Implementation of Real-Time Virtual CPU Partition on Linux

Aloysius K. Mok, Xiang Feng
Computer Science Department, University of Texas-Austin
Austin, Texas-78713, U.S.A
{mok,xf}@cs.utexas.edu

Zhengting He
Elec. & Comp. Engr. Dept., University of Texas-Austin
Austin, Texas-78713, U.S.A
zhe@ece.utexas.edu

Abstract

A real-time virtual resource is an abstraction for resource sharing where the application task groups sharing a resource must meet timing constraints in the absence of knowledge of all the timing requirements of all the task groups, thus prohibiting a global schedulability analysis. RTVR addresses the issues of application isolation in the open system environment, as well as satisfying the timeliness requirements of all the task groups. Based on the theoretical framework in previous work [10], [11], [4], we present in this paper the first implementation of a real-time virtual resource for the CPU. We shall describe two real-time virtual resource (RTVR) prototypes that are based on the Linux 2.4.18.3 kernel. The first RTVR implementation uses a static resource level scheduler which can be applied to systems with predefined application task sets. The second implementation has a novel dynamic resource level scheduler under which task groups (temporal partitions) can join and leave dynamically. We shall report some experimental results on measuring system performance in various aspects such as the effect of the scheduling quantum size, interrupt request response time and scheduling overhead. The experiments demonstrate that RTVR can be efficiently implemented while satisfying its theoretical properties.

1 Introduction

The concept of a *Real-Time Virtual Resource* (RTVR) was introduced in [10] for abstracting resource sharing where a physical resource such as a CPU is time-shared by two or more applications. Each application has a group of tasks that are scheduled by its own application-specific scheduler to meet the application's timeliness requirements. Application task groups may be subject to a variety of timing constraints and so in general cannot be efficiently scheduled by a centralized scheduling policy. On the other hand, the application task groups may interfere with one another in the absence of centralized control. RTVR provides the basis for an application programmer interface that facilitates distributed control and is aimed at achieving the following properties:

- To ensure application isolation so that each application task group may be programmed as if it had dedicated access to a physical resource, i.e., without interference from other task groups due to resource sharing. This property is crucial for complex real-time systems such as avionics control systems both for maintaining timeliness properties of subsystems and also for containing faults within each subsystem in the event of subsystem failures.

- To investigate the problem in the *open system* [3] environment where: (1) tasks may join and leave task groups dynamically, and thus, global knowledge of the task groups might not be available *a priori*; (2) in case of hardware failures, task groups may be relocated by coexisting with other task groups on the diminished pool of shared physical resources.

- To be highly scalable so that RTVR may be employed in large scale real-time systems with a huge number of subsystems and applications,

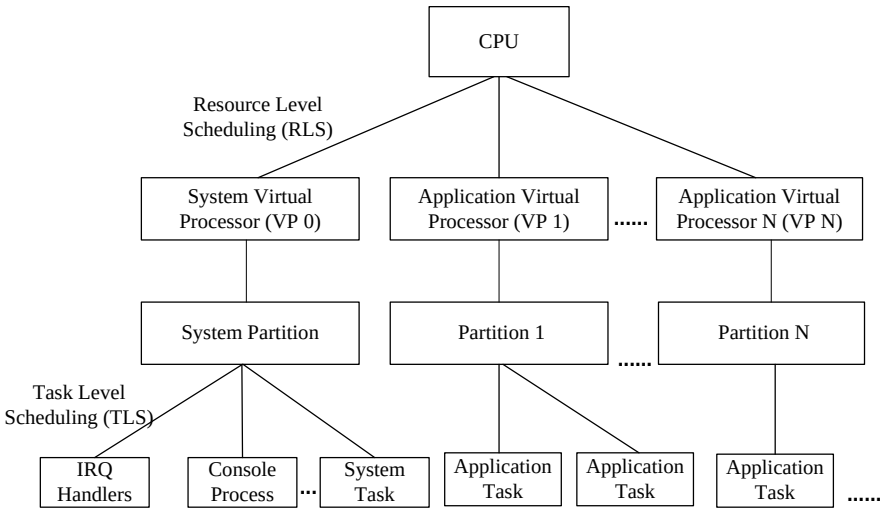

FIGURE 1: *RTVR Structure Overview*

and system integration and testing (debugging) may be greatly expedited.

Towards these goals, a RTVR is defined by two key parameters: (1) the Availability Factor (also called the Rate) α of its corresponding physical resource which is reserved by the RTVR and (2) the Delay Bound Δ of of the RTVR. The delay Δ is a design parameter, a non-negative number that is specified by the application programmer. A correctly implemented RTVR will provide at least $\alpha \times L$ units of the resource's time in any interval of length $L + \Delta$ for any value of L. An application task group is scheduled on its own RTVR. RTVR time-shares the physical resource with other RTVRs such that the task group that is scheduled on this RTVR may be programmed as if it had dedicated access to a physical resource. Tasks within the same application task group are scheduled by a task level scheduler (TLS) that is specialized to the real-time requirements of the tasks in the group. Each application task group is then assigned to run on one partition. Finally, RTVRs on the same physical resource are scheduled by a resource level scheduler (RLS).

In [10], we showed how schedulability analysis can be performed for the earliest-deadline-first (EDF) and the fixed-priority (FP) schedulers in the context of the real-time virtual resource, where a real-time task group is characterized by the standard real-time task model of Liu and Layland [8]. We showed that the schedulability analysis can be performed without global knowledge of all the task groups that share the physical resource. For each task group, the schedulability analysis only needs to take into account the delay bound of the real-time virtual CPU on which the task group executes; as long as the sum of the worst-case response time of a task and the jitter introduced by the delay bound does not exceed the task's deadline, the task is schedulable. This is important for the open system environment[3] where global knowledge of the task groups is assumed to be unavailable.

In [11], we first introduced the mathematical concept of *regularity* and discussed how to exploit it for realizing the delay bounds of real-time virtual resources. For the case of *regular partitions*, we showed that the utilization bounds of both fixed-priority scheduling and dynamic-priority scheduling remain unchanged from those for dedicated resources. We determined the utilization bounds for the more general case of *irregular partitions*. In particular, both types of partitions can be efficiently constructed by exploiting the compositionality properties vis-a-vis the *regularity* measure.

Based on the theoretical results obtained from those papers, we have implemented two real-time virtual resource (RTVR) prototypes based on Linux 2.4.18.3 kernel. We used the Linux operating system because its open source code was available and because of its popularity in the personal computing and embedded system area. Our implementation architecture is shown in Figure 1. A CPU is partitioned into $N + 1$ virtual processors (VP). VP 0 is called system virtual processor (SVP) on which all system tasks run. All other N virtual processors are called application virtual processors (AVP). The first prototype implements static resource level scheduling (Static RLS) which can be applied to systems with a predefined application task set. The second one im-

plements dynamic RLS (Dynamic RLS) where partitions can be dynamically join and leave the system. As a general purpose OS, the Linux kernel is not preemptive in kernel mode, which makes accurate CPU partition and deadline guarantee for real-time applications impossible. In both of our prototypes, (1) the kernel is made to be preemptive by spawning a kernel thread for each interrupt service routine (ISR) except for ISR 0 which is the routine for timer interrupt; (2) under the original Linux scheduler, a RLS is inserted to temporally partition the CPU, and the original Linux scheduler becomes task level scheduler (TLS); (3) system calls and utilities are provided to assist applications to specify real-time parameters and pass them to kernel.

The main contribution of this paper is fourfold.

1. This paper presents a first implementation of RTVR. Several practical issues were identified and discussed in details.

2. Dynamic RLS scheme is introduced so that not only tasks but also partitions could join and leave the system dynamically.

3. A novel two-level partitioning scheme is introduced so that even partitions with small rates with regard to their partition delay could be scheduled efficiently.

4. A prototype implementation of network virtualization is presented to show the effectiveness of a set of RTVRs.

2 Related Work

There have been many initiatives to make Linux real-time. Two general categories of solutions have been proposed and provided on the market. The first one is to modify the current Linux kernel or replace it with a new one. The new kernel implements and then keeps the full set of original kernel API which contains a full set of system calls. Examples using this approach include TimeSys (Linux/RK) [12], Red-Linux [15] and Qlinux[5].

Another creative approach is to impose another level of kernel (called sub-kernel) on the top of existing Linux kernel. In this way, Linux is treated as the lowest priority task of the sub-kernel OS. RTLinux [2] and RTAI [1] are examples of this category.

While our approach falls into the first category, it differs from others mainly in the way how it models real-time.

Linux/Resource Kernel (RK) [12] allows applications to specify only their resource demands and leaves the kernel to satisfy those demands to hidden resource management schemes. The resource demands are usually expressed in a form of (Computation Time, Period, Deadline).

Red-Linux [15] aims to provide a general scheduling framework to integrate three paradigms, namely, priority-driven, time-driven and share-driven paradigms. In order to do that, Red-Linux identifies four scheduling attributes, i.e., priority, start-time, finish time and budget.

Constant Bandwidth Server (CBS) [7] uses a deadline postponing scheme to provide bandwidth isolation. A CBS Server is described by two parameters: Budget and Period. The bandwidth is calculated as Budget over Period and the period also serves as deadline.

The main difference between our approach and previous ones is that we minimize the interaction between the resource level scheduler and the task level scheduler to a simple interface. Unlike previous approaches, our resource level scheduler does not require knowledge of the task level deadlines or their derivatives in partition scheduling. In the other direction, the task level scheduler may need to know at most the delay bound of the partition it executes on. More importantly, the related delay bound of the partition allows the application task scheduler to determine not only compliance with deadline requirements but also event-separation types of constraints. If the application task groups are not all specified in one common task model such as Liu and Layland periodic tasks, our partition model can still be used.

Our work is also differentiated from QLinux (Start-time Fair Queuing - SFQ) [5] and Fluctuation Constrained Server (FCS) [6, 16]. Both SFQ and FCS have the similar notion of supply deviation for any time interval as our work. However, SFQ aims to minimize the delay to achieve near-optimal fairness while our work makes sure the delay could guarantee the schedulability of real-time tasks. Furthermore, SFQ depends on the number of threads (which are equivalent to partitions in our work) being scheduled, while the resource level delay in our work is specified by the partition request, thus providing stronger guarantee. FCS is intended to be on per stream basis while our work is on per task group basis. FCS does not provide any real-time schedulability analysis if a group of tasks instead one single task (stream) is running on FCS, whereas task level scheduling is a major problem that real-time virtual resource addresses as discussed in [10] and [9].

In Hierarchical Loadable Scheduler (HLS) [13] schedulers may be converted to one another by means of service guarantee. Their goal is also quite different from ours since HLS aims at constructing a hierarchy of schedulers while our work is to construct a hierarchy of virtual resources.

Shin and Lee recently presented their work of a periodic resource model [14], whereas a resource allocation of X time units every Y time units is guaranteed. Our approach differs from theirs mainly in that we do not require periodicity, thus providing a more general model for real-time applications.

The rest of the paper is organized as following. Section 3 introduces the background concept and definitions. Section 4 describes the static RLS. Section 5 describes the dynamic RLS. Section 6 gives an overview to the Linux kernel scheduler and discusses some related implementation issues. Some experiments are conducted and the results are provided in Section 7. Section 8 concludes our writing.

3 Background Definition

In this section we shall review a few key preliminary definitions concerning the bound-delay resource partition model presented in [10] before we proceed to the resource level scheduling. For more details and examples pertaining to this section, please refer to [10], [11].

Definition 1 *[10] The Availability Factor (rate) of a resource partition* Π *is the percentage of the total time of a resource is available to this particular partition. Obviously, the rate of service provision of a dedicated resource is* 100%.

Definition 2 *[10] The Supply Function* $S(t)$ *of a partition* Π *is the total amount of time that is available to* Π *from time 0 to time t.*

Definition 3 *[10] The Partition Delay* Δ *of Partition* Π *is the smallest d so that for any* t_0 *and* t_1, $(t_1 \geq t_0)$, $(t_1 - t_0 - d)\alpha(\Pi) \leq (S(t_1) - S(t_0)) \leq (t_1 - t_0 + d)\alpha(\Pi)$.

Partition Delay measures the largest deviation of a partition on any time interval with regard to the amount of supply it is supposed to receive when there is no delay at all.

Definition 4 *[10] A Bounded Delay Resource Partition* Π *is a tuple* (α,Δ) *where* α *is the rate of the partition and* Δ *is the partition delay.*

Definition 5 *[11] The Supply Regularity* R_s *of Partition* Π *is the smallest d so that for any* t_0 *and* t_1, $(t_1 \geq t_0)$, $(t_1 - t_0)\alpha(\Pi) - d \leq (S(t_1) - S(t_0)) \leq (t_1 - t_0)\alpha(\Pi) + d$.

Similar to partition delay, supply regularity also measures the largest deviation of a partition on any time interval. The difference is that partition delay is expressed in terms of time while supply regularity in terms of amount of supply. From the definitions, we could easily conclude that $R_s/\Delta = \alpha$.

4 Static RLS

Static RLS is applicable to systems where all partitions are fixed and their parameters are known. The concept of static RLS was introduced in [11]. The essential idea of the algorithm is to compose the target partition by combining several partitions which are efficiently schedulable. Two theorems are involved to support constructing the scheduler.

Theorem 1 *Regular partitions whose availability factors are all powers of some number and whose total availability factor* ≤ 1.0 *are schedulable.*

Example 1 *Regular partitions* Π_i $(1 \leq i \leq 4)$ *with availability factors of* $1/2, 1/4, 1/8, 1/8$ *respectively can be easily scheduled on a dedicated resource with the period of* 8 *and the time slot assignment of* $(1, 2, 1, 3, 1, 2, 1, 4)$ *where i indicates* Π_i.

Theorem 2 *When k partitions each with supply regularity of* 1 *are combined together they form a partition with supply regularity of k.*

Definition 6 *Given Partition* Π *with availability factor of a and supply regularity of* R_s*, the Adjusted Availability Factor* $AAF(a, R_s)$ *is the total of the availability factors of partitions that are used to compose* Π.

For a system with N partitions where the i_{th} partition has rate $= \alpha_i$ and supply regularity $= R_{si}$ $(1 \leq i \leq N)$, the partition table generation algorithm is as follows:

- For each partition Π_i, calculate its Adjusted Availability Factor $AAF(\alpha_i, R_{si})$ as follows:
  If $R_{si} = 1$, $AAF(\alpha_i, R_{si}) = \frac{1}{2^k}$,w
  where $k = \left\lfloor \log_{\frac{1}{2}} \alpha_i \right\rfloor$
  else $AAF(\alpha_i, R_{si}) = \frac{1}{2^{i_1}} + \frac{1}{2^{i_2}} + ... + \frac{1}{2^{i_{R_{si}}}}$
  where $i_1 = \left\lceil \log_{\frac{1}{2}} \alpha_i \right\rceil$, if $\log_{\frac{1}{2}} \alpha_i$ is not an integer
  otherwise $i_1 = \left\lceil \log_{\frac{1}{2}} \alpha_i \right\rceil - 1$
  for $k \in [2, R_{si} - 1]$
  if $\log_{\frac{1}{2}}(\alpha_i - \sum_{z=1}^{k-1} \frac{1}{2^{i_z}})$ is not an integer,
  $i_k = \left\lceil \log_{\frac{1}{2}}(\alpha_i - \sum_{z=1}^{k-1} \frac{1}{2^{i_z}}) \right\rceil$
  otherwise,
  $i_k = \left\lceil \log_{\frac{1}{2}}(\alpha_i - \sum_{z=1}^{k-1} \frac{1}{2^{i_z}}) \right\rceil - 1$,
  $i_{R_{si}} = \left\lceil \log_{\frac{1}{2}}(\alpha_i - \sum_{z=1}^{R_{si}-1} \frac{1}{2^{i_z}}) \right\rceil$
  Record $2^{i_{R_{si}}}$ as P_i which denotes the partition period for Π_i.
- If $\sum_{i=1}^{N} AAF(\alpha_i, R_{si}) > 1.0$, program exits.

- Allocate M time slots where $M = max\{P_i \mid 1 \leq i \leq N\}$. M is the partition table period.
- For each Π_i, $1 \leq i \leq k$, we have
  $$AAF(\alpha_i, R_{si}) = \frac{1}{2^{i_1}} + \frac{1}{2^{i_2}} + ... + \frac{1}{2^{i_{R_{si}}}}$$
  For each $k \in [1, R_{si}]$, among the M time slots, assign 1 out of every i_k time slots to Π_i.
- Assign the rest time slots to non-real-time tasks or system partition.
- Combine all neighboring slots that are assigned to the same partition.

Example 2 *The following example shows how a partition table is generated for a system which has 3 AVPs with real-time requirements as* $\{(\alpha_i, R_{si}), i \in [1,3] \mid (0.375, 2), (0.25, 2), (0.25, 1)\}$.

- *Calculate* $AAF(\alpha_i, R_{si})$
  $$AAF(\alpha_1, R_{s1}) = 0.25 + 0.125; \; P_1 = 8$$
  $$AAF(\alpha_2, R_{s2}) = 0.125 + 0.125; \; P_2 = 8$$
  $$AAF(\alpha_3, R_{s3}) = 0.25; \; P_3 = 4$$
- $\sum_{i=1}^{N} AAF(\alpha_i, R_{si}) = 0.875$.
- $M = max\{P_i \mid 1 \leq i \leq N\} = 8$.
- *Allocating time slots.*
- *Combine the neighboring slots belonging to the same VP.*

5 Dynamic RLS

In the previous section, we described static RLS. This approach can efficiently schedule partitions with different delay requirements. However, it cannot accommodate partitions joining and leaving dynamically. This is due to: (1) the partition delay property holds with regard to only one static scheduling table; (2) when a new partition request arrives, the entire scheduling table has to be recomputed again. To solve this problem, we shall introduce the concept of dynamic RLS. In this section, we shall first discuss how to devise dynamic RLS algorithms ; then we consider the effects of quantum-based scheduling where there is a minimum size limit on CPU time slices; finally we discuss a two-level scheduling approach for scheduling a special type of partitions.

5.1 Dynamic RLS Algorithm

A dynamic scheduling algorithm, as its name indicates, allows partitions to join and leave dynamically. This requirement implies that 1), there is no global scheduling analysis for RLS. 2) there is no restriction on the scheduling algorithms as long as the resultant partitions meet their timing requirements. The similarity between RLS and TLS leads us to try to bridge them by converting RLS problem to a conventional scheduling problem that has been extensively studied in the context of TLS. In this way, we may apply those results from task scheduling to resource scheduling as well.

Theorem 3 *Suppose the execution of a real-time task with computation time of* C *and period of* P *is considered as the execution of a partition. The resultant partition tuple* (α, Δ) *is* $(C/P, 2P - 2C)$ *regardless of which scheduling algorithm is used.*

Corollary 1 *To schedule a partition* (α, Δ) *is equivalent to schedule a task with* $(\Delta\alpha/(2(1-\alpha)), \Delta/(2(1-\alpha)))$ *as computation time and period, respectively.*

Example 3 *To schedule Partition* Π $(0.2, 40ms)$, $C = \Delta\alpha/(2(1-\alpha)) = 0.2 \times 40/(2 \times (1-0.2)) = 5ms$ $P = \Delta/(2(1-\alpha)) = 40/(2 \times (1-0.2)) = 25ms$ *Therefore, to schedule* Π *is equivalent to schedule a task with computation time of 5ms and period of 25 ms.*

The admission test for new partitions could be expressed as $\sum_{i=1}^{n} \alpha_i \leq Utilization\ Bound$.

The utilization bound depends on which scheduling algorithm is used. It would be 1.0 for earliest deadline first (EDF) and $m(2^{\frac{1}{m}} - 1)$ for rate monotonic scheduling (RM) where m is the number of partitions.

5.2 Considering Scheduling Quantum

In the discussion above, we assume that scheduling (especially task switching) can be performed with time being real numbers. However, many systems have a lower limit on the smallest time allocation unit, namely, the scheduling quantum. The scheduling quantum may also be viewed as the unit for specifying the precision of time measurements. It is imperative to consider the effects of scheduling quantum when we implement the scheduling scheme on real systems. In this subsection we shall reexamine the issues that we discuss in the previous subsection and we shall also show the interdependence among rate, partition delay and the scheduling quantum.

Theorem 4 *To schedule a partition* (α, Δ) *on a system with scheduling quantum of* Q *is equivalent to schedule a task with* $(\lceil \Delta\alpha/(2Q(1-\alpha)) \rceil \times Q, \lfloor \Delta/(2Q(1-\alpha)) \rfloor \times Q)$ *as computation time and period, respectively.*

Because of the interdependence of rate, partition delay and scheduling quantum that is discussed in [4], the introduction of scheduling quantum also puts some constraints on rate and partition delay.

- $\alpha \geq 1/(1 + \lfloor \Delta/Q \rfloor)$: The partition will get at least one quantum available time after the actual partition delay $\lfloor \Delta/Q \rfloor$ happens. This puts a lower bound on the rate which means that the partition rate could not be infinitely small.

- $\Delta \geq Q$: The partition delay should be no less than the scheduling quantum.

Again the admission test with quantum consideration could be expressed as following:
$\sum_{i=1}^{n} \frac{\lceil \Delta\alpha/(2Q\times(1-\alpha))\rceil \times Q}{\lfloor \Delta/(2Q\times(1-\alpha))\rfloor \times Q} \leq Utilization\,Bound$

5.3 Two-Level Resource Scheduling

During the implementation, we observed that the lower bound on the rate could lead to inefficient usage of the resource. Typically, the scheduling quantum of Linux is set as 10 ms. For example, we measured that a typical MP3 application should not be delayed more than 190 ms which could be considered as its partition delay. On the other hand, it needs less than 1 ms every 190 ms of time. However, we know from the relation between scheduling quantum and rate, that for such a partition delay to be achieved, the lowest rate is $(1/(1+190/10)) = 5\%$. The actual CPU usage of MP3 is just around 0.5%. Similarly, ISRs also have these characteristics. To solve this problem, we introduce a novel method of two-level partitioning.

The essential idea of this scheduling method is to group all those partitions with small partition delays as well as small rates together and consider them as one partition with small partition delay but with larger rate (the sum of the rates of the small partitions). When this partition is scheduled, it divides the scheduling quantum into mini time slots and distribute them among the original small partitions. A more detailed description is given below:

1. Partition Grouping: Small partitions (α_i, Δ_i) are grouped together as one partition Π (α, Δ) where $\alpha \geq \sum \alpha_i$ and $\Delta \leq \Delta_i - Q$.

2. Partition Scheduling: When a time slot assigned to Π, it will be split into N mini time slots. $\lceil (N\alpha_i/\alpha) \rceil$ mini time slots will be assigned to Partition Π_i. The time slots will run in the order of i.

3. Partition Admission: If there are n time slots unused among those N mini time slots. A new partition Π' (α', Δ') could be admitted by Partition Π if

   (a) $\Delta' \geq (\Delta + Q)$

   (b) $\alpha' \leq (n/N) \times \alpha$

4. Partition Leaving: If a partition leaves, instead of leaving a "hole" among the mini time slots, all partitions of the order higher (later) than this partition will run earlier than this partition. It also means all active partition will run first and if there are mini time slots that are either unused or left by leaving partitions, they will always run latest among all the mini time slots. It is similar to the compression of mini time slots.

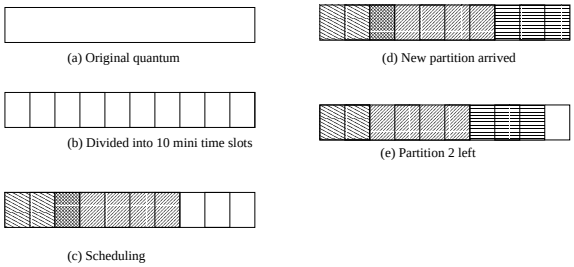

FIGURE 2: *Illustrations of Example 4*

Example 4 *Schedule* Π_1 *(0.05, 40ms),* Π_2 *(0.02, 43ms),* Π_3 *(0.10, 48ms) using the two-level scheduling described above. Assume* $N = 10$.

1. *Group:* Π *(0.05+0.02+0.10, 40-10ms) = (0.17, 30ms). Because of the quantum size is 10ms and as we showed above that quantum size will put a lower bound to rate, the* Π *will get actually (.25, 30ms).*

2. *Scheduling: Every time slot of* Π *is further divided into 10 mini time slots as shown in Figure 2 (a) and (b),* Π_1 *then will get* $\lceil (10 \times 0.05/0.25) \rceil = 2$ *mini time slots as shown in Fig 2 (c). Similarly,* Π_2 *will get 1 slots and* Π_3 *4 slots. The remaining 3 slots could be used for non-real-time partition and could be assigned to new partition(s) arriving later. When* Π *gets to run, the 2 time slots of* Π_1 *will run first, then the 1 time slot of* Π_2*, 4 slots of* Π_3*, finally the remaining 3 time slots.*

3. *Suppose a new partition* Π' *(0.06, 45ms) requests to join, because (1)* $45 > 30 + 10$*, (2)* $0.06 < 3/10 \times 0.25$*, it will be granted admission and* $\lceil (10 \times 0.06/0.25) \rceil = 3$ *will be assigned to* Π' *as shown in Figure 2-(d). Those time slots of* Π' *will run latest among all partitions because* Π' *joins the latest.*

4. *Suppose* Π_2 *leaves,* Π_3 *will run right after* Π_1 *instead of letting the resource idling for one mini time slot which was assigned to* Π_2*. Therefore, that one mini time slot will be postponed till the last of the 10 mini time slots as shown in Figure 2 (e).*

Notice that this two-level resource scheduling method is different from the multi-level resource scheduling introduced in [4]. In [4], resource scheduling is recursively applied to partitions because large resources might need to be partitioned more than once. The two-level resource scheduling introduced here is specifically designed to address the problem of scheduling partitions with rates smaller than the lower bound imposed by the scheduling quantum. As for scalability, this method is not as scalable as that in [4].

6 Implementation Issues

Linux is an open source, Unix-style general purpose operating system that has become popular not only in server market, but also in the personal computing and embedded system area. Our RTVR prototypes are built based on Linux 2.4.18.3 kernel. However, Linux does not support RTVR directly. Therefore, in this section, the detailed description of RTVR kernel scheduling implementation will be given and some related issues will be discussed.

6.1 RTVR Kernel Structure

The kernel structure of RTVR is shown in Figure 3. A RLS is inserted beneath the original Linux scheduler which now becomes the TLS (Task Level Scheduler). When a timer interrupt occurs, the RLS updates system time information and runs the RLS to activate a particular VP according to the partition table. Then TLS will schedule the tasks on that VP.

Each VP can have its own customized TLS or share a common TLS with other VPs. As mentioned above, all VPs share the original Linux scheduler as TLS in our prototypes. The original Linux scheduler provides 3 scheduling algorithms: POSIX real-time processes can be scheduled either by SCHED_FIFO or by SCHED_RR (round-robin) policy; Other non-real-time processes are scheduled by SCHED_OTHER policy.

Each VP owns a dedicated process run queue. When a particular VP is active, TLS will schedule processes in the corresponding run queue. In this way the RLS and TLS are cleanly separated and any customized TLS can be easily plugged in without extra structural modification.

The original Linux kernel is not able to provide realtime guarantee in the sense that when an un-masked interrupt occurs, kernel jumps into the corresponding ISR and service it immediately. This limitation makes accurate CPU partition impossible, which is required by RTVR. In our prototypes, an ISR thread is spawned for each non-timer-interrupt service routine by *init* process during system initialization. All the real work is done in the ISR threads except a minimum amount of code directly interfacing hardware, i.e., acknowledge the interrupt controller. In this way ISRs can be scheduled and run in appropriate time, which achieves accurate CPU partition.

ISR threads are assigned to the highest priority by default in our prototypes. Whenever an interrupt occurs, the corresponding ISR thread will be waken up and inserted to the head of a run queue which is the one associated with SVP by default. When the corresponding VP is activated by the RLS, TLS will guarantee to pick the ISR thread first and execute it. After the interrupt is serviced, the ISR thread will be de-queued from the run queue and put into sleep state. Since all ISR threads are assigned running on SVP by default, disturbance on applications running on other AVPs by these interrupts is kept to a minimum, which is highly desirable for hard real-time applications. Users are allowed to change the priority of any ISR thread and set it to run on any AVP if necessary so that flexibility is also achieved.

6.2 Scheduling Implementation

Due to the page limit, we shall describe only the essential data structure of the dynamic RLS kernel. The static RLS kernel is simpler and can be viewed as a subset of dynamic RLS kernel.

- structure *partition parameters [MAX PARTITION NUMBER]*, which contains:
  - float *rate*;
  - float *partition delay*;
- int *scheduling table[2][MAXIAML LENGTH]*;
- int *effective table length*;

The first member data *partition parameters* is a collection of parameters of partitions. When a new partition is requested, an admission test will be performed based on the information of existing partitions. Once it is admitted, the parameters of an equivalent task are computed. Scheduling tables will be computed based on those task parameters as if a task were being scheduled. The actual scheduling and the admission test are dependent on the scheduling algorithm.

The second member data *scheduling table* shows that scheduling tables used in dynamic RLS. Scheduling according to a table is typically used in time-driven scheduling systems. It is conceptually simple and proven to be run-time efficient. Meanwhile, event-driven scheduling algorithm which is used for dynamic RLS is notorious for its large

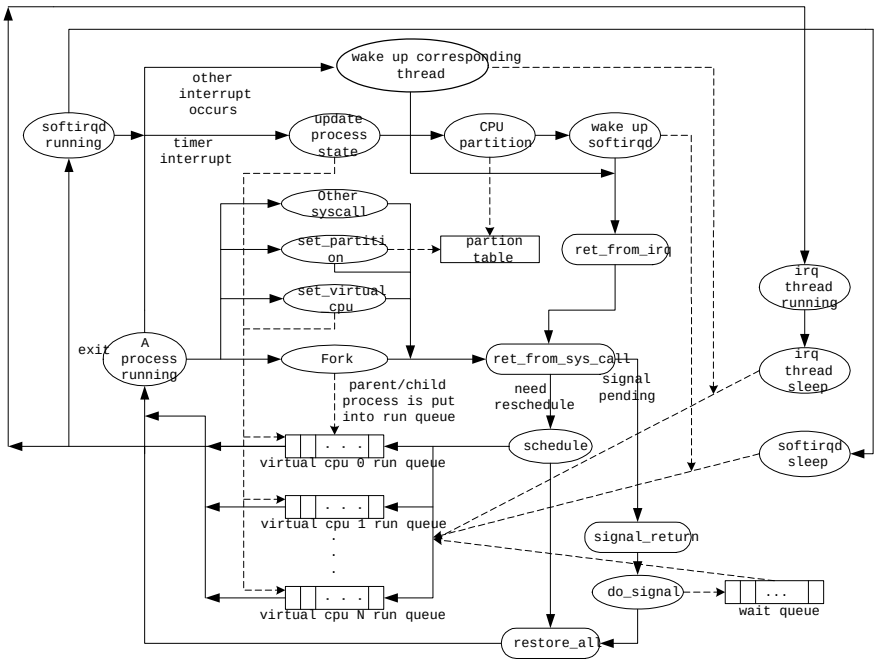

FIGURE 3: *New Kernel Process Scheduling and State-Transition Diagram*

scheduling overhead. Take the EDF scheduling algorithm for example, every time when a new job is released or a job finishes the queue of deadlines will be re-evaluated and the job with the earliest deadline will be chosen to run next. In order to utilize the advantages of both time-driven and event-driven scheduling, we chose to use tables in the dynamic partitioning.

Two sets of tables are employed in the implementation. The table computation task is assigned to system partition exclusively, which is created to deal with system tasks. When the OS is scheduling according to one set of table, system partition will work on computing the other set using the scheduling scheme that we discussed in the previous section. Earliest Deadline First scheduling algorithm is used to schedule the tasks that correspond to partitions. When the table for scheduling is exhausted, the two tables will then alternate. This process is shown in Figure 4. In this way, scheduling decision making is done in a batch fashion when system partition is running and all other real-time partitions will run in the efficient time-driven mode.

There are several noteworthy issues related to the tables.

First, system partition has to finish computing one set of table before the other set is exhausted. By checking the length of tables and the partition delay parameter of system partition, this would be easily done.

Second, adding or removing partitions will not be effective till a new set of tables is computed accordingly and is ready for use in scheduling. This would induce a bounded delay for partition related operations. Assuming that partitions in their nature are more static and are requested also in a more predictable fashion than tasks, this type of delay is acceptable. Even if the length of delay is too large for certain systems, the problem could be solved by changing the effective table length or by immediate recomputing a new set of table and activating it.

Third, in the case that system partition is the only active partition, the tables will not be used and the system partition will have exclusive access to the processor. The system performance would be the same as original Linux. The transition diagram is shown in Figure 4.

The third member data *effective table length* mandates the effective length of each table. There are basically three cases that the effective length might need change. One is that the delay of partition related operation as mentioned above is too large. Second, if the scheduling period is less than the total length of these two tables, the effective length could

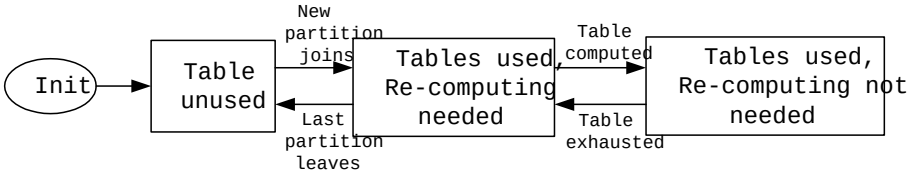

FIGURE 4: *Scheduler State-Transition*

be set accordingly, thus eliminating the need to recompute the tables till any change happens to the partitions as a way of optimization. Third, adjusting the effective length dynamically provides memory-size-sensitive systems another leverage to balance between memory usage and system performance.

6.3 Network Virtualization

So far, our discussion is limited to virtualizing processor. In an operating system, there are many other types of resources such as network, IO and storage. Among those resources, network is of particular interest to us because 1), network is preemptible on the packet level, 2), there is strong functionality support for networking in Linux. Actually, our implementation takes advantage of Linux network traffic control.

In Linux, traffic control provides queueing systems and scheduling mechanism to manipulate the time and order how packets are transmitted onto the network. By default, Liunx has a single FIFO queue to collect packets and dequeue them as fast as the network hardware could accept. Other example traffic control options include SFP, CBQ and GRED.

We implement a new queueing discipline called Real-Time Virtual Networking (RTVN). Similar to RTVR, RTVN provides guarantees for the amount of packets transmitted within an interval.

The implementation of RTVN is similar to that of Token Bucket Filter. The major difference between these two is that TBF limits the highest speed of transmission while RTVN bounds the lowest speed of transmission by reservation. Partition delay in RTVN is mapped into the queue depth in TBF and admission control exists only in RTVN.

6.4 Others

New system calls are built into the kernel of the prototypes. Utilities that performs appropriate system calls are also implemented to assist the user to manage the partition table and pass application real-time requirements to the kernel.

SVP has to be given fair amount of CPU utilization because all newly forked processes and system tasks such as ISR threads are running on it by default. A low CPU sharing utilization leads to a large IRQ response time which may cause the whole system unstable. For example, if SVP shares 5% of the CPU, the minimal interval between two neighboring partition allocated to SVP will be 190 ms, assuming scheduling quantum is 10ms. In our implementation, we enforce that SVP will share at least 10% of the CPU supply, which leaves enough CPU supply for applications on AVPs but is the minimum to ensure the system reliability.

Two-level resource partition is implemented in dynamic RLS. SVP can be configured as a small partition and grouped with other small partitions to form a normal partition so that CPU can be shared more efficiently in this way. For example, if a normal partition shares 10% of the CPU and SVP occupies 1 mini slot,it actually only shares 1% of CPU but the IRQ response time is almost the same as if it had shared 10% of CPU in static RLS. The implementation of two-level resource partition is achieved by dynamically adjust the scheduling quantum size which is the reverse of the timer interrupt frequency.

7 Experiments

Several experiments were carried out to study the performance of our prototypes. (1) The interrupt response time both on static and dynamic RLS were measured. (2) We measured the CPU partition scheduling overhead. (3) An H.263 video decoder and a g-nibble game were selected as real-time applications that coexist to compete for CPU cycles. We qualitatively evaluate their performance on RTVR and compare with that on Linux. (4) A video encoder application which requests mainly CPU cycles and network bandwidth was selected. We shall see how RTVR achieved better performance by appropriate co-partitioning of both resources.

Experiment (1)-(3) were carried on Toshiba Satellite 1100 laptop with an Intel Pentium III Celeron 1.33 GHz processor and 256 MB RAM. (4) were made on a desktop PC with Pentium III 600MHz processor and 256 MB RAM. After we

present the measurement results, we shall discuss system performance-tuning problem.

7.1 Keyboard Response Time

To study the typical interrupt response on our RTVR, we measured keyboard response time. The keyboard uses IRQ 1 in our IBM-PC compatible laptop. The system is set to have a SVP and an AVP. Instrumentation code is inserted to measure the IRQ response time. Figure 5 shows the average response time to IRQ 1 over 10 seconds.

In the static RLS prototype, the availability factor of AVP is set to $i/8$, where i is an integer in the range $[0, 7]$. The supply regularity is set to the minimal value which can accurately represent the corresponding availability factor. For example, if availability 7/8 is assigned to AVP, to accurately represent 7/8, supply regularity has to be at least 3 because $7/8 = 1/2 + 1/4 + 1/8$. In the dynamic RLS prototype, the AVP is set as a small partition. The rate of AVP is set to $i/10$, where i is also an integer in the range $[0, 9]$.

Unsurprisingly, dynamic RLS significantly reduces the IRQ response time by adding a small partition layer in RLS, which demonstrates that it can efficiently accommodate tasks with small rate and small delay.

7.2 Scheduling Overhead

The typical CPU partition scheduling overhead consists of RLS execution, TLS execution and memory operation such as stack swap. Figure 6 shows the maximal context-switching time measured over 10 seconds on both prototypes. Note that there are 1328940 CPU cycles/ms, which means that the context-switching overhead increased by adding a RLS layer is less than 0.4%.

The execution time of RLS in static RLS only consists of table lookup operation, which is almost constant. In dynamic RLS, the maximal execution time also consists of partition table generation. which is conducted once every effective table length. The average RLS execution time and context-switching time, is directly related to it. Figure 7 plots the average RLS execution time versus effective table length in dynamic RLS. Obviously, a longer table reduces scheduling overhead but consumes more memory. The trade-off between table length and memory consumption should be tuned on an individual platform basis.

7.3 Applications Compete for the Same Resource

An H.263 video decoder and a g-nibble game were selected as real-time applications coexisting to compete for CPU cycles. We qualitatively measure their performance on RTVR and compare with that on Linux. The H.263 decoder is a real-time application which decodes and displays 25 CIF (352 x 288 pixels) frames per second. The g-nibbles game is an interactive program which requests small CPU rate but also small delay. It can be used to subjectively test keyboard response. Figure 8 shows the maximal number of video decoder threads running on each platform while the keyboard response is still satisfactory at the same time. Adding one more video decoder on any platform will either cause the video pictures to suffer from delays and/or jerkiness, or perceptibly degrade the g-nibble game.

The static RLS prototype is set up with 1 SVP and 3 AVPs. The availability factor and supply regularity of each AVP is set to 0.25 and 1, respectively. Two video decoder threads can be run on each AVP with good quality while still satisfying the g-nibble.

The dynamic RLS prototype is set up with a SVP and 2 AVPs, each of which is a small partition. The first one shares 40% CPU and runs 3 decoder threads. The second one shares 50% CPU and runs 4 decoder threads.

This experiment convincingly showed that our RTVR prototypes can accommodate more real-time tasks when they compete for the same resource (CPU).

7.4 Applications Compete for More than One Resource

In this experiment, three identical video encoder were executed simultaneously on our desktop PC for 10 seconds. Each one encodes the same video file and then sends the coded stream out. Thus, they mainly compete for CPU computation and network bandwidth. We show how RTVR achieves better performance by co-partitioning both resources.

To simplify the test, the video encoder is set to code 1 I frame and then 4 P frames at the normal rate of 25 frames/s. The average output bandwidth request by each stream is 50 Kbps. Each encoder can buffer 5 frames waiting to be sent at most. To see how network scheduling affect the performance, we set the total available bandwidth to 300 kbps. Each packet is 1k bytes in length. For each frame, we measure the time difference (in ms) between when it is actually sent and when it is supposed to be sent. The delay from previous frames will accumulate and result in delay of later ones. We set the initial delay

as 1000 ms; thus the deadline of frame k to be sent out can be calculated as $1000 + 40k$ ms.

Figure 9-12 showed the result with all four combinations of resource partition, that is, with/without CPU/network partition. Figure 9 can be seen as the result when they run on original linux kernel. Each stream has about 50 % frames missing the deadline. Figure 10 is the performance measured with network partition only. We reserve 100 kbps bandwidth with latency 40 ms for each stream. the performance is much better than case 1, but still stream 2 and 3 has almost 20% deadline miss. Figure 11 is the performance measured with CPU partition only. There are no deadline miss. The gap between deadline and the actual time when packets are sent increases as simulation continues, which means it is possible to support more stream(s). 12 is the performance measured with both CPU and network partition. As expected, there are no deadline miss. And, the"gap" is larger compared to case 3, which means the best performance is achieved by proper resource co-partitioning.

7.5 RTVR Performance Tuning

We now discuss how to tune a RTVR system to achieve optimal performance.

The relationship between availability factor and partition delay is shown in Figure 13 in static RLS. For a given availability factor α, the actual partition delay D depends on the scheduling quantum Q and the supply regularity R_s. For example, given Q as 10 ms and α as 1/16, if R_s is 1, it is partitioned as 1 out of every 16 quantum which results in D equal to 150 ms. If R_s is 2, it is partitioned as 2 out of every 32 quantums and D can be as large as 300 ms. In Figure 13, the solid line and dash line represent the minimal and maximal partition delay for any given α in static RLS, respectively. In dynamic RLS, partition delay cannot be bounded by the rate only since the EDF algorithm is employed in RLS.

Take the H.263 decoder used in the experiment as an example. We measured its periodic deadline as 40 ms and rate as 1/9. To guarantee the 40 ms deadline, α on static RLS should be at least 1/4 and R_s be 1 as the point A in Figure 13. Since each one only requires rate 1/9, two decoders can run on the same partition, which explains the experiment setup of static RLS. Clearly, in the static RLS case, the deadline of the H.263 decoder is the constraint.

In dynamic RLS, the scheduling quantum of a small partition can be as small as 1 ms, which makes the rate to become the constraint. To maximize the number of the decoder threads running, we set two small partition AVPs. One shares 40% of CPU and accommodates 3 threads; the other shares 50% and accommodates 4. Note that theoretically we can group 8 decoder threads on an AVP sharing 90% CPU. However, as expected, too many threads grouped together will interfere with each other which makes some displays un-smooth.

Again, different supply regularity results in different partition delay even with the same availability factor in static RLS, such as point B and C in Figure 13. A larger supply regularity yields a larger partition delay which may violate the deadline of real-time tasks. On the other hand, it provides more scheduling flexibility, and possibly reduces scheduling overhead. For example, a RM scheduler can accommodate a task set $\{(C, P) \mid (4, 7), (2, 8), (2, 32)\}$, where C is the computational time and P is the task period. However, it cannot accommodate a task set such as $\{(C, P) \mid (4, 7), (2, 8), (1, 16)\}$.

Given a set of tasks with different real-time requirements, it is an interesting question how to tune a RTVR system to achieve optimal performance. We believe that all responsible factors such as partition delay, scheduling quantum, task grouping should be carefully investigated.

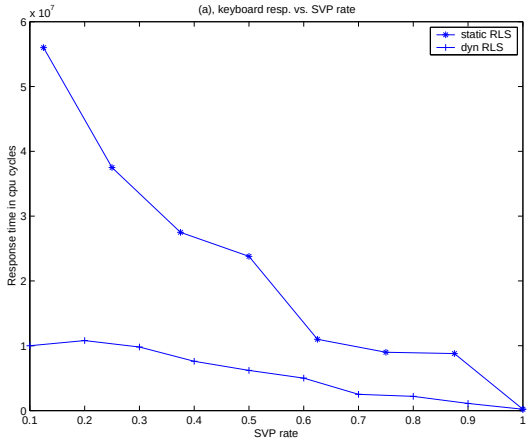

FIGURE 5: *Keyboard Response*

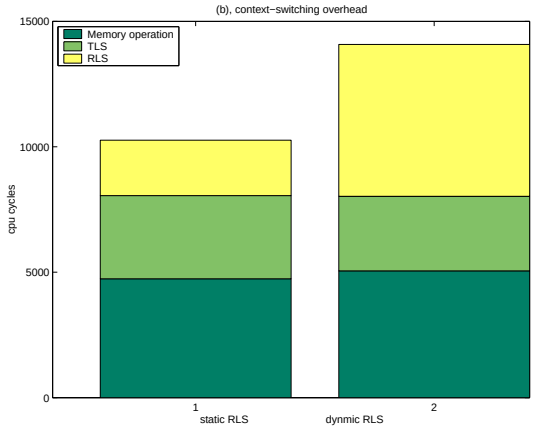

FIGURE 6: *Context-switching Overhead*

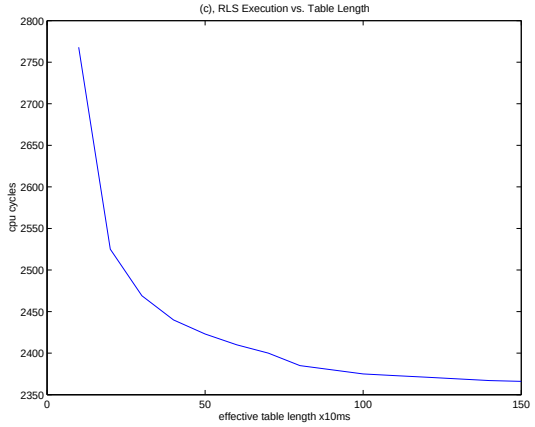

FIGURE 7: *RLS Execution Time*

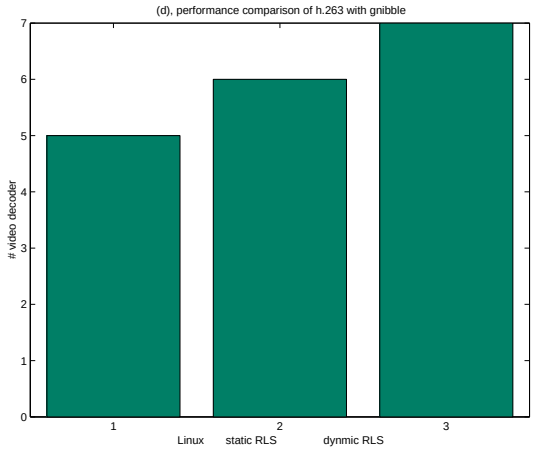

FIGURE 8: *Performance*

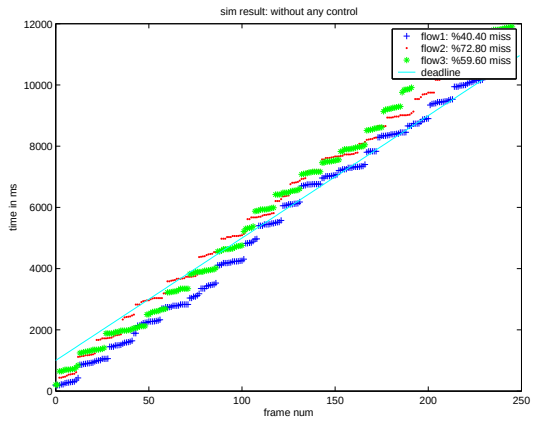

FIGURE 9: *Without Any Scheduling*

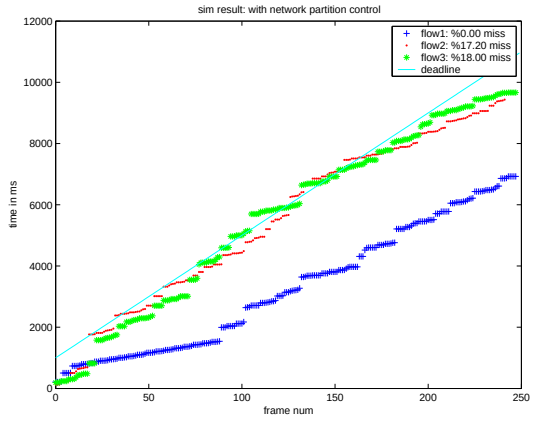

FIGURE 10: *Network Scheduling*

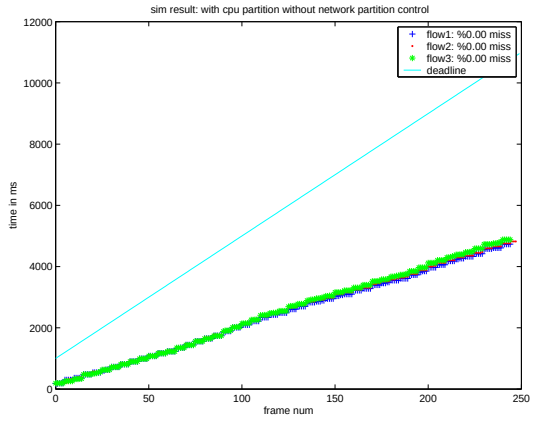

FIGURE 11: *CPU Scheduling*

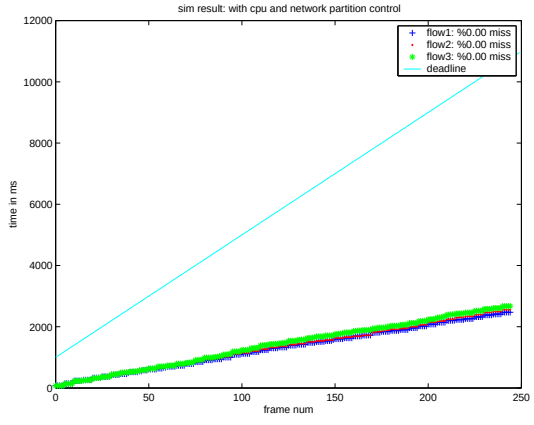

FIGURE 12: *CPU + Network Scheduling*

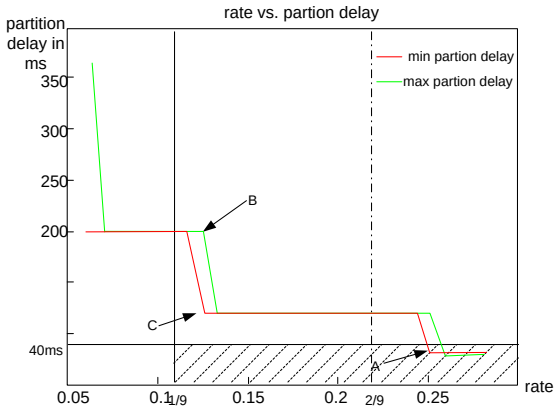

FIGURE 13: *Partition Rate vs. Delay*

8 Conclusions

In this paper, we have described two prototypes of the RTVR (Real-Time Virtual Resource) concept. The prototypes were implemented based on the Linux 2.4.18.3 kernel.

The RTVR concept is an elegant abstraction for programming real-time applications in an open environment. It was an open question, however, whether its theoretical advantages could actually be realized in practice. The work reported herein gives some credence to a positive answer to the implementability question.

Some highlights of our implementation are:

- The design of resource level scheduler was intensively investigated. High level description of static resource level scheduling and dynamic resource level scheduling were presented. In dynamic RLS, partitions are allowed to join and leave the system dynamically.
- Modifications made to Linux in order to support RTVR were explained in detail. The kernel scheduler implementation was emphasized. Device virtualization with network as a prototype is also discussed.
- Several experiments were conducted to measure system performance in various aspects including IRQ response time, scheduling overhead and memory consumption.

We are continuing our work on improving the implementation. Even though we have shown that our implementation can deal with more than one resource type, our longer-term goal is to be able to apply the real-time virtual resource concept to all types of resources that an operating system may need to manage.

References

[1] RTAI:http://www.rtai.org/

[2] RTLinux:http://www.fsmlabs.com/

[3] Z. Deng and J. Liu, 1997, "*Scheduling Real-Time Applications in an Open Environment*", in IEEE Real-Time Systems Symposium, pages 308-319.

[4] X. Feng and A. Mok, 2002, "*A Model of Hierarchical Real-Time Virtual Resources*", in IEEE Real-Time Systems Symposium, pages 26-35.

[5] P. Goyal and Harrick M. Vin and Haichen Cheng, 1996, "*Start-Time Fair Queuing: A Scheduling Algorithm for Integrated ServicesPacket Switching Networks*", Technique report, Dept. of Computer Sciences, Univ. of Texas at Austin (ftp://ftp.cs.utexas.edu/pub/techreports/tr96-02.ps.Z).

[6] K. Lee, 1995, "*Performance Bounds in Communication Networks with Variable-Rate Links*", in SIGCOMM, pages 126-136.

[7] G. Lipari and S. Baruah, 2001, "*A Hierarchical Extension to the Constant Bandwidth Server Framework*", in Real-Time Technology and Applications Symposium, pages 26-35.

[8] C. L. Liu and James W. Layland, 1973, "*Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*", in Journal of ACM, 20(1).

[9] A. K. Mok and X. Feng, 2001, "*Towards Compositionality in Real-Time Resource Partitioning Based on Regularity Bounds*", in IEEE Real-Time Systems Symposium, pages 129-138.

[10] A. Mok and X. Feng and D. Chen, 2001, "*Resource partition for real-time systems*", in Real-Time Technology and Applications Symposium, pages 75-84.

[11] A. K. Mok and X. Feng and D. Chen, 2001, "*Resource Partition for Real-Time Systems*", Technique report, Dept. of Computer Sciences, Univ. of Texas at Austin (ftp://ftp.cs.utexas.edu/pub/amok/UTCS-RTS-2001-01.ps).

[12] R. Rajkumar and L. Abeni and D. De Niz and S. Gosh and A. Miyoshi and S. Saewong, 2000, "*Recent developments with Linux/RK*", in PROCEEDINGS OF THE REAL TIME LINUX WORKSHOP".

[13] J. Regehr and J. Stankovic, 2001, "*HLS: A Framework for Composing Soft Real-Time Schedulers*", in IEEE REAL-TIME SYSTEMS SYMPOSIUM, pages =3-14.

[14] I. Shin and I. Lee, 2003, "*Periodic Resource Model for Compositional Real-Time Guarantees*", in IEEE REAL-TIME SYSTEMS SYMPOSIUM, pages 3-12.

[15] Yu-Chung Wang and Kwei-Jay Lin, 1999, "*Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel*, in IEEE REAL-TIME SYSTEMS SYMPOSIUM, pages 246-255.

[16] Geoffrey G. Xie and Simon S. Lam, 1995, "*Delay guarantee of virtual clock server*", in IEEE/ ACM TRANSACTIONS ON NETWORKING, 3(6): 683-689.