ENERGY SAVING SCHEDULING FOR EMBEDDED REAL-TIME LINUX APPLICATIONS

Claudio Scordino and **Giuseppe Lipari**
Scuola Superiore Sant'Anna
Viale Rinaldo Piaggio, 34 - 56025 Pontedera - Pisa, Italy
{scordino@gandalf.sssup.it,lipari@sssup.it}

Abstract

The problem of reducing energy consumption is becoming very important in the design of embedded real-time systems. Many of these systems, in fact, are powered by rechargeable batteries, and the goal is to extend, as much as it is possible, the autonomy of the system. To reduce energy consumption, one possible approach is to selectively slow down the processor frequency.

In this paper we propose a modification of the Linux kernel to schedule aperiodic tasks in a soft real-time environment. The proposed solution consists in a new scheduling strategy based on the Resource Reservation Framework [9], which introduces very little modification to the Linux API. Our scheduler is based on Algorithm GRUB (Greedy Reclamation of Unused Bandwidth), presented by Baruah and Lipari [5].

After presenting the algorithm, we describe its implementation on a Intrinsyc CerfCube 250, which uses a Intel PXA250 processor. We show with an example of multimedia application that, by using our approach, we save up to 38% of energy with respect to an unmodified Linux.

1 Introduction

Although there is a mathematical theory to exactly formalize the behaviour of a real-time system, sometimes the design of these systems is still made in a rough way. Some designers consider that a fast enough system is always able to respond in a satisfactory way, and do not consider other factors (like size, cost, or energy consumption) that are decisive in the embedded systems sector.

The problem of reducing energy consumption is becoming very important in the design and implementation of embedded real-time systems. Many of these systems are powered by rechargeable batteries, and the goal is to extend as much as it is possible the autonomy of the system. A similar problem is being addressed in normal workstation PCs. As processors become more and more powerful, their energy consumption increases correspondingly, and it becomes a problem to dissipate the heat produced by the processor.

To reduce energy consumption, one possible approach is to selectively slow down the processor frequency. By reducing the frequency, it is also possible to reduce the voltage at which the processor is functioning, so reducing the power consumption. For example, if the processor has little to do for a certain interval of time, by reducing the frequency we slow down the processor speed but we can still complete all activities in time.

However, by reducing the frequency and the processor speed, we increase the load of the system. In particular, a certain task will take more time to be executed. In real-time systems, this means that some important task may miss its deadline. Therefore, it is important to identify the conditions under which we can safely slow down the processor without missing any deadline.

In this paper we propose a modification of the Linux scheduler that is able to change the processor frequency reducing the energy consumption, guaranteeing at the same time that no task will miss its deadline. The proposed solution consist in a new scheduling strategy based on the Resource Reservation Framework , which introduces very little modification to the Linux API. Basically, a new scheduler, called SCHED_CBS is available to the user of an application. In this way, it is possible to apply the same technique also to legacy Linux code.

After presenting the algorithm, we describe its implementation on a Intrinsyc CerfCube 250, which consists of a Intel PXA250 processor with 32Mb of Flash ROM and 64 Mb of SDRAM. We configured the system to support three different frequencies, 100Mhz, 200Mhz and 400 Mhz. The modified OS is Linux 2.4.18. We show an experiment in which we compare an unmodified version of Linux with our modified version. In the experiment, we run a multimedia application where tasks are dynamically activated and suspended with a highly variable load. We show that by using our approach, we save up to 38% of energy with respect to an unmodified Linux.

This work has been done in the context of the OCERA project (IST-35102), which is financially supported by the European Commission.

2 Related Work

It's not easy to compare our work with other existing solutions. This is essentially due to two elements of difference: the kind of real-time system, and the kind of experimental tests.

We work in an aperiodic environment, so the problem is not so trivial as in a periodic context, where if a task stops, then it will not execute until the next period (as in [7]). In our system a task could activate at any time. This assumption reduces the set of time points in which we can safely reduce the frequency clock.

Moreover, our results are not based upon some simulation (as in other works): we actually implemented our scheduling policy in Linux (creating a soft real-time system) and we actually measured the amount of energy used by the system.

3 Algorithm GRUB

We chose to implement the *Greedy Reclamation of Unused Bandwidth* algorithm ([5]), which is an improvement of the *Constant Bandwidth Server* ([2]).

GRUB is an algorithm belonging to the class of *aperiodic servers with dynamic priorities*. This technique consists in creating an entity, referred as server, that manages a set of tasks. However, a limit of some aperiodic servers with dynamic priorities is that they rely on the knowledge of execution times of served aperiodic tasks. In some cases, though, the execution time of a task is unknown, or extremely variable from an instance to another (consider, for example, a MPEG player). In these cases, the use of a hard real-time system to manage this kind of applications would be unsuitable for two reasons:

- First, the worst case execution time (WCET) of the job could be much higher than its average execution time. Since the guarantees for hard real-time tasks are given on the basis of the WCET (and not on the basis of the average execution time), this kind of applications could cause an enormous waste of resources. In fact, the system is sized according to the WCET of each real-time task, and this leads to a very partial utilisation of the performance that it could offer.

- Second, it's hard to provide an exact evaluation of the WCET. The fact that the real-time guarantees depend on the evaluation of the WCET of each job, makes the hard real-time system weak respect to some mistake in this evaluation. If a job doesn't respect the evaluated execution time, another task could miss its deadline.

GRUB is not affected by this problem because it doesn't rely on an evaluation of the WCET. It guarantees temporal isolation among tasks: as a consequence, a task can't affect the performance of another task.

In our model, each server is characterised by two parameters, $(U_i,\ P_i)$, where U_i is the server bandwidth (or fraction of the processor utilisation) and P_i is the period. Algorithm GRUB provides an abstraction of "slower processor": the task served by a server S_i with bandwidth U_i executes as it were executing on a dedicated slower processor with a minimum speed equal to U_i times the speed of the real processor.

The period P_i represents the *granularity* of the time from the point of view of the server. The smaller P_i, the closer is the virtual time to the real-time.

Each task τ_i executing on server S_i generates a sequence of jobs $J_i^1,\ J_i^2,\ J_i^3,\ \ldots,$ where J_i^j becomes ready for execution (arrives) at time a_i^j ($a_i^j \leq a_i^{j+1} \forall i, j$), and requires a computation time of c_i^j. We assume that, inside each server, these jobs are executed in FIFO order, i.e. J_i^j has to finish before J_i^{j+1} can start executing.

We make the following requirements of our scheduling discipline:

- The arrival times of the jobs (the a_i^j's) are not *a priori* known, but are only revealed on line during system execution. Hence, our scheduling strategy cannot require knowledge of future arrival times.

- The exact execution requirements c_i^j are also not known beforehand: they can only be determined by actually executing J_i^j to completion. (Nor do we require an *a priori* upper bound (a "worst-case execution time") on the value of c_i^j.)

- We are interested in integrating our scheduling methodology with traditional real-time scheduling — in particular, we wish to design a scheduler that is a minor variant of the classical *Earliest Deadline First* scheduling algorithm (EDF) [1].

In this paper, we will consider a system comprised of n servers $S_1, S_2, \ldots, S_n$, with each server S_i characterized by the parameters U_i and P_i as described above. Furthermore, we restrict our attention to systems where all of these servers execute on a single shared processor (without loss of generality, this processor is assumed to have unit processing capacity) — we therefore require that the sum of the processor shares of all the servers sum to no more than one; i.e.,

$$\left(\sum_{i=1}^{n} U_i\right) \leq 1 \ .$$

The following theorem formally states the performance guarantee that can be made by Algorithm GRUB *vis a vis* the behaviour of each server when executing on a dedicated processor. For a proof of this theorem, see [5].

Theorem 1 *Suppose that job J_i^j would begin execution at time-instant A_i^j, if all jobs of server S_i were executed on a dedicated processor of capacity U_i. In such a dedicated processor, J_i^j would complete at time instant $F_i^j \overset{def}{=} A_i^j + (e_i^j/U_i)$, where e_i^j denotes the execution requirement of J_i^j. If J_i^j completes execution by time-instant f_i^j when our global scheduler is used, then it is guaranteed that*

$$f_i^j \leq A_i^j + \left\lceil \frac{(e_i^j/U_i)}{P_i} \right\rceil \cdot P_i \ . \qquad (1)$$

For the previous inequality, it follows that $f_i^k < F_i^k + P_i$. This is what we intend when we say that P_i is the temporal granularity of the server: by using algorithm GRUB, every job finishes at most P_i time units later than the completion time on a dedicated slower processor.

Several other server-based global schedulers (e.g., CBS [2]), can offer performance guarantees somewhat similar to the one made by Algorithm GRUB. However, Algorithm GRUB has an added feature that is not to be found in many of the other schedulers — an ability to *reclaim* unused processor capacity ("bandwidth") that is not used because some of the servers may have no outstanding jobs awaiting execution.

Now, we describe the algorithm.

Algorithm Variables. For each server S_i in the system, Algorithm GRUB maintains two variables: a *deadline* D_i and a *virtual time* V_i.

- Intuitively, the value of D_i at each instant is a measure of the *priority* that Algorithm GRUB accords server S_i at that instant — Algorithm GRUB will essentially be performing earliest deadline first (EDF) scheduling based upon these D_i values.

- The value of V_i at any time is a measure of how much of server S_i's "reserved" service has been consumed by that time. Algorithm GRUB will attempt to update the value of V_i in such a manner that, *at each instant in time, server S_i has received the same amount of service that it would have received by time V_i if executing on a dedicated processor of capacity U_i.*

Algorithm GRUB is responsible for updating the values of these variables, and will make use of these variables in order to determine which job to execute at each instant in time.

At any instant in time during run-time, each server S_i is in one of three states: Inactive, Active Contending, or Active Non Contending. The initial state of each server is Inactive. Intuitively at time t_o a server is in the Active Contending state if it has some jobs awaiting execution at that time; in the Active Non Contending state if it has completed all jobs that arrived prior to t_o, but in doing so has "used up" its share of the processor until beyond t_o (i.e., its virtual time is greater than t_o); and in the Inactive state if it has no jobs awaiting execution at time t_o, and it has *not* used up its processor share beyond t_o.

At each instant in time, Algorithm GRUB chooses for execution some server that is in its Active Contending state (if there are no such servers, then the processor is idled). From among all the servers that are in their Active Contending state, Algorithm GRUB chooses for execution (the next job needing execution of) the server S_i, whose deadline parameter D_i is the smallest.

While (a job of) S_i is executing, its virtual time V_i increases (the exact rate of this increase will be specified later); while S_i is not executing V_i does not change. If at any time this virtual time becomes equal to the deadline ($V_i == D_i$), then the deadline parameter is incremented by P_i ($D_i \leftarrow D_i + P_i$). Notice that this may cause S_i to no longer be the earliest-deadline active server, in which case it may surrender control of the processor to an earlier-deadline server.

State Transitions. Certain (external and internal) events cause a server to change its state (see Figure 1).

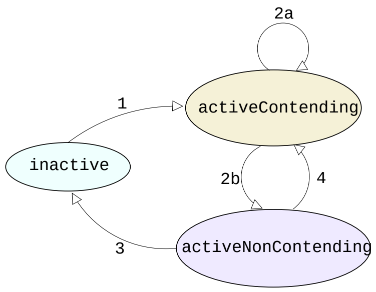

FIGURE 1: *State transition diagram.*

1. If server S_i is in the Inactive state and a job J_i^j arrives (at time-instant a_i^j), then the following code is executed

$$V_i \leftarrow a_i^j$$

$$D_i \leftarrow V_i + P_i$$

   and server S_i enters the Active Contending state.

2. When a job J_i^{j-1} of S_i completes (notice that S_i must then be in its Active Contending state), the action taken depends upon whether the next job J_i^j of S_i has already arrived.

   (a) If so, then the deadline parameter D_i is updated as follows:

$$D_i \longleftarrow V_i + P_i \ ;$$

   the server remains in the Active Contending state.

   (b) If there is no job of S_i awaiting execution, then server S_i changes state, and enters the Active Non Contending state.

3. For server S_i to be in the Active Non Contending state at any instant t, it is required that $V_i > t$. If this is not so, (either immediately upon transiting into this state, or because time has elapsed but V_i does not change for servers in the Active Non Contending state), then the server enters the Inactive state.

4. If a new job J_i^j arrives while server S_i is in the Active Non Contending state, then the deadline parameter D_i is updated as follows:

$$D_i \longleftarrow V_i + P_i \ ,$$

   and server S_i returns to the Active Contending state.

5. There is one additional possible state change — if the processor is ever idle, then *all* servers in the system return to their Inactive state.

Algorithm GRUB mantains a global variable *total system utilisation* that, at every instant, is equal to

$$U = \sum_{i=1, S_i \neq Inactive}^{n} U_i$$

where n is the number of servers in the system.
This variable is initialised to 0 and it is updated every time a server enters in or exits from state Inactive. In particular, when S_i exits from state Inactive U is increased of U_i, whereas when S_i enters state Inactive it is decreased of U_i.
The rule for updating the virtual time of every server is as follows:

$$\frac{d}{dt} V_i = \begin{cases} \frac{U}{U_i} & \text{if } S_i \text{ is executing} \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

Let us make an example to understand the way the algorithm works. Consider a server S_1 with bandwidth $U_1 = 0.25$ and period $P_1 = 20$msec that serves a MPEG player that needs to visualise 25 frames per second. If the system is fully utilised (i.e. the total system bandwidth U is equal to 1), then Equation 2 tells us that the virtual time is increased at a rate of $1/0.25 = 4$. By looking at the algorithm rules, we see that the server executes approximately $P_1/4 = 5$msec every period P_1.
In general, the bandwidth U_1 can be computed using some rule of thumb, or by performing a careful analysis of the application code. For our purpouses, in this example we assume that in the worst case 5msec are enough to visualise a frame in most cases.
However, suppose that at some point the total system utilisation U is equal to 0.75. Then, server S_1 can execute more than 5msec every period, because we can reclaim the spare bandwidth. According to Equation 2, the virtual time is increased at a rate of $0.75/0.25 = 3$. This means that our server will be able to execute for $P_1/3 = 6.66$msec every period.
Thus, if our application sometime requires more than 5msec to display a frame, it can take advantage of the reclaimed bandwidth and still execute inside the period boundary. This property can help us in setting the server bandwidth U_1 to a lower value. For example we can decide to set U_1 equal to the average bandwidth required by the application. Algorithm GRUB ensures that our application will take advantage of the spare bandwidth and execute more than $U_1 P_1$ in most cases. This property of GRUB is called "reclamation", because we are giving the spare bandwidth to the needing servers.

3.1 Power-aware scheduling

We can use GRUB to decide when the frequency of the processor can be changed. As first step, let us assume that the processor speed can be varied continuosly, from a maximum speed factor of 1 (i.e. the processor works at its maximum speed) to a minimum of 0 (i.e. processor is halted). As explained previously, GRUB maintains a global variable U that is the sum of the bandwidths of all servers that are not in the Inactive state. The key idea is that, if we set the speed factor of the processor to be equal to U, no server will miss its deadline.

In practice, if the processor is not fully utilised ($U < 1$) the exceeding bandwidth ($1 - U$) can be used in two ways:

1. To execute the active servers for a longer time, so that they can execute faster and finish earlier. This is the "reclamation" property, and it is the original goal the GRUB algorithm was designed for.

2. To slow down the processor. Each active server will still execute for a longer time, but they will execute at a slower speed. The net effect is that their performance is not degraded.

Let us make an example to understand how it works. Suppose that we have a server S_1 with bandwidth $U_1 = 0.2$ and period $P_1 = 10$msec. If $U = 1$, it means that the system is fully utilised, i.e. there are many servers in the system that are not in the Inactive state, and the sum of their bandwidth is equal to 1. Under these conditions, our server S_1 will be allowed an execution of 2msec every period $P_1 = 10$msec. If the system utilisation U goes down to 0.5 (for example because some server has no job to execute and is in the Inactive state), then our server S_1 is allowed to execute for $(U_1/U)P_1 = 4$msec units of time. If we slow down the processor speed to a factor of 50%, the server still executes the same amount of code as in the first case, because it executes twice as much (4msec instead of 2msec) but at half the speed. Therefore, the performance of the tasks served by server S_1 does not change.

Of course, no existing processor can vary its speed with continuity. So, we set some "thresholds" on the values of the total system bandwidth. If the system bandwidth goes below a certain threshold we can lower the processor frequency, and hence the processor speed. The implementation details of the algorithm are explained in the following sections.

4 Implementation details

4.1 Generic Scheduling

Since we want to limit as much as it is possible the modifications to the standard Linux scheduler, we decided to apply a small patch (called *Generic Scheduler Patch*) that exports the necessary kernel symbols. Then, we implemented our scheduler as a loadable kernel module.
Our scheduler needs to "intercept" the job arrival (i.e. tasks that are unblocked) and the job finishing (i.e. tasks that are blocked). Moreover, the scheduler must know when tasks are created and when tasks terminate.
Therefore, we decided to export an interface to the scheduler through the standard `sched_setscheduler()` system call, adding a new scheduling policy, called `SCHED_CBS`, and extending the structure `sched_param`.
For this reason, the Generic Scheduler Patch exports the following *hooks* that can be used to intercept the interesting scheduling events:

block_hook is invoked when a task is blocked, such that the scheduler understands that the current job has finished.

unblock_hook is invoked when a task is unblocked, such that the scheduler is informed of the arrival of a new job.

fork_hook is invoked when a new task is created by a fork() and a pointer to the task is passed as parameter.

cleanup_hook is invoked when a task is terminated, such that the scheduler can free the internal resources.

setsched_hook is invoked when the system calls `sched_setscheduler()` or `sched_setparam()` are called by the user.

All the hooks, except `setschedsched_hook` have a parameter that is a pointer to the structure `task_struct` of the corresponding task.
The patch inserts a new field called `private_data` in the `task_struct`, of type `void *`. It is a pointer used by our scheduler to access the private real-time data of every task. In our case, it is a pointer to the server that handles the task. If necessary, the scheduler must set this field to the appropriate data structure during the `fork_hook`. When the module is removed, it must ensure that all tasks have their `private_data` set to NULL.
Our dynamically loadable scheduler modifies the task priority, raising the selected task to the maximum priority, and then calls the standard Linux

scheduler. Based on the information received by the hooks, our scheduler selects which task has to be executed and sets its policy to `SCHED_FIFO` or `SCHED_RR` and the `rt_priority` to the maximum real-time priority + 1. Then, it invokes the Linux scheduler.

In practice, the Linux scheduler acts as a dispatcher for our scheduler. Thus, the modification to the standard Linux scheduler are minimal. In particular, our implementation will take advantage of the more efficient scheduler that is being provided by Ingo Molnar for the next version of Linux.

Note that in this implementation, the scheduling algorithm does not assume any periodic behaviour of the task. As matter of fact, the scheduler only intercepts the blocking/unblocking events of a task, and it is the task's responsibility to implement a periodic behaviour, if required. Thus, our scheduler is able to serve any kind of task, from non-periodic legacy Linux processes to periodic soft real-time tasks.

4.2 CPU clock frequency assignment

The hardware we used in our experiments is a Intrinsyc CerfCube 250, consisting of 32 MB Flash ROM, 64 MB SDRAM, and a Ethernet 10/100 Mbps. The processor is an Intel PXA250. It is a superpipelined 32 bits RISC processor based on the Intel Xscale micro-architecture. This architecture permits a on-the-fly switch of the clock frequency and a sophisticated power consumption management.

In particular, the processor can be in one of the following states:

1. Turbo Mode: the processor core works at the peak frequency.
2. Run Mode: the processor core works at its "normal" frequency. In this mode, it is assumed that the processor frequently accesses external memory, so it is convenient for it to work at a frequency lower than the Turbo Mode frequency.

The register in which it is possible to select the clock frequency is called CCCR (*Core Clock Configuration Register*) and can be found at the address `0x41300000`. The CCCR register manages the core clock frequency which the memory controller clock, the LCD clock and the DMA clock depend upon. In this register, the following parameters are specified:

- Frequency multiplier from the quartz frequency to the memory controller (L)
- Frequency multiplier from the memory frequency to the CPU frequency in Run Mode (M)
- Frequency multiplier from the CPU frequency in Run Mode to the CPU frequency in Turbo Mode (N)

The value of L is chosen depending on the constraints of the external memory and of the LCD and it is usually constant, while the values of M and N can change in order to change the speed of the processor. Value M is chosen based on the bus speed constraints and on the minimal performance requirements. Value N is based on the values of peak performance.

To modify the system clock frequency, register CCLKCFG can be used, that is register number 6 of the co-processor 14 (which is dedicated to power management at the lower level). It is a 32 bit register that is used to enter the Turbo Mode and the Frequency Change Sequence. Register CCLKCFG can only be modified through the following assembler instructions.

- To read the value of the register and put it in `R0`, you must use

```
MCR p14, 0, R0, c6, c0, 0
```

- To copy the content of register `R0` into register CCLKCFG, you must use

```
MRC p14, 0, R0, c6, c0, 0
```

To ensure that the Turbo bit does not change when we enter the Frequency Change Sequence, we must perform a read-modify-write sequence of assembler instructions.

As you can see, there is a great deal of flexibility in setting the clock frequencies. We had to choose how to implement our algorithm, which frequency to use as base frequencies, and so on. We decided to use only 3 levels for the processor clock frequency (100 MHz, 200 MHz, 400 MHz), whereas the memory frequency does never change. By using these 3 levels, we are able to use the minimum possible frequency (100 Mhz) and the maximum one (400 Mhz). Therefore, we have two thresholds, $U_{th1} = 1/4$ and $U_{th2} = 1/2$.

One detail that must be taken into careful consideration is the overhead of changing frequency. Changing frequency is not "for free", as the processor take some time (order of tens to hundreds of micro-seconds) to adjust to the new frequency. Althought this is not so high, it cannot be ignored. In particular we want to avoid limit situations in which the processor keep changing its frequency up and down, because this would completely trash the system.

When an increase of the total system bandwidth U goes over one of the thresholds, we immediately increase the processor frequency. Indeed, even if it could be a short transitory peak we cannot be sure and we do not want to risk a degradation of the system performances. When a decrease of the total system bandwidth U goes below one of the threshold, instead, we do not change the frequency immediately. In fact, in case of a short temporary decrease of the bandwith, we could end up changing the frequency very often up and down. Therefore, when U goes below a threshold, we set a timer. If the timer expires and U is still below the threshold we lower the frequency.

Now we compute the maximum overhead of the frequency switch. Let δ be the maximum time it takes to switch frequency and let Δ be the timer expiration interval. We can have a maximum of 2 frequency switches every Δ, one to go down and another one to go up. Therefore, in the worst case this accounts for a bandwidth reduction of $\frac{2\delta}{\Delta}$.

In our implementation, the timeout duration is a customisable parameter of the algorithm, called PWR_TIMEOUT which specifies the timer's duration in seconds.

Every threshold level is described by the following struct:

```
struct pwr_level {
   int bandwidth;
   int run_mode;
   int turbo_mode;
   int selected;
   struct t_data_struct pwr_timer;
};
```

The first parameter (bandwidth) is the maximum bandwidth that this level can support (the bandwidth is expressed as an integer because we use a fixed point representation). The second and third parameter are, respectively, the Run Mode multiplier (M) and the Turbo Mode multiplier (N). Variable selected is used to handle the timer. When the total system utilisation goes below bandwidth, we set the timer pwr_timer and the variable selected. If U goes above the threshold bandwidth before the timer expiration, selected is set to false, and the timer is canceled.

A global variable

```
struct pwr_level* current_pwr_level;
```

points to the current power level.

5 Experimental results

One may argue that varying the processor frequency only, without touching the peripherals frequencies (like memory, for example) does not bring appreciable advantages. We will show that this is not the case.

Our study is particularly focused on multimedia applications. Therefore, we decided to evaluate the performance of our system using a multimedia application. However, our approach can be used for a large range of different applications, because it is completely transparent to the application characteristics.

Unfortunately, our system, the Intrinsyc CerfCube, does not present a video output. So, we decided to focus our attention to a audio decoder. We selected the decoder provided by the *Xiph.Org Foundation* that is a "non-profit organisation dedicated to protecting the foundations of Internet multimedia from control by private interests". *Ogg Vorbis* is a non-proprietary compressed audio format that provides high quality (from 8 to 48 bits, polyphonic) with fixed or variable bitrate that ranges from 16 to 128 Kbps per channel. It is in direct competition with MPEG-4 et similia.

To execute the first test, we decompressed some audio stream at 44100 Hz and two channels, measuring the time necessary to decompress every stream under the different fixed clock frequencies.

From the obtained values, we extracted how much the speed of decompression is related to the speed of the CPU. The result is shown in Figure 2, where we show on the x-axis the speed of the processor, and on the y-axis the decompression speed. As you can see the relationship is almost linear. This justifies our assumption that by doubling the processor speed, the computation time of one task's job halves. In the Figure we also show the 99% confidence interval.

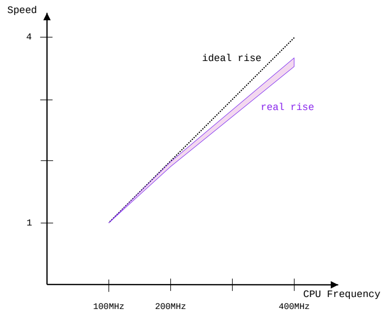

FIGURE 2: *Decompression speed related to CPU speed (99% confidence interval).*

Then, we evaluated the power consumed by our system under different conditions, with or without our algorithm. We measured the current entering into the CerfCube with a circuit powered separately by a 9 V battery that measures the current and sends it to a host computer via serial communication. The circuit puts a very small resistor in series with the CerfCube and measures the voltage at the ends of the resistor.

By using our algorithm, we measured the temporal evolution of the current under different loads. In Figures 3, 4, 5, we show the temporal evolution of the input current when the total load is $U = 0.25$, $U = 0.5$ and $U = 1$ respectively.

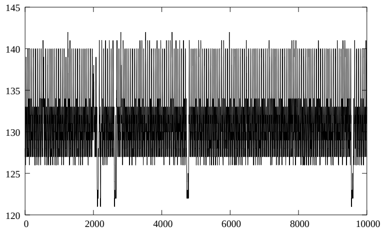

FIGURE 3: *Temporal evolution of the current with system bandwidth U = 0.25*

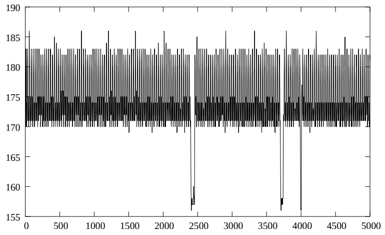

FIGURE 4: *Temporal evolution of the current with system bandwidth U = 0.5*

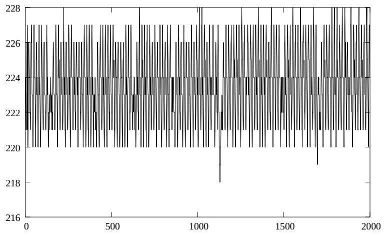

FIGURE 5: *Temporal evolution of the current with system bandwidth U = 1*

We computed the average values of the input current, reported in the table 1.

CPU clock frequency	Current
100 MHz	446.0 mA
200 MHz	508.5 mA
400 MHz	579.9 mA

TABLE 1: *Average values of the input current.*

In Figure 6 we also report what happens with our algorithm when the total bandwidth goes from 0.1 to 1. As we said previouosly, the processor frequency is changed from 100 Mhz to 400 Mhz. The temporal evolution of the current is reported in Figure 6.

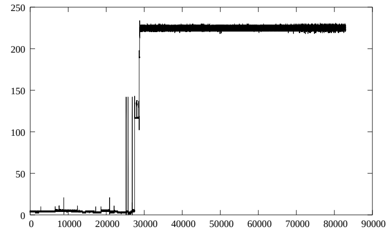

FIGURE 6: *Temporal evolution when the total bandwidth goes from 0.1 to 1*

Our algorithm brings an advantage in two distinct situations.

Idle System. When the system is idle, we can lower the frequency to 100 Mhz. Without the frequency scaling mechanism, it is necessary to mantain the frequency to 400 Mhz all the time because we must guarantee maximum performance under peak load. The input current when the system is idle is shown in Figure 7. The average value of the input current is 250.5 mA.

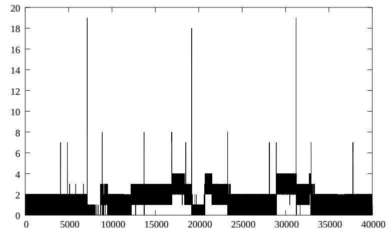

FIGURE 7: *An idle system with voltage scaling*

If the voltage scaling algorithm is not activated and the system is idle, the input current is shown in Figure 8. The average value of the input current is 406.8 mA. We can say that we save up to 38.4% of power in the average when the system is idle.

Non idle systems. When the system is not idle and the required total bandwidth is less than 1, it is possible to find a clock frequency less than 400 Mhz that respects the performance of the applications. We run an application that decodes an audio stream using a bandwidth of 0.15. By using our voltage scaling algorithm we measured an average current of 343.6 mA, whereas without voltage scaling, the average current was 433.3 mA. Therefore, we saved up to 20.7%.

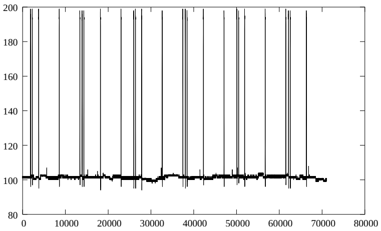

FIGURE 8: *An idle system without voltage scaling*

The most general case is when we have many applications that activate and deactivate themselves many times. In that case, the load of the system is highly variable, and we can take fully advantage of the voltage scaling algorithm.
We executed the audio decoder many times on different audio streams. Every instance is assigned a different value of the bandwidth, so that the total bandwidth of the system goes up and down, as shown in Figure 9. We expect a similar behaviour in the temporal evolution of the input current.

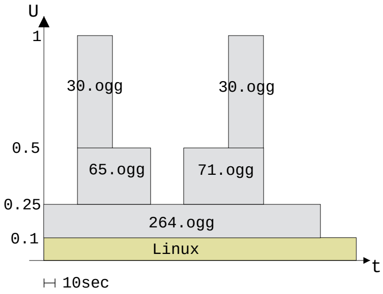

FIGURE 9: *Variation of the bandwidth during the test.*

In Figure 10 we show the input current measured in the presence of the voltage scaling algorithm. You can verify that the temporal evolution is similar to the one of Figure 9. The average input current is 413.3 mA.

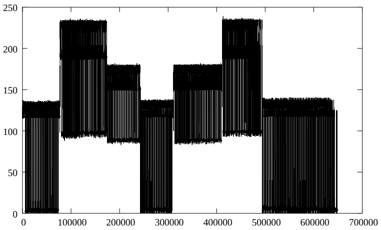

FIGURE 10: *Input current with voltage scaling*

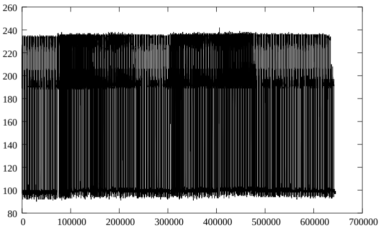

FIGURE 11: *Input current without voltage scaling*

The input current measured with no voltage scaling is shown in Figure 11. The average input current is 461.2 mA. Therefore, we saved up to 10.4 %.

6 Conclusions

We modified the Linux OS in order to guarantee soft real-time tasks and at the same time reduce the power consumption. The resulting system is highly modular because is based on the Loadable Kernel Module feature of Linux. It currently supports Intel PXA250 based architecture. We believe that it could be easily portable on different hardware architectures. The code is distributed under the GPL and it can be dowloaded from *http://www.ocera.org.*

References

[1] C.L. Liu, and J. W. Layland, January 1973, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of ACM, Vol. 20, No. 1, pp. 46-61.

[2] L.Abeni and G.Buttazzo, December 1998, *Integrating Multimedia Applications in Hard Real-Time Systems*, Proceedings of the IEEE Real-Time Systems Symposium, Madrid, Spain, pp. 4-13.

[3] Luca Abeni and Giuseppe Lipari, December 2002, *Implementing Resource Reservations in Linux*, Real-Time Linux Workshop, Boston (MA).

[4] Intel corporation, February 2002, *Intel PXA250 and PXA210 Application Processors Developer's Manual*, order number 278522-001.

[5] Giuseppe Lipari and Sanjoy Baruah, June 2000, *Greedy reclamation of unused bandwidth in constant-bandwidth servers*, Proceedings of the EuroMicro Conference on Real-Time Systems, Stockholm, Sweden, pp. 193-200.

[6] Giuseppe Lipari and Sanjoy Baruah, May 2001, *A hierarchical extension to the constant bandwidth server framework*, PROCEEDINGS OF THE REAL-TIME TECHNOLOGY AND APPLICATIONS SYMPOSIUM, TAIPEI, TAIWAN.

[7] H.Aydin, R.Melhem, D.Moss, P.Mejia-Alvarez, December 2001, *Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems*, PROCEEDINGS OF REAL-TIME SYSTEM SYMPOSIUM, LONDON, UK, pp.95-105.

[8] Ismael Ripoll, Pavel Pisa, Luca Abeni, Paolo Gai, Agnes Lanusse, Sergio Saez, and Bruno Privat, 2002, *WP1 - RTOS State of the Art Analysis: Deliverable D1.1 - RTOS Analysis*, OCERA (http://www.ocera.org).

[9] R. RajKumar, K. Juvva, A. Molano, and S. Oikawa, January 1998, *Resource kernels: A resource-centric approach to real-time systems*, PROCEEDINGS OF THE SPIE/ACM CONFERENCE ON MULTIMEDIA COMPUTING AND NETWORKING.

[10] Alessandro Rubini and Jonathan Corbet, *Linux Device Drivers 2nd edition*, O'REILLY.