SOLVING THE LOCK HOLDER PREEMPTION PROBLEM IN A MULTICORE PROCESSOR-BASED VIRTUALIZATION LAYER FOR EMBEDDED SYSTEMS

Hitoshi Mitake, Yuki Kinebuchi, Tsung-Han Lin and Tatsuo Nakajima
Department of Computer Science and Engineering, Waseda University, Tokyo, Japan

Keywords: Virtualization Technologies, Real Time Systems, Embedded Systems.

Abstract: In this paper, we explain the reason why the Lock Holder Preemption(LHP) problem is serious when using a multi-core processor based virtualization layer. Then, we introduce two new techniques for avoiding the LHP problem. The existing techniques and new proposed techniques have been implemented on our virtualization layer called SPUMONE, and we measured the results showing that the proposed new techniques reduce the semantic gap to use a virtualization layer on a multi-core processor in embedded systems.

1 INTRODUCTION

As predicted by the Moore's law, today's computer systems have become significantly powerful. The powerful computing platforms make it possible to virtualize the platforms to execute multiple operating systems on a single processor. In server side computing, virtualization already became a de-facto standard technology, mainly for integrating multiple servers. In this field, virtualization significantly reduces the cost of engineering, management, and hardware resources.

Increasing the number of processor cores is becoming popular trend in current embedded systems. This trend is very attractive because multi-core processors containing several cores that run at the low clock frequency require less energy than processors containing only one core and running at the high clock frequency, if the parallelism of applications is well exploited. This benefit, reducing energy consumption, is especially important for embedded devices because they may run with limited batteries.

However, developing dedicated software for every rich functional device introduces a significant engineering cost. Reusing existing software is the most important approach not to increase the cost of highly functional embedded systems. The virtualization technology offers a possibility to reuse existing software without modifying it. Therefore, this approach reduces the development cost significantly.

A traditional RTOS is suitable for executing real-time applications, but lacks the huge software library that GPOSes like Linux have. Developing modern embedded devices with both rich interfaces and guaranteed real-time responsiveness by using either GPOS or RTOS is a very difficult task. So running the two types of OSes on the same device is a promising approach to combine the best of both worlds. Especially on multicore processor based embedded systems, sharing one core by several OSes is effective to use the CPU resource efficiently. If the interference between OSes is severe, the cores should be statically assigned to respective OSes. But this approach might produce high amounts of processors' idle time, and from a hardware cost perspective, it is not economical.

For increasing the throughput of GPOS, SMP OSes are becoming popular. For example, Linux currently supports SMP very well, and many applications on Linux are already parallelized to exploit it. Because of the today's trend of cloud computing, web browsers became especially important for client side computers including mobile terminals. For example, Jones et al. showed that a web browser is similar to a compiler because it uses a large amount of processing power for lexical analyzing, syntax parsing and rendering web pages, and has lots of potential parallelism (Christopher Grant Jones and Bodik, 2009). So parallelizing web browsers is an efficient approach to reduce energy consumption and to utilize SMP OS efficiently. The virtualization layer allows both SMP OS and RTOS to coexist on the same multi-core pro-

Mitake H., Kinebuchi Y., Lin T. and Nakajima T..
SOLVING THE LOCK HOLDER PREEMPTION PROBLEM IN A MULTICORE PROCESSOR-BASED VIRTUALIZATION LAYER FOR EMBEDDED SYSTEMS.
DOI: 10.5220/0003800603690377
In *Proceedings of the 2nd International Conference on Pervasive Embedded Computing and Communication Systems* (PECCS-2012), pages 369-377
ISBN: 978-989-8565-00-6

cessor. As described above, this approach reduces the development cost by reusing software significantly.

When RTOS and SMP GPOS share one SMP system, there is a possibility that the RTOS preempts the SMP GPOS even when the SMP GPOS executes a code in a critical section. This preemption may cause critical performance degradation of the SMP GPOS. The problem is called the *lock holder preemption* (LHP) problem. The existing solution for solving the problem is called the *delayed preemption* technique (Uhlig et al., 2004). When a kernel thread of SMP GPOS executes a critical section, RTOS is prohibited to preempt the SMP GPOS. Thus, the LHP problem does not occur, and there is no throughput degradation of GPOS. However, the technique decreases the real-time responsiveness of RTOS as described in Section 5 because the critical section in GPOS like Linux is not enough short.

In this paper, we propose two new techniques for avoiding the LHP problem. Both techniques rely on the *vCPU migration* mechanism to migrate a virtual core implemented by a virtualization layer among physical cores. The first technique is called the *trap based migration* technique, and the second technique is the *on demand migration* technique. The two techniques have different tradeoffs in terms of real-time responsiveness and overhead. Each system needs to choose the suitable one by taking into account the tradeoffs of both techniques and the requirements of each system. We have implemented the *delayed preemption* technique and the proposed two techniques on SPUMONE, a virtualization layer developed in our research group. We also show the evaluation results of the three techniques. The results present the merits and demerits of each technique clearly.

The rest of the paper is structured as follows. We first explain the LHP problem and show the effect on SPUMONE in Section 2. In Section 3, we present an overview of SPUMONE. Section 4 presents the *delayed preemption* technique and the effect on SPUMONE. In the section, two new techniques are also proposed and we show how to implement them in SPUMONE. The evaluation of the new techniques to avoid the LHP problem is shown in Section 5 and Section 6 summarizes the paper.

2 THE LOCK HOLDER PREEMPTION(LHP) PROBLEM IN A VIRTUALIZATION LAYER

The LHP problem occurs in SMP OS in the following situation: on a virtualization layer, multiple OSes may run simultaneously on the same core. We assume that RTOS uses one virtual core, and SMP GPOS uses two virtual cores offered by a virtualization layer. We also assume that the virtual core for RTOS and one virtual core of SMP GPOS share the first physical core and the other virtual core used by SMP GPOS that runs on the second physical core. Now, a virtual core used by SMP GPOS holds a spin lock, and the virtual core used by RTOS becomes ready and preempts the execution of the virtual core of SMP GPOS. When another virtual core of SMP GPOS tries to acquire the same spin lock, it needs to wait for the virtual core to release the lock after RTOS becomes idle.

We also assume that the priority of RTOS is higher than the priority of GPOS. This is a natural configuration to use both RTOS and GPOS simultaneously. So, the preempted GPOS cannot resume the execution until all activities in RTOS becomes idle. Therefore, there is a high possibility that the lock holder waits for a long time to be resumed and that other physical cores are also stopped until RTOS becomes idle. This degrades the throughput of SMP GPOS significantly. Of course, the LHP problem is well discussed is the case when multiple SMP GPOSes run on a multi-core processor. However, the combination of RTOS and SMP GPOS may cause more serious performance degradation.

There is also another problem related to the LHP problem. Typical SMP GPOSes like Linux use the inter core interrupt mechanism to synchronize between physical cores. For example, the TLB shutdown uses the mechanism to keep the consistency of TLBs of all physical cores. GPOSes usually assume that the synchronization cannot be preempted by other activities. Therefore, the preemption of the synchronization also causes significant performance degradation, and in the worst case, it may cause the deadlock in the GPOS kernel.

We demonstrate the effect of the LHP problem with a virtualization layer called SPUMONE.

In this demonstration, we are using SMP Linux as SMP GPOS and TOPPERS/JSP (which we simply call "TOPPERS") (Toppers, 2011) as RTOS[1]. Figure 1 shows the result of running the hackbench benchmarking program (Hackbench, 2011) on Linux, when TOPPERS consumes CPU time every 500ms. A virtual core is assigned to TOPPERS and four virtual cores are assigned to Linux. The virtual core for TOPPERS and one virtual core for Linux shares one physical core. Three other virtual cores for Linux use the remaining three respective physical cores.

[1]TOPPERS is a open source RTOS that offers μITRON interface, and it is used in many Japanese commercial products.

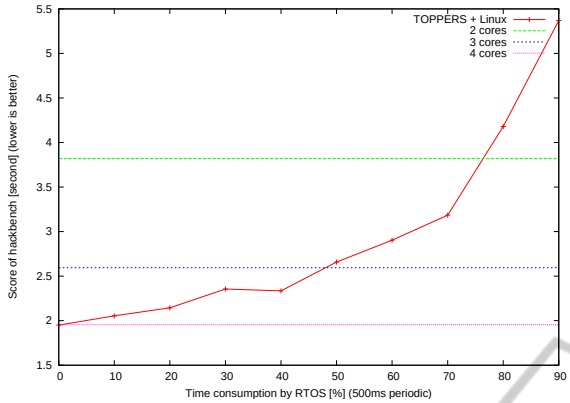

Figure 1: Score of hackbench on SPUMONE when the LHP problem does not take into account.

The X axis indicates the rate of CPU consumption by TOPPERS, where the unit is 10%. The Y axis represents time in seconds which was required to complete the execution of hackbench, where the lower value means the better score with better throughput. This graph contains three horizontal lines. Each of these horizontal lines describes the score of hackbench when Linux dominates 2, 3 or 4 physical cores without executing TOPPERS. As the graph shows, when the CPU consumption of TOPPERS is lower than 50%, the performance of Linux is better than the situation where Linux dominates three dedicated physical cores. However, when the CPU consumption of TOPPERS becomes higher than 50%, the result of hackbench becomes worse than when Linux dominates three dedicate physical cores. Moreover, when the CPU consumption of RTOS exceeds 80%, the result is worse than when Linux dominates only two dedicated physical cores. The result means that the throughput of SMP Linux is significantly degraded when the CPU utilization of RTOS become high and that there is a possibility that the execution of the Linux kernel is stopped for a long time until the lock holder in Linux is resumed.

3 AN OVERVIEW OF SPUMONE

3.1 Basic Design Principle

SPUMONE is a virtualization layer for single and multi-core processor based embedded systems. In the design of SPUMONE, our design is to satisfy the requirements for developing a virtualization layer for embedded systems described in (Armand and Gien, 2009). SPUMONE offers the para-virtualized interface to guest OSes because most of processors for embedded systems do not offer hardware virtualization supports like x86. As shown in [2], the size of SPUMONE is very small, and the overhead is also very small. Minimum modification of guest OSes is one of the most important requirements described in (Armand and Gien, 2009). In our design, we decide modify only the initialization code and the interrupt dispatching mechanism in each guest OS. The approach was adopted VirtualLogic VLX(Armand and Gien, 2009), and the virtualization layer has been adopted in many commercial embedded system products. VLX is not open source software, so we decided to develop SPUMONE. The architectures of SPUMONE and VLX is very similar when used on a single core processor, but the architecture of SPUMONE is dramatically different on a muticore processor. As described in IV.B, our architecture does not have data structures shared by multiple physical cores, so SPUMONE does not require to use the complex multiprocessor synchronization mechanism that may cause problems in real-time systems. Also, our architecture can use physical core more efficiently by moving a guest OS according to its workload, and unused physical core can be turned off to reduce energy. L4(Heiser, 2009) is another virtualization layer for embedded systems. L4 offers a high level para-virtulization interface, and it offers the isolation among guest OSes. A guest OS on L4 needs to replace all privileged instructions, so the amount of modification of guest OSes becomes large. In SPUMONE, guest OSes directly invoked privileged instructions, and radically new mechanisms to isolate a virtualization layer and guest OSes using multicore processors without increasing the modification of guest OSesare offered. Therefore, SPUMONE is a promising platform for multicore processor basedembedded systems than traditional virtualization layers because the amount of modification of guest OSes is significantly less than other virtualization layer.

The most important abstraction offered by SPUMONE is vCPU. vCPU is a virtual core, and multiple vCPUs can be multiplexed on a physical core. Each guest OS requires a necessary number of vCPUs, and the total number of vCPUs can exceed the number of physical cores. Figure 2 shows an overview of SPUMONE. In the figure, SPUMONE runs on a single core processor, and offers two vCPUs. One vCPU is used for RTOS and another vCPU is used for GPOS. Each guest OS contains its own scheduler to multiplex a set of processes implemented in the guest OS. Therefore, each guest OS maintains its own scheduling policy to schedule its own pro-

[2]The modification of a guest OS is less than 100 lines, and the overhead is less than 2%.

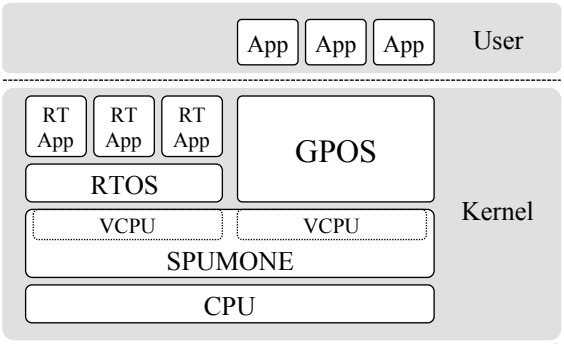

Figure 2: The structure of SPUMONE on a single core processor.

cesses.

SPUMONE also offers a scheduler to schedule multiple vCPUs. In the current implementation, the fixed priority scheduling is used to schedule RTOS and GPOS. vCPU for RTOS has always a higher priority than the priority of vCPU for GPOS. The vCPU for RTOS can preempt the vCPU for GPOS anytime to ensure the real-time responsiveness of RTOS. The interrupt is also virtualized by SPUMONE. SPUMONE intercepts all interrupts, and decides which interrupt should be delivered to respective guest OSes. For ensuring the real-time responsiveness of RTOS, even the interrupt handlers of GPOS are always preempted by RTOS. When multiple vCPUs executing the SMP Linux kernel are multiplexed on the same physical core, the vCPUs are scheduled by the timesharing scheduler.

The current target processor of SPUMONE is the SH4a architecture, which is the high-end processor in the SuperH (Corporation, 2011) RISC processor family. Linux, TOPPERS, and L4 currently run on SPUMONE.

3.2 Supporting Multicore Processors

SPUMONE is currently supporting a shared memory-based multi-core processor. Figure 3 shows the structure of SPUMONE on a multi-core processor. The most important feature for supporting multi-core processors is to adopt the distributed model, where each core has its own instance of a virtualization layer. The approach is significantly different from the traditional approach that has only one instance shared by all physical cores. The distributed model offers better scalability in terms of a number of physical cores (Baumann et al., 2009). Also, the model does not require the synchronization among cores to access most of key data structures. Thus, the single core version and the multicore version can share the same binary code. This improves the maintainability of the

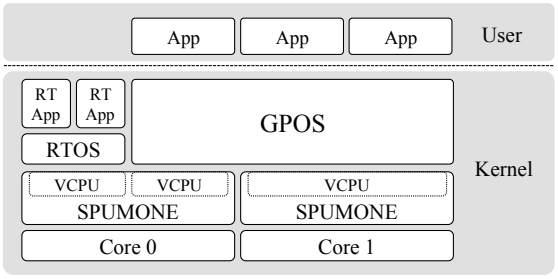

Figure 3: The structure of SPUMONE on a multi-core processor.

virtualization layer significantly. Traditional virtualization layers like Xen suffer significant scalability problems because shared data structures are accessed from multiple cores simultaneously. This is an important design issue to reduce the overhead of virtualization layer when they are used on multicore processors. However, as described in Section II, the LHP problem need to be solved to utilize multicore processors effectively.

In Figure 3, we assume that SPUMONE runs on a dual core processor. Each core executes a separate instance of SPUMONE. The SPUMONE instance running on core 0 offers two vCPUs and the instance running on core 1 offers one vCPU. One vCPU of core 0 is used by RTOS and another vCPU of core 0 is used by GPOS. The vCPUs of core 1 is also used by GPOS. Thus, GPOS has two vCPUs. The configuration may cause the LHP problem when the vCPU on core 0 for GPOS is preempted by the vCPU for RTOS.

In SPUMONE, the mapping between vCPUs and physical cores is dynamically changed according to the current situation of guest OSes. If the total utilization of guest OSes becomes low, all vCPUs may share only one core and the power of other cores can be turned off. This approach offers a possibility to reduce the power consumption significantly. Also, when RTOS becomes idle, GPOS can use the entire utilization of a multicore processor. This means that multiple vCPUs used by an SMP GPOS may share the same physical core to utilize multicore processors more efficiently according to the current situation. However, for achieving the maximum throughput, the LHP problem should be taken into account.

For realizing the flexible management of a multicore processor, SPUMONE offers a mechanism called the *vCPU migration* mechanism. The mechanism moves vCPU from one physical core to another physical core. The images of guest OSes reside in the shared memory, so the mechanism just copies only the register states between different SPUMONE instances. The mechanism uses inter core interrupts to synchronize between physical cores. A more detailed

implementation will be explained in the next section because the *vCPU migration* mechanism is a key underlying infrastructure for the new techniques to avoid the LHP problem.

The multicore version of SPUMONE runs on the MSRP1 board developed by Hitachi and Renesas. The board contains a multicore processor called RP1, which consists of four SH4a cores and 128MB DRAM as main memory. The memory is shared by all cores.

4 IMPLEMENTATION TO AVOID THE LHP PROBLEM ON SPUMONE

Currently, we are using Linux as SMP GPOS and TOPPERS as RTOS. In this section, we describe how we implemented techniques to avoid the LHP problem on SPUMONE.

4.1 Implementation of the Delayed Preemption Technique

In order to compare the effectiveness of the delayed preemption technique with our new techniques based on the *vCPU migration* technique, we implemented the *delayed preemption* technique on SPUMONE.

Our current implementation exploits the internal structure of Linux. Every thread in Linux has its own data structure for management. This data structure is named *struct thread_info*, and it has a field named *preempt_count*. *preempt_count* indicates whether the thread is in the IRQ context and how many locks the thread holds. We implemented the *delayed preemption* technique by using the *preempt_count* field. When the *preempt_count* field of the currently running thread becomes bigger or equal to 1, our modified Linux kernel invokes SPUMONE API to notify to disable the preemption of Linux. When the *preempt_count* field of the thread reaches to 0, Linux invokes SPUMONE API to enable the preemption of Linux.

When RTOS becomes ready, it can usually preempt GPOS anytime. However, the *delayed preemption* technique does not allow RTOS to preempt GPOS while GPOS holds a lock. Thus, the thread dispatch of RTOS is delayed until the lock is released. This means that the dispatch latency is degraded according to the length to hold a lock.

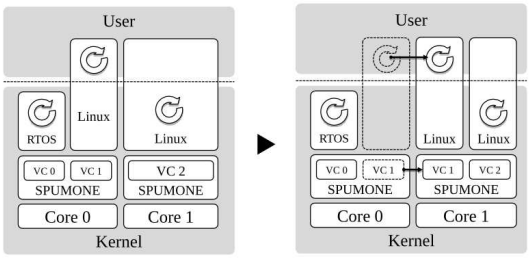

Figure 4: The vCPU migration mechanism.

4.2 Avoiding the LHP Problem using the vCPU Migration Mechanism

In this section, we first describe a brief overview of the vCPU migration mechanism of SPUMONE. Then, we show two new techniques based on the *vCPU migration* mechanism to avoid the LHP problem.

4.2.1 Implementation of the vCPU Migration

SPUMONE provides the *vCPU migration* mechanism for moving vCPUs owned by guest OSes. By using this mechanism, guest OSes can change the physical core that executes their vCPUs.

The vCPU migration in SPUMONE has two types. One is *departure migration*, and another is *return migration*. In Figure 4, vCPU VC 1 runs on core 0 in the left figure. *departure migration* moves VC1 on core 0 to core 1 as shown in the right figure, and *return migration* moves VC1 from core1 to core0. The vCPU migration is invoked by using the inter core interrupt mechanism.

The source core that migrates a virtual core executes *spm_cpu_migrate()* to initiate the migration, and the destination core executes *accept_immigrant()* to resume the migrated vCPU. The overhead of the migration is the sum of the two functions. As shown in Section 5.4, the overhead of the vCPU migration is small.

4.2.2 Trap based Migration Technique

As described in Section 2, the LHP problem occurs when one of vCPUs for Linux is preempted by TOPPERS while it executes a critical section in the kernel space. In Figure 4, the situation shown in the left figure may cause the LHP problem. However, the LHP problem occurs only when the Linux kernel executes the kernel code. If Linux does not invoke the kernel, the problem does not occur.

This technique does not allows Linux to execute the kernel code on core 0. When Linux invokes a trap

instruction or receives an interrupt, *departure migration* is invoked and the vCPU of Linux running on core 0 is moved to core 1. After returning from the trap or interrupt, *return migration* is invoked and the vCPU is moved from core 1 back to core 0. This technique can be easily implemented on SPUMONE because SPUMONE intercepts all traps and interrupts before forwarding them to guest OSes.

When using this technique, RTOS can always preempt Linux without causing the LHP problem. Thus, it does not degrade the real-time responsiveness. However, it requires to move the vCPU every time traps and interrupts are invoked. The overhead of moving the vCPU may become a problem if the frequency of traps and interrupts becomes high. At least, every interrupt causes a vCPU migration even if there is no user level activity.

4.2.3 On Demand Migration Technique

When using the technique, a vCPU for Linux is migrated from core 0 to core 1 while RTOS becomes active on core 0. When the RTOS becomes idle, the vCPU of Linux is backed from core 1 to core 0.

When RTOS becomes ready, *departure migration* is invoked, and *return migration* is invoked when the RTOS becomes idle. This technique can solve the LHP problem because before RTOS preempts Linux, Linux is migrated to another core.

However, before handling an interrupt of RTOS, Linux needs to be migrated to another physical core. The preemption of RTOS needs to wait for the completion of *departure migration* to move the vCPU from core 0 to core 1. This means that the technique increases the interrupt latency, but this increased latency can be bounded by the worst case latency of the *departure migration*. The technique requires to invoke the *vCPU migration* mechanism, whenever RTOS becomes active or idle. This means that every timer interrupt causes the vCPU migration even if there is no active thread on RTOS.

5 EVALUATION

In this section, we show the evaluated results of the delayed preemption technique and of the new techniques based on the *vCPU migration* mechanism. We especially measured the following two performance aspects:

- Dispatch latency of RTOS.
- Maximum throughput of GPOS.

In our evaluation, the dispatch latency of RTOS means the elapsed time to activate the highest priority thread after an interrupt that makes the thread ready is received by a processor.

The maximum throughput of GPOS shows the effect of the proposed solution. In the measurement, there are two possibilities to degrade the throughput of GPOS. The first possibility is caused by the LHP problem, and the second possibility is caused by the overhead of the proposed technique. Our solutions can solve the LHP problem, but if the overhead is big, the solutions may degrade the throughput.

In the following subsections, we show the results of the two aspects, and interpret their significance.

5.1 Evaluation Environment

When executing both TOPPERS and Linux on the multicore processor, one physical core multiplexes one vCPU for TOPPERS and one vCPU for Linux. The other three vCPUs for Linux run on dedicated physical cores. *departure migration* moves the vCPU of Linux to another physical core. In this case, two vCPUs for Linux share the same physical core. In the measurement, at the begginning, we do not take into account the LHP problem caused when multiple vCPUs for Linux are executed on one physical core. Our focus is the LHP problem when SMP Linux is preempted by TOPPERS, but as shown later, the LHP problem within SMP Linux also causes serious performance degradation, and needs to be taken into account.

5.2 The Impact on RTOS Dispatch Latency

In this experiment, a periodic task runs every 1ms. It is sampled 100,000 times during the measurement. The dispatch latency is the time spent from the interrupt triggered until the periodic task starts its execution. Only the periodic task is executed on TOPPERS which means that no other task on TOPPERS will prevent the execution of the periodic task.

Figure 5, 6 and 7 show the dispatch latency in TOPPERS where running hackbench on Linux[3]. Our approach improves the dispatch latency significantly compared to the *delayed preemption* technique. The reason of this improvement is that our approach does not execute the Linux kernel with RTOS at the same time. When RTOS becomes runnable, vCPU executing Linux is migrated to another core. The source of the increase of interrupt latency is the time to disable

[3]The avarage is 24.09 μ when using the *delayed preemption* technique, 2.30 μ when using the *trap based migration* technique, and 4.71μ when using the *on demand migration* technique.

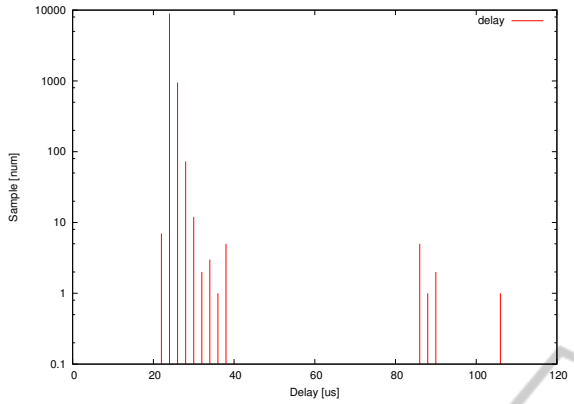

Figure 5: Dispatch latency with the *Delayed Preemption* technique.

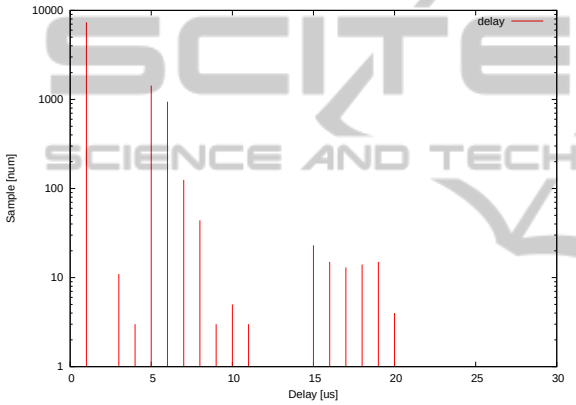

Figure 6: Dispatch latency with the *Trap based Migration* technique.

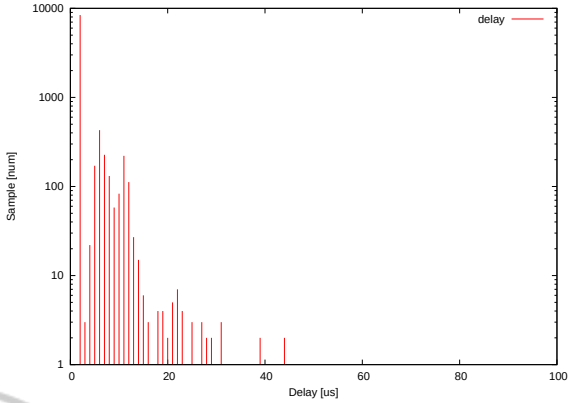

Figure 7: Dispatch latency with the *On Demand Migration* technique.

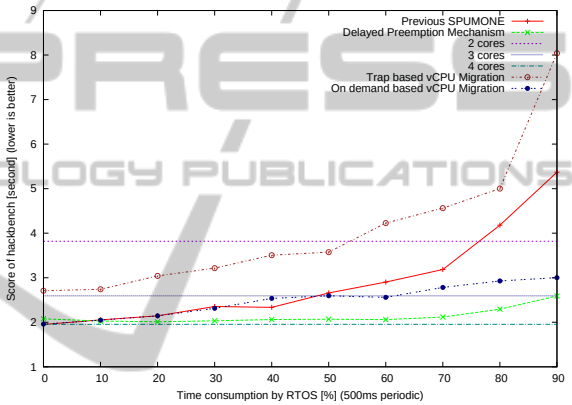

Figure 8: The hackbench scores in Four Configurations(1).

interrupts. In the Linux kernel, there are many places to disable interrupts and they have a significant impact on the dispatch latency.

As shown in , the dispatch latency without the migration based techniques is almost the same as the latency when using them. Thus, the techniques solve the LHP problem significantly, but also they do not degrade the dispatch latency.

5.3 The Impact on GPOS Throughput

We compared the score of the hackbench benchmark which evaluates the scalability of the number of cores with Linux running on the top of four dedicated cores (indicated as four cores in the Figure 8 and Figure 9), Linux running on the top of three dedicated cores and one core shared with TOPPERS in various workloads (xx% in the figures), and Linux running on the top of three dedicated cores (indicated as three cores in the figures). The task on TOPPERS is executed in the cycle of 500 ms. The percentage shows the ratio of the execution time of the periodic task against the cycle (30% means that the task is executed for 150 ms continuously).

The hackbench program executing on SMP Linux that has four vCPUs. One of the vCPUs shares a physical core with the vCPU of TOPPERS. In the evaluation, we change the utilization of a periodic task on RTOS.

When the utilization of RTOS is high, the possibility to preempt critical sections and cause the LHP problem becomes high. Hackbench creates many processes that communicate each other. Hackbench is executed in the kernel almost of the entire time. The score becomes better when the kernel overhead is low. When the LHP problem occurs, the kernel remains busy waiting for a long time. Thus, the LHP problem makes the score of hackbench bad. We consider that this benchmark is suitable to measure the worst case effect of the LHP problem.

Figure 8 shows the score of hackbench in four configurations. The first configuration does not use any techniques to avoid the LHP problem. The result

shows that the LHP problem significantly degrades the throughput of hackbench. The second configuration shows the result when the *delayed preemption* technique is used. The result indicates the technique solves the LHP problem, and the utilization of RTOS proportionally affects the score of hackbench. However, as shown in the previous section, the technique increases the dispatch latency of RTOS significantly. The third configuration adopts the *trap based migration* technique. The result is not good as we expected due to the overhead of virtual core migration mechanism because hackbench invokes system calls very frequently. The last configuration adopts the *on demand migration* technique. This configuration can improve the throughput significantly because the approach solves the LHP problem, and the overhead is small.

However, the results of the figure show that the throughput achieved by our techniques is not as good as the *delayed preemption* technique is used. When using our proposed approach, some virtual cores used by SMP Linux share the same physical core. Because the vCPUs are scheduled by the time sharing scheduler in SPUMONE, the execution of a critical section in the Linux kernel may be preempted by the other vCPUs executing the SMP Linux kernel, thus it might cause another LHP problem. For solving this LHP problem within SMP Linux, we modified SMP Linux to yield the vCPU when the length of busy waiting for entering a critical section exceeds a pre-determined threshold. Figure 9 shows the results when applying the technique. In this case, the LHP problem is completely solved without degrading real-time responsiveness.

The above discussion shows that the *trap based migration* technique solves the LHP problem, but also that the overhead to invoke frequent vCPU migrations is high, so the GPOS throughput does not improved when the utilization of RTOS is high. On the other hand, the *on demand migration* technique improves the GPOS throughput dramatically without degrading real-time responsiveness. Thus, the results show that the *on demand migration* technique is well fit to be used in embedded systems.

5.4 Overhead of the vCPU Migration Mechanism

As we described in Section 4.2.1, the main source of overhead of the vCPU migration mechanism occurs in accept_immigrant() and smp_cpu_migrate(). Figure 12 shows the measured costs of the two functions in the *departure migration* and *return migration*. The sum of the costs of the two functions indicates the actual cost of the vCPU migration, which is about 50μs.

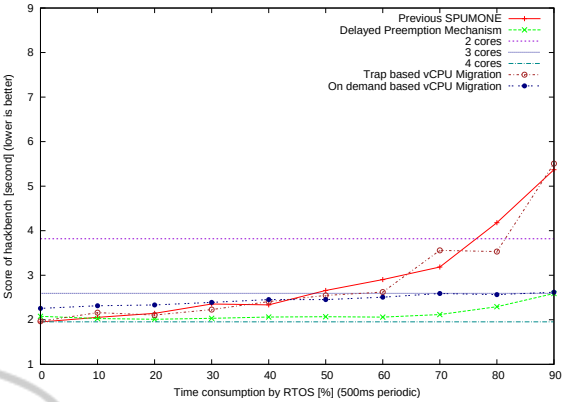

Figure 9: The hackbench scores in Four Configurations(2).

Migration path	spm_cpu_migrate()	accept_immigrant()
Departure migration	24.9μs	28.0μs
Return migration	38.1μs	12.0μs

Figure 10: Overhead of the vCPU migration.

If the frequency of vCPU migration is increased, the overhead to avoid the LHP problem is also increased. The result shows that the effectiveness of the proposed techniques depends on the workload running on Linux. Also, it depends on the workload of real-time applications. Especially, the utilization of real-time activities and the frequency of resuming and suspending RTOS have significant impact on the effectiveness of the proposed techniques.

6 CONCLUSIONS

When a virtualization layer supports a shared memory based multi-core processor, the LHP problem becomes very serious. The existing technique called the *delayed preemption* technique solves the problem and exploits the maximum merits of multicore processors. However, this technique decreases the real-time responsiveness of RTOS. The existing solution is adopted in virtualization layers for enterprise servers because the maximum throughput is the most critical design criteria in this area. However, is is not appropriate for embedded systems, which need to satisfy the real-time constraints. We proposed two new techniques based on the *vCPU migration* mechanism to avoid the LHP problem. The measure results show that the *trap based migration* technique reduces dispatch latency and solves the LHP problem. However, it does not improve the GPOS throughput due to the overhead of frequent vCPU migration due to system

calls. On the other hand, the *on demand migration* technique solves the LHP problem and reduces the dispatch latency. Also, the overhead is not large, so the GPOS throughput is not degraded. Therefore, the *on demand migration* technique is well fit to be used in embedded systems.

REFERENCES

Hackbench. (2011). http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c

Toppers project. (2011). http://www.toppers.jp/en/index.html

Armand, F. and Gien, M. (2009). A practical look at micro-kernels and virtual machine monitors. In *Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*, CCNC'09, pages 395–401, Piscataway, NJ, USA. IEEE Press.

Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. (2009). The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA. ACM.

Christopher Grant Jones, Rose Liu, L. M. K. A. and Bodik, R. (2009). Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*.

Renesas Electronics Corporation. (2011). Superh risc engine family. http://www.renesas.com/products/mpumcu/superh/superh_landing.jsp

Heiser, G. (2009). Hypervisors for consumer electronics. In *Proceedings of the 6th IEEE Consumer Communications and Networking Conference*.

Ousterhout, J. K. (1982). Scheduling techniques for concurrent systems. In *Proceedings of Third International Conference on Distributed Computing Systems, 1982*.

Uhlig, V., LeVasseur, J., Skoglund, E., and Dannowski, U. (2004). Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, Berkeley, CA, USA. USENIX Association.