Robust Lightweight Embedded Virtualization Layer Design with Simple Hardware Assistance

Tsung-Han LIN^{†a)}, Yuki KINEBUCHI[†], *Nonmembers*, and Tatsuo NAKAJIMA[†], *Member*

SUMMARY In this paper, we propose a virtualization architecture for a multi-core embedded system to provide more system reliability and security while maintaining performance and without introducing additional special hardware supports or implementing a complex protection mechanism in the virtualization layer. Embedded systems, especially consumer electronics, have often used virtualization. Virtualization is not a new technique, as there are various uses for both GPOS (General Purpose Operating System) and RTOS (Real Time Operating System). The surge of the multi-core platforms in embedded systems also helps consolidate the virtualization system for better performance and lower power consumption. Embedded virtualization design usually uses two approaches. The first is to use the traditional VMM, but it is too complicated for use in the embedded environment without additional special hardware support. The other approach uses the microkernel, which imposes a modular design. The guest systems, however, would suffer from considerable modifications in this approach, as the microkernel allows guest systems to run in the user space. For some RTOSes and their applications originally running in the kernel space, this second approach is more difficult to use because those codes use many privileged instructions. To achieve better reliability and keep the virtualization layer design lightweight, this work uses a common hardware component adopted in multi-core embedded processors. In most embedded platforms, vendors provide additional on-chip local memory for each physical core, and these local memory areas are only private to their cores. By taking advantage of this memory architecture, we can mitigate the above-mentioned problems at once. We choose to re-map the virtualization layer's program on the local memory, called SPUMONE, which runs all guest systems in the kernel space. Doing so, it can provide additional reliability and security for the entire system because the SPUMONE design in a multi-core platform has each instance installed on a separate processor core. This design differs from traditional virtualization layer design, and the content of each SPUMONE is inaccessible to the others. We also achieve this goal without adding overhead to the overall performance.

key words: operating system, embedded system, virtualization, local memory, scratch-pad memory

1. Introduction

As the needs of embedded products have recently increased, the short latency, i.e., the real-time system capability, is no longer the only requirement. Designers and end-users want their systems to have rich graphic supports, feature-rich off-the-shelf device drivers, and most importantly, the dependability. Moreover, powerful networking capability, which, according to Bell's Law, is this generation's computing class, is also a must for such devices. A GPOS such as Linux, by its nature, cannot handle real-time tasks well enough (hard real-time) in embedded devices. The virtual-

ization technique is thus used to solve this type of problems by isolating these different systems from each other, where each system is an independent domain. Now system developers can easily apply their applications on an embedded device and cooperate with those legacy software or hardware without considerable costs in system construction. By virtualizing the underlying physical hardware into several virtual instances, different systems can co-exist in a single embedded device and be monitored and controlled by a single VMM (Virtual Machine Monitor). Virtualization, however, is not a new technique: it was first proposed by IBM in the 1960s, and is commonly used in data centers and desktop environments, including VMWare [1], and VirtualBox [2], etc.

For embedded environments, the virtualization technique must make some adjustments to handle different design challenges, including limited hardware resources and power management requirements. Efficient using these resources becomes an important design consideration. Embedded systems, e.g., ARM [3], have recently and widely used the multi-core processor. With this additional computing power, embedded virtualization systems can achieve better performance. Power consumption could also be reduced in the multi-core platform, which is even more important for embedded devices. With proper configurations to virtualization systems, e.g., letting two guest OSes in low operation modes share a core and lower another core's frequency, can also save considerable power.

Embedded virtualization design is usually categorized in two ways. The first is using a traditional VMM system, as described. It virtualizes most system resources, and guest OSes can run on it without any significant system changes. VMM itself, however, is too complicated for embedded system design. To allow all guest OSes act as though they were running on a bare machine, the VMM must provide functionalities such as instruction emulation, memory access control, spatial isolation among guest OSes, and interrupt. These operations all involve heavy computing efforts and result in considerable power consumption, which is critical to the embedded system. If there were no special hardware to help perform some operations, such as the page table virtualization [4], the traditional VMM design would be a burden to the embedded environment.

The other approach is to use a microkernel approach that provides a modular system construction fashion. Using the microkernel system, however, causes guest systems to suffer from many code modifications, as the microker-

Manuscript received December 29, 2011.

Manuscript revised May 20, 2012.

[†]The authors are with DCL Lab., Department of Computer Science, Waseda University, Tokyo, 169–8555 Japan.

a) E-mail: johnny@dcl.info.waseda.ac.jp

DOI: 10.1587/transinf.E95.D.2821

nel offers high-level abstraction, e.g., process and memory regions, and guest OSes must tame the abstraction to execute them on the abstraction. A microkernel construction example is given below. Another concern of the microkernel approach is that guest systems run in the user space. Most RTOS kernel and application codes run in the kernel space and are thus difficult to put in the user space.

Most importantly, these two design approaches have only one virtualization image, which makes the whole system easily breakable. The corruption of a single virtualization instance could propagate to all guest systems if no mechanism exists to isolate the virtualization layer and the guest systems. When designing a virtualization system, one must ensure that it is lightweight enough, which requires a limited number of complicated operations and sufficient isolation and reliability between the virtualization layer and the guest systems.

Motivated by the above design considerations, we focus on the following topics, which are the main contributions of this paper in the multi-core virtualization environment:

- *Analyzing the design considerations of the embedded virtualization system.* This paper discusses some design perspectives and uses an in-house developed project, SPUMONE, as a discussion example. SPUMONE design properties are also examined to further analyze those issues and the distributed SPUMONE design is the most important feature. In this design, each available physical core contains exactly one copy of a virtualization instance, and a single virtualization instance crash will not affect the others [5]. It is also easily adapted into the multi-core SMP platform, thus increasing its scalability. Using this architectural advantage, we further improve the reliability and security of the whole virtualization system.
- *Isolation using local memory to improve reliability.* To provide better isolation among virtualization layers and guest systems, a common hardware in the embedded platform, called local memory, is used; this hardware is introduced in the next section. Data in the local memory are only accessible to their physical core, and for this reason, faults in one virtualization instance will not propagate to the whole system, thus further improving reliability. Isolation using the local memory is a new idea, and we found no previous research that uses local memory to provide isolation in the embedded virtualization design. There is also no performance degradation when altering the system with local memory installation. Some evaluations of the performance similarities are given below to support this.

This paper is organized as follows. Section 2 motivates our design approach of a virtualization system. Section 3 provides the base virtualization architecture in this paper. Section 4, which is the main contribution of the paper, introduces new virtualization consolidation using local memory and gives some use scenarios. Some evaluations

about performance similarity are performed in the Sect. 5, and the Sect. 6 discusses related works. Section 7 concludes the work and provides suggestions for future studies.

2. Platform and Design Issues

In this section, we first introduce the platform used for the embedded virtualization system design and examine some design issues when working on the embedded virtualization system. Finally, an in-house developed embedded virtualization architecture is given to address the issues being analyzed.

2.1 Platform

2.1.1 Multi-Core Architecture

Increasing the clock frequency of a processor has been proven unsuitable for enhancing performance because the power consumption problem becomes the bottleneck, especially in deep-submicron design that contains many leakage sources. Vendors opt to provide more computing units to increase performance. This multi-core design trend also extends to embedded platforms, especially the power consumption issue is critical for most consumer electronic devices. Other co-processors, i.e., like GPU (graphics processing unit) or DSP (digital signal processor) generally exist in an embedded platform, also referred to as ASMP (asymmetric multiprocessing) architecture; this work focuses on the SMP (symmetric multiprocessing) architecture, where all processor cores are identical and tied to the same memory bus.

2.1.2 Local Memory

Some low-end embedded platforms, such as MCU (Micro Controller Unit) do not provide a cache system for developers. They instead provide an on-chip SRAM, the so-called local memory[†], which is typically only a few KBytes, private to a processor core, and accessed using an on-chip internal high-speed bus.

Table 1 shows the local memory configurations of some common embedded vendors or architectures.

Most embedded processor vendors have provided this type of local memory, so most embedded applications can be built for special requirements and performance. Embedded system developers can install all mp3 decoding programs

Table 1 Local memory configuration.

Architecture	Configuration
805x	128 or 256 KB
ARM11	4 to 256 KB
MIPS32 4KE Family	Up to 1 MB
SuperH RP1	Instruction:8 KB Data:16

[†]Some vendors also call it Scratchpad or Tightly-coupled memory.

in the local memory area to enhance the entire decoding process and performance because all operations and data fetches are in the same chip and accessed using the same high-speed bus.

Local memory differs from cache memory, although they both reside in the same chip of a processor core. The hardware controls the cache, and there is some contention for the cache line. Data in the cache are only a copy of their original image in the external main memory, so other programs or systems can access to this information. Conversely, local memory is totally controlled by software or developers mapping the desired program sections or data into this area. The contents in the local memory are not replaced or swapped to the external main memory, and other systems or programs thus cannot read them. Only their own physical core, the core on the same chip as the local memory, has the right to access to them. As a result, the contents in the local memory area have higher security than those in cache. The local memory power consumption [6], [7] is also smaller than that of the traditional cache system.

Not only can performance and power consumption be improved using local memory, but security and reliability can also be enhanced. Using local memory in a multi-core platform virtualization system, e.g., including some critical hypervisor[†] parts or a guest system running on one core into the local memory, developers can partition the whole system into different domains for different purposes without concern that one may harm the others, as local memory contents are accessible only to their physical core. This also simplifies the multi-core embedded virtualization design by not requiring the implementation of complicated functionalities to provide protection mechanisms that only enlarge the TCB (Trusted Computing Base) of the virtualization system and make the whole system more prone to attack. The entire embedded virtualization system can thus benefit from using local memory from both the performance (if there is no cache system) and security perspectives.

2.2 Virtualization System Consolidation in a Multi-Core Platform

2.2.1 Traditional VMM in Embedded System

Many great existing VMM works can also be adapted to embedded environments. Due to their original design strategies, as well as target platforms with more computing power and memory space, directly using them in an embedded system could seriously impact system performance. The traditional VMM design has the following important functionalities to support multiple guest systems running on top of it:

- *Instruction Emulation or Binary Translation.* To provide user with a convenient virtualization environment setup, most VMM frameworks provide this feature to users so that they can directly install systems without any modification; this is usually called the *full-virtualization* technique.

- *Shadow paging.* Handling paging systems in a virtualization environment is another issue because the hypervisor must also provide sufficient isolation and protection to the guest systems. Multiple guest systems attempt to use the same memory source, so the hypervisor must manipulate these memory operations.

The traditional VMM features described above indicate that they are all CPU-intensive computation jobs and therefore consume considerable energy, which is critical for the embedded system. In recent VMM designs, system developers turn to special virtualization hardware components for help, but the architecture environment of desktop or data center computers differ from those of embedded devices. Most embedded devices do not have that degree of special hardware support. Determining a new way to make hypervisor design and operations more lightweight is crucial.

2.2.2 Guest OS Kernel in the User or Kernel Space

To isolate the hypervisor and the guest systems in a traditional virtualization system, guest systems may be configured to reside in the second-most privileged level, and the hypervisor is installed in the most privileged level, such as VMWare in the x86 system. In an embedded system, however, there are typically two protection levels. To separate the guest systems from the hypervisor, guest systems must be put into the user space. This leads to many guest system modifications, as most systems, especially RTOSes, have their kernel and application codes in the same address space; some application codes in a RTOS are highly integrated with privileged instructions or device drivers. As a result, porting these codes to the user space is difficult; however, this kind of consolidation does provide some isolations between guest systems and the virtualization layer.

Another alternative is to allow these guest systems to run in the kernel space alongside the hypervisor. This saves considerable engineering efforts and allows new systems to be added easily. Isolate the guest systems' kernels from the hypervisor becomes an issue because these kernels are now in the same address space and could affect each other. While some processor vendors provide special hardwares to implement isolation using hardware page table manipulation and thus increase the ability of the hypervisor to protect the system, such as Intel's Intel VT-x [8] and AMD's AMD-V [9], again, it is uncommon to see this type of hardware support in embedded systems.

2.2.3 Single or Multiple Hypervisor Image

In the traditional virtualization design, one hypervisor image hosts all guest systems. The advantage of using this configuration is that it provides a more straightforward approach for developers. Nevertheless, when some intrusions

[†]We use the terms virtualization layer and hypervisor interchangeably in this paper.

control this single image through a guest system, these intrusions into the virtualization layer affect the guest systems because this layer is shared. The hypervisor became a medium for propagating faults because no isolation mechanism exists in the hypervisor to stop the faults.

Having multiple virtualization layer instances could reduce this issue to a certain degree. It is also easier for developers to adopt this configuration on the multi-core or SMP platform by limiting the information shared or synchronized among instances, thus increasing system scalability. Even so, these hypervisor images are still run on the shared memory region, which remains accessible to others. Protecting each virtualization instance and achieving higher security is also a concern for this approach. When a single image is broken, the whole system must ensure that the remaining hypervisor instances can help the guest systems continue working and provide a mechanism to recover the broken system.

2.3 Chosen Virtualization Architecture: Distributed Design with Local Memory

To mitigate the above issues and keep the virtualization layer implementation small enough without requiring complex functionalities, we opt to use the *local memory*, as shown in Fig. 1, to help construct the system and provide sufficient reliability and security. We called this system construction *Distributed Design* because the hypervisor is split into several instances and installed on all available physical CPU core local memories.

- To address the issue in the Sect. 2.2.2, we choose to have guest systems' kernels run with the hypervisor in the kernel space. This allows the guest systems to suffer few modifications. Meanwhile, isolation among guests and the hypervisor is provided by installing the hypervisor into the local memory. As Fig. 1 shows, although the guest system's kernel and virtualization layer are in the kernel space, the virtualization layer

is in the local memory's address space, and the guest system's kernel remains in the main memory's address space. As contents in the local memory are only accessible to its physical core, other cores' programs' access is prohibited, as shown in Fig. 1. Even when a guest system on the same core tries to access the contents, it must first obtain permission to do so.

- Using the local memory in the embedded multi-core platform can also allow the distributed design to easily be adopted on the local memory, and can solve the single hypervisor instance protection problem simultaneously. As Fig. 1 shows, each virtualization instance is installed in the local memory of a physical core. When a fault in a hypervisor instance occurs that is caused by its guest systems, the fault only stays in that physical core domain; the rest of the system stays unaffected. A monitoring service can recover the fault instance independent of the other instances.

This design method can also be implied on other existing hypervisor works to enhance the system reliability. Some common routines, including VCPU[†] scheduling and interrupt dispatch, do not need to involve all physical CPUs for execution; thus these operations can run normally on each physical CPU. Even if these CPUs need to communicate or exchange information, they can use inter-processor communication, which is often a simple operation. Microkernel-based hypervisors can thus use this distributed local memory approach easily. Basic system control servers of a microkernel-based hypervisor, such as device server and file server, can run independently on each physical CPU and be protected by the local memory. Inter-server communication can be done by the original hypervisor IPC mechanism. Other operations or system consolidations, such as Xen [10], that have centralized operations, such as memory protection or Xen's domain 0, where all other guest domains must communicate for execution, would make applying distributed design difficult. Hypervisor works that would prefer to use the distributed design with local memory approach have some limitations, and must satisfy the following:

- No centralized control. Each hypervisor on a physical CPU can perform the same functionalities as it did when there was only one hypervisor image. If the hypervisor need or want to communicate with the other hypervisors, such as device sharing between guest OSes or communication, inter-processor or shared memory-based mechanisms must be used, and this may need some additional code changes.
- Small and simple enough design. A hypervisor can use the computing power of only one physical CPU; thus, the load on that physical CPU is larger than when the all-system-one-hypervisor approach assigned its load

Fig. 1 Local-memory-protected virtualization architecture.

[†]VCPU stands for virtual CPU and is a data structure that contains the guest system's context and register information; it is also the basic hypervisor scheduling unit.

to all available physical CPUs. Additional computation and design complexity may be added to the one-hypervisor approach. To use our proposed local memory consolidation, due to the local memory size constraint, the per-CPU hypervisor image must be small to be put into local memory for protection. Small size hypervisor also means that the number of functionalities is limited, and this is a trade-off between reliability and functionality. We also further discuss the size issue in the Sect. 5.2.1.

Based on these rules, we have developed our own distributed virtualization layer work, called SPUMONE, and its new enhancement using local memory, which is introduced in more detail in the next two sections.

3. Base Virtualization Architecture: SPUMONE

This paper uses a virtualization layer called SPUMONE [11] (Software Processing Unit, Multiplexing ONE into two or more), developed in our project for embedded systems, as a base system. It can run on the multi-core platform; a single-core version of SPUMONE is also provided. SPUMONE is a composition kernel that multiplexes a single physical CPU core into multiple virtual CPU instances [5]. It aims to provide a lightweight virtualization environment to its guest OSes where the impact of this virtualization layer on the guest OSes would be as small as possible. Figure 2 shows the overall SPUMONE architecture, and the following points summarize and analyze some important SPUMONE features.

- *Small emulation overhead.* This is the most important feature of SPUMONE. As Fig. 2 shows, SPUMONE and guest OSes run together at the most privileged level; it thus further lightens the workload of the entire virtualization layer due to small privileged instruction emulation efforts.
- *Small modification.* As mentioned above, guest OSes of the SPUMONE system all run in the most privileged mode with SPUMONE itself. We do not have to modify many portions of guest OSes, which is especially important when considering rapidly developed open-source OSes such as Linux. This can considerably reduce engineering costs. Table 2 gives some comparisons using the LOC (lines of code) added or removed from SH-Linux and RTOS (TOPPERS/JSP in this case).
- *Interrupt virtualization.* To prevent the co-existing GPOS affect the real-time responses of RTOS, SPUMONE first investigates all interrupts and assigns these interrupts to an appropriate destination OSes, according to its interrupt destination table. SPUMONE also first handle OS traps, but they are directly returned to the currently executing virtual processor.
- *Distributed design.* In the multi-core SPUMONE version, each physical core contains a dedicated SPUMONE instance, and each instance can have dif-

Fig. 2 SPUMONE architecture.

Table 2 Code changes for guest OSes.

Guest OS	Lines changed
SH-Linux	26
TOPPERS/JSP	38

ferent configurations for different purposes. Figure 2 shows that one SPUMONE instance has two VCPUs and can provide them to both GPOS and RTOS, while the other instance has two VCPUs that are dedicated to GPOS. This is radically different from the traditional hypervisor designs that have only one hypervisor instance across all available physical cores. The distributed design provides additional isolation because a single SPUMONE instance failure only affects its guest system. The other instances can still work safely if developers choose to separate guests OSes from SPUMONE instances. Another advantage of this design strategy is that it is much easier for developers to put critical SPUMONE data in local memory without concern for the synchronization problem, thus increasing the reliability of the entire system. More details about the local memory use scenarios and discussion of this distributed SPUMONE design are given below.

4. Hardware-Assisted Virtualization Design

4.1 Applying SPUMONE Distributed Architecture

We use the Renesas RP1 processor, developed by Renesas and Waseda University, to perform the work. The RP1 processor contains four physical SH4a cores and each has their own local memory area. There are 8 KBytes ILRAM and 16 KBytes OLRAM in each core for instruction and data operand use, respectively. Developers can access them by mapping portions of their program into these areas.

In SPUMONE's multi-core platform design, each physical core is coupled with a SPUMONE instance (Fig. 3 (b)). This differs from most traditional virtualization layer designs on multi- or many-core platforms, as shown in

Fig. 3 SPUMONE and traditional VMM comparison.

Fig. 4 Domain isolation.

Fig. 3 (a), which have only one virtualization instance covering all physical cores. The advantage of using the traditional multi-core virtualization design is that it provides more straight forward approaches, e.g., it does not require communication with other virtual layer instances. However, it might be a risk for a system to explore the whole set of information on virtualization to all of guest systems because one fatal error in a virtualization layer can propagate to the entire system. Most embedded processors also have fewer privilege levels in a core than desktop processors, such as the Intel x86 families, so the additional hardware protection is not available for most embedded virtualization designs. As indicated in the depiction of SPUMONE's architecture in Fig. 2, all guest systems and SPUMONE run at the same privilege level. It is thus important to have a distributed virtualization layer design to separate these instances among physical cores.

Using SPUMONE's architecture as a basis, we change the mapping address of each instance to each core's local memory area instead of using main memory. This can further improve the whole system's reliability. Figure 4 considers a guest SMP Linux configured to support four VCPUs running on two CPUs without sharing any physical cores with other guest OSes. In this case, Linux covers more than one SPUMONE instance because VCPUs are assigned across different physical cores. Guest OSes of SPUMONE are also modified to be aware of the existence of SPUMONE through the para-virtualization technique [12] used in the SPUMONE design, so attacks on a guest OS could harm the important data located in the local memory that contains SPUMONE parts. This type of corruption or attack may thus propagate across SPUMONE instances, and this part of the system becomes an unsecure sub-domain. The remaining system, however, is not corrupted because of the dis-

tributed assignment of SPUMONE instances. In this case, RTOS can continue to run even if the SPUMONEs coupled with SMP Linux are broken. Thus, if developers attempt to put some versions of guest OSes that are under testing or are easily compromised but have functionalities in the platform, system reliability will be maintained to some extent.

Another scenario of using local memory to store SPUMONE data occurs when some attacks jump directly into the shared main memory area and attempt to modify SPUMONE data. These attacks do not have a opportunity to do so because, unlike the data in the cache, there is no need to copy data on the local memory to the shared memory area.

If fewer physical CPU cores are available than the number of guest OSes, i.e., an over-committed system, or the system is configured such that some guest OSes must share the same physical core, as shown in Fig. 2, developers may all configure their systems so that the guest OSes share a physical core. When system developers want to configure their systems in this way, they usually want to achieve better performance or consume less power. The VCPUs of guest OSes' VCPUs are put into the same virtualization layer and physical core, either by statically configuration or by dynamically migrating VCPUs. The performance enhancement requirement conflicts with the need of reliability in that resources or systems must be shared. This is a design tradeoff between achieving better performance and achieving higher reliability by isolation, and we emphasize reliability in this paper while keeping the same performance. We make this statement on top of the multi-core embedded environment. If there are only two physical cores, the system can still be configured into two different domains, with each domain remaining unaffected by the other domain.

In the embedded virtualization system, various guest

Table 3 Size of sections of SPUMONE (in bytes).

Section Name	Size
<i>.start</i>	8 k
<i>.text</i>	11 k
<i>.data</i>	412
<i>.bss</i>	6 k
<i>.rodata</i>	4 k

systems perform in different and unpredictable ways, especially legacy in-house designed OSes. Developers should thus seriously consider isolation and system reliability.

4.2 Local Memory Management

A linker script performs address re-mapping control. Each program that forms part of the final image file contains some special areas, e.g., a *.text* section for instructions and a *.data* section for initialized data. After compiling, the linker merges these object files into a final execution image and puts these sections at the proper memory address using the guidance of the linker script.

The implementation in this paper maps a SPUMONE image onto each physical core's local memory area. Table 3 shows each program section's size in a SPUMONE instance. In the platform used, which is shown below, the total size of each physical core's local memory is sufficient to hold all program sections.

5. Evaluation and Discussion

5.1 Evaluation

A virtualization system with the proposed local memory consolidation provides an easy way to enhance overall system reliability. Meanwhile, by altering the location of the program image, system performance remains unchanged, as described in this section. Conclusions regarding why the performance is maintained in all experiments are given at the end of this section because the rationale is the same for each experiment. The evaluation is performed using the MSRP1BASE2 board which contains an RP1 quad-core 600 MHz processor and 128 MB memory. Each core has 8 KBytes, 16 KBytes and 128 KBytes of ILRAM, OLRAM and URAM (also a on-chip local memory in this platform), respectively. We use TOPPERS/JSP 1.3 as a guest RTOS and Linux 2.6.16 as a guest GPOS running on top of SPUMONE instances.

SPUMONE has responses for two important operations: OS switch and interrupt delivery. The guest system's performance is greatly affected by how quickly SPUMONE can perform these two operations. We compare some benchmarking results related to these two operations between the original SPUMONE implementation without local memory usage (*Normal*) and the one with local memory (*LMEM*) to show that no performance degradation occurs. We first measure the interrupt dispatch delay differences between these implementations on RTOS to show that the SPUMONE im-

Fig. 5 Dispatch delay with SPUMONE on local memory. **Fig. 6** Dispatch delay with the original implementation of SPUMONE.

age alteration to the local memory will not affect the interrupt delivery in the timing-sensitive RTOS. RTOS is still running a periodic task every 1 ms and is sampled 100,000 times in a measurement run while Linux is performing a kernel compiling job. Figures 5 and 6 show the result. The measurement on the local memory implementation (Fig. 5) shows that the dispatch delay distribution is almost the same as that of the original SPUMONE implementation. The average dispatch delay for the local memory implementation is $7.1 \mu\text{s}$ and $8.54 \mu\text{s}$ for the original SPUMONE.

We use some benchmarks in Linux to demonstrate that the performance is maintained with the local memory approach, which means that the image location alteration also does not affect guest systems switching or their original performance. Instead of running different applications, we use several tools from some micro-benchmarks to better cover the operation details that form the bases of most applications and thus determine the overall performance. The following operations are the most important operations to which most applications are related, and all are evaluated below:

- Scheduling ability and context switch

Fig. 7 Hackbench results w/ and w/o local memory.

Fig. 8 Context switch latency in the linux (Lower is better).

- CPU operations
- Memory access
- Process manipulation

First, the *hackbench* benchmark program is used to measure latency and performance variations on the Linux scheduler when RTOS is also running its periodic job, as described in the first evaluation setting. The *hackbench* program creates process sets according to the user's given parameter. Up to 10 groups are used in this evaluation due to platform limitations. Figure 7 shows the result, where execution times for both settings are nearly identical. This shows that no extra scheduling latency is introduced when using different SPUMONE image installations.

We then compare the context switch latency in Linux between these two SPUMONE image settings. LMBench [13] provides some tools to measure the performance variations in Linux. Here, we use its context switch tool, *lat_ctx*, to perform the comparing tasks. This program generates a program set[†] for a given size,^{††} and these processes also communicate through the Unix pipe. The program itself measures context switch time latency, as shown in Fig. 8. The difference between the two implementations is not obvious, though the local memory version performs better in some cases than in others. For most cases, we can easily see that the latencies are close to each other, which demonstrates that a different SPUMONE image installation strategy does not affect the context switch latency.

Memory access performance is also shown using a simple program named *memspeed* [14]. This program allocates a fixed size memory area, 32 MB in our experiment, and performs different strides for reading and writing memory in the area. Figures 9 and 10 give the results of read and write access, respectively. There is a threshold of 32 k for

Fig. 9 Memspeed -32 MB read.

Fig. 10 Memspeed -32 MB write.

Fig. 11 Memory mapping latency.

access, which is the data cache size. Accessing stride larger than this threshold causes throughput degradation because the access is made in the main memory area. Nearly identical results in both figures also show that SPUMONE image location change has no effect on this system's operation.

Another memory access-related test is also given using LMBench's *lat_mmap* program. This program maps and unmaps parts of a file into and from the main memory. It acts similar to a shared library mapping in a running process, which can also show whether memory manipulation is affected. The latency in Fig. 11 for both the original and local memory SPUMONE versions indicates that there is no performance degradation if SPUMONE is installed in the local memory because of similar memory access performance.

^{††}From 2 to 16 in our case.

^{††}We use 0 K, 8 K and 16 K to perform the experiments. 0 K means this process does not pass anything through the pipe.

Table 4 Process creation latency (*us*).

Type	fork	exec	shell
<i>Normal</i>	1995	10939	32058
<i>LMEM</i>	2061	10991	32006

Table 5 IPC latency (*us*).

Type	tcp	udp	unix socket
<i>Normal</i>	365.92	123.14	74.80
<i>LMEM</i>	351.15	116.28	71.38

Fig. 12 UnixBench score (Higher is better).

We measure another facet of the guest system with respect to process operation using *lat-proc* and a set of IPC (interprocess communications) benchmarks in the LM-bench. The former measures the latency of the fork, exec, and shell process operations, and the latter allows processes to use different IPC protocols to communicate with each other and measures their latencies. If the SPUMONE image location change affects the overall virtualization system, then this part of the guest system's operations is also affected. Tables 4 and 5 show no performance degradation regardless of SPUMONE image location.

We also use some Unixbench's [15] tools to observe the performance variations. We use its CPU benchmark, *dhry2reg*, and two other system performance tools, *pipe* and *syscall*, to evaluate differences between the two implementations. Figure 12 scores these three different system-wise benchmarking results, which again show that no side-effect is introduced regarding SPUMONE local memory usage.

We have provided different benchmarking results here to show a lack of performance degradation in different system operations, which demonstrates that SPUMONE local memory usage does not affect overall performance. The reason behind this performance similarity is that, for a SPUMONE operation, the only difference between the original and local memory versions is the place from which the operation retrieves its data or instructions. Data or instruction access behavior at the local memory is similar to that at the main memory or cache using addressing. Only the access speeds for these memories are different. If a system provides cache and local memory and if the size of a hypervisor is small enough, e.g., if SPUMONE can reside entirely in the local memory or cache, then those access operations

have basically the same access speed, as the cache and local memory are located inside the same processor chip. They should have the same or similar access speed and can be accessed within a single processor cycle regardless of architecture. Consequently, no obvious performance variation is shown, and this change does not affect the guest systems, as shown in the evaluation results. Moreover, if a low-end system has no cache but the local memory is provided, installing the hypervisor image into the local memory provides performance upgrades for a quicker access speed than that in the main memory.

5.2 Discussion

5.2.1 Hypervisor Size Concern

When considering platforms where the local memory size is far less than the desired hypervisor image, which are common in embedded systems, and that do not have cache systems, some selection strategies must be made. One of the most important operations for SPUMONE is to switch OSES when necessary. This operation involves heavy memory access for storing one preempted OS CPU context and loading another OS context to execution. Assume that one guest OS is running its own tasks when an interrupt arrives, which the other guest OS should handle, as it registers this interrupt. In this case, SPUMONE first searches its vector table, which is in the *.bss* section of the program, and decides which guest OS should handle this interrupt. If handling a destination guest OS requires OS-switching, SPUMONE saves the VCPU states of the current guest OS, which contains the current execution context and is also in the *.bss* section, and VCPU values of the chosen guest OS are executed.

In its implementation, SPUMONE manipulates its own stack area instead of using the specific OS stack area. Some applications running on top of a guest OS may have their own stack management. To avoid corruption from this type of operation, the SPUMONE implementation chooses to have its own stack area.

These operations are all memory-intensive operations, and the latency between the CPU and the main memory is considerable because of differing bus frequencies. Most importantly, they are all critical parts of SPUMONE and should be well protected. Based on the above discussion, we can choose to re-map only these parts of a SPUMONE image onto local memory while providing enough reliability to the entire system. Other virtualization implementations can use this type of selection strategy to customize their system constructions. It is also simple for developers to make these changes using the linker script alteration given in the Sect. 4.2 when building the system.

5.2.2 Additional System Protection by Auxiliary Subsystem

To further construct a robust virtualization environment,

some additional consolidation choices can be used to enhance the overall system protection using some auxiliary sub-systems.

According to the typical setup in Fig. 4, a virtualization environment can be divided into RTOS and GPOS parts. An RTOS is usually small enough to be installed almost entirely into the local memory and can thus be protected from a corrupted GPOS attack. When an RTOS cannot be entirely installed, the above selection strategy can alternatively be applied, and only critical portions are brought into the local memory. This strategy may leave some space in the local memory, so a small auxiliary system can be inserted to run with the RTOS and check its correctness. Here, a small detecting system [16] called *Secure Pager*, which can be installed entirely in the local memory with RTOS, is being developed for this purpose. This system is responsible for performing a hash calculation of the portions of the guest system being promoted to the local memory to increase the loaded part of the system's reliability and security.

Conversely, for GPOS, we offer an additional detection program, a monitoring service [17], to detect abnormal SMP Linux activities. As in some Linux rootkits that try to insert a kernel module to change the execution of important system calls or modify the system call table, GPOS intrusions could also take control of SPUMONE's interrupt manipulation or other operations. When this type of intrusion occurs, this monitoring service can recover the system by rebooting it without affecting the normal operations of RTOS' secure domain. While other recovery mechanisms may exist, they are beyond the scope of this paper. Implementing a monitoring operation in SPUMONE might be another possible method for protecting the system from an intrusion, but it also increases SPUMONE size and makes it prone to attack.

6. Related Work

6.1 Hypervisor

The L4 microkernel [18] provides a flexible way to construct a virtualization system in embedded environments using the modular design approach. It leaves all guest systems running in the user space and communicating with each other and VMM itself using an inter-process communication mechanism. This modifies the guest systems in many codes and makes it hard for the microkernel to catch up with rapidly developed open-source systems. Because most RTOS kernel and application codes are written and executed in the kernel space, porting them on top of the microkernel requires considerable effort.

VirtualLogix's VLX [19] is also a widely used embedded virtualization system and is especially known for its good real-time responsiveness. Its design approach is similar to that of the SPUMONE system, as it also allows guest systems to run in the kernel space without having to modify too many codes; it also has the application codes reside in fixed memory areas separately, called a "sandbox", so simple isolation among guest systems is provided. Similar to

other traditional VMMs, VLX has only one virtualization layer instance. It makes this layer a new fault propagation medium if an attack controls this layer. However, there are some special hardware supports in the embedded system to provide isolation, such as ARM's Trustzone [20] technique. It uses a special hardware to separate the system into secure and unsecure domains, and the hardware can restrict access to the secure domain. Developers can use an additional monitoring service to further enhance system reliability. This type of special hardware is still rarely seen in most embedded systems.

SPUMONE's distributed design makes fault propagation impossible because only the bad instances are affected. Moreover, each SPUMONE instance is laid on the physical core's local memory area, which is inaccessible from the other areas. SPUMONE's distributed model is similar to the *Multikernel* proposed by Baumann et al. [21]: it has better scalability because the future hardware development is trending toward multi-core configurations. The main differences between SPUMONE and Multikernel are as follows:

1. SPUMONE is a virtualization system that runs on a hardware platform that has the same architecture. The Multikernel, which is not a virtualization system, addresses the distribution system's multi-core problem, and each OS on it runs directly on a processor core.
2. Guest OSs on SPUMONE may share a physical core, whereas OSs on Multikernel do not share a core with each other.

There are many designs for the embedded virtualization systems mentioned above. Each design has its own concerns and has taken some design trade-offs. We have discussed basic pros and cons and why we chose to implement SPUMONE as described in our previous work [22]. In addition to those basic design strategies, we further focus on the reliability issue and distributed virtualization design. We take advantage of SPUMONE's distributed design and use local memory to provide more isolation and security. Centralized or single-image hypervisor design, as discussed in Sect. 2.2.3, gives system maintainers more intuitive controls and makes communications among guest systems more straightforward. The distributed design lacks this type of powerful management that uses a shared-memory mechanism, as discussed in Sect. 2.3. As a trade-off, the distributed design provides better scalability that is more suitable for the embedded multi-core platform. Moreover, the proposed local memory approach in the virtualization system is easy for developers to use because of the common availability of the local memory in the embedded platform. The only disadvantage using local memory is to modify the image location in the linker script at the compile time, which is a small cost.

6.2 Local Memory

Using a local memory can be categorized into two main objectives. The first is to enhance system performance, and the other tries to minimize power consumption.

Local memory is generally used to reduce the external bus traffic and increase performance because it resides in the same processor core chip, especially in embedded system design. Embedded applications usually need some special performance adjustments, and hardware resources are not similar to those on desktop or data center computers. By exploring the memory architecture and appropriately assigning parts of a program to an on-chip or local memory, the entire system would gain more improvements from a performance perspective. Panda et al. [23]–[26] attempt to partition programs and put some program portions in local memory by estimating memory performance. They have applications for running critical parts on on-chip memory to achieve better performance. In contrast to developers handling program mapping, work in [27] leaves the program partition to the compiler level. Before using a compiler and linker to map program data to certain memory addresses, they feed profiling data to make the compiler and linker put specific data into on-chip SRAM.

Power consumption has recently gained more attention, as it is widely used in consumer electronic devices. These devices provide considerable functionality to end users, though they must be designed carefully to use hardware resources. Accessing data in the main memory is a main power consumption source. If a developer can identify or properly select program data for transfer onto on-chip local memory, significant power would be saved. In [6], a runtime program data placement is proposed. This placement uses an extra hardware controller and a new instruction set to perform runtime data selection and to put the data on the scratchpad memory. The results show that this placement can gain up to 50% energy improvement while increasing overall system performance. [28] also propose a method that identifies some frequently used basic blocks and inserts special instructions to dynamically move data and instructions to the scratchpad memory. These authors also claim that the number of instructions being copied to scratchpad memory is reduced. The overall energy consumption is also considerably reduced.

This paper presents a different way of providing more reliability and security to the embedded virtualization system. Because the local memory is only accessible to its processor core, it provides additional isolation and security to the whole system if developers carefully arrange the whole system, especially for a virtualization system. We take the advantage of SPUMONE's design, where each SPUMONE instance is installed in a different processor core and the whole SPUMONE image is put into local memory. Advanced hardware units such as Trustzone or Intel VT-x provide more convenient ways to consolidate the system and are also more secure than SPUMONE's approach because more privileged modes are introduced to avoid further system intrusion. The local memory approach, however, provides another security alternative for low-end devices that do not have special hardware units. The overall performance is also sustained without any degradation.

7. Conclusion and Future Work

In this paper, we proposed the usage of a local memory on the embedded virtualization design to build up a robust embedded virtualization environment and a new virtualization architecture for multi-core processor based embedded system. Hardware resources are relatively poor in embedded systems, so developers have to make more efficient use of these resources. By taking the advantage of the SPUMONE's distributed design in a multi-core platform, we chose to map the program sections on the local memory and this mapping can further increase the system reliability for not letting some compromised systems to destroy others because the local memory is private to each other and can only be accessed by its corresponding physical core. Data security can also be guaranteed for not exposing themselves at public shared main memory region. This is achieved without additional implementation of some software-based memory isolation approaches. In a multi-core embedded platform, however, the power consumption is another issue that has to be addressed. The virtualization layer which provides VCPUs to guest systems can also be implemented to have VCPU migration among physical cores to save more energy. With the use of the local memory in SPUMONE, the data located in local memory have to be transferred to another physical core as well. RP1 platform provides a direct inter-local memory transfer mode, but with this operation, it may increase the design complexity of the SPUMONE layer. We have to make more evaluations about this work more carefully in the future.

References

- [1] "VMware, Inc.," <http://www.vmware.com>
- [2] "Oracle Corporation," <http://www.virtualbox.org/>
- [3] "ARM Ltd.," <http://www.arm.com/>
- [4] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," *Proc. 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII*, pp.26–35, New York, NY, USA, ACM, 2008.
- [5] T. Nakajima, Y. Kinebuchi, A. Courbot, H. Shimada, T.H. Lin, and H. Mitake, "Composition kernel: A multi-core processor virtualization layer for rich functional smart products," *8th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, 2010.
- [6] A. Janapsatya, S. Parameswaran, and A. Ignjatovic, "Hardware/software managed scratchpad memory for embedded system," *ICCAD '04: Proc. 2004 IEEE/ACM International Conference on Computer-Aided Design*, IEEE Computer Society, pp.370–377, Washington, DC, USA, 2004.
- [7] R. Banakar, S. Steinke, B. sik Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," *Tenth International Symposium on Hardware/Software Codesign (CODES)*, pp.73–78, ACM, Estes Park, 2002.
- [8] "Intel VT-x," <http://www.intel.com/technology/virtualization>
- [9] "AMD-V," <http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx>
- [10] "The Xen Project, XenSource, Inc.," <http://www.xen.org/>

- [11] Y. Kinebuchi, T. Morita, K. Makijima, M. Sugaya, and T. Nakajima, "Constructing a multi-os platform with minimal engineering cost," Analysis, Architectures and Modelling of Embedded Systems: Third IFIP TC 10 International Embedded Systems Symposium, pp.195–206, Sept. 2009.
- [12] A. Whitaker, M. Shaw, and S.D. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," Proc. USENIX Annual Technical Conference, 2002.
- [13] "LMBench - Tools for Performance Analysis," <http://lmbench.sourceforge.net/>
- [14] "memspeed," <ftp://ftp.netbsd.org/pub/NetBSD/misc/gmcgarry/bench/memspeed.c>
- [15] "byte-unixbench - A Unix benchmark suite," <http://code.google.com/p/byte-unixbench/>
- [16] Y. Kinebuchi and T. Nakajima, "Core-local memory assisted protection," Proc. 16th IEEE Pacific Rim International Symposium on Dependable Computing, 2010.
- [17] H. Shimada, A. Courbot, Y. Kinebuchi, and T. Nakajima, "A lightweight monitoring service for multi-core embedded systems," IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp.202–209, 2010.
- [18] "L4 Microkernel," <http://l4ka.org>
- [19] "VirtualLogix," <http://www.virtuallogix.com>
- [20] "TrustZone technology, ARM Ltd," <http://www.arm.com/products/processors/technologies/trustzone.php>
- [21] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhanian, "The multikernel: A new os architecture for scalable multicore systems," Proc. ACM SIGOPS 22nd symposium on Operating systems principles, pp.29–44, ACM, SOSP '09, New York, NY, USA, 2009.
- [22] H. Shimada, Y. Kinebuchi, T.H. Lin, A. Courbot, and T. Nakajima, "Design issues in composition kernels for highly functional embedded systems," Proc. 2011 ACM Symposium on Applied Computing, SAC '11, pp.338–345, ACM, New York, NY, USA, 2011.
- [23] P.R. Panda, N.D. Dutt, and A. Nicolau, "Architectural exploration and optimization of local memory in embedded systems," ISSS '97: Proc. 10th International Symposium on System Synthesis, pp.90–97, IEEE Computer Society, Washington, DC, USA, 1997.
- [24] N. Dutt, "Memory organization and exploration for embedded systems-on-silicon," Proc. 1997 International Conference on VLSI and CAD, 1997.
- [25] P. Panda, N. Dutt, and A. Nicolau, "Local memory exploration and optimization in embedded systems," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.18, no.1, pp.3–13, Jan. 1999.
- [26] P.R. Panda, N.D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems," ACM Trans. Des. Autom. Electron. Syst., vol.5, no.3, pp.682–704, 2000.
- [27] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," ACM Trans. Embed. Comput. Syst., vol.1, no.1, pp.6–26, 2002.
- [28] A. Janapsatya and A.I.S. Parameswaran, "A novel instruction scratchpad memory optimization method based on concomitance metric," ASP-DAC '06: Proc. 2006 Conference on Asia South Pacific Design Automation, pp.612–617, ACM Press, 2006.

Tsung-Han Lin received B.E. and M.E. degree in Computer Science and Engineering from Yuan Ze University, Taoyuan, Taiwan, in 2005 and 2007, respectively. He is a Ph.D. student in the Department of Computer Science, Waseda University, Japan. He was a staff engineering in Andes Technology, Taiwan in 2009–2010. His current research interests include operating system, virtualization system.

Yuki Kinebuchi received B.E., M.E. and Ph.D. degree in Computer Science and Engineering from Waseda University in 2005, 2007 and 2012, respectively. His current research interests include operating system, virtualization system and system security.

Tatsuo Nakajima is a professor in Department of Computer Science, Waseda University. He was a researcher in School of Computer Science, Carnegie Mellon University in 1990–1993, a researcher engineer in AT&T Laboratories, Cambridge in 1998–1999 and a visit 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 program co-chair of RTCSA 2002 and ISORC 2003, and general chair of RTCSA 2003 and ISORC 2005. His research interests include distributed systems, operating systems, ubiquitous computing, information appliances.