve all of the problems but we can clear up problems in kernel with KTAS.

In the future, we move to focus on improving the proposed system architecture to analyze various reasons of problem accurately and to show the visual data more easily. In addition, we are currently using an offline way to detect problems, we are going to apply real-time way to discover problems and save the data in real-time. This real time KTAS provide solution to detect and analyze the problem in kernel more quickly and accurately.

References

[1] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, A measurement-based analysis of the real-time performance of the Linux kernel. In Real-Time Technology and Applications Symposium (RTAS 2002), Sept. 2002.

[2] Tim Bird, Learning the kernel and finding performance problems with kfi. In CELF International Technical Conference, 2005.

[3] Mathieu Desnoyers and Michel R. Dagenais, The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In OLS (Ottawa Linux Symposium) 2006, July.

[4] Thomas Gleixner, Douglas Niehaus, Hrtimers and Beyond: Transforming the Linux Time Subsystems. Ottawa Linux Symposium 2006.

[5] LTTng project, http://ltt.polymtl.ca/

[6] G.Anzinger, High resolution timers project. http://high-res-timers.sourceforge.net/.

[7] Yaghmour K. and Dagenais M. R., Measuring and characterizing system behavior using kernel-level event logging. In Proceedings of the Annual Technical Conference on USENIX Annual Technical Conference, 13_26, 2000.

[8] System Director mevalet, http://www.nec.co.jp/cced/mevalet

[9] Mathieu Desnoyers and Michel Dagenais, Low disturbance embedded system tracing with Linux Trace Toolkit Next Generation. In ELC (Embedded Linux Conference) 2006.

[10] Martin Bligh, Mathieu Desnoyers and Rebecca Schultz, Linux Kernel Debugging on Google-sized clusters. Proceedings of the Linux Symposium June, 2007, Ottawa, Ontario in Canada.

[11] Linux Kernel State Tracer, http://lkst.sourceforge.net

[12] Debugging with Data Display Debugger, User Guide and Reference Manual First Edition, for DDD Version 3.3.9. 15 January, 2004.An Introduction to the Real-time OS \& Nucleus PLUS Training Guide. Accelerated Technology Inc.

[13] Kernel Function Trace, http://elinux.org/Kernel\Function \Trace.

[14] Nucleus, An Introduction to the Real-time OS & Nucleus PLUS Training Guide. Accelerated Technology.

[15] Yu-Chung and K.-J Lin, Enhancing the Real-Time Capability of the Linux Kernel. In Proceedings of the IEEE Real Time Computing Systems and Applications, Hiroshima, Japan, October 1998.

[16] Mark Wilding and Dan Behman, Self-Linux Mastering -The Art of Problem Determination.

[17] High Precision Event Timers Specification, Intel Corporation.

[18] Bryan M. Cantrill, Michael W.Shapiro, and Adam H.Leventhal. Dtrace: Dynamic instrumentation of production system. In USENIX04, 2004.

[19] SystemTAP: Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In OLS05 (Ottawa Linux Symposium) , 2005.

[20] Mathieu Desnoyers and Michel R. Dagenais, Deploying LTTng on Exotic Embedded Architectures at Embedded Linux Conference 2009.

[21] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, W.-Y. Ma, "Automated Known Problem Diagnosis with Event Traces", Microsoft Research Technical Report MSR-TR-2005-81, Jun. 2005.

[22] Stephen Atkins, IBM pSeries Technical Support, IBM Software Group, http://www.ibm.com/developerworks/aix/ library/au-nmon_analyser/.

[23] Lauterbach, Integrated Run and Stop Mode Debugging for Embedded System, Embedded System Conference 2007.

[24] The Linux Advantage (Join the Linux revolution), Sage Software, Inc.

[25] IA-PC HPET (High Precision Event Timer) Specification, Intel Corporation, 2004.

[26] Embedded World, http://www.embeddedworld.co.kr/english/.

Authors

Ki-Duk Kwon received M.S. degrees in Information and Communication Engineering from University of Hanyang, P.R. South Korea in 2001 and 2003, respectively. Presently, he is a doctoral candidate of the Graduate School of Computer Science, Waseda University. His research interests include operating systems, embedded systems, and the kernel monitoring.

Midori Sugaya is a researcher of Dependable Embedded OS R&D Center, Japan Science and Technology Agency (JST). She has work experience in software industry. She received Master Degree in computer science from Waseda University, Japan, 2004, and also a candidate of the PhD in computer science from Waseda University. Her research interests include operating and dependable systems and proactive fault management system.

Tatsuo Nakajima is a professor in the Department of computer Science, Waseda University. He was a researcher in School of Computer Science, Carnegie Mellon University in 1990-1993, a research engineer in AT&T Laboratories, Cambridge in 1998-1999 and a visiting research fellow in Nokia Research Center, Helsinki in 2005. He was also an associate professor in School of Information Science, Japan Advanced Institute of Science and Technology in 1993-1999. He was a program co-chair of RTCSA 2002 and ISORC 2003, and a general chair of RTCSA 2003 and ISORC 2005. His research interests include distributed systems, operating systems, ubiquitous computing, and information appliances.