In appear in: *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, CA, May 2004.

Supporting Dynamic QoS in Linux

Xin Liu   Steve Goddard
Department of Computer Science and Engineering
University of Nebraska — Lincoln
Lincoln, NE 68588-0115
{lxin, goddard}@cse.unl.edu

Abstract

This work is an application of the Variable-Rate Execution (VRE) model in Linux to support dynamic Quality of Service (QoS). Based on conventional time-sharing scheduling algorithms, Linux does not adequately support QoS requirements. The VRE scheduler can assign a specified execution rate to any application, and dynamically adjust the execution rate during runtime. Rate controller components are introduced to adjust a task's execution rate based on predefined rules and runtime feedbacks, such as the suspension time, the queue length, and so on.

A significant feature of this work is its ability to support legacy applications at the binary level. On conventional operating systems, millions of applications have been built under time-sharing schedulers, which we call legacy applications. Under the VRE model, a legacy application can obtain a guaranteed variable execution rate. We also designed a simple default rate controller for legacy multimedia applications.

The Linux kernel was slightly modified in our implementation to achieve reconfigurability. Both the VRE scheduler and the default rate controller are implemented as Linux loadable modules, which can be dynamically loaded into the kernel to replace the Linux scheduler or change the behavior of the scheduler. We provide a set of interfaces for users to design and use their own schedulers and rate controllers.

1 Introduction

In recent years, many QoS-sensitive applications, typically multimedia applications, have brought out many QoS-supported execution models [2, 12, 16, 20] and schedulers [4, 25, 11, 29]. However, only static QoS support is not enough. In practice it is hard to determine the execution rate. Many applications have variable execution rates. For example, a multiple-target multiple-sensor radar tracking system pays more attention to fast-moving targets than to slow targets. And the platforms also affect the execution rate. Obviously, a program runs faster on an 800MHz P3 processor than on a 400MHz P2 processor.

On conventional operating systems, millions of applications have been built under time-sharing schedulers. We call those applications *legacy applications*. Many legacy applications, especially multimedia applications, already have QoS demands. However, conventional time-sharing systems do not provide any QoS support. Considering the millions of legacy applications, it is actually infeasible to rebuild all of those applications upon a completely new platform. A likely solution is to support legacy applications at the binary level.

The Variable-Rate Execution (VRE) model [10] was initially designed to solve the above problems. The VRE model is an extension of the Rate-Based Execution (RBE) model [12]. While the RBE model schedules tasks at a fixed average rate, the VRE model supports variable execution rates by allowing variable *period* and *execution time* parameters during runtime. It also allows tasks to join and leave the system at arbitrary times.

Work similar to the VRE model is the *rate-based earliest deadline* (RBED) scheduler presented in [6], which was independently and simultaneously developed. Although the theoretical model is not the main focus of this paper, we highlight the difference between the RBED scheduler and the VRE model in Section 2.

This paper mainly focuses on solving three problems: (i) modeling legacy applications under the VRE model; (ii) how to adjust the execution rates during runtime; (iii) the implementation and programming interface.

To be compatible with the time-sharing schedulers, we split the execution of a legacy application into sequential time slices. The difference in our execution model is that each task is described by two parameters, *period (p)* and *size of time slice (c)*, and each time slice is assigned a deadline based on the period. The Earliest Deadline First (EDF) scheduler guarantees that the task will receive a time slice of size *c* in every period of *p* time units. We adjust the values of

c and p to control the execution rate of a legacy application.

The *rate controller* component is introduced to solve problem (ii). Since different applications can have different execution patterns, it is difficult to provide a universal rate controller for all applications. Therefore, we provide a set of interfaces for users to customize their specific rate controllers. The construction of specific rate controllers is beyond the scope of this work. But, we designed a simple rate controller as a default rate controller for legacy multimedia applications.

This work, including the VRE scheduler and the rate controller mechanism, was implemented in Linux. We slightly modified the Linux kernel to be able to dynamically load user-customized schedulers. The *loadable scheduler* mechanism separates specific scheduling policies from the scheduling mechanism. Users can load their specific schedulers without rebooting the system as long as the schedulers are built in compliance with the *scheduler* interface introduced in Section 4.1. Moreover, users can change the execution pattern of a VRE task by attaching a specific rate controller.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces the processing model assumed in this work. Section 4 gives an overview on the programming interface. Section 5 presents the experiments and results. We conclude with a summary and discussion of future work in Section 6.

2 Related Work

In practice many QoS-supported systems are derived from proportional share scheduling algorithms. The SMART [25] and BERT [4] are derived from the Weighted Fair Queueing (WFQ) [9] algorithm, which is also known as packet-by-packet generalized processor sharing (GPS). The WFQ scheduler associates a weight to each task; all the tasks are executed in proportion to their weights. Each task is tagged with a dynamic expected finish time. When the first job of a task finishes, the expected finish time is updated according to the size of the next job. The WFQ scheduler always selects the task with the earliest finish time. The Worst-case Fair Weighted Fair Queuing (WF^2Q) [5] extends the WFQ to prevent a task from getting executed faster than expected in a perfect fair share scheduler.

QLinux [11, 29] employs the Hierarchical Start-time Fair Queuing (H-SFQ) algorithm, which is similar to WFQ. Rather than using an expected finish time in WFQ, each job is assigned a *start time* tag which is computed in a similar manner as the virtual finish time. The Start-time Fair Queuing scheduler always chooses the job with the earliest *start time* tag. QLinux partitions all threads into groups; each group reserves a bandwidth in proportion to its weight. Each group also has a scheduler that schedules the group members in proportion to their weights.

The proportional share scheduling algorithms make no QoS guarantees if the sum of total weights grows very large. The *Constant Bandwidth Server* [2] algorithm avoids this problem by using *utilization* instead of *weight*. Applications are run on a CBS server which is assigned a fixed bandwidth. The execution of an application is decomposed into a sequence of time slices, called a budget in the CBS server, and each time slice is assigned a deadline which is represented by the deadline of the CBS server. While a server is executing, its budget is reduced until it reaches zero. The server gets replenished when its budget expires. The VRE model follows the CBS method to handle legacy applications. To some extent, we can view a VRE task as a *variable bandwidth server*.

Several researchers have developed techniques for supporting variable computation times and/or release patterns (e.g., [20, 21]). However, each of these provides relatively strict bounds on how much these parameters are allowed to vary, as compared to the VRE model. Researchers have also proposed methods for reducing task execution rates or computation times in overload conditions (e.g., [1, 14, 15, 23]).

The first work to provide explicit increasing and decreasing hard QoS guarantees on a task-by-task basis appears to be the *elastic* task model created by Buttazzo, Lipari, and Abeni [8]. In the elastic task model, a task is parameterized by a five-tuple $(C, T_0, T_{min}, T_{max}, e)$ where C is the tasks's WCET, T_0 is the nominal period for the task, T_{min} and T_{max} denote minimum and maximum periods for the task, and e is an elastic coefficient. The elastic coefficient e "specifies the flexibility of the task to vary its utilization" [8]. In this case, the utilization is varied by changing the length of the period, which is allowed to "shrink" to T_{min} or "stretch" to T_{max}, depending on the system load. The VRE model used in this work also allows the period of a task to shrink or stretch. In the VRE model, however, no bounds on the length of the period are defined a priori. Moreover, the VRE model also supports increasing and decreasing the WCET, which is not supported by the elastic task model.

Other researchers have taken a system-level approach to support adaptive real-time computing (e.g., [7, 18, 19, 28, 13, 26]). Most of these systems focus on over-load conditions and use various combinations of value-based scheduling, mode changes, and/or feedback mechanisms to shed or reduce load in an attempt to meet the most critical deadlines.

The work most similar to the VRE model is the *rate-based earliest deadline* (RBED) scheduler presented by Brandt et al. in [6]. In that work, the authors try to "flatten the scheduling hierarchy" by supporting hard real-time, soft real-time, and non-real-time tasks with a single scheduler. Their algorithm allows periodic tasks to dynamically change utilizations and periods. However, the RBED scheduler does not provide any method to adjust pending deadlines. Thus, any rate change has to delay until all related pending jobs are finished. For example, suppose a best-effort task, T_1, is the only task in the system at time 0, then T_1 utilizes all computing capacity ($u_1 = 1$ where u_1 is T_1's utilization). Suppose T_1's deadline is $D_1 = 10$ and a hard real-time task T_2 arrives with utilization $u_2 = 0.5$ at time 5, T_2 has to wait until T_1 terminates and releases its bandwidth, which might be fatal for hard real-time tasks. In the VRE model, however, we can immediately accept T_2 by changing D_1 to 15. Generally speaking, the total utilization of a hybrid system is always 1 as long as there exist any non-real-time tasks. The acceptance of new tasks or rate change of existing tasks implies immediately changing the utilizations of other best-effort tasks.

Moreover, the underlying task model assumed by Brandt et al. in [6] is a generalization of the Liu and Layland periodic task model [17]. As stated, previously, the VRE model is a generalization of the RBE task model, which is a generalization of Mok's sporadic task model [22]. The VRE task model reduces to the task model in [6] when $x_i(t) = 1, \forall i, t$, and jobs are released with a strictly (variable) periodic pattern rather than a (variable) sporadic pattern.

3 The Task Model

This section discusses the processing model. Section 3.1 introduces the VRE model. Section 3.2 presents how legacy applications are processed under the VRE model. Section 3.3 discusses the schedulability condition. Section 3.4 introduces the concept of rate controllers.

3.1 Variable Rate Tasks

The VRE model is an extension of the RBE model [12] which schedules tasks at their average rates. A RBE task is specified by a four-tuple (x_i, y_i, d_i, c_i) of integer constants.

- The pair (x_i, y_i) is referred to as the rate specification of a RBE task; x_i is the maximum number of executions expected to be requested in any interval of length y_i.
- Parameter d_i is a response time parameter that specifies the maximum desired time between the release of a task instance and the completion of its execution (i.e., d_i is the relative deadline of the task).
- Parameter c_i is the maximum amount of processor time required for any job of task T_i to execute to completion on a dedicated processor.

A RBE task set is schedulable if there exists a schedule such that the j^{th} release of task T_i at time t_{ij} is guaranteed to complete execution by time $D_i(j)$, where

$$D_i(j) = \begin{cases} t_{ij} + d_i & \text{if } 1 \le j \le x_i \\ \max(t_{ij} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \tag{1}$$

The second line of Equation (1) prevents the processor from being saturated by early job releases.

The VRE model extends the RBE model to support variable execution rate. Following the notation of the RBE model, a VRE task is described by four parameters $(x_i(t), y_i(t), d_i(t), c_i(t))$, which allows the parameters to change during runtime. Thus, the deadline assignment equation evolves to Equation (2).

$$D_i(j) = \begin{cases} t_{ij} + d_i(t) & \text{if } 1 \le j \le x_i(t) \\ \max(t_{ij} + d_i(t), D_i(j - x_i(t)) + y_i(t)) & \text{if } j > x_i(t) \end{cases} \tag{2}$$

where t_{ij} is the release time of job J_{ij}.

The variable execution rate is achieved primarily by adjusting either or both the *period* ($y_i(t)$) and the *execution time* ($c_i(t)$) parameters. As an example, a video player might change its QoS requirements by either reducing resolution or skipping frames, which requires a change to the $c_i(t)$ or $y_i(t)$ parameter respectively.

There might be pending jobs when a VRE task wants to change its rate. Several options are available to handle the pending jobs. A lazy and feasible way is to delay the rate change until all pending jobs are done. That is the approach Brandt et al. adopted in [6]. As we already mentioned in Section 2, there exist cases where we want the rate change to immediately take effect. Thus, we change the deadlines of the pending jobs as follows.

- *Rule 1: $c_i(t)$ changes at time t_x.* Let $D_i(j)$ be a pending deadline, $D'_i(j)$ be the modified deadline according to the rate change, and $s_i(t_x)$ be the received execution time by time t_x, then,

  $$D'_i(j) = t_x + max((D_i(j) - t_x) \cdot \frac{c_i(t_x - 1)}{c_i(t_x)}, c_i(t_x - 1) - s_i(t_x))$$

- *Rule 2: $y_i(t)$ changes at time t_x,*

  $$D'_i(j) = t_x + ((D_i(j) - t_x) \cdot \frac{y_i(t_x)}{y_i(t_x - 1)}, c_i(t_x - 1) - s_i(t_x))$$

- *Rule 3: $x_i(t)$ changes at time t_x,*

  $$D'_i(j + m) = t_x + y_i(t_x) \cdot (\lfloor \frac{m}{x_i(t_x)} \rfloor + 1),\ 0 \le m \le k$$

In [10] and this work, we assume $y_i(t) = d_i(t)$ to obtain an efficient schedulability test that can be used on-line as an admission control condition. See [10] for more details.

3.2 Modelling Legacy Applications

The intended applications of this work are legacy applications which have been built on conventional time-sharing schedulers. Since legacy applications are sequential instruction streams, the processing of a legacy application under the VRE task model follows the way of conventional time-sharing systems, being executed by time slices. The difference in the VRE model is that each time slice is treated as a job and assigned a deadline. A legacy task T_i is inserted into the ready queue with other VRE tasks, and scheduled with the EDF scheduling algorithm. When job J_{ij} of task T_i is dispatched (i.e., begins to execute), an execution timer is set to preempt the execution of job J_{ij} after $c_i(t)$ time units. If task T_i is preempted by another task, the execution timer state is saved with the context of task T_i and restored when job J_{ij} resumes execution. When the timer set for job J_{ij} expires, task T_i is preempted and, as though one job had completed and a new job released, a new deadline is set for job J_{ij+1} using Equation (2) and the rate parameters of T_i, which is similar to the method used in [2] when a request overruns the server's budget.

The deadline assignment of a legacy application requires two parameters, $p_i(t)$ and $c_i(t)$, where $p_i(t)$ is the period and $c_i(t)$ is the size of a time slice. To be consistent with the VRE model, a legacy application is described as $(1, p_i(t), p_i(t), c_i(t))$, which means the task will be allocated 1 time slice of size $c_i(t)$ every period of length $p_i(t)$. The rate adjustment is implemented by the adjustment of either the $p_i(t)$ parameter or the $c_i(t)$ parameter.

A main problem with the legacy applications is that conventional time-sharing systems are unaware of the execution rate or the time constraints. On this problem, the VRE model shows a significant advantage over other algorithms (CBS, RBE, etc.). Users have no need to know the exact execution rate in advance. Instead, they can just assign an approximate execution rate and adjust the rate during runtime. In practice, most multimedia applications have a fixed period. Suppose in a video, people will notice excessive jitter if the rate is less than 15 frames per second, then $66ms$ can be used as its period ($p_i(t)$). The $c_i(t)$ parameter is a variable which we can change during runtime. The rate adjustment is like a feedback-based control loop. We allocate an initial rate to a legacy application, monitor the actual execution and adjust the rate based on the feedbacks, for example, the suspension time in a given interval or the percentage of missed deadlines.

Figure 1 and Figure 2 are two simple examples that illustrate how the variable rate execution model works. For simplicity, the rate changes in these examples are made at task deadlines, but this is not required. In Figure 1, the initial execution rate is $(1, 4, 4, 2)$, and the $c_i(t)$ parameter is adjusted during runtime. At time $t = 4$, the WCET is changed from 2 to 1. Thus, execution rate changes to $(1, 4, 4, 1)$, and the next two execution intervals each require at most 1 time unit. At time $t = 12$, the task's $c_i(t)$ parameter is changed to 2, and the execution rate changes back to its initial specification: $(1, 4, 4, 2)$. This example might represent a scenario in which a video player changes its resolution and needs more or less execution time in an interval of $y_i(t)$ time units.

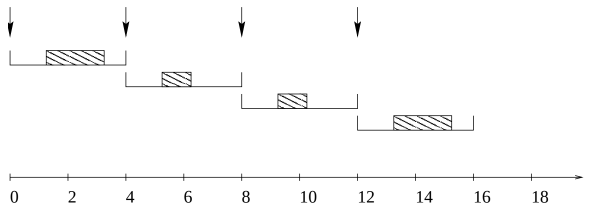


Figure 1. The initial execution rate is $(1, 4, 4, 2)$. At time 4, the execution rate changes to $(1, 4, 4, 1)$, and the execution rate changes back to $(1, 4, 4, 2)$ at time 12.

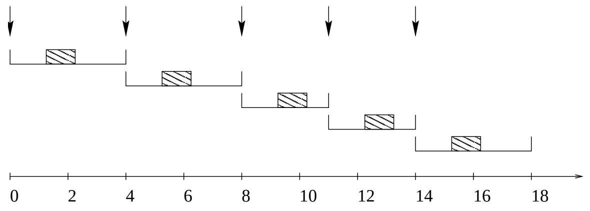


Figure 2. The initial execution rate is $(1, 4, 4, 1)$. At time 8, the execution rate changes to $(1, 3, 3, 1)$, and the execution rate changes back to $(1, 4, 4, 1)$ at time 14.

A scenario in which a video player skips frames is shown in Figure 2. In this case, the $y_i(t)$ parameter is adjusted during runtime. The initial execution rate specification is $(1, 4, 4, 1)$, and at time $t = 8$ the execution rate changes to $(1, 3, 3, 1)$. The execution rate changes back to $(1, 4, 4, 1)$ at time $t = 14$.

3.3 Schedulability Condition

A sufficient schedulabity condition for the VRE task set was given in [10]. We summarize the schedulability theorems from [10] in this section.

Lemma 3.1 defines a loose upper bound for the demand of a variable rate task in any given interval; Theorem 3.2 gives a sufficient but not necessary schedulability condition for a VRE task set.

Lemma 3.1. [10] *Let V_i be a variable rate task $(x_i(t), y_i(t), y_i(t), c_i(t))$. If no job of V_i released before time $t_0 \geq 0$ requires processor time in the interval $[t_0, l]$ to meet a deadline in the interval $[t_0, l]$, then*

$$\forall l > t_0, \ \widehat{dbf}([t_0, l]) = \int_{t_0}^{l} f_i(t)dt \qquad (3)$$

is an upper bound on the processor demand in the interval $[t_0, l]$ *created by* V_i *where* $f_i(t)$ *is the fraction function of* V_i *computed by* $f_i(t) = \frac{x_i(t) \cdot c_i(t)}{y_i(t)}$.

Theorem 3.2. [10] *Let the task set* $\mathcal{V} = \bigcup_{t=0}^{\infty} V(t)$ *be a set of variable rate tasks with* $d_i(t) = y_i(t), 1 \le i \le n$. *Preemptive EDF will succeed in scheduling* $\mathcal{V}$ *if*

$$\forall L > 0, L \ge \sum_{j \in \mathcal{V}} \widehat{dbf}_j(L) \tag{4}$$

Corollary 3.3. [10] *Let the task set* $\mathcal{V} = \bigcup_{t=0}^{\infty} V(t)$ *be a set of variable rate tasks with* $d_i(t) = y_i(t), 1 \le i \le n$. *Preemptive EDF will succeed in scheduling* $\mathcal{V}$ *if Equation* (5) *holds.*

$$\forall t, \sum_{i \in V(t)} f_i(t) \le 1 \tag{5}$$

Equation (5) looks like the necessary and sufficient condition of EDF in [17], but it is actually different. The VRE model supports a dynamic task set in which tasks are allowed to release jobs early. This means we can have intervals of time in which the utilization function is greater than 1 adjacent to intervals of time in which the utilization function is less than 1, and the task set may still be schedulable. Thus, Equation (5) is only sufficient, and not necessary. To develop a tighter condition, which is both sufficient and necessary, the actual times of rate changes must be known a priori. Thus, it is infeasible to evaluate such a condition.

Corollary 3.3 can be used as the condition for admission and rate-change control. When a new variable rate task arrives or an existing variable rate task requests to change its rate, the system will recompute the sum of the fractions. If the sum is less than or equal to 1, accept the request; otherwise, reject the request.

3.4 Rate Controller

The rate adjustments might be complicated. In this section, we introduce a *rate controller* component to automatically accomplish rate adjustments. The role of the rate controller is to monitor the execution and adjust the rate based on specific feedback and predefined rules.

In practice, different applications can have different execution patterns, and we believe it is infeasible to construct a general-purpose rate controller. For example, we were trying to solve the *receive livelock* problem in a recent work [30]. In that work, we adaptively adjust the execution rate of a network application according to the buffer utilization of its socket layer queue. We provide a set of interface for users to design their own rate controllers. Section 4.3 introduces the programming interface for rate controllers. Each variable rate task can have its own rate controller adjusting the execution rate.

We designed a simple rate controller as the default rate controller for legacy multimedia applications. The design assumes a legacy application will suspend itself when it runs faster than its need. Take a video decoder as an example, the decoder shall decode 30 frames in a second. If the decoder decodes 30 frames in the middle of a second, then it will suspend itself for the rest of the second.

Suppose $V_i = (x_i(t), y_i(t), d_i(t), c_i(t))$ is a variable rate task. Let $s_i(t)$ be the received execution time of V_i by time t and $S_i(t)$ be the expected execution time by time t, $S_i(t) = \int_0^t f_i(t) d_t$ where $f_i(t) = \frac{x_i(t) \cdot c_i(t)}{y_i(t)}$. The default rate controller acts as follows.

- periodically check the execution time ($s_i(t)$). Suppose we check the execution time at time t ($s_i(t)$), then we check the execution time after p time units ($s_i(t+p)$).
- compute the received execution time in the period $[t, t+p]$, $\Delta s_i = s_i(t+p) - s_i(t)$.
- compare Δs_i with the expected execution time ($\Delta S_i = S_i(t+p) - S_i(t) = p \cdot f_i(t) = p \cdot \frac{x_i(t) \cdot c_i(t)}{y_i(t)}$).
  - If $\Delta S_i - \Delta s_i > \delta_u$ where δ_u is a pre-defined upper bound, then reduce the execution rate, either decreasing $c_i(t)$ or increasing $y_i(t)$.
  - If $\Delta S_i - \Delta s_i < \delta_l$ where δ_l is a pre-defined lower bound, then increase the execution rate, either increasing $c_i(t)$ or decreasing $y_i(t)$.
  - Otherwise, keep the execution rate.

4 Programming Interface

This section introduces some implementation details and the programming interface. Section 4.1 presents the loadable scheduler mechanism. Section 4.2 introduces the programming interface for variable rate tasks. Section 4.3 introduces the programming interface for rate controllers.

4.1 Loadable Schedulers

The loadable scheduler mechanism is implemented by pre-planting several hooks in the original Linux kernel. The hooks are pre-planted as follows:

```
int do_fork(unsigned long clone_flags,
            unsigned long stack_start,
            struct pt_regs *regs,
            unsigned long stack_size)
{
...
#ifdef LOADABLE_SCHEDULER
        p->policy=SCHED_EDF;
        if(p_sched!=NULL)
                (*p_sched->sched_fork)(p);
#endif
...
}
```

When a scheduler is loaded, the *p_sched* pointer is directed to that scheduler. Then, the user-defined *sched_fork* function will be invoked when a thread is created by the *do_fork* function. When the scheduler is removed, it sets the *p_sched* pointer back to *NULL*.

The interface for user-customized schedulers is defined by a data structure *scheduler*, which is shown in the following code.

```
/*scheduler structure*/

/*this structure defines the hook table in kernel*/

struct scheduler{
    struct prio (*sched_goodness)
        (struct task_struct * p, int this_cpu,
         struct mm_struct *this_mm);
    void (*sched_fork)
        (struct task_struct *task);
    void (*sched_exit)
        (struct task_struct *task);
    void (*sched_sleep)
        (struct task_struct *task);
    void (*sched_wakeup)
        (struct task_struct *task);
    struct task_struct *(*sched_choose_next)
        (struct task_struct*, int);
    void (*sched_nice)
        (struct task_struct*, int, void*);
    void (*sched_handle_ticks)
        (struct task_struct*, unsigned long );
    void (*sched_replenish)
        (struct task_struct*);
};
```

4.2 Variable Rate Threads

The programming model of variable rate threads is consistent with conventional time-sharing systems. Thus, legacy applications need no modification to run under the VRE scheduler. Variable rate threads bridge the gap between legacy time-sharing applications and QoS-based applications. The system call $set\_execution\_rate$ transfers a legacy application to a variable rate thread and set its execution rate. Typically, we set the rate of a legacy application as follows.

```
set_execution_rate(pid, 1, y, y, c);
```

The above statement sets the execution rate of thread pid to $(1, y, y, c)$ which means pid will run c time units every y time units.

4.3 Rate Controller

A rate controller is currently defined as follows:

```
struct rate_ctlr{
   unsigned long period;
   void (*control)(struct task_struct *p_task);
}
```

where *period* is the period that the function *control* is invoked.

Three system calls dealing with rate controllers are shown below.

- *add_controller(char *name_of_ctlr, struct rate_ctlr *ctlr)* adds a rate controller *ctlr* into the system;
- *rm_controller(struct rate_ctlr *ctlr)* removes a rate controller *ctlr* from the system;
- *set_controller(pid_t pid, char *name_of_ctlr, unsigned long period)* assigns a rate controller *name_of_ctlr* to a thread *pid*.

5 Evaluation

The programming and task models have been implemented in Linux and evaluated with non-real-time and legacy multimedia applications. The scheduler is implemented as a loadable Linux module on a Redhat 8.0 distribution with a 2.4.18 Linux kernel. Most functions are implemented by the loadable modules; only a small modification is made to the kernel.

Our experiments focus on three aspects: QoS support, the variable rate mechanism and the default rate controller. The experiments were done on a PC with an Athlon Thunderbird 1GHz processor. We selected *MPlayer* as the legacy multimedia application.

The first experiment is to evaluate the QoS support. We ran *MPlayer* on both the original Linux scheduler and our new scheduler. In our scheduler, the execution rate of *MPlayer* is set to $(1, 10, 10, 1)$. That is, the *MPlayer* shall receive 1 time tick (10 microseconds) every 10 time ticks, which is actually faster than its required execution rate. As shown in Figure 3, the actual execution rate of *MPlayer* is close to $(1, 12, 12, 1)$. Every 15 seconds, we check the execution time of *MPlayer* and create 20 non-real-time processes which do nothing but execute infinite loops. As we can see in Figure 3, the original Linux scheduler provides no QoS guarantee; the execution rate (the slope) decreases when new processes are created. But our model provides constant service quality when new processes are created.

The second experiment is on the execution pattern of legacy multimedia applications under the rate adjustment mechanism. We changed the execution rate of *MPlayer* at selected points and checked the execution time of *MPlayer* every 3 seconds.

Table 1 shows the execution rate adjustments and the times when they are made. We use different time units for the two columns, which might be confusing at first. We use a *second* as the time unit in the *Time* column, while in the *Rate* column we use a *tick* (which is $10ms$ in Linux by default) as the time unit. For example, the second entry in Table 1

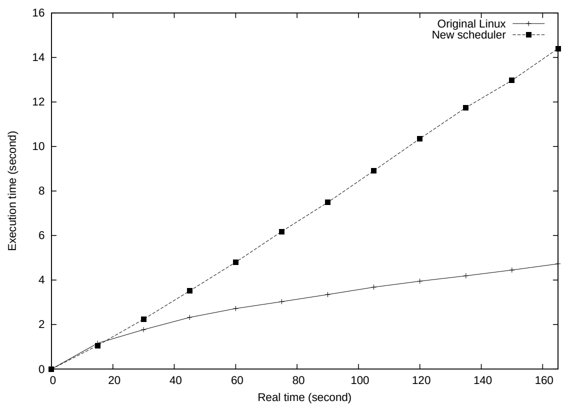


Figure 3. The execution rate in the original Linux scheduler decreases as the system load increases while the new scheduler maintains a constant execution rate. The actual execution rate in our scheduler is close to $(1, 12, 12, 1)$, which is slower than the assigned rate.

Time(in seconds)	Rate (x,y,d,c) (in 10ms ticks)
0	(1,50,50,1)
30	(1,30,30,1)
60	(1,20,20,1)
90	(1,10,10,1)
120	(1,10,10,2)

Table 1. Rate adjustment.

means we change the rate to 1 tick every 30 ticks at the 30th second.

The execution time of *MPlayer* with rate adjustment is shown in Figure 4. To highlight the execution rate of *MPlayer*, the null program execution is not included in Figure 4. We can verify that the rate changes are consistent with our adjustments shown in Table 1.

An interesting thing in Figure 4 is the rate change around time 155 where we did not adjust the rate. It appears that rate $(1, 10, 10, 2)$ is faster than the actual rate of job releases. When the execution rate was changed to $(1, 10, 10, 2)$ at time 120, *MPlayer* had to finish the pending jobs accumulated through the interval $[0, 120]$. Thus, *MPlayer* executed at its full rate $(1, 10, 10, 2)$ in the interval $[120, 155]$. When the accumulated pending jobs were finished, its execution rate dropped to the actual rate of job releases.

The third experiment is on the default rate controller. We initially set *MPlayer* to run at a low rate ($(1, 20, 20, 1)$). The actual execution rate is a little bit slower than the assigned rate because of the roundoff in the implementation. Then we attached the default rate controller to *MPlayer* at time 45. As we can see from Figure 5, the default controller immediately detected the contention and accelerated the execution rate. When all pending jobs were finished, the execution rate

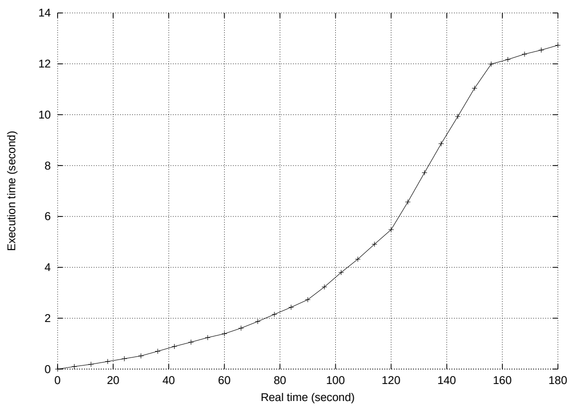


Figure 4. The actual execution rate increases when we increase its rate specification, but falls around time 155 when the rate specification is greater than the actual job release rate.

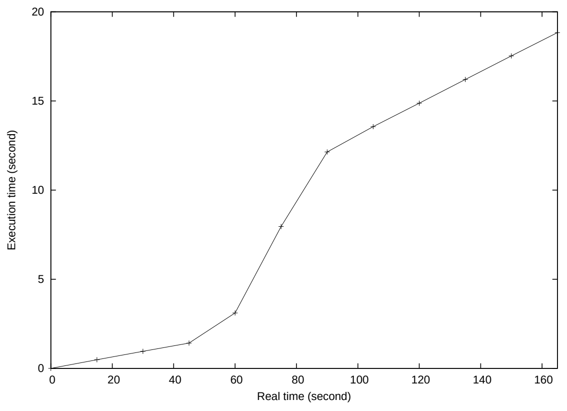
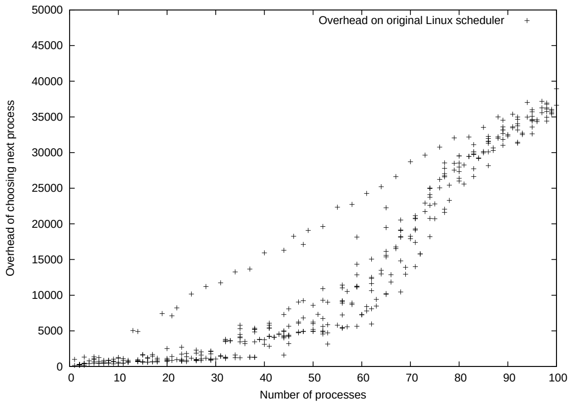


Figure 5. The default rate controller detected contention at time 45 and accelerated the execution rate.

dropped to the regular rate. The results look similar to the second experiment, but the rate adjustment was done automatically by the controller.

In the Linux kernel, all running processes are put in a list called *runqueue*. The Linux scheduler scans the entire list and selects the process with the highest priority. Our implementation also follows this pattern though another implementation might be more efficient. Thus, the overhead shall be a linear function of the number of running processes. We measured the overhead of our scheduler, and compared it with the overhead of the original Linux scheduler. The overhead was measured in CPU cycles, which was retrieved by the "*rdtsc*" instruction (*read timestamp counter*). Figure 6 shows our results of the measurements. These results are consistent with Brandt et al in [6] where a slightly simpler variable rate task model was implemented in Linux, with the change made to the kernel rather than as a loadable module.



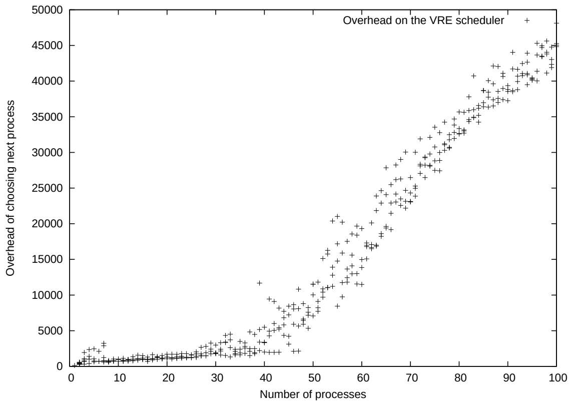


Figure 6. The overhead is a linear function of the number of processes under both the original Linux scheduler and our scheduler. The overhead of our scheduler is a little bit higher than the original Linux scheduler.

6 Conclusion

This work enhanced the Linux kernel to dynamically load user-customized schedulers. A variable rate execution scheduler is implemented to provide dynamic QoS support. Variable rate tasks run at a variable rate which is subject to change. Each variable rate thread can attach a rate controller to adjust its execution rate during runtime. A schedulability condition is given for admission control.

Legacy applications are modeled as specific VRE tasks. The VRE model does not require the exact execution rate to be known in advance. Users can assign an initial execution rate to a legacy application and adjust its rate later.

The implementation is done on a Redhat 8.0 distribution. Our experiments show that it is possible to approximate the execution rate of a legacy application without knowing the internal time constraints.

References

[1] Abdelzaher, T.F., Atkins, E.M., and Shin, K.G. "QoS Negotiation in Real-Time Systems and Its Applications to Automated Flight Control," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[2] Abeni, L., Buttazzo, G., "Integrating Multimedia Applications in Hard Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.*, Madrid, Spain, Dec. 1998.

[3] Bavier, A., Montz, A., and Peterson, L., "Predicting MPEG execution times," *Proc. of the Joint International Conf. on Measurement and Modelling of Computer Systems*, Madison, WI, Jun. 1998, pp. 131-140.

[4] Bavier, A., Peterson, L. and Mosberger D., "BERT: A scheduler for best effort and real-time tasks," Technical Report TR-602-99, Department of Computer Science, Princeton University, Mar, 1999.

[5] Bennett, J., Zhang, H., "WF2Q: Worst-case Fair Weighted Fair Queueing," IEEE INFOCOM '96, San Francisco, CA, Mar. 1996, pp. 120-128.

[6] Brandt, S. A., Banachowski, S., Lin, C., and Bisson, T., "Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes," *Proceesings of IEEE Real-Time System Symposium*, December 2003, pp. 396-407.

[7] Burns, A., D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Stringini, "The Meaning and Role of Value in Scheduling Flexible Real-Time Systems," *Journal of Systems Architecture*, 2000.

[8] Buttazzo, G. C., Lipari, G., and Abeni, L., "Elastic Task Model for Adaptive Rate Control," *Proceesings of IEEE Real-Time System Symposium*, December 1998, pp. 286-295.

[9] Demers, A., Keshav, S., and Shenker, S., "Analysis and simulation of a fair queuing algorithm," *Proceedings of the SIGCOMM '89 Symposium*, Sep. 1989, pp. 112.

[10] Goddard, S., Liu, X., "A Variable Rate Execution Model," *in Proceedings of the 16th Euromicro Conference on Real-Time Systems*, Catania, Sicily, Italy, June 2004.

[11] Goyal, P., Guo, X. and Vin, H. M., "A hierarchical CPU scheduler for multimedia operating systems", In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, October 1996.

[12] Jeffay, K., Goddard, S., "A Theory of Rate-Based Execution," *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1999, pp. 304-314.

[13] Jehuda, J. and A. Israeli, "Automated Meta-Control for Adaptable Real-Time Software," *Real-Time Systems Journal*, 14(2), pp. 107-134, Mar. 1998.

[14] Kuo, T.-W., and Mok, A. K, "Load Adjustment in Adaptive Real-Time Systems," *Proceedings of the 12th IEEE Real-Time Systems Symposium*, December 1991.

[15] Lee, C., Rajkumar, R., and Mercer, C., "Experiences with Processor Reservation and Dynamic QoS in Real-Time Mach," *Proceedings of Multimedia Japan 96*, April 1996.

[16] Lipari, G., Baruah, S., "Greedy reclamation of unused bandwidth in constant-bandwidth servers", *Proceedings of the EuroMicro Conferences on Real-Time Systems,* pp. 193-200, Stockholm, Sweden. June 2000.

[17] Liu, C., Layland, J., "Scheduling Algorithms for multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, Vol 30., Jan. 1973, pp. 46-61.

[18] Lu, C., J. Stankovic, T. Abdelzaher, G. Tao, S. Son, and M. Marley, "Performance Specifications and Metrics for Adaptive Real-Time systems," *Proceedings of IEEE Real-Time Systems Symposium*, pp. 13-22, Nov. 2000.

[19] McElhone, C., and A. Burns, "Scheduling Optional Computations for Adaptive Real-Time Systems," *Journal of Systems Architectures*, 2000.

[20] Mercer, C. W., Savage, S., and Tokuda, H., "Processor Capacity Reserves for Multimedia Operating Systems," *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[21] Mok, A. K., and Chen, D., "A multiframe model for real-time tasks," *Proceedings of IEEE Real-Time System Symposium*, Washington, December 1996.

[22] Mok, A.K.-L., *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*," Ph.D. Thesis, MIT, Department of EE and CS, MIT/LCS/TR-297, May 1983.

[23] Nakajima, T., "Dynamic QOS Control and Resource Reservation," *IEICE, RTP'98*, 1998.

[24] Nett, E., M. Gergeleit, and M. Mock, "An Adaptive Approach to Object-Oriented Real-Time Computing," *Proceedings of ISORC*, April 1998.

[25] Nieh, J., Lam, M., "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications," *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malô, France, Oct. 1997, pp. 184-197.

[26] Rosu, D., K. Schwan, S. Yalamanchili, and R. Jha, "On Adaptive Resource Allocation for Complex Real-Time Applications," *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 320-329, Dec. 1997.

[27] Rosu, D., K. Schwan, and S. Yalamanchili, "FARA-A Framework for Adaptive Resource Allocation in Complex Real-Time Systems," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1998.

[28] Steere, D., A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A Feedback-driven Proportion Allocator for Real-Rate Scheduling," *Proceedings of the Symposium of Operating Systems Design and Implementation*, 1999.

[29] Sundaram, V., Chandra, A., Goyal, P., Shenoy, P., "Application Performance in the QLinux Multimedia Operating System," *Proceedings of the Eighth ACM Conference on Multimedia*, Los Angeles, CA, November 2000, pp. 127-136.

[30] Wang, C., "The Design, Implementation and Evaluation of APS: Adaptive Proportional Share Scheduling," Master Thesis, University of Nebraska - Lincoln, Department of Computer Science and Engineering, August 2003.