Composition Kernel: A Multi-core Processor Virtualization Layer for Rich Functional Smart Products

Tatsuo Nakajima, Yuki Kinebuchi, Alexandre Courbot, Hiromasa Shimada, Tsung-Han Lin, Hitoshi Mitake

Composition Kernel: A Multi-core Processor Virtualization Layer for Rich Functional Smart Products

Tatsuo Nakajima Yuki Kinebuchi Alexandre Courbot Hiromasa Shimada
Tsung-Han Lin Hitoshi Mitake

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

Abstract. Future ambient intelligence environments will embed powerful multi-core processors to compose various functionalities into a smaller number of hardware components. This makes the maintainability of intelligent environments better because it is not easy to manage massively distributed processors.
A composition kernel makes it possible to compose multiple functionalities on a multi-core processor with the minimum modification of OS kernels and applications. A multi-core processor is a good candidate to compose various software developed independently for dedicated processors into one multi-core processor to reduce both the hardware and development cost. In this paper, we present SPUMONE which is a composition kernel for developing future smart products.

1 Introduction

Multi-core processors are being increasingly adopted for embedded systems because they improve performance, power consumption and lower development cost. Composing multiple operating systems on a multi-core processor enhances the reusability of software when developing rich functional embedded systems. For example, a new product may require to use the new version of an operating system, but to ensure the compatibility with legacy software, the old version may also need to be present. Multiple OS environments enable the product to use two versions of an operating system at the same time. In order to build multiple OS environments, a virtualization layer specialized for embedded systems is necessary, since most of processors for embedded systems support only two protection levels, and there is no hardware support for virtualization. In traditional approaches, an OS kernel runs at the user level to isolate the respective OS kernels, but this approach requires heavy modifications to the guest OSes. Especially, device drivers need to be rewritten and may degrade the performance significantly. Therefore existing solutions is not preferred by the embedded system industry.

In this paper, we propose a composition kernel where multiple OS kernels are running on top of a very thin hardware abstraction layer. The hardware abstraction layer multiplexes underlying physical processor cores into virtual cores

which can be dynamically migrated among the physical cores. A composition kernel can reduce the engineering cost of developing an embedded system by reusing existing OS kernels and application with minimum modification. It also supports real-time interrupt responsiveness, a feature that is difficult to support for large and highly functional monolithic OS, like Linux and Windows. In addition, flexible virtual core migration can help reducing the power consumption of the processor.

Our project is developing SPUMONE which is a composition kernel for embedded systems, and currently focuses on the following three issues.

- Mapping and scheduling of virtual cores on physical cores dynamically to balance the tradeoff among real-time constraints, performance and energy consumption.
- Reducing interrupt latency without degrading real-time performance in a single and multi-core processor.
- Detecting the integrity violations in OS kernels, and repairing them by rebooting the kernels independently.

SPUMONE offers a scheduling algorithm to execute a general purpose OS without affecting the timing constraints of real-time OSes. Also, the execution of general purpose OSes should utilize the maximum performance of multi-core processors. However, when the system load becomes low, SPUMONE reduces the number of used physical cores by migrating virtual cores to a small number of physical cores. The unused physical cores can be turned off to reduce the power comsumption. The overview of multi-core resource management is described in Section 4.1.

For satisfying timing constraints of real-time OSes, SPUMONE carefully coordinates interrupt handling to reduce the effect of disabling interrupts in a single processor case. However, in a multi-core processor case, SPUMONE migrates a virtual core that executes a general purpose OS on another physical core when a real-time OS becomes runnable. Also, lock holder preemption is a serious problem to run an SMP OS kernel on SPUMONE. The migration of virtual cores can solve the problem as described in Section 4.2.

A monitoring service increases the security and reliability of the entire system. On SPUMONE, each OS kernel can be rebooted independently when the kernel is crashed. The monitoring service enables a kernel to be rebooted proactively. When the monitoring service detects an anomaly in the kernel by checking the integrity of some of its data structures, it tries to restore the integrity of these data structures. If the anomaly cannot be fixed, the monitoring service will reboot the OS kernel to recover its integrity completely. This approach can be used to remove kernel rootkits that modify the behavior of operating system kernels. The overview of reliability and security issues in SPUMONE is described in Section 4.3.

2 Motivation

In the near future, a variety of daily objects near us will become smart products. 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, tables, toothbrushes, and mirrors [12]. These products have the surfaces to encourage people to motivate desirable behavior [6, 7], or provide the economic incentives [10]. This offers us a big opportunity to make traditional products more attractive.

There are two characteristics to develop these future smart products. 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 only one operating system. Current smart products adopt various types of operating systems to satisfy different requirements. For example, products 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 implementing highly responsive services with tight timing constraints.

Diverse hardware platforms are the second characteristic. Especially, future smart products 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, it is easy to satisfy all real-time requirements of these activities on a single CPU core by migrating all operating systems on the 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. The mapping the execution of operating systems and physical CPU cores should be flexible according to the current workload.

Dependability is one of the most important requirements in future smart products. 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 button and push it 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, the appliance can usually be avoided to crash while a user is interacting with it by repairing a small amount of damaged kernel's data structure. The kernel will be restarted for achieving a complete repair after a user stops to use the appliance.

3 Composition Kernel: SPUMONE

Multi-core processors will become more and more common for future information appliances. Composing multiple OS functionalities is a promising approach to use

multi-core processors effectively. As the number of functionalities increases, the system requires more computation power to execute them without violating their performance requirements. Since different functionalities are often implemented on different operating systems, an underlying software platform needs to execute multiple operating systems without having to reimplement them on one single operating system. This approach enables a sytem to reuse existing application and operating system code. Thus, it allows to develop smart products with rich functionalities easily and at a low cost.

There are several traditional approaches to execute multiple operating systems on a processor in order to compose multiple functionalities. Microkernels execute guest operating system kernels at the user level. In this case, various privileded instructions, traps and interrupts need to be virtualized by replacing their code. Also, since operating system kernels are to be executed as user level tasks, application tasks need to communicate with the operating system kernel via inter-process communication. Moreover, when emulating Unix-based operating systems, implemeting signals using this approach is very hard. Therefore, the operating system needs to be modified to a significant amount. Virtual machine monitors are another approach to execute multiple operating systems. If a processor offers a hardware virtualization support, all instructions that need to be virtualized trigger traps to the virtual machine monitor. This makes it possible to use the operating system without any modification. But, if the hardware virtualization support is incomplete, some instructions still need to be complemented by replacing some codes to virtualize them.

Most of processors used for embedded systems only have two protection levels, and MMU cannot usually be used in the kernel address space. So, when operating system kernels are located in the kernel address space, they are hard to be isolated. On the other hand, if the operating system kernels are located in the user address space, the kernels need to be modified siginificantly. Most of embedded system industries prefer not to modify a large amont of the operating systems' code, so it is desirable to put them in the kernel address space. Also, the virtualization of MMU requires significant overhead if the virtualization is implemented by software[1]. Therefore, we need altenative mechanisms to ensure the security and reliability of the kernels.

In a traditional virtual machine monitor, handling I/O devices causes a significant problem. When isolating device drivers from operating systems, the engineering cost is very serious. I/O ports can be emulated by virtual machine monitors, but such an approach causes serious performance degradation. If microkernels are used, device drivers can be implemented in user-level OS kernels. In this case, it is possible to allow the OS kernels to access kernel memory through DMA. Although device drivers can be separated from the OS kernels, the drivers need to be re-implemented with a significant engineering cost. Finally, device drivers can be implemented inside virtual machine monitors or microkernels, but this approach decreases the reliability and security due to potential bugs in device drivers.

[1] Xen incurs 30% overhead if MMU is virtualized using the shadow paging

In order to execute multiple operating system kernels on a multi-core processor, the assignment of OS kernels to physical cores should be taken into account. The underlying platform offers virtual cores to OS kernels. Virtual cores are used to schedule all activities in the OS kernels, and the mapping between physical cores and virtual cores should be completely transparent to OS kernels without increasing engineering cost.

In [11], Armand and Gien present several requirements for hardware virtualization for embedded systems:

- Run an existing operating system and its supported applications in a virtualized environment, such that modifications required to the operating system are minimized (ideally none), and performance overhead is as low as possible.
- It should be straightforward to move from one version of an operating system to another one; this is especially important to keep up with frequent Linux evolutions.
- Reuse native device drivers from their existing execution environments with no modifications.
- Support existing legacy often real-time operating systems and their applications while guaranteeing their deterministic real-time behavior.

In our project, we are developing a composition kernel called SPUMONE. SPUMONE (Software Processing Unit, Multiplexing ONE into two or more) is a thin software layer for multiplexing a single physical CPU core into multiple virtual ones. In this section, several characteristics are shown as follows.

Virtualization Strategies

Unlike typical hypervisors or virtual machine monitors, SPUMONE itself and OS kernels are executed in the privileged mode as mentioned in the previous section. Executing SPUMONE and OS kernels in the privileged mode contributes to minimize the overhead introduced to and the amount of modifications required to the OS kernels. Furthermore it makes the implementation of SPUMONE itself simple. Executing OS kernels in the user mode is known to complicate the implementation of the virtualization layer, because various privileged instructions need to be emulated. In our approach, the majority of the kernel and application instructions, including the privileged instructions, are executed directly by the real CPU core, and only a minimal set of instructions are emulated by SPUMONE. These emulated instructions are invoked from the OS kernels using simple function calls. Since the interface has no binary compatibility with the original CPU core interface, we simply modify the source code of OS kernels, a method known as the paravirtualization.

For isolating multiple operating systems, if it is necessary, SPUMONE assumes that underlying processors support the mechanisms to protect physical memories used by respective operating systems like VIRTUS [4]. The approach may be suitable for enhancing the reliability of the OS kernels on SPUMONE without increasing significant overhead.

SPUMONE does not virtualize IO devices because traditional approaches incur significant overhead that most of embedded systems could not tolerate. In

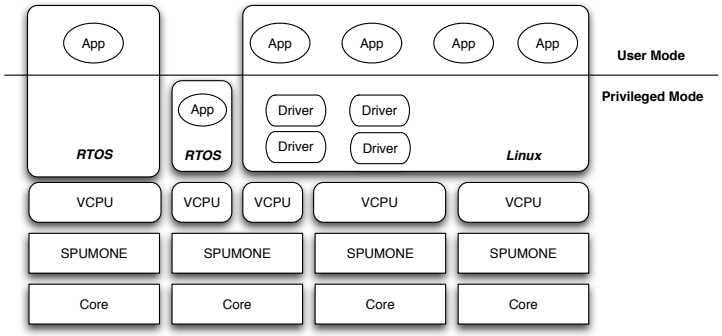

Fig. 1. Composition Kernel

SPUMONE, since device drivers are implemented in the kernel address space, they do not need to be modified when the device is not shared by multiple operating systems.

Interrupt/Trap Delivery

Interrupt virtualization is a key feature of SPUMONE. Interrupts are investigated by SPUMONE before they are delivered to each OS. SPUMONE receives an interrupt, then looks up the interrupt destination table to make a decision to which OS it should be delivered. The destination virtual core is statically defined for each interrupt when the kernels are built. Traps are also delivered to SPUMONE first, then are directly forwarded to the currently executing virtual core.

The interrupt delivery process on a multi-core platform works basically like the one on a single-core platform. Each SPUMONE instance delivers interrupts to their destinations. In order to deliver interrupts to a virtual core running on a different core, the assignments of interrupts and physical cores are switched along with virtual core migrations.

Virtual Core Scheduling

A CPU core is multiplexed by scheduling the execution of virtual cores. The execution states of OSes are managed by a data structure that we call a *vcpu*. When switching the execution of virtual cores, all the hardware registers are stored into the corresponding vcpu's register table, and then loaded from the table of the next executing vcpu. The mechanism is similar to the process implementation of a classical OS, but in addition, SPUMONE saves the entire processor state, including the privileged control registers.

The scheduling algorithm of virtual cores is the fixed priority preemptive scheduling. When the real-time OS and the general purpose OS share a physical core, the virtual core bound to the RTOS would gain a higher priority than the virtual core bound to the general purpose OS in order to maintain the real-time responsiveness. This means the general purpose OS is executed only when the

virtual core for the real-time OS is in an idle state and has no task to execute. The process or task scheduling is left up to OS so the scheduling model for each OS is maintained as-is. The idle real-time OS resumes its execution when it receives an interrupt. When virtual cores assigned to the general purpose OS are migrated to execute on a shared core, those cores are scheduled with a timesharing scheduler.

4 Highlights in SPUMONE

In the current implementation, we adopted Toppers[2] as a real-time OS and Linux as a general purpose OS. Also, we modified SMP Linux to use multiple virtual core offered by SPUMONE to exploit multi-core processors.

We have implemented SPUMONE on the Hitach/Renesas RP1 experimental multi-core board. The processor contains four SH4-A cores which can communicate with a shared memory. Currently, Linux, Toppers and OKL4 are running on SPUMONE. The modification of guest OSes is usually less than about 100 lines. The worst case interrupt latency of Toppers is less than 35μs while executing Linux both on a single and multi-core processor.

4.1 Dynamic Multiple Cores Management

SPUMONE for multi-core processors is designed in a distributed model. A dedicated instance of SPUMONE is assigned to each physical core as shown in Fig.1. This design is chosen in order to eliminate the unpredictable overhead of synchronization among multiple CPU cores. In addition, the basic lock mechanism can be shared between single-core and multi-core version, which may simplify the design of SPUMONE. It also enables the system to scale on multi-core and many-core processors as discussed in [1].

As described in the previous section, SPUMONE enables to multiplex multiple virtual cores on physical cores. The mapping between physical cores and virtual cores is dynamically changed to balance the tradeoffs among real-time constraints, performance and energy consumption. In SPUMONE, a virtual core can be migrated to another core according to the current situation. There are several advantages of our approach.

The first advantage is to change the mapping between virtual cores and physical cores to reduce energy consumption. As shown in Fig. 2, we assume that a processor offers two physical cores. Linux uses two virtual cores, and the real-time OS uses one virtual core. When the utilization of Toppers is high, two virtual cores of Linux are mapped on one physical core (Left Top). When Toppers is stopped, each virtual core of Linux uses a different physical core (Right Top). Also, one physical core is used by a virtual core of Linux and another physical core is shared by Linux and Toppers when the utilization of Toppers is low

[2] Toppers is an open source real-time OS used in various Japanese embedded system products. Toppers implements the μITRON interface specification that is a Japanese standard for the real-time OS.

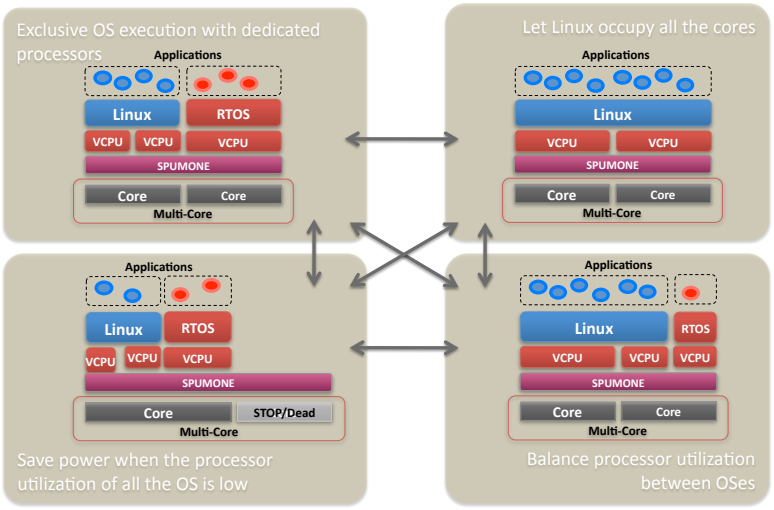

Fig. 2. Dynamic Multiple Cores Management

(Right Below). Finally, when it is necessary to reduce energy consumption or one of physical cores is dead, all virtual cores run on one physical core (Left Below). This approach enables us to use very aggressive policies to balance real-time constraints, performance, and energy consumption.

4.2 Reducing Interrupt Latency

In order to minimize interrupt delay of Toppers while sharing a physical core by Linux and Toppers, we proposed two approaches for a single and multi-core processor respectively.

The first approach that is for a single core processor is replacing the interrupt enable and disable instructions with the virtual instruction interface. A typical OS disables all interrupt sources when disabling interrupts for atomic execution. Our approach leverages the interrupt mechanism of the processor: we assign the higher half of the interrupt priority levels (IPLs) to Toppers and the lower half to Linux (Fig.3: Left). The instructions enabling and disabling interrupts are typically provided as kernel internal APIs. They are typically coded as inline functions or macros in the kernel source code.

When the Linux tries to block the interrupts, it modifies its interrupt mask to the middle priority. Toppers may therefore preempt Linux even if it is disabling the interrupts (Fig.3: Right (1)). On the other hand when Toppers is running, the interrupts are blocked by the processor (Fig.3 : Right (2)). These blocked interrupts could be immediately delivered when Linux is dispatched.

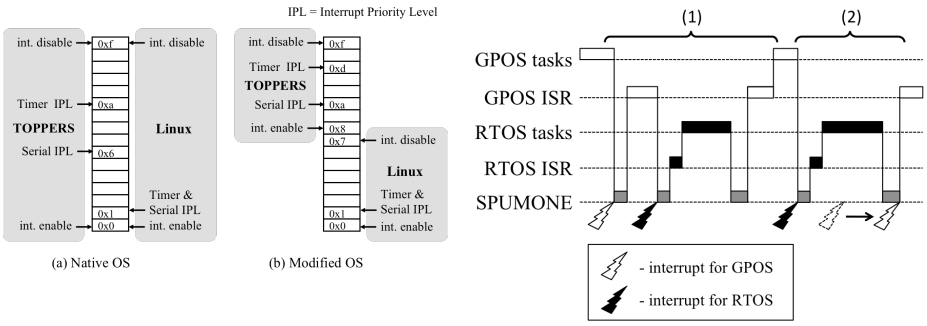

Fig. 3. The interrupt priority levels assignment and Interrupt Delivery Mechanism

The second approach that is for a multi-core processor is based on virtual core migration. As we implemented the first approach described in the previous paragraph, we found that some paths in the Linux kernel gained a highest lock priority unexpectedly (e.g. bootstrap, idle thread). This suggests us the possibility that some device drivers or kernel modules programmed in a bad manner gain a high IPL and interfere with the activity in Toppers. This means that careful coordination of IPLs requires high engineering cost. We modified SPUMONE to proactively migrate a virtual core, which is assigned to Linux that shares a physical core with Toppers, to another physical core when it traps into the Linux kernel or interrupts are triggered. In this way, only the user level code of Linux is executed concurrently on a shared physical core, which will never modify the IPLs. Therefore, Toppers may preempt Linux immediately without separating IPLs used in the first approach (Fig.4).

The approach can also minimize the effect of lock holder preemption. When the lock holder in the Linux kernel is preempted by Toppers, the Linux kernel executed on other physical cores must wait until Toppers becomes idle, and the lock owned by the preempted Linux kernel is released. This significantly degrades the performance of Linux. The virtual core migration ensures that the execution of the Linux kernel is always migrated to other physical cores that do not execute Toppers.

4.3 Security and Reliability

In SPUMONE, guest OS kernels share the same priviledged address space to reduce the amount of modifications and performance impact as much as possible. As a consequence, we need another approach to enhance security and reliability without relying on familiar isolation mechanisms. The basic approach is to use a monitoring service to detect the violation of the integrity of each kernel, and recover by rebooting the kernel. In our approach, each guest OS kernel can be rebooted independently even though all OS kernels reside in the same address space.

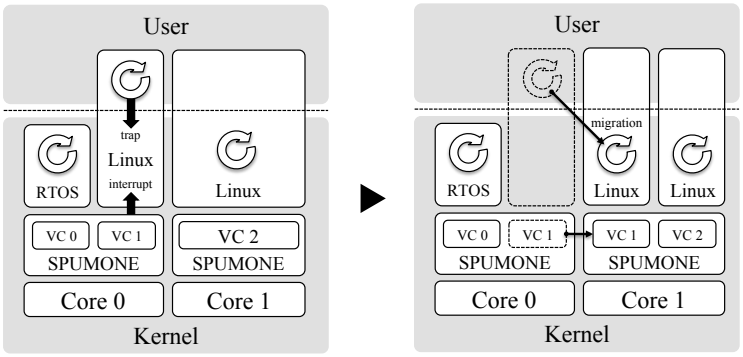

Fig. 4. Virtual core migration

The monitoring service checks the integrity of several data structures in the OS kernel periodically. The integrity is specified as constraints of each data structures. If the monitoring service detects the violation of the constraints, the service invokes a recovery function that is defined for each data structure to recover the integrity. The repair procedure may not repair the system completely - some garbage may remain in the kernel space or the repair procedure may even fail. In this case, the guest OS kernel is rebooted proactively.

When the Linux kernel causes an error while executing the kernel, the error can be translated to a system call error or an application signal. Of course, the optimistic approach may leak some resources in the kernel. In this case, Linux is rebooted when the kernel becomes idle. This approach is very effective in some embedded systems. For example, when the user is using a mobile phone, the Linux kernel does not need to be rebooted immediatly if some errors occur in the kernel, but the kernel can be rebooted when the user puts the phone in his pocket.

We are also considering an alternative approach. When the monitoring service detects some anomalies, it saves the states of application processes. Then, the Linux kernel is rebooted, and the states of processes are reconstructed. The applications can continue to run even though the kernel is restarted, which is similar to the checkpoint/recovery approach. Toppers and its applications can be simply rebooted when some anomalies are detected. The rebooting time can be improved by storing some important states in a persistent memory by using a similar techniques presented in [5]. Usually, most applications on Toppers contain a small amount of states, and rebooting the applications and Toppers is very fast. Also, the rebooting does not affect the functionality of the embedded system. This approach improves user satisfaction dramatically because the user is not aware of the reboot.

In our approach, if the Linux kernel is attacked, the attacker can invade other OS kernels. In order to attack other kernels, an attacker needs to insert code into the Linux kernel address space. Various traditional approaches can detect the modifications of the kernel easily. Recently, attacks tend to use kernel rootkits.

Kernel rootkits try to stealth themselves, and various security tool cannot find them. For example, some rootkits may modify kernel data structures that are used to manage processes. In our approach, the monitoring service checks the data structures that the rootkits try to modify, and repairs them to allow security tools to detect the rootkits.

Currently, the monitoring service checks some typical data structures that various rootkits are known to modify, and shows that our approach can remove many well known rootkits. We are also working on the synchronization mechanism between the monitoring service and the Linux kernel. Our approach uses optimistic synchronization because we cannot modify the Linux kernel to exclude shared data structures between the Linux kernel and the monitoring service [9].

Of course, the monitoring service should be protected from the Linux kernel. There are various mechanisms to protect the monitoring service. For example, we can use a co-processor or a special device to execute the monitoring service. The multi-core processor that we are using (SH4-A) for building embedded systems contains a local memory in each CPU core. This local memory can only be accessed by its CPU core. In our approach, a CPU core is dedicated to execute the monitoring service. Thus, the Linux kernel cannot access its local memory, but the CPU core executing the monitoring service can access all the memory used by the Linux kernel.

5 Current Status and Future Direction

SPUMONE can execute multiple operating systems without suffering a large amount of overhead and engineering cost. Although most of processors for embedded systems are not suitable to implement the virtualization layer to offer the complete isolation between guest OSes because it requires a large amount of overhead without virtualization hardware supports. Since SPUMONE and OS kernels run in the same privileged space, our approach increases the possibility of the kernel corruption, but a monitoring service detects the corruption in SPUMONE and guest OSes and heals them by rebooting. Also, introducing the virtualization layer in embedded systems offers additional advantages. For example, proprietary device drivers can be mixed with GPL codes without license violation. This solves various business issues when adopting Linux in embedded systems.

We are currently enhancing our implementation to support various policies to consider the tradeoff among power consumption, performance and timing constraints. The monitoring systems should be enhanced in the near future. Especially, we are interested in using Daikon [2] to detect the invariance inside the kernel automatically.

Asymmetric multicore processors are a promising approach to reduce the power consumption [3]. In SPUMONE, we are considering to hide the heterogeneity inside the SPUMONE and offer virtual homogeneous multicore processors to operating systems. However, it is not easy to hide the heterogeneity completely. We are also considering to develop the MapReduce-based applica-

tions on SMP Linux [8], and coordinate the middleware and SPUMONE to hide the heterogeneity completely.

References

1. Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schuepbach, Akhilesh Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems", In Proceedings of the 22nd ACM Symposium on Operating Systems Principles, 2009.
2. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao, "The Daikon System for Dynamic Detection of Likely Invariants", Science of Computer Programming, Vol. 69, No. 1–3, Dec. 2007, pp. 35-45.
3. Alexandra Fedorova, Juan Carlos Saez, Daniel Shelepov, Manuel Prieto, "Maximizing Power Efficiency with Asymmetric Multicore Systems". Communication of the ACM, Vol.52, No.12, pp.48-57, 2009.
4. Hiroaki Inoue, Junji Sakai, Masato Edahiro, "Processor virtualization for secure mobile terminals", ACM Transaction on Design Automation of Electronic Systems, Vol. 13, No. 3, 2008.
5. Hiroo Ishikawa, Alexandre Courbot, Tatsuo Nakajima, "A Framework for Self-Healing Device Drivers", In Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, pp.277-286, 2008.
6. Tatsuo Nakajima, Vili Lehdonvirta, Eiji Tokunaga, Hiroaki Kimura, "Reflecting Human Behavior to Motivate Desirable Lifestyle", In Proceedings of the Conference on Designing Interactive Systems, pp.405-414, 2008.
7. Tatsuo Nakajima, Hiroaki Kimura, Tetsuo Yamabe, Vili Lehdonvirta, Chihiro Takayama, Miyuki Shiraishi, Yasuyuki Washio, "Using Aesthetic and Empathetic Expressions to Motivate Desirable Lifestyle", In Proceedings of the Third European Conference on Smart Sensing and Context, pp.220-234, 2008.
8. Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems", In Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture, 2007.
9. Hiromasa Shimada, Alexandre Courbot, Yuki Kinebuchi, Tatsuo Nakajima, "A Lightweight Monitoring Service for Multi-Core Embedded Systems", In Proceedings of the 13th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2010
10. Tetsuo Yamabe, Vili Lehdonvirta, Hitoshi Ito, Hayuru Soma, Hiroaki Kimura, Tatsuo Nakajima, "Applying Pervasive Technologies to Create Economic Incentives that Alter Consumer Behavior", In Proceedings of the 11th International Conference on Ubiquitous Computing, pp.175-184, 2009.
11. Franois Armand, Michel Gien, "A Practical Look at Micro-Kernels and Virtual Machine Monitors", In Proceedings of the IEEE 6th Consumer Communications and Networking Conference 2009, pp.1-7, 2009.
12. Fahim Kawsar, Tatsuo Nakajima, Kaori Fujinami, "Deploy Spontaneously: Supporting End-Users in Building and Enhancing a Smart Home", In Proceedings of the 10th International Conference on Ubiquitous Computing, pp.282-291, 2008.