Implementation of Compositional Scheduling Framework on Virtualization

Jungwoo Yang, Hyungseok Kim, Sangwon Park, Changki Hong and Insik Shin
Department of Computer Science, KAIST,
373-1 Gusung-Dong, Yusung-Gu, Daejeon 305-701, South Korea
{*jungwoo.yang, witbring, hudoni, ckhong, insik.shin*}*@kaist.ac.kr*

***Abstract*—Virtualization has been receiving increasing attention in embedded real-time systems. However, real-time systems, whose correctness depends on timing requirements, are not easily applicable to virtualization since virtualization mainly focuses on functional correctness. A hierarchical scheduling framework (HSF) provides a method of composing the complex timing requirements of real-time systems. There have been several works on the implementation of the HSF. Although the scheduling framework of virtualization directly corresponds to the HSF, they did not consider implementing the HSF on virtualization. In this paper, we implement a two-level HSF, where components use a periodic interface model using virtualization. We use an L4/Fiasco micro-kernel as a virtual machine monitor (VMM) and an L4Linux as a virtual machine (VM) and extend these to support real-time properties. The experimental results show that the HSF is suitable for the virtualization environment.**

***Keywords*-Real-time scheduling, Compositional real-time guarantees, Virtualization**

I. Introduction

Virtualization has been receiving an increasing amount of attention in embedded real-time systems [1]. Virtualization is a methodology of dividing the resources of a computer into multiple execution environments, and it offers many benefits. First, it supports heterogeneous operating system environments, even on a single processor. The general purpose OSs and real-time OSs are allowed to run on the same processor concurrently. Second, it supports architectural abstraction so that the same software architecture can be migrated, essentially unchanged, between different systems. Third, a scalable hypervisor is able to adjust the number of cores for an application domain in a multicore environment. Fourth, virtualization can provide security. Even if one of the guest OSs has security flaws, the other guest OSs are able to remain safe. Moreover, it supports the efficient distribution of application software by shipping the program together with its own OS image.

Virtualization uses a layer of software that provides an illusion of a real machine to multiple instances of a virtual machine (VM) or guest OSs. This layer is traditionally called a virtual machine monitor (VMM) or hypervisor. The hypervisor allows multiple operating systems to run concurrently on a host computer. The scheduling framework of virtualization directly corresponds to a hierarchical scheduling framework (HSF) [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. The HSF is introduced to support CPU time sharing among applications (components) under different scheduling services. A typical example of the HSF is a two-level scheduling framework consisting of a system scheduler and a number of component schedulers (see Figure 1). Under the system-wide scheduling, the system scheduler allocates the CPU to components. Under the component-wide scheduling, the component scheduler inside each component subsequently allocates a share of CPU (given to the component by the system scheduler) to its own internal tasks. In Figure 1, the system scheduler of the system corresponds to the VMM's scheduler, and each component of the system corresponds to the VM running on top of the VMM. Since each VM has its own operating system, the OS's scheduler can be considered as the component scheduler in the two-level hierarchical scheduling framework. Applications running on top of the VM correspond to internal tasks inside the component.

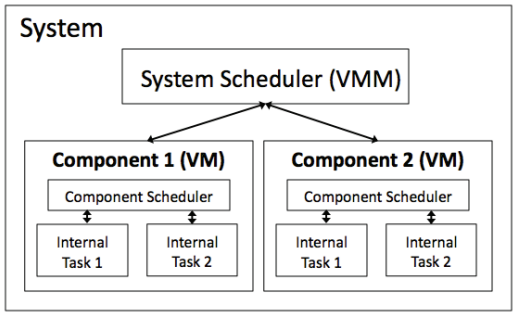

Figure 1. The architecture of the hierarchical scheduling framework.

There have been several works on the implementation of the HSF. Behnam *et al.* [13] implemented a two-level HSF in a commercial real-time operating system, VxWorks. Each component in the system is a set of periodic tasks, and each component specifies the collective real-time requirements of its internal tasks in the form of a periodic interface model. By using the periodic interface model in the HSF, the system scheduler provides the component with CPU resources according to the timing requirements imposed by its component interface. When a component receives the CPU, a user scheduling routine (USR), which is shared by every component, is triggered to schedule internal tasks depending on a real-time scheduling algorithm of the component. The

USR changes the priority of each internal task in terms of the scheduling algorithm, and then puts a tasks in the ready queue (priority queue) of the system scheduler. Heuvel *et al.* [14] implemented the HSF with SRP-based synchronization protocols in a real-time operating system, μC/OS-II. They implemented the stack resource policy (SRP) as a component synchronization protocol and implemented HSRP and SIRAP as the system SRP-based synchronization protocol. Moreover, they investigated the system overhead induced by the synchronization primitives of each protocol. Both works implemented the HSF in real-time operating systems, but they did not consider virtualization.

In this paper, we implement a two-level HSF, where components use the periodic interface model using virtualization. We use an L4/Fiasco micro-kernel as a VMM and an L4Linux as a virtual machine. We extend the L4/Fiasco micro-kernel to provide the separation between the L4/Fiasco micro-kernel and the L4Linux and support the schedulability in a compositional manner from components to the system. We extend the L4Linux to support real-time scheduling algorithms. In addition, we revise the L4Linux micro-kernel such that it can compose the collective timing requirements of its internal tasks into its periodic interface and pass the interface to the L4/Fiasco micro-kernel. Our evaluation shows that it is feasible to implement HSF over virtualization for compositional schedulability with small overheads. In particular, it shows that periodic component interfaces can be efficiently derived out of a set of internal tasks, and it can effectively enforce VM scheduling over the L4/Fiasco micro-kernel at low cost.

The rest of this paper is organized as follows: Section 2 presents an overview of the system. Section 3 describes system architecture and implementation details. In Section 4 we provide comparisons among our implementation and other systems in terms of the deadline miss, and then compute an overhead of our approach.

II. Overview

Figure 2 illustrates a hierarchical real-time system with virtualization. A hypervisor runs on the actual machine with proper abstraction towards its guest OSs. More than one guest OSs can be executed upon the hypervisor. They are scheduled by the real-time scheduler of the hypervisor. We consider that each guest OS supports periodic real-time tasks, and it has its own internal real-time scheduler so that the real-time tasks running on the guest OSs can be scheduled under the particular scheduling policy of their guest OS.

In such an environment, we consider that the hypervisor schedules its guest OSs to satisfy the real-time requirements of the individual internal periodic tasks of each guest OS. The hypervisor may fail to allocate the processor to guest OSs satisfying such real-time requirements if the hypervisor does not explicitly consider the requirements in its scheduling policy. For example, suppose that the scheduling algorithm of a hypervisor is the 'round robin algorithm'. Each guest OS will be scheduled in a fair manner, but without consideration of its internal tasks' real-time constraint. In this case, a guest OS can miss a deadline if the guest OS has an internal task approaching its deadline with remaining execution, while the hypervisor schedules other guest OSs until the deadline. Hence, it entails a scheduling framework where the hypervisor is aware of the urgency of individual guest OSs. We assume that there is no dependency between guest OSs, as well as real-time tasks.

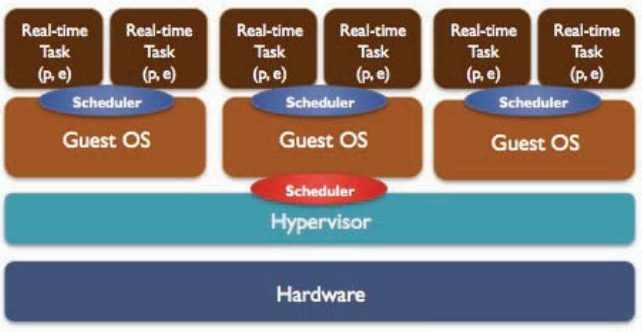

Figure 2. Hierarchical Real-Time System Architecture.

Note that the hierarchical real-time system with virtualization naturally brings up the HSF. Each guest OS is associated with a component, and the scheduler of the hypervisor is associated with the system scheduler. Each guest OS composes the collective timing requirements of its real-time internal tasks into its periodic interface and passes the interface to the hypervisor. We define *periodic interface* I as (Π, Θ^+), where Π is the guest OS period and Θ^+ is the collective timing requirement needed for the real-time tasks of the guest OS. The hypervisor supports the timing requirements by scheduling guest OSs according to its periodic interfaces.

III. System Architecture

For the virtualization environment, we use an L4/Fiasco and an L4Linux as a hypervisor and a para-virtualized guest OS, respectively. The overall system architecture looks like Figure 2. We revised L4Linux server to support the periodic model and real-time scheduling. The L4Linux servers, on which real-time tasks are running, provide their periodic interfaces to the L4/Fiasco so that the L4/Fiasco schedules the L4Linux servers by establishing communication between components and the L4/Fiasco micro-kernel for passing the component interfaces.

The design issues of the L4/Fiasco are similar to L4Linux's in terms of that each of them schedules the periodic tasks. For the L4/Fiasco and the L4linux, we consider following two issues:

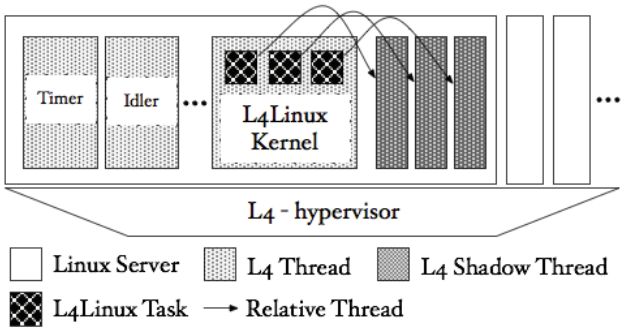

Figure 3. The virtualization environment of an L4/Fiasco

1) Supporting a periodic thread(task),
2) Scheduling periodic threads(tasks).

Also, some hierarchical scheduling issues between the L4/Fiasco and the L4linux are considered.

We define 'thread' as a scheduling unit of the L4/Fiasco, 'task' as a scheduling unit of the L4Linux, and 'L4Linux server' as a collection of L4/Fiasco threads associated with one L4Linux to reduce confusion.

A. Hypervisor: L4/Fiasco

Figure 3 illustrates the virtualization environment of an L4/Fiasco. An L4Linux server consists of several L4/Fiasco threads such as a linux kernel thread, a timer interrupt thread, and an idler thread. Once a task is generated in an L4Linux, a corresponding L4/Fiasco thread is generated in its L4/Fiasco, as well. We define the corresponding L4/Fiasco thread as a 'L4/Fiasco shadow thread'. When the L4Linux kernel schedules an L4Linux task, the L4Linux kernel wakes up its corresponding L4/Fiasco shadow thread through IPC[1], and the L4/Fiasco shadow thread executes user code on behalf of the L4Linux task.

Such the properties generate many issues to consider. In this section, we consider changing the L4Linux kernel thread to a periodic thread and scheduling L4Linux servers based on the timing requirement of its L4Linux kernel thread under the virtualization environment of the L4/Fiasco.

1) Periodic Thread: Supporting the periodic model in the L4/Fiasco has already been done and included in the L4/Fiasco source code developed by Steinberg [15]. The L4/Fiasco supports two modes, one is a periodic mode and the other is a conventional mode so that a periodic thread can coexist with an non-periodic thread. Figure 4 shows the sequence of the states for the periodic model. In this way, an L4Linux kernel thread is able to enter a periodic thread just by requesting permission from the admission server. If a thread wants to be a periodic thread, two functions are required, *l4_rt_begin_periodic()* and *l4_rt_next_period()*. *l4_rt_begin_periodic()* is invoked

[1]IPC - Inter Process Communication

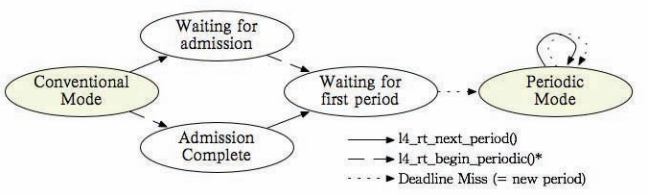

Figure 4. Periodic Model State Diagram.

to permit a thread to work on the periodic mode by an admission server that is responsible to admit an L4/Fiasco user thread to enter the periodic mode. *l4_rt_next_period()* is used when a thread is ready to enter the periodic mode or it finishes its periodic job.

A thing we have to consider is how to set up the admission server. We have two choices, one is to make a new server for only admission. The other choice is to add admission server functionalities into any existing server. In this paper, we have used the latter case since there is a server that is always loaded before any L4Linux is loaded. an L4/Fiasco loader is used to start an L4/Fiasco server such as an L4Linux server at runtime, so we extend the L4/Fiasco loader to operate as the admission server.

2) Scheduling for periodic threads: Although the L4/Fiasco supports the periodic model as mentioned, its scheduler should be changed for a reason. From the viewpoint of the L4/Fiasco, the L4Linux consists of one or more L4/Fiasco threads: one L4/Fiasco thread for the L4Linux kernel itself and the others for L4/Fiasco shadow threads that corresponds to L4Linux tasks individually. The periodic interface of the L4Linux component specifies the collective timing requirements of its internal tasks. Thus, an L4/Fiasco is supposed to schedule the L4Linux kernel thread and other L4/Fiasco shadow threads according to the component interface. This requires an accounting method that charges the execution of the L4Linux kernel thread and L4/Fiasco shadow threads into the execution budget of a corresponding component.

To simplify the problem, we make some assumptions. First, a priority of the L4/Fiasco scheduler can be preserved for the L4Linux kernel thread or L4/Fiasco shadow threads so that we can distinguish a thread corresponding to the L4Linux by checking priority. Second, when a thread of one L4Linux is scheduled, it cannot be preempted by the other L4Linuxes' threads except for the interrupt threads. This assumption is reasonable since we use the RM algorithm as system scheduling policy, and each L4Linux composes the interface with the same period.

To solve the problem, we separate the threads of an L4Linux server. The interrupt threads are not grouped with an L4Linux kernel thread and L4/Fiasco shadow threads since interrupt handlers should be executed as soon as it requested. Hence, interrupt threads have higher priority than

the others so that we consider only the thread corresponding to an L4Linux kernel and L4/Fiasco shadow threads as a scheduling group. We assigned the preserved priority into an L4Linux's thread other than interrupt threads to identify its scheduling group so that every thread corresponding to one L4Linux has same priority. The remaining problem is how the L4/Fiasco scheduler identifies such a thread's L4Linux server for accounting. Each L4/Fiasco shadow thread has a thread number of its L4Linux kernel as its pager, and the L4Linux kernel thread number is unique. Therefore, each linux group can be identified with a distinct number.

A new timeout called a "realtime_timeout" is introduced for the group scheduling. The timeout is set as Θ^+ of a linux's interface whenever any task of the linux is first scheduled in a period. When the timeout is set, the other linux threads cannot be scheduled, other than interrupt threads. After the timeout expires, the other linux threads can be scheduled in the same way. By doing this, even though an L4Linux kernel causes the system to sleep, it can be awakened immediately whenever its task reaches the ready state since other L4Linux servers cannot be scheduled while the realtime timeout is being set.

B. Guest OS: L4Linux

An L4Linux supports the priority preemptive scheduling policy with 120 different priority levels, but it does not provide the periodic model. To support the periodic model in the L4Linux, a timing property representing the period and the deadline is added into task structure so that tasks are classified into two types through existence of the property (i.e., either periodic task or non-periodic task). In internal real-time scheduling policy, periodic tasks have the highest priority to assure timing requirements within a guest OS. We apply the RM scheduling algorithm so a periodic task can be preempted by any periodic tasks which have shorter period.

For the hierarchical scheduling, a guest OS should notify its periodic interface to its hypervisor. When a guest OS spawns a periodic task, its bounded execution time should be reevaluated, and then the guest OS notifies the change to its hypervisor. The hypervisor allocates the CPU to guest OSs according to the periodic interfaces.

1) Periodic Task: For a periodic task, "task_struct" which is a data structure for a linux task is modified so that L4Linux kernel can notice the timing requirement of the periodic task, and several system calls are added as follows,

```
set_periodic_task(pid, period, WCET)
wait_next_period(pid)
```

"set_periodic_task()" is invoked to set the period and the worst case execution time of a task 'pid'[2], and a periodic task calls "wait_next_period()" to release CPU voluntarily when it finishes its periodic job. These system calls make it easy to use a programming model for the periodic task. A pseudo-code for periodic tasks as follows,

Source 1. Periodic Task Example

```
BEGIN
  // Set periodic task as periodic model
  set_periodic_task(pid, period, WCET)
  WHILE CONDITION
    // Wait for next period
    wait_next_period(pid)
    // Consume resource in WCET
    do_peridic_job()
  END WHILE
END
```

2) Scheduling for periodic tasks: We refer to a study conducted to support a periodic scheduling policy [13]. In the work, they suggest a user scheduling routine (USR) to make a task ready at appointed time, and the task information is stored in a time event queue (TEQ). When it comes to the next activation time, the USR occurs by checking the TEQ.

We use the basic mechanism of Behnam's work [13] to implement the TEQ. When a periodic task is created, its next activation time is registered in the TEQ. When the next activation time of the task comes, its guest OS checks whether or not the task misses its deadline, and a new next activation time of the task is re-inserted into the TEQ. After that, the periodic task enters the ready state immediately. Checking the TEQ can be done in $O(1)$, and each activation time of a task can be popped and pushed from/to the TEQ in $O(\log n)$, where n is the number of tasks in the TEQ, since it is implemented as a priority queue sorted by the next activation time.

For periodic tasks, we modify the highest priority ready queue to suit real-time scheduling since all periodic tasks will be located in the highest priority ready queue. It is implemented as a priority queue, whereas others are not. The priority queue contains periodic tasks in sorted order according to the RM. We are able to simply implement the EDF by replacing the sort criteria with deadline of each task.

Figure 5 illustrates the implementation of the RM scheduler in an L4Linux. When a periodic task finishes its periodic job, it enters pending state until its next activation time comes. The periodic task enters ready state and is scheduled by the RM scheduler if its activation time comes. Once a periodic task enters ready state, the RM scheduler verifies that the running periodic task has a higher priority than the highest priority periodic task in the ready queue. If the verification fails, preemption occurs. Since every periodic tasks are located in the highest priority ready queue, it is scheduled in advance of non-periodic tasks.

[2]The worst case execution time is for the calculation of the L4Linux's interface

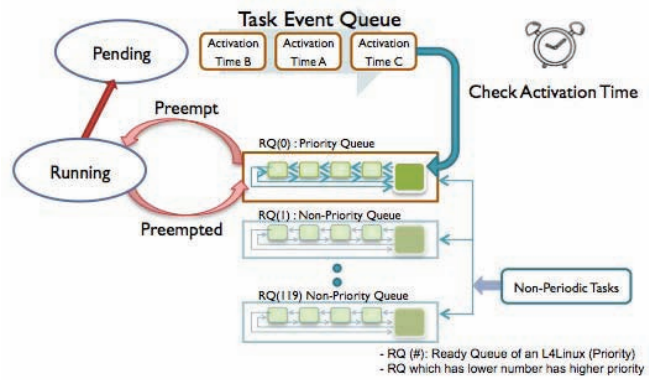

Figure 5. Implementation of the RM scheduling in an L4Linux

3) Supporting Hierarchical Scheduling: A guest OS that has periodic tasks should be considered as a real-time thread by its hypervisor, and the guest OS should pass its periodic interface $I(\Pi, \Theta^+)$ to the hypervisor when it is changed. To be specific, when the first periodic task is spawned, the guest OS should be transformed into a periodic thread and its interface should be set. After that, the guest OS just reevaluates the bounded execution time of its interface whenever a new periodic task is generated. Finally the interface is given to the hypervisor as a collective timing requirement of the guest OS. The hypervisor schedules each guest OS as the periodic thread according to its interface.

However, the interface dynamically changed raises several issues. First, schedulability issue arises. We assume that there is no task set whose L4Linux is not schedulable since the schedulability check can be done by additional routine based on some theoretical approaches, and a newly generated periodic task that is not scheduable can be rejected if necessary. The second issue is the time when the changed interface is reflected in the hypervisor. For this issue, we assume that the newly generated task in an L4Linux can be pending until the next period of the L4Linux so that the new task can be scheduled from the next period since the hypervisor schedules the L4Linux according to the changed interface from next period.

C. Compositional Real-time Guarantees

Shin *et. al.,* [2] introduced a compositional real-time model. Compositional real-time model indicates that when there are some real-time components having their own scheduler, each component can be composed as a periodic interface $I(\Pi, \Theta^+)$. Thus each component can be handled as periodic real-time task. We have implemented the RM scheduler in the L4Linux, so we could exploit theorem 6 in [2].

Theorem (Periodic Capacity Bound for RM) *For a given periodic workload set W, period Π under the RM scheduling algorithm, bounded execution time Θ^+ for a periodic partition resource Γ is*

$$\Theta^+ = \max_{\forall T_i \in W} \left(\frac{-(p_i - 2\Pi) + \sqrt{(p_i - 2\Pi)^2 + 8\Pi I_i}}{4} \right), \quad (1)$$

where

$$I_i = e_i + \sum_{T_k \in HP(W,T_i)} \left\lceil \frac{p_i}{p_k} \right\rceil \cdot e_k, \quad (2)$$

p_i is period of task i, and e_i is execution time of task i.

In order to use the theorem, a given fixed period Π should be provided to the L4Linux. In our implementation, we decide the period Π as $0.5s$.[3] Whenever a real-time task is generated in the L4Linux, it calculates its own new Θ^+ by using the theorem and its workloads. Since the linux kernel does not support a floating division, we use *do_div()* function that divides unsigned 64bit integer, which represents time property in nano seconds.

The calculated Θ^+ is sent to its L4/Fiasco kernel through a hypercall. The hypercall of the L4/Fiasco is exactly the same as system call in the L4/Fiasco, so the L4Linux just invokes *l4_rt_change_timeslice()* system call with a new Θ^+ to apply its new interface.

IV. Experiments

We evaluate the performance and overheads of our HSF implementation to the L4/Fiasco micro-kernel. We measured deadline miss ratio to see whether the HSF implementation can be applicable real-time systems. We also measured how much overheads our HSF implementation introduces.

The environments are L4/Fiasco-1.2, L4Env revision of 467, and L4Linux-2.6.31. The system for experiment is equipped with an AMD Athlon Dual Core 2.0Ghz and 3GB of main memory. Since the L4/Fiasco micro-kernel supports only a uni-processor, our experiments take place only on a single core, even though the system is equipped with multi-cores.

A. Deadline Miss

In this section, we present the measured deadline miss under diverse scheduling policies. For the experiments, a hypervisor consists of two VMs that have the real-time workloads. We assume that each component has an interface period p of 0.5 second. During the experiments, each periodic task is a processor-intensive task. Each job of the task is dropped if its previous job misses a deadline. Otherwise, it is possible to continue to generate deadline misses. We have the following scheduling policies in experiments,

1) the round robin scheduling policy ρ_1,

[3]The relative overheads generated by other system threads such as interrupt thread is large if its period is too short. On the other hand, longer period of the L4Linux generates large abstraction overhead in the periodic resource model.

2) a fixed priority-based scheduling policy ρ_2, and
3) the compositional real-time scheduling policy ρ_3.

The round robin scheduling policy ρ_1, the basic policy of the L4/Fiasco, schedules each VM in a fair manner. The priority based scheduling policy ρ_2 schedules each VM with a distinct fixed-priority that is assigned according to the utilization of its workloads. The compositional real-time scheduling policy ρ_3 is a HSF based on periodic component interfaces: the system scheduler schedules the components according to the periodic interfaces under the RM algorithm.

We perform experiments with a couple of scenarios as follows,

Scenario 1	VM1	$T_1(1.0, 0.2)$	$T_2(1.2, 0.2)$	$T_3(1.5, 0.2)$
	VM2	$T_4(20, 2)$	$T_5(30, 2)$	
Scenario 2	VM1	$T_1(8, 1.5)$	$T_2(10, 2)$	
	VM2	$T_3(2, 0.1)$	$T_4(3, 0.1)$	

Table I
SCENARIOS FOR EXPERIMENT

Table II shows the results of our experiments for scenario 1. For the scenario 1, ρ_1 only generates deadline miss. Figure 6 illustrates a process generating deadline miss for scenario 1 under ρ_1. Suppose that every task enters ready state at the same time so that VM2 has enough tasks to execute at that time. The processor is allocated to both of VMs in a fair manner, but deadline miss can be generated.

Policy	VM1			VM2	
	Task 1	Task 2	Task 3	Task 4	Task 5
ρ_1	0	3	25	0	0
ρ_2	0	0	0	0	0
ρ_3	0	0	0	0	0

Table II
DEADLINE MISS FOR SCENARIO 1 IN 200 PERIODS.

Table III shows the occurrence of the deadline miss for the scenario 2. ρ_2 only generates a deadline miss. For the scheduling policy, $VM2$ cannot get CPU until $VM1$ is idle since $VM1$ has a higher priority than $VM2$ due to utilization of workloads[4]. When all tasks enter ready state at the same time, the VM1 holds processor resource in $3.5s$ and T_3 and T_4 generates deadline miss.

For compositional real-time scheduling, it schedules all above scenarios as well as other diverse scenarios without deadline miss. Consequently, the HSF with periodic interface is suitable for the virtualization system in terms of real-time scheduling.

[4]Utilization - VM1 : 0.39, VM2 : 0.28

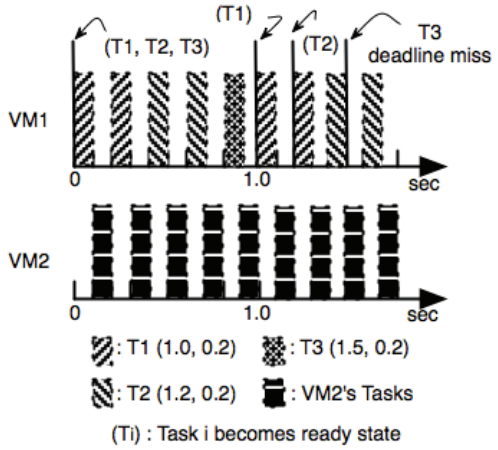

Figure 6. Scheduling for VM1(3) and VM2(4) in L4/Fiasco.

Policy	VM1		VM2	
	Task 1	Task 2	Task 3	Task 4
ρ_1	0	0	0	0
ρ_2	0	0	14	5
ρ_3	0	0	0	0

Table III
DEADLINE MISS FOR SCENARIO 2 IN 200 PERIODS.

B. Overhead

We have measured the overhead by using the RDTSC instruction on an AMD architecture. The instruction returns the number of clock cycles. The parts generating overhead are the phase of "selecting a next thread in the ready queue," "setting the real-time timeout," and "calculating the composed interface."

The overhead of "selecting a next thread in the ready queue" exists since we have modified the L4/Fiasco source code to disable other L4Linux threads during the execution time of one L4Linux. The L4/Fiasco checks whether a next thread in the ready queue belongs to the L4Linux when the real-time timeout is set. If the thread does not belong to the L4Linux, the L4/Fiasco chooses a next thread repeatedly. We define the additional clock cycles to search for a proper next thread in the ready queue linearly as the overhead. Table IV indicates the measured clock cycles to choose a next thread in the basic L4/Fiasco and the revised L4/Fiasco.

Type	Worst Case	Best Case	Average Case
Basic	20952	69	219.33
Revised	20628	82	266.56

Table IV
OVERHEADS FOR SELECTING A NEXT THREAD

From this result, we can roughly think that the average of the overhead is less than 50 clock cycles (about 25 μs) on two VMs. This overhead depends on how many VMs are running on the L4/Fiasco since only one thread of each L4Linux server in the L4/Fiasco can be located in the ready queue at the same time. This overhead occurs in every context switching.

The overhead of "setting the real-time timeout" is needed to support the dedicated scheduling during the VM's execution time of each period. While this timeout is set, the tasks of other L4Linux cannot be scheduled. The overhead is showed in Table V.

Worst Case	Best Case	Average Case	Unit
2110	131	475.75	(clocks)
1047	65	236	(μs)

Table V
OVERHEAD FOR SETTING TIMER

This overhead should be paid for every period of each L4Linux. That means that when two VMs are running, we have to pay about 500 μs to set the timer every 0.5s.

Finally, the overhead of "calculating the composed interface" occurs when every real-time task is generated in an L4Linux. This overhead includes IPC communication between the L4loader and the L4Linux as well as calculating interface of the L4Linux. This depends on the number of real-time tasks of the same L4Linux. Figure 7 indicates the overheads for the number of real-time tasks. Even though

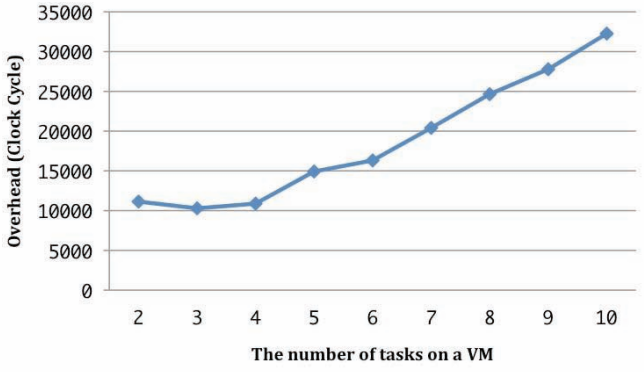

Figure 7. Overhead for calculating bound execution time.

the overhead is quite large due to IPC communication, it is reasonable in that it occurs when only a new real-time task is generated.

The overheads are generated depending on the number of VMs and real-time tasks linearly and the overhead is relatively small.

V. CONCLUSION

This paper presents the design and implementation of our hierarchical scheduling framework in the virtualization environment. The L4/Fiasco and the L4Linux are chosen as the hypervisor and the guest OS, respectively. The L4/Fiasco kernel is revised to schedule its L4Linux server according to its periodic interface.

We show a couple of scenarios whose timing requirements are not satisfied by some scheduling policies, but supported by our HSF implementation. We demonstrate the effectiveness of this approach with overhead measurement. Our experimental results show that the HSF with a periodic interface is acceptable in the virtualization environment.

In our implementation, each guest OS shares the same period Π. However, the best Π of each guest OS cannot be fixed since it will be changed according to the timing requirements of its dynamically changing workloads and the context switching overheads between the guest OSs. Finding a suitable Π dynamically may generate additional overheads that can affect real-time guarantees. This raises the problem of how and how often to find a good Π. The research of this problem remains as future work.

REFERENCES

[1] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, April 2008.

[2] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the 24th IEEE Real-Time System Symposium*, December 2003.

[3] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *Proceedings of the 25th IEEE Real-Time System Symposium*, December 2004.

[4] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: response-time analysis and server design," in *Proceedings of the 4th ACM International Conference on Embedded Software*, September 2004.

[5] R. Davis and A. Burns, "Hierarchical fixed priority preemptive scheduling," in *Proceedings of the 26th IEEE Real-Time System Symposium*, December 2005.

[6] Z. Deng and J. Liu, "Scheduling real-time applications in an open environment," in *Proceedings of the 18th IEEE Real-Time System Symposium*, December 1997.

[7] X. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in *Proceedings of the 23rd IEEE Real-Time System Symposium*, December 2002.

[8] T.-W. Kuo and C.-H. Li, "A fixed-priority-driven open environment for real-time applications," in *Proceedings of the 20th IEEE Real-Time System Symposium*, December 1999.

[9] G. Lipari and S. Baruah, "Efficient scheduling of real-time multi-task applications in dynamic systems," in *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium*, May 2000.

[10] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, July 2003.

[11] S. Matic and T. Henzinger, "Trading end-to-end latency for composability," in *Proceedings of the 26th IEEE Real-Time System Symposium*, December 2005.

[12] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein, "Analysis of hierarchical fixed-priority scheduling," in *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, June 2002.

[13] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. Bril, "Towards hierarchical scheduling on top of vxworks," in *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, December 2008.

[14] M. M.H.P., van den Heuvel, R. J. Bril, J. J. Likkien, and M. Behnam, "Extending a hsf-enabled open-source real-time operating system with resource sharing," in *Proceedings of the 6th International Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2010.

[15] U. Steinberg, "Quality-assuring scheduling in the fiasco microkernel," Master's thesis, Dresden University of Technology, March 2004.