An Operating System Architecture for Future Information Appliances

Tatsuo Nakajima, Hiroo Ishikawa, Yuki Kinebuchi, Midori Sugaya, Sun Lei, Alexandre Courbot, Andrej van der Zee, Aleksi Aalto, and Kwon Ki Duk

Department of Computer Science and Engineering
Waseda University
3-4-1 Okubo Shinjuku Tokyo 169-8555, Japan
tatsuo@dcl.info.waseda.ac.jp

Abstract. A software platform for developing future information appliances requires to satisfy various diverse requirements. The operating system architecture presented in this paper enhances the flexibility and dependability through virtualization techniques. The architecture allows a system to use multiple operating systems simultaneously, and to use multi-core processors in a flexible way. Also, dependability mechanisms in our architecture will avoid crashing or hanging a system as much as possible in order to improve the user experience when defects in the software are exposed. We present a brief overview of each component in the operating system architecture and some sample scenarios that illustrate the effectiveness of the architecture.

1 Introduction

Information appliances become more and more complex for supporting a large number of new functionalities. For example, current Japanese mobile phones contain about ten million lines of source code and support a variety of functionalities such as an electronic wallet, a media player, an Internet browser/e-mail, and a photo camera. Other information appliances like televisions and car navigation systems contain almost the same size of software. Moreover, information appliances will need to satisfy diverse requirements for supporting various future services. Also, different types of information appliances will need to take into account diverse hardware platforms. In the near future, multi-core processors will become common although current operating systems for information appliances still have many issues to support the processors.

This paper proposes an operating system architecture to satisfy the diverse requirements for building attractive future information appliances. The uniqueness of the architecture is as follows:

- The architecture accommodates multiple operating systems on a multi-core processor simultaneously. This enables us to reuse a large amount of software on existing operating systems. Also, the number of CPU cores to execute an operating system will be changed dynamically due to energy consumption requirements.

U. Brinkschulte, T. Givargis, and S. Russo (Eds.): SEUS 2008, LNCS 5287, pp. 292–303, 2008.

– The architecture virtualizes crashing or hanging of an operating system as much as possible. The information appliance that adopts the architecture tries to postpone the crash until the user does not interact with the appliance anymore. Thus, the user is not aware of crashing or hanging of the appliance, and the user experience will be improved significantly.

In Section 2, we present some examples of future information appliances, and the requirements for the operating system architecture for building the appliances. In Section 3, we show the structure and components of the architecture. Section 4 shows some example scenarios that illustrate the effectiveness of our architecture. Section 5 concludes the paper.

2 Future Information Appliances and Their Requirements

In the near future, a variety of daily objects near us will become information appliances. These artifacts are connected to the Internet and enhance our daily activities. In our research group, we have enhanced various daily objects such as chairs [1], tables [3], toothbrushes [6], and mirrors [2]. These future information appliances will spontaneously collaborate with each other to compose more complex services from existing services [5] [6]. A variety of middleware infrastructures is necessary to develop various application services rapidly [4].

There are two characteristics to develop future information appliances. The first is to offer a huge amount of functionalities that need to satisfy diverse requirements to offer various attractive services. These diverse requirements cannot be implemented on any one operating system. Current information appliances adopt various types of operating systems to satisfy different requirements. For example, appliances controlling a variety of devices have used small operating systems that include only a real-time thread scheduler and some device drivers. The operating systems usually do not support memory protection domains, but are suitable for highly responsive services with tight timing constraints. The future operating systems architecture should support multiple operating systems running on a multi-core processor simultaneously, without violating real-time requirements of application services.

The second characteristic is diverse hardware platforms. Especially, future information appliances will need to use a multi-core processor dynamically to save energy consumption. As described in the previous paragraph, multiple operating systems should be executed on a multi-core processor. Each operating system allocates a suitable number of CPU cores according to the current workload. Let us assume a mobile phone that uses a multi-core processor. While a user does not use the mobile phone, only one CPU core is used to execute several background application services on multiple operating systems simultaneously. In this case, there are a few activities on these operating systems, and it is easy to satisfy all real-time requirements of these activities on a single CPU core. However, when a user starts watching a TV program, multiple CPU cores become active and most of them are used to process the TV program.

Dependability is one of the most important requirements in future information appliances. Crashing or hanging of a service on an appliance will degrade user experience significantly. For example, if the service is hung, a user needs to find a reset switch and push the switch to restart the appliance. Usually, a user interacts with information appliances for a short time. Although some errors inside a kernel may damage the kernel, usually the appliance can be avoided to crash while a user is interacting with it by repairing the damaged kernel data structure. The kernel will be restarted for a complete repair after a user finishes to use the application service.

3 Operating System Architecture

This section presents an operating system architecture for future information appliances. We describe the structure of the proposed architecture and show why the architecture satisfies the requirements described in the previous section.

3.1 Overview of Architecture

Figure 1 shows the structure of our operating system architecture. The architecture consists of six components. The first component is the SPUMONE hardware abstraction layer. The second is the L4 micro-kernel [10]. The third is the ArcOS dependable real-time operating system. The fourth is the monitoring service, and the fifth one is the anomaly detection service. The last one is the Linux kernel.

The SPUMONE hardware abstraction layer runs on a multi-core processor, and it is an infrastructure to satisfy the diverse requirements described in the previous section. SPUMONE takes into account the diversity of application services and hardware platforms.

The L4 micro-kernel is a small real-time kernel. The kernel offers only basic functionalities such as process and memory management. Most of operating system services such as a file service and device drivers are implemented in independently schedulable protection domains. Also, L4 offers a powerful IPC mechanism that makes it is easy to develop application services decomposed into multiple protection domains.

The ArcOS dependable real-time operating system runs on the L4 micro-kernel. Its purpose is to increase the dependability of control processing services running on ArcOS. The most important concept in ArcOS is self-healing. The integrity of a system is maintained without help of a system administrator, and is recovered automatically when the inconsistency inside the application service is exposed.

The monitoring service is running on ArcOS. It maintains the integrity of the Linux kernel to enhance its dependability, and recovers the integrity automatically by observing critical data structures inside the Linux kernel. The monitoring service is isolated from the Linux kernel to avoid the effect of the inconsistency in the Linux kernel.

The anomaly detection service is implemented in the Linux kernel. It enhances the dependability of the Linux application services. The service uses the

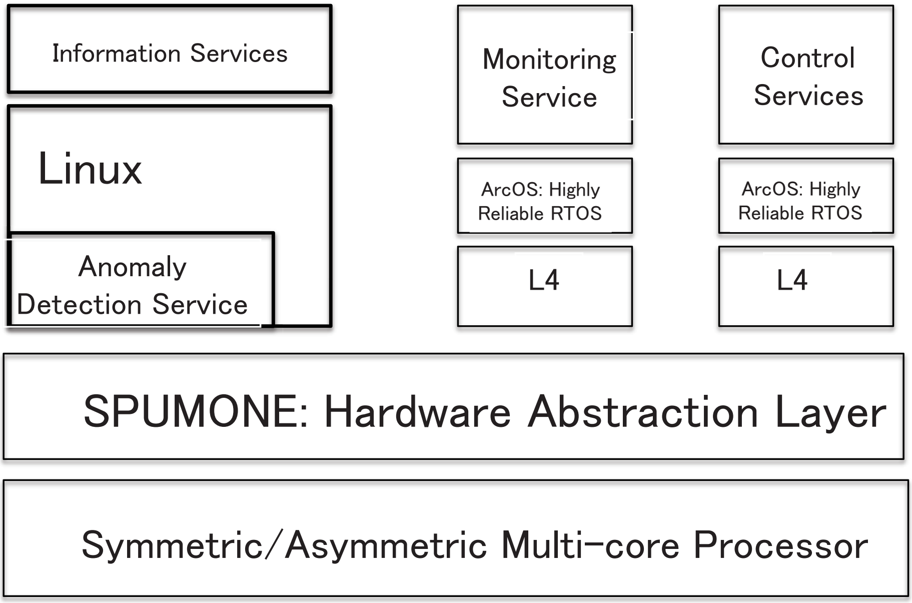
Information Services
Linux
Anomaly Detection Service
Monitoring Service
ArcOS: Highly Reliable RTOS
L4
Control Services
ArcOS: Highly Reliable RTOS
L4
SPUMONE: Hardware Abstraction Layer
Symmetric/Asymmetric Multi-core Processor

Fig. 1. An Overview of Our Operating System Architecture

monitoring and tracing facilities offered by the Linux kernel to detect anomalies in application services.

The architecture also improves user experience when an anomaly in a system occurs. ArcOS increases the robustness of the monitoring service, and the monitoring service repairs the anomaly in the Linux kernel. When SPUMONE detects crashing or hanging in Linux, it converts them to recoverable errors that can be handled in applications services. However, we do not assume that the monitoring service repairs the damage of the kernel completely. Thus, the monitoring service will restart the Linux kernel while a user does not interact with an appliance. The anomaly detection service detects the anomaly in application services on Linux, and restarts the abnormal services to maintain the integrity of the services.

3.2 SPUMONE: Hardware Abstraction Layer

SPUMONE offers the abstraction called virtual processors to satisfy the diverse requirements by using multiple operating systems as described in the previous section. One or multiple virtual processors are assigned to each guest OS. If a guest OS is configured as an SMP operating system, multiple virtual processors may be multiplexed on a single CPU core or be executed on different CPU cores. An instance of SPUMONE is created on each CPU core and may schedule multiple virtual processors. For reducing the overhead of switching virtual processors, both guest operating systems and SPUMONE are located in the same privileged address space. Thus, each guest OS can invoke privileged instructions without virtualizing them.

A guest OS uses its own process scheduler. When the OS becomes idle, SPUMONE changes the status of the virtual processor executing the OS to passive, and the virtual processor will not be selected to be executed until an interrupt will change its status to active. The physical memory in a processor is divided into multiple memory areas, and each area is assigned to a different guest OS. The memory area used by the guest OS can be protected by other guest OSes for avoiding malicious memory access.

Each virtual processor is assigned a different priority, and SPUMONE chooses an active virtual processor that has the highest priority to be executed on a CPU core. The number of virtual processors used by each guest OS is statically determined when the system is booted. Also, the number of virtual processors on each CPU core is statically determined. For reducing the power consumption, SPUMONE can be stopped and resumed independently on each CPU core, so the number of CPU cores can be dynamically changed according to the system's power consumption policy. Also, a guest OS may change the status of its virtual processor to idle dynamically according to the current condition of the processor for satisfying real-time constraints of application services on multiple operating systems.

An interrupt or exception from a hardware device is interposed by SPUMONE. The mapping between interrupt sources and virtual processors is statically specified in SPUMONE. In traditional OSes, an interrupt processing is scheduled before the execution of all processes. However, in SPUMONE, low priority interrupt processing can be delayed while executing a virtual processor with a high priority. This means that the priority of interrupt processing is integrated with the priorities of virtual processors. Thus, an interrupt with a low priority does not affect the execution of a virtual processor with a high priority.

SPUMONE has three mechanisms to increase its flexibility. The first mechanism is the priority integrity mechanism. To guarantee different guest OS's timing constraint requirements, each guest OS changes the priority of its virtual processor dynamically according to the priority of the executing process. When a CPU core multiplexes several virtual processors, the priority of the executing process is mapped to its virtual processor. Thus, the virtual processor on which a guest OS runs to schedule a process with highest priority is selected to be executed on a CPU core. The second mechanism is a CPU accounting mechanism to limit the CPU capacity used by each virtual processor. This is useful to protect the capacity of a CPU core from the monopolization by a specific guest OS. The third mechanism is a mechanism to virtualize errors occurring on each CPU core. When SPUMONE detects hanging or crashing on the CPU core, it changes the current program counter and stack pointer to the initially specified value to convert the errors to easily recoverable errors. For example, when hanging or crashing is detected during the execution of a system call, the system call raises an exception to an application process, and the process will recover from the exception by re-executing the system call or executing an alternative system call.

Currently SPUMONE runs on a single core SH4A architecture processor. Linux, L4 and TOPPERS are running on SPUMONE and can be run concurrently on a single CPU core. We are working on supporting a multi-core processor that contains four SH4A CPU cores.

3.3 ArcOS: Dependable Real-Time OS

ArcOS is an operating system that has an ability of automatic self-healing. In ArcOS, each software component is implemented in an isolated protection domain that can be independently restarted when hanging or crashing occurs in these components. ArcOS is a multi-server operating system built on top of the L4 micro-kernel. System components such as a file system and device drivers run in independently isolated protection domains, and are also restarted in case these system components violate their integrity and the violation is exposed by some errors.

The most fundamental functionalities of ArcOS are involved in the root servers that we assume to be highly reliable. P3, which is one of the root servers, is a dedicated memory manager of each software component. An error detector and a persistent storage mechanism described below are implemented in P3, so that each component can use the mechanism as an infrastructure service.

A component implemented on ArcOS tries to recover an error caused by the anomaly through various exception handling strategies. Also, ArcOS tries to virtualize the error to convert the unrecoverable error to the recoverable error if possible. Since the aggressive recovery strategies used in ArcOS may not resolve the inconsistency in the component, the inconsistency should be completely removed by scheduling the restart of the component when the effect of the restart becomes minimal. In case the exposed error cannot be recovered with any aggressive recovering strategies, ArcOS will restart the component immediately.

Since each service is decomposed into multiple software components in ArcOS, the integrity of a system is maintained by restarting respective components using the micro-rebooting technique [8]. The most important issue when using the micro-rebooting technique is to quickly restart a software component to reconstruct internal data structures. Candea et. al.[8] have proposed crash-only software to make is easy to build highly reliable Internet services. However, for building information appliances, it takes a long time to reconstruct data structures in a restarted component if it is implemented as crash-only software. ArcOS assumes that each component stores some critical data in a persistent storage. Also, the component defines a recovery procedure, which is invoked when the component is restarted. The procedure reconstructs internal data structures quickly by retrieving data from the persistent storage. The persistent storage stores critical data structures that may take a long time to recover. Also, session data maintained by system services are stored in persistent storage. For example, a file service keeps data about currently opened files for reconstructing the data when it is restarted. Some errors caused by the anomaly in a component may make the data in the persistent storage inconsistent. Since transactional updates are not suitable to implementing various services on information appliances, ArcOS virtualizes the errors to prevent data in the persistent storage from becoming inconsistent. A component continues to execute even when errors are exposed after the completion to access data structures and the consistency is maintained.

ArcOS offers three mechanisms to maintain the system's integrity. The first mechanism is to invoke a recovery procedure before a software component is restarted. As described in the previous paragraph, the mechanism enables the component to be recovered very quickly. The second mechanism is a persistent storage that is used to store critical data for supporting quick restarting. The last mechanism is a failure detection service that detects a crash or hang in a component and restarts it. ArcOS also offers a programming framework that hides details to use the persistent storage to develop self-healing application services in an easy way.

In the current implementation, we have developed a couple of device drivers on ArcOS to show the effectiveness. The device driver is a source of the fragileness of the current operating systems. As described in previous research [11], it is not easy to develop self-healing device drivers even when using micro-kernel-based operating systems. ArcOS enables various device drivers to be recovered quickly and makes it possible to use the micro-rebooting technique in information appliances.

3.4 Monitoring Service

The motivation for our monitoring service is driven by handling system anomaly and security attacks. When information appliances will become more complicated, they may behave in a strange or unexpected way due to undisclosed bugs or faults that are not handled correctly, which is known as system anomaly. Currently once the anomaly of information appliances occurs, it is difficult to perform either anomaly analysis or further recovery. In a similar way, information appliances suffer from security attacks. Virus programs might inject malicious codes into the target host OS through the Internet and then compromise it. Because most of the end users lack enough technical knowledge, they usually cannot solve such security problems themselves and even cannot notice that the system has been compromised. The monitoring service is designed for satisfying the above requirements by providing both the inconsistency detection inside the OS kernel and the automatic recovery mechanism.

Conventional solutions usually suffer from the high memory overhead. In prior researches, the backward-recovery technique takes snapshots at checkpoints to perform a recovery from some fatal errors. On every checkpoint, the system will make a memory snapshot of some specific processes, which introduces the overhead of memory resources. In signature-based intrusion detection systems, a large mount of persistent memory has been used to track suspicious activities. Moreover, there will be more overhead when the monitored object behaves more complicated and nondeterministically, e.g. in the Linux kernel. Current information appliances are still limited by system resources to reduce its costs. Obviously, the above solutions are neither suitable for developing information appliances, nor for the detection and recovery of the anomaly in the Linux kernel. Moreover, our monitoring service is more light-weight.

The monitoring service observes the integrity in the Linux kernel by monitoring the consistency of its critical data structures. When the service detects an inconsistency, a repair function is invoked. A similar technique described in [7] is used for repairing consistent data structures. In the repair function, we do not assume that the consistency is completely recovered. There is always the possibility that inconsistencies like memory-leakage remains. The consistency will be recovered completely after the entire Linux kernel is restarted while a user does not interact with an appliance.

Since the monitoring service is implemented as quickly restart-able components by using the ArcOS framework, each component of the service can be efficiently restarted automatically when a component crashes or hangs. The monitoring service is developed as several Linux kernel modules and a single independent inconsistency detection module running on the L4 micro-kernel and ArcOS. The kernel modules are in charge of the kernel data structure repair. The inconsistency detection module can access the memory of the Linux kernel at runtime through shared memory between the Linux kernel and the inconsistency detection module. The Linux kernel's internal critical data structures are periodically checked for consistency against the built-in anomaly detection database. Once inconsistency in some data structures has been detected, the corresponding repair functions will be invoked. Since the inconsistency detection module runs outside the Linux kernel, and it is completely isolated and hidden from Linux, the faults or bugs inside the Linux kernel does not affect the detection process running on ArcOS.

Currently, several case studies have been carried out to show the effectiveness of the monitoring service, such as the kernel-level hidden process detection and memory leak recovery.

3.5 Anomaly Detection Service

The role of the anomaly detection service is to detect anomalies in a Linux application service before the inconsistency caused by the anomaly is exposed. The user level application services implemented on the Linux kernel becomes more and more complex in future information appliances. These application services may misbehave temporally or can be attacked by malicious programs. For making the behavior of the services more stable, detecting the anomaly of the services is very important.

The service records the fine-grained CPU resource usage of each process and a variety of detailed events traced inside the Linux kernel such as invoking system calls or occurring interrupts and exceptions. The information allows the anomaly detection service to analyze the behavior of applications services without modifying them. For example, the communication patters among user processes can be extracted by analyzing the kernel events traced in the network protocol module transparently. The approach is similar to Magpie [13], but our approach also uses the fine-grained resource usage to make it easy to reduce the overhead to analyze a large amount of kernel events.

The anomaly detection service can be used to detect abnormal behavior in application services, and increase the dependability of a system significantly by restarting abnormal services before the violation of the integrity becomes serious. It learns the normal pattern of the behavior of a target application service by using a variety of machine learning techniques. When it detects the abnormal pattern, the target service is restarted to recover its integrity. For example, the fine-grained CPU resource usage of each process may be used to detect intrusion inside an information appliance. Also, by using a similar approach described in [12], the service detects the anomaly caused by software bugs in application services. The approach may enable us to detect unknown anomaly by classifying abnormal behavior from normal behavior.

The anomaly detection service consists of four modules. The first module is a kernel tracing module that traces a variety of events inside the Linux kernel. Currently, LLTng [14] is used as the kernel event tracing module. The second module is an accounting system that calculates the fine-grained CPU resource usage for each process [15]. Of course, the information can be extracted from the information generated by the kernel event tracing module, but our approach can reduce the overhead caused by analyzing a large amount of kernel events, and it is more suitable for developing information appliances. The third module is a logging module that records the information generated in the first and second modules. The fourth module is an analysis module that is a set of user processes retrieving a model from the logs stored in the logging module. A different process is implemented to retrieve a different model from the logs. When the analysis module detects abnormal behavior, the causing process will be restarted to recover the integrity of an information appliance.

4 Sample Scenarios

The section presents several sample scenarios showing the effectiveness of the proposed operating system architecture.

Robust Control Processing Service: ArcOS enables us to develop robust control processing services. The services can be decomposed into multiple components and each component can be restarted independently. By using the framework offered by ArcOS, a service recovers its integrity by restarting it when an inconsistency is detected.

The execution of Linux is isolated from the execution of a control processing service by SPUMONE. SPUMONE configures the execution in Linux to be postponed when the execution of control processing services is active. The approach makes it possible that the temporal effect of Linux does not affect the execution of control processing services. Also, restarting Linux does not affect the execution of the control processing services. For example, the execution of a continuous media processing service on ArcOS does not violate its timing constraints even when Linux is restarted.

Virtual Dependability: If a user is not aware of the restart of an appliance, the appliance becomes virtually dependent on the user. The monitoring service and

SPUMONE repair the Linux kernel's internal integrity to continue the execution even when a crash or hang in the Linux kernel occurs. However, since our repair takes an optimistic approach, the inconsistency that does not affect the execution of the kernel for a short time will remain in the kernel. The inconsistency can be removed completely by restarting the kernel. SPUMONE schedules the reboot of the kernel when a user does not interact with an appliance.

SMP Emulation on SPUMONE: Interrupt processing time can be very long in the current Linux kernel when burst network traffic is received while a single CPU core is used. This violates the timing constraints of real-time applications. Our approach to solve the problem is that SPUMONE creates two virtual processors on a single CPU core, and switches the contexts of the virtual processors in a fine-grained way.

In this approach, one virtual processor executes all interrupt processing, and another virtual processor executes all other activities. Thus, when a virtual processor starts to execute a long processing interrupt in a high priority, the processing is blocked when the interrupt processing time exceeds the specific threshold. The CPU accounting mechanism in SPUMONE makes the configuration easy.

Using a Multi-core Processor: SPUMONE enables each CPU core to be turned on or off dynamically according to the current workload in Linux and the policy of the power management. For example, when the workload of an appliance is very low, one CPU core is active to execute both Linux and L4. On the other hand, if the workload is high, all CPU cores will be used to process the workload.

When several application services on multiple operating systems need to ensure their timing constraints, the integrity of priorities should be maintained. SPUMONE supports two approaches to maintain the priority integrity. The first approach is to coordinate all priorities in multiple operating systems by mapping them in global priorities. The approach requires taking into account all real-time activities on the multiple operating systems to ensure their timing constraints. The second approach uses different CPU cores for executing multiple operating systems. When Linux starts a real-time application, Linux and L4 use different CPU cores to schedule their applications independently. In this case, since different operating systems use different CPU cores, each operating system can schedule real-time activities without considering the real-time activities in other operating systems.

5 Conclusion and Future Directions

This paper presents an operating system architecture for future information appliances. Currently, we are implementing the operating system architecture on several hardware platforms that use Hitachi SH4 processors. One of them contains a multi-core processor with four SH4 processors.

There are a couple of future directions in our research. Of course, one of the most important directions is to build an actual information appliance using the proposed architecture. We are planning to build a simple audio player on ArcOS in such a way that Linux can be rebooted anytime without disturbing the execution of the audio player.

The second future direction is to use the monitoring service to monitor the hardware devices. This is very useful to detect various abnormal conditions of the appliance. For example, if the source of the anomaly is correctly localized, it may be possible to replace the damaged hardware easily. This makes the model-based diagnosis[9] possible. Also, if the damage is serious and has the potential to cause serious a accident in the near future, the appliance disables to avoid the risk of an accident. This is very useful to preserve sustain-ability.

The last future direction is to implement ArcOS on Linux. It makes it easy to decompose an application service into multiple processes. When the anomaly detection service detects abnormal processes, the processes are restarted to recover the integrity of the service. However, current Linux does not support sufficient functionalities to implement ArcOS. Thus, we will add some system calls for solving the problem.

References

1. Yamabe, T., Fujinami, K., Nakajima, T.: Experiences with Building Sentient Materials Using Various Sensors. In: Proceedings of 24th International Conference on Distributed Computing Systems Workshops (IWSAWC 2004) (2004)
2. Fujinami, K., Kawsar, F., Nakajima, T.: AwareMirror: A Personalized Display Using a Mirror. In: Proceedings of International Conference on Pervasive Computing (Pervasive 2005) (2005)
3. Iwasaki, S., Hirakawa, Y., Mase, H., Tokunaga, E., Nakajima, T.: Towards computer-supported face-to-face knowledge sharing. In: Extended Abstracts Proceedings of the 2006 Conference on Human Factors in Computing Systems (2006)
4. Nakajima, T., Fujinami, K., Tokunaga, E., Ishikawa, H.: Middleware design issues for ubiquitous computing. In: Proceedings of the 3rd International Conference on Mobile and Ubiquitous Multimedia (MUM 2004) (2004)
5. Nakajima, T., Satoh, I.: A software infrastructure for supporting spontaneous and personalized interaction in home computing environments. Personal and Ubiquitous Computing 10(6), 379–391 (2006)
6. Nakajima, T., Lehdonvirta, V., Tokunaga, E., Kimura, H.: Reflecting Human Behavior to Motivate Desirable Lifestyle. In: Proceedings of The 6th ACM Conference on Designing Interactive Systems (DIS 2008) (2008)
7. Demsky, B., Rinard, M.C.: Goal-Directed Reasoning for Specification-Based Data Structure Repair. IEEE Transactions on Software Engineering 32(12) (2006)
8. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot - A Technique for Cheap Recovery. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004) (2004)
9. Peti, P., Obermaisser, R., Ademaj, A., Kopetz, H.: A Maintenance-Oriented Fault Model for the DECOS Integrated Diagnostic Architecture. In: Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005) (2005)
10. L4 eXperimental Kernel Reference Manual, Version X.2, Revision 6, System Architecture Group, Department of Computer Science, Universität Karlsruhe (2006)

11. Herder, J.N., Bos, H., Gras, B., Homburg, P., Tanenbaum, A.S.: Failure Resilience for Device Drivers. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007) (2007)
12. Chen, M.Y., Accardi, A., Kiciman, E., Patterson, D., Fox, A., Brewer, E.: Path-based failure and evolution management. In: Proceedings of the 1st, USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004) (2004)
13. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for Request Extraction and Workload Modelling. In: Proceedings of the International Symposium on Operating Systems Design and Implementation (OSDI 2004) (2004)
14. Desnoyers, M., Dagenais, M.R.: The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In: Proceedings of the Ottawa Linux Symposium (2006)
15. Sugaya, M., Oikawa, S., Nakajima, T.: Accounting System: A Fine-Grained CPU Resource Protection Mechanism for Embedded System. In: Proceedings of the 9th IEEE International Symposium on Object and Component-oriented Real-Time Distributed Computing (ISORC 2006) (2006)