EDF-DVS Scheduling on the IBM Embedded PowerPC 405LP

Aravindh V. Anantaraman, Ali El-Haj Mahmoud, Ravi K. Venkatesan, Yifan Zhu and Frank Mueller
North Carolina State University, Center for Embedded Systems Research
Departments of Computer Science / Electrical and Computer Engineering
phone: +1 (919) 515-7889, e-mail:mueller@cs.ncsu.edu

Abstract

Power consumption has become a major concern, both for processor design with high clock rates and embedded systems that rely on batteries to operate. Recent support for dynamic frequency and voltage scaling (DVS) in contemporary architectures allows software to affect power consumption by varying both execution frequency and supply voltage on the fly. However, processors generally enter a sleep state while transitioning between frequencies/voltages. In the following, we describe an experimental framework for studying DVS for a processor that continues to execute during frequency/voltage transitions. For this purpose, we developed an infrastructure for investigating hard real-time DVS schemes on the IBM PowerPC 405LP. Task scheduling was performed using four earliest-deadline-first (EDF) DVS schemes, including our feedback real-time DVS algorithm that, prior to this work, had only been evaluated in simulation. Voltage and current of the processor core were depicted through an oscilloscope, and the energy consumption was assessed through a data acquisition board. Measurements indicate a considerable potential for real-time DVS scheduling algorithms to lower energy consumption up to 54% over naïve DVS schemes. The benefits of continued execution during frequency/voltage switching provide up to 5% energy savings for frequent switches.

1. Introduction

Energy management has become a vital design constraint in embedded systems for a long time. The demand for efficient energy management is increasing in hand-held devices, where battery life is important. The battery life limits the energy budget in these devices and imposes severe restrictions on energy consumption. This means that energy savings becomes a necessity rather than convenience. Dynamic Voltage Scaling (DVS) is a widely used energy management technique for extending battery life. DVS dynamically scales processor core voltage up or down, depending on the computation demand of the system. Reducing the supply voltage results in a lower maximum transistor switching speed, and this also allows lowering the clock frequency of the device. Assuming that voltage and frequency are linearly related, scaling down both voltage and frequency results in cubic reduction of power consumption ($P \propto V^2 \times f$).

DVS algorithms have been intensively studied for both non real-time and real-time systems [15, 2, 11, 7, 8, 14, 18]. In the case of real-time systems, the DVS algorithm calculates a safe operation frequency that provides just enough processor computation power to finish a given task before its deadline. The goal is to save the maximum possible amount of energy and yet maintain safe operation of the hard real-time system where all tasks are guaranteed to meet their deadlines.

In this work, we implemented a DVS infrastructure for a real-time system using IBM's PowerPC 405LP. The 405LP processor provides the hardware support required for DVS and allows software to scale voltage and frequency via user-defined operation points ranging from a high end of 266 MHz at 1.8V to a low end of 33 MHz at 1V [13, 3, 10]. The IBM PowerPC 405LP was especially attractive for DVS since it has the ability to execute instructions even when the frequency/voltage is being changed, much in contrast to any other processors with DVS support that we know of where the processor has to enter the sleep mode during frequency/voltage transitions. We implemented a real-time earliest deadline first (EDF) scheduling policy as part of a user-level threads package under the supported Linux operating system. Then, we extended the capabilities of the infrastructure to support four hard real-time software DVS techniques (static, cycle-conserving, look-ahead and feedback) that leverage the already available hardware DVS support in the PowerPC processor. We used an oscilloscope and an analog data acquisition board to measure the voltage and current supplied to the processor core. Custom changes to the development board allow us to separately measure the voltage and current of the processor, memory and I/O components. From voltage and current, we calculate the power consumption of the system during application runs for each of the four DVS algorithms that we implemented.

2. Embedded Platform and DVS Support

The PowerPC 495LP runs on a diskless MontaVista Embedded Linux variant, which is based on the 2.4.21 stock kernel but has been patched to support DVS on the PPC 405LP. The board has also been modified for 50% reduced capacitance, which allows DVS switches to occur more rapidly, i.e., switches are bounded by at most a 200 microseconds duration from 1V to 1.8V during which execution continues. Switches may occur in a synchronous (blocking) manner, as traditionally supported by DVS-capable processors. Alternatively, switches may be asynchronous (non-blocking) such that execution may proceed during the switch. Figure 1 depicts the changes in current (lower curve) and voltage (upper curve) of the processor core during an asynchronous switch. This unique

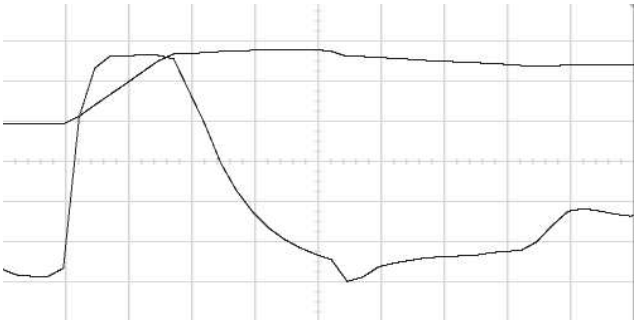


Figure 1. Current and Voltage Transition During DVS

feature of asynchronous switching is realized by a system call that, when switching to a higher voltage/frequency, first reprograms the voltage to ramp up towards the maximum as fast as possible (30 degree voltage ramp on the upper curve). As a side effect, this ramp-up results in an instant current surge (steep ramp of the lower curve). In addition to initiating the ramp-up, the time to reach a voltage level at least as high as required by the new frequency is estimated. A high-resolution timer is programmed to interrupt when this duration expires, prior to which execution within the application can still continue after returning from the system call (executing instructions during the 30 degree ramp-up). Once the timer interrupt triggers its handler (at the end of the 30 degree ramp on the upper curve), the power management unit is reprogrammed to settle at the target voltage level, and the new processor frequency is activated before returning from the handler. The voltage then settles (in case it overshot) in a controlled manner to the new operating point. The current also settles in a controlled manner depending on the actual processing activity.

This unique DVS facility is supported by a dynamic power management (DPM) facility, which is provided as an enhancement to the Linux kernel [3]. DPM *operating points* define stable frequency/voltage pairs (as well as related system parameters), which we experimentally determined. *Operating states* describe system states, such as idle, task activity and sleep. Each operating state can be associated with an operating point via a DPM *policy* that defines a set of operating points, one for each operating state. In such a manner, operating points can be selected depending on activities chosen by the kernel. To allow applications to also select operating points, applications can define a DPM *task state* through a system call. Each of these task states can then be associated with an operating point within a given policy. Overall, the DPM enhancements allow both kernel-level and user-level DVS in a flexible manner. However, the implementation details are more subtle. Changing a policy results in synchronous DVS switching where execution is blocked inside the Linux kernel till a safe, new operating point is established. In contrast, changing a task state results in asynchronous DVS switching with continuing execution between the system call and the following timer interrupt.

Our aim is to assess this advanced technology for the suitability and benefits in power consumption for real-time systems using DVS scheduling algorithms. Unfortunately, stock Linux does not support any real-time schedulers. Hence our first step was to implement an EDF scheduler on the 405LP under Linux. The DVS scheduling is independent of task scheduling, *i.e.*, the scheduler can be envisioned to be comprised of (1) EDF scheduling and (2) DVS scheduling. These two components are completely independent. This means that, even though we plan to identify and implement a specific DVS algorithm, our scheduler will work with any existing DVS algorithm. We implemented an EDF and DVS algorithm on the user level and analyzed the power benefits. We chose a user-level solution over a kernel-based one due to the simplicity of design and the fact that OS background activity is minimal on this embedded board, which ensures the validity of our results. To expose the processor to a real-time task load, we developed a framework composed of tasks with synthetic CPU board activity whose execution time is predictable and controllable. We then exposed a variety of task sets to the EDF-DVS scheduler for different scheduling policies within this framework.

3. A Light-Weight Thread Library

The first step in implementing a DVS system is providing support for pre-emptive threads. Since we are dealing with a hard real-time system, these threads should be scheduled according to some real-time scheduling algorithm (EDF in

our case). This raises a number of interesting issues and design choices, which we will discuss briefly below.

Light-weight processes can be implemented under Unix as either kernel-space or user-space threads [16, 12, 1]. In the case of kernel-space, scheduling and dispatching of the threads (including context switching) is performed as part of the kernel and is transparent to the application. On the contrary, user-space threads are scheduled and dispatched as part of an application library transparently to the kernel. Each approach has its pros and cons. Kernel-space threads require complex kernel modifications, which are often error-prone. However, kernel modifications provide more control over the course of execution (especially with real-time considerations). User-space threads are simpler to implement as a part of an application library but they provide less control over execution and resources since the kernel is not aware of their existence. Another important issue is portability. Although we are targeting a very specific system (the PowerPC development board running a specific version of Linux), it would be much easier to develop a generic and portable threads library under any Unix system, which provides better debugging tools, eventually reducing development time. Hence, we opted for user-level threads instead of kernel-level threads.

After deciding on implementing a user-level threads library, the next question is how to actually approach the implementation. Again, we are concerned here about portability mainly to facilitate ease of implementation and debugging. The literature reveals that the most portable technique to implement user-space threads is via using the standard C functions `setjmp()` and `longjmp()` [4]. `Setjmp()` stores the current machine context (program counter, stack pointer, and registers) in a `jmp_buf` structure, and `longjmp()` restores the context saved in a `jmp_buf` structure. Although these functions are available for any system supporting the standard ANSI C, unfortunately, the details of the `jmp_buf` structure implementation is machine dependent. From a programmer's point of, this should not be a problem provided that the saved and restored contexts are of the same structure on a given machine. We have to treat the `jmp_buf` structure as an opaque structure.

Each thread must execute using its own private stack to enable pre-emptive scheduling. The UNIX `sigaltstack()` system call forces a signal handler function to run on a specified alternative stack [6]. By generating a new stack for each spawned thread and using a signal handler running on the alternative stack to create the thread's context using `setjmp()`, we ensure that every thread will have its private execution context, including its private stack. This eventually allows us to safely implement pre-emptive scheduling.

The threads library provides the following API:

- `initThreads(void)`: Initializes the data structures used by the threads library. Must be called from the application before using any other threads functions.
- `spawnThread (void(*func)(void), unsigned int WCET, unsigned int period, unsigned int deadline, unsigned int phase)`: Creates a new thread running the function provided as the first argument. The characteristic of the task (WCET, period, deadline, and phase) are used by the real-time scheduler to release new instances of the task, take scheduling decisions, and confirm all deadlines are met. The `spawnThread()` function only creates a new thread and saves its context (including the private stack), but it does not start its execution. We use `SIGUSR1` signal to run the function `CreateNewContext()` as its signal handler and set it to run using an alternative stack (with the `sigaltstack()` call). At this point, saving the machine context for this thread using `setjmp()` will also save this private newly created stack info.
- `threadYield()`: Must be called by the task to indicate to the scheduler that the current job instant of the task has finished execution. The context of this task is saved at this point, and this will be the starting context of the next released instance of this job. This behavior is intentional (and correct) for periodic tasks since their repetitive behavior is modeled as infinite while loops, where each loop iteration corresponds to one instance of the task.
- `Start()`: Called from `main()` after spawning all threads. Time starts (t = 0) when the `Start()` function is called, and threads are activated, scheduled, and dispatched according to the provided scheduling algorithm.

4. Real-Time Scheduler Integration

The scheduler we implemented is the standard earliest-deadline-first (EDF) scheduler. The EDF scheduling policy is one in which task priorities are dynamically assigned such that the task with the earliest deadline has the highest priority. The EDF scheduler is invoked on two occasions: (1) when a task completes, and (2) a task is released. When a task completes, the EDF scheduler activates the next active task with the earliest deadline. If there are no active tasks, then the processor transitions into an idle state. At the moment, the idle state is modeled as an idle task that is an infinite loop. If the scheduler is invoked due to a task release, the scheduler decides if the released task has to run next or if the previous (interrupted) task has to be resumed. When

the scheduler is invoked, it decides the next task to be executed and also sets a timer interrupt to be triggered at the next release time of any task(s) in the task set. When the timer interrupt is triggered, the released task(s) is/are activated and the scheduler then assigns priorities for the tasks.

Example

The following example is depicted in Figure 2 below. It shows the interaction between the threads library and the preemptive scheduling algorithm. The steps are indicated by numbers on the Figure. We assume that there are two threads in the system running the functions T1 and T2, respectively. The `main()` function is called and initializes the threads library by calling `initThreads()`.

1. The first thread is spawned by calling the function `spawnThread()` and
   - Space is allocated for a new alternative stack;
   - the `SIGUSR1` signal handler (`CreateNewContext()` function) is configured to use the alternative stack (using the `sigaltstack()` function). The old stack value is saved;
   - the `SIGUSR1` signal is raised, which will:
2. Call the `CreateNewContext()` function running on the newly created stack (`T1_stack`). The context of the machine at this point is saved in the structure `T1_jmp_buf` by calling the `setjmp()` function.
3. After returning from `CreateNewStack()`, the signal handling stack is set back to the original stack.
4. Control is returned back to `main()`.
5. Same as (2) above, but a new stack is created for T2.
6. Same as (3) above, but T2 context is saved in `T2_jmp_buf`.
7. Same as (4) above.
8. The `Start()` function is called. `Start()` will initialize the data structures used by the scheduler (mainly, timers), and call the `Scheduler()` function.

Let us assume that T2 has the highest priority and should run now. The saved context T2_jmp_buf is restored by calling a longjmp() to that context. This will transfer the control again to the CreateNewContext(), specifically to the setjmp(). The return value of setjmp() will indicate that it has been called by a longjmp(), and the actual function T2 is called (labeled B in Figure 2).

While T2 is running, a timer interrupt goes off (indicating the release of a new task). The timerhandler() function is called (labeled C on the figure), where the context of the interrupted task T2 is saved in the structure T2_jmp_buf, then the Scheduler() is called. Assume that the scheduler decides that T2 is still the highest priority job in the system, so it dispatches it again by calling a longjmp() to T2_jmp_buf. This will take us back to the saved context at the timerhandler() function at setjmp(). The return value of setjmp() will indicate it was called by a longjmp(), which means we need to resume the job from the point, where it was interrupted when the timerhandler() was first triggered by the alarm signal. This is achieved by just returning from timerhandler(), which will transfer control back to T2() at the interrupted point (labeled D).

The current instant of T2 continues execution without any further interruptions till it finishes. At this point, the thread calls the threadYield() function (labeled E). The execution context is saved (this will be the starting point of the new released instance of the task), and the scheduler is called.

C and E indicate the two situations at which the scheduler is invoked, namely task release and task completion.

Note that timerhandler() and threadYield() functions will run on the interrupted calling thread's stack, respectively.

5. Worst-Case Timing Analysis

We currently lack tool support to derive safe upper bounds of the worst-case execution time of tasks on the PPC405 processor, such as a static timing analysis tool would provide [9]. Hence, we resorted to the more conventional empirical analysis using dual-loop timing, as given in the following code snippet:

```
t1 = gettimeofday();
for( i = 0; i<N; i++);
t2 = gettimeofday();
work();
t3 = gettimeofday();
for(i=0; i<N; i++)
  work();
t4 = gettimeofday();
```

The timing loop computes the average execution time of N instances of the task taking into account the loop overhead and cold cache misses. The execution time is calculated as:

$$t_{work} = \frac{(t4 - t3) - (t2 - t1)}{N}.$$

6. Dynamic Voltage Scaling Algorithms

We implemented four dynamic voltage scaling algorithms: The static, cycle-conserving and look-ahead developed by Pillai and Shin [15] as well as the feedback scheme developed in-house by Zhu *et al.* [20, 19, 5]. All of these DVS schedulers interact with the EDF scheduler as follows. Once a scheduling decision is demanded, the EDF scheduler invokes a DVS scheduler that looks at the amount of work completed so far, the actual processor requirement, static slack (due to under-utilization) and dynamic slack due to early completion) available. The DVS scheduler then

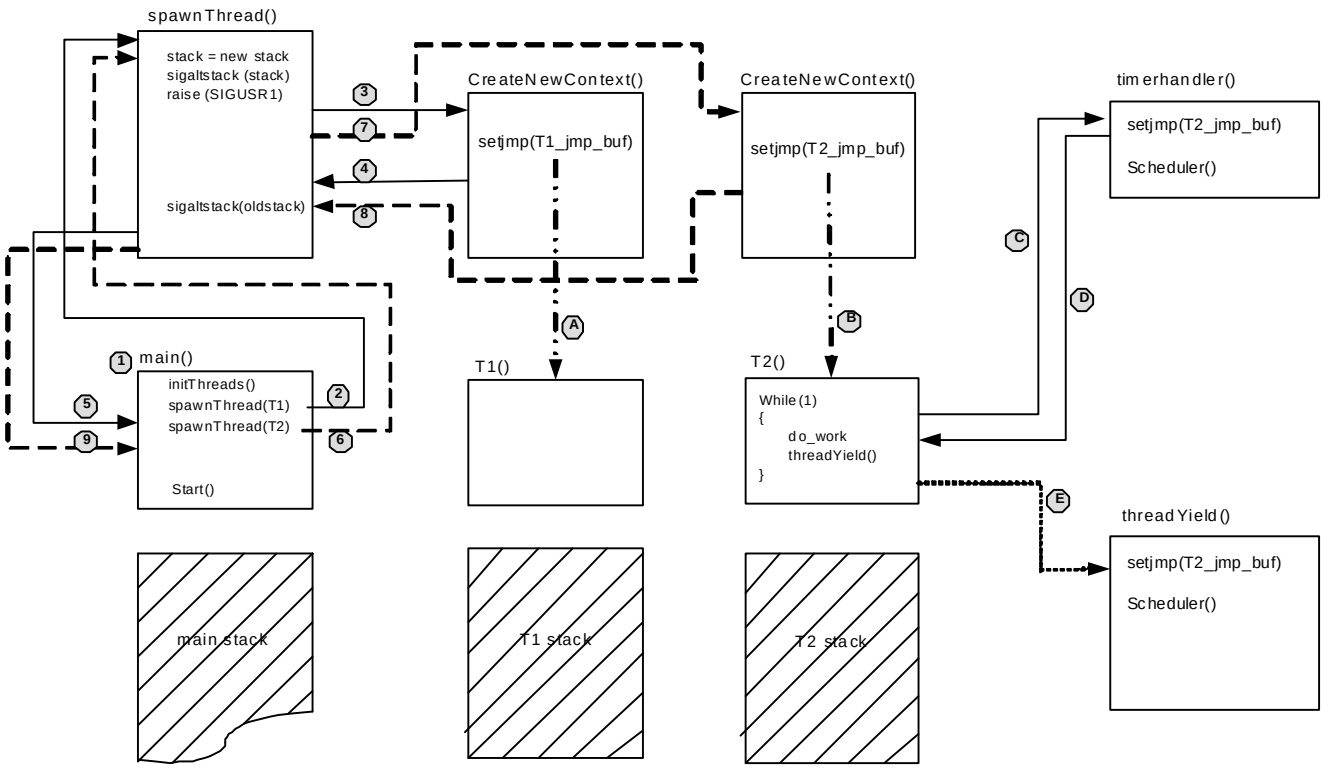


Figure 2. Interaction between the Threads Library and the Preemptive Scheduling Algorithm

computes a safe frequency and voltage that can meet the real-time requirements of the task set. The DVS scheduler also issues the appropriate primitives to reduce the frequency/voltage. We used system calls to the patched Linux PowerPC kernel that change the frequency as well as voltage. The user-level library that was responsible for thread creation and management also has primitives to dynamically change the frequency and voltage.

Next, the DVS scheduling algorithms are introduced in increasing level of computational overhead (for making a scheduling decision) and increasing potential for energy savings.

6.1. Static Voltage Scaling under EDF

In this algorithm, the lowest possible operating frequency is selected while still meeting deadlines in EDF scheduling for a given task set. The system utilization is computed based on the worst-case execution times. This provides the amount of static slack that is available, and this value is used as the scaling factor to scale the frequency accordingly. Hence, the frequency is set statically and is not changed unless the task set is changed. Consequently, the DVS scheduler is invoked when a task set is released for execution (see Figure 3).

The remaining schemes are dynamic scheduling policies: cycle-conserving and look-ahead as well as our feedback RT-DVS EDF.

Find the lowest frequency f_k with
$\alpha = \frac{f_k}{f_m} \wedge f_k \in \{f_1, ..., f_{max}\}$
which satisfies EDF: $\sum_{i=1..n} \frac{C_i}{P_i} \leq \alpha$

Figure 3. Static RT-DVS EDF

6.2. Cycle-Conserving RT-DVS EDF

In Cycle-Conserving RT-DVS (EDF), initially the task set is scheduled based on the worst-case execution times. When one task in the task set completes, the system utilization is recomputed using the actual execution times of the completed task and the worst-case execution times of the remaining tasks (see Figure 4). This is repeated when each task in the task set completes. Hence, the frequency is re-

Upon release of T_i, set $U_i = \frac{C_i}{P_i}$ and select_frequency().
Upon completion of T_i with actual execution time cc_i,
set $U_i = \frac{cc_i}{P_i}$ and select_frequency().
select_frequency: find lowest f_k s.t. $\sum_{i=1..n} U_i \leq \frac{f_k}{f_{max}}$.

Figure 4. Cycle-Conserving RT-DVS EDF

duced by computing the amount of dynamic slack due to early task completion in addition to static slack. Thus, the DVS scheduler is invoked upon task releases and completions.

6.3. Look-Ahead RT-DVS EDF

This is the most aggressive of the three DVS algorithms by Pillai and Shin and exploits both static and dynamic slack. The look-ahead scheme determines future computation needs and defers task execution. The cycle-conserving scheme discussed above assumes the maximum frequencies initially until tasks complete and then reduces the operating frequency and voltage.

In contrast, the look-ahead scheme tries to defer as much work as possible in a greedy manner. It sets the operating frequency to meet the minimum work that must be completed now to ensure all future deadlines are met. In theory, the look-ahead scheme may necessitate increased frequencies at a later stage in order to complete deferred work in time. But actual execution times are usually much smaller than worst-case execution times [17]. This means that, on the average, earlier lower frequencies typically do not result in later pressure forcing high frequencies in order to meet deadlines (see Figure 5).

Upon release of T_i,
  set remaining time $c\_left_i = C_i$ and defer().
Upon completion of T_i, set $c\_left_i = 0$ and defer().
During execution of T_i, decrement c_left_i.
defer execution:
  let $U = \sum_{i=1..n} \frac{C_i}{P_i}, s = 0$
  for (i = 0..n s.t. $T_i \in \{T_1, ..., T_n | D_1 \geq ... \geq D_n\}$)
    $U = U - \frac{C_i}{P_i}$
    $x = max(0, c\_left_i - (1 - U)(D_i - D_n))$
    $U = U + \frac{c\_left_i - x}{D_i - D_n}$
    $s = s + x$
  end for
  select_frequency($\frac{s}{D_n - current\_time}$)
select_frequency(α): find lowest f_k s.t. $\alpha \leq \frac{f_k}{f_{max}}$

Figure 5. Look-Ahead RT-DVS EDF

6.4. Feedback RT-DVS EDF

This algorithm exploits both dynamic and static slack in an even more greedy manner than the look-ahead algorithm discussed above. The algorithm anticipates an actual execution time of each task invocation similar to the execution time used in previous invocations. It then splits the execution budget of a task into two parts as depicted in Figure 6, the anticipated actual time C_A (scaled at the lowest possible frequency) and the remaining time C_B (scaled at maximum frequency). All future tasks are deferred as long as

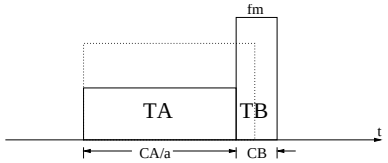


Figure 6. Task Splitting

possible using a maximal (worst-case) schedule, which is related to the actual schedule to derive the currently available slack s_k for task k. Thus,

$$\alpha = \frac{C_A}{C_A + s_k}$$

indicates the scaling factor and the corresponding lowest possible frequency using `select_frequency(`α`)` as in Figure 5. The algorithm is capable of capturing changes in actual execution times using a PID-feedback scheme. Preemptions of the current task have to be anticipated via future slot allocations in the schedule implemented in a backward sweep to fill idle and early completion slots from a task's deadline backwards (see Zhu and Mueller for algorithmic details [20]). Due to the even more greedy approach than any of the previous schemes, the algorithm is reported to exhibit additional energy savings, particularly for medium utilization systems, which are quite common [5]. Even more substantial savings have been observed for fluctuating execution times where PID-feedback provides new opportunities for aggressive scaling [20]. By evaluating this algorithm on the 405LP, we wanted to assess the true potential for energy savings in an actual system as opposed to a simulation environment. Also, we wanted to determine if the lower frequencies chosen by the feedback scheme outweighs the higher computational overhead required to make scheduling decisions.

7. Experimentation Methodology

The DVS algorithms (static, cycle-conserving, look-ahead and our feedback) were exposed to the DVS capabilities of the 405LP board. Specifically, the scheduling algorithms could chose any frequency/voltage pair from the set depicted in Table 1. This set of pairs was constrained by a need to have a common PLL multiplier of 16 relative to the 33MHz base clock and a divider of two or any multiple of 4. Changing the multiplier results in additional overhead for switching, which we wanted to eliminate in this study.

In order to assess power consumption, we needed to monitor processor core voltage and current at a high rate.

Setting	0	1	2	3	4
Frequency (MHz)	33	44	66	133	266
Voltage (Volts)	1.0	1.0	1.1	1.3	1.7

Table 1. Valid Frequency/Voltage Pairs

Due to the rate requirements, it was not feasible to employ multimeters for measuring the voltages since their precision is generally low and very coarse-grained. The error rates in voltage readings could also result in measuring incorrect voltages. Hence, we used a high-frequency analog data acquisition board to gather data for (a) the processor core voltage and (b) the processor current. The latter was measured as a voltage level over a resistor with a 1V drop per 360mA. Data acquisition allowed us to experiment with longer-running applications to assess the energy consumption of the processor. We also employed an oscilloscope for visualizing the voltages and currents, again with high precision in voltage and current readings. The snapshots in the oscilloscope shots depict the phase just after a simultaneous release of all tasks at the beginning of a hyperperiod.

8. Results

We first assessed the overhead of different DVS techniques supported by the test board and the dynamic power management extensions of the operating system. Table 2 reports the overhead for synchronous switching (by changing the DPM policy) in a time range bounded by two extremes: (a) Switching between adjacent frequency/voltage levels and (b) switching between the lowest and highest levels. Furthermore, the overhead for initiating an asynchronous system call (by changing the DPM task state) leading to a switch and the subsequent handler overhead are reported for a range of the highest and the lowest processor frequencies. The results indicate that a synchronous DVS switch delaying execution during voltage/frequency transition has about an order of a magnitude larger overhead than a system call triggering an asynchronous switch where execution proceeds during the switching interval. The system call is complemented by an interrupt that occurs when the voltage has ramped up sufficiently, and the new frequency is set in the interrupt handler. This handler overhead, comparable to a signal handler, increases the overhead of the asynchronous approach only insignificantly.

activity	sync. DVS	async. DVS	signal handling
overhead	117-162 μsec	8-20 μsec	0.07-0.6 μsec

Table 2. DVS overhead

In a second set of experiments, we measured the power of executing three medium utilization task sets depicted in Table 3. Task first task set is harmonic, *i.e.,* all periods are integer multiples of the smallest period, which facilitates scheduling. This often allows scheduling algorithms to exhibit an extreme behavior, typically outperforming any other choice of periods. The second and third task sets are non-harmonic with longer and shorter periods, respectively. Actual execution times were half that of the WCET for each task for this experiment.

Each of the previously discussed real-time DVS algorithm were subsequently exposed to execution under these task sets. As a baseline for comparison, we also support a naïve DVS scheme where task execution and idle alternate between maximum and minimum frequency/voltage, respectively.

We executed task sets one, two and three for five, one and seven hyperperiods for a total of twelve, ten and ten seconds, respectively. We measured the voltage and current of the processor during this time using an analog data acquisition board. This allowed us to integrate instant power consumption of the execution time. We also performed all experiments for synchronous and asynchronous DVS switching to assess the savings attributed to the latter.

Table 4 depicts the energy consumption in mWatt-hours of each experiment. The naïve DVS algorithm serves as a base for comparisons for each of the subsequent DVS algorithms. For task set one, static DVS reduces energy consumption by about 29% over the naïve scheme. Cycle-conserving saves 47% energy. Look-ahead RT-DVS saves over 50%, and our feedback method saves about 54% energy compared to naïve DVS. This clearly shows the tremendous potential in energy savings for real-time scheduling.

The savings for each algorithm are lower for task set two peaking at about 23% for our feedback scheme. As mentioned before, task set one is harmonic, which typically results in the best scheduling (and energy) results since execution is more predictable. Task set three lies in between the other two with peak savings of 37% for our feedback scheme.

The results also demonstrate that the overhead for calculations inherent to scheduling algorithms is outweighed by the potential for energy savings. This is underlined by the increasing overhead in execution time for each of the scheduling algorithms (from left to right in Table 4) accompanied by decreasing energy consumption.

Another noteworthy result is the comparison between synchronous and asynchronous DVS switching depicted in the last row for each task set in Table 4. For each of the scheduling algorithms, we see additional savings of 1-5% due to the ability to commence with a task's execution while frequency and voltage are changing. We also ran experiments with task sets that had an order of a magnitude smaller deadlines and execution times. Surprisingly,

	Task Set 1		Task Set 2		Task Set 3	
task	Period (P_i)	WCET (C_i)	Period (P_i)	WCET (C_i)	Period (P_i)	WCET (C_i)
1	2,400	400	600	80	90	12
2	2,400	600	320	120	48	18
3	1,200	200	400	40	60	6

Table 3. Task Set, times in msec

algorithm	naïve DVS	static RT-DVS	static save	cycle-cons.	c-c save	look-ahead	l-a save	our feedback	fdbk save
Task Set 1									
sync.	4.47	3.2	28.41%	2.38	46.61%	2.21	50.56%	2.04	54.21%
async.	4.43	3.13	29.35%	2.327	47.51%	2.12	52.07%	2.00	54.70%
savings	0.89%	2.19%		2.51%		3.92%		1.95%	
Task Set 2									
sync.	0.544	0.5056	7.06%	0.4713	13.36%	0.424	22.06%	0.4089	24.83%
async.	0.5276	0.5025	4.76%	0.4622	12.40%	0.4218	20.05%	0.4064	22.97%
savings	3.01%	0.61%		1.93%		0.52%		0.61%	
Task Set 3									
sync.	0.595	0.5616	5.61%	0.4799	19.34%	0.4043	32.05%	0.3708	37.68%
async.	0.5802	0.5496	5.27%	0.4547	21.63%	0.3912	32.57%	0.3671	36.73%
savings	2.49%	2.14%		5.25%		3.24%		1.00%	
Task Set 2 vs. Task Set 3									
change	9.07%	8.57%		-1.65%		-7.82%		-10.71%	

Table 4. Energy [$mW - hrs$] consumption per RT-DVS algorithm

the synchronous vs. asynchronous saving remained approximately the same, even though DVS switches occur ten times as often. These results seem to indicate that the benefit of continuous execution during DVS switching, although not negligible, is secondary to trying to minimize the overhead of switching itself.

We also compared task sets two and three in terms of their absolute energy readings, which is valid since they executed for the same amount of time (ten seconds), the same actual to worst-case execution time ration and the same utilization, albeit at seven times more context switches. This change is depicted in the last row of Table 4 for the asynchronous case. Not surprisingly, the energy with naïve DVS is about 9% higher for task set three than for set two due to the higher context switch overhead of the latter. Quite interestingly, this overhead turns into a *reduction* in energy as DVS schemes become more aggressive peaking at close to 11% energy savings of task set three relative to set two for our feedback scheme.

Finally, we performed two sets of experiments to depict voltage and current fluctuations on an oscilloscope. In the first experiment, the worst-case execution times were inflated compared to the actual execution times to create a considerable amount of dynamic slack that can be utilized by some DVS algorithms (cycle-conserving, look-ahead and our feedback method). In the second experiment, the actual execution times of the tasks were very close to the worst-case execution times used by the schedulability analysis.

In each case, we ran three different task sets with high (0.9), medium (0.5) and low (0.1) utilization, respectively. In what follows, we briefly present the snapshots obtained by the oscilloscope for medium utilization and comment on the indications of those results.

8.1. Loose task sets

These task sets have considerable of dynamic slack dues to the fact that their WCETs are inflated compared to their actual execution times. We experimented with a high, medium and a low utilization. While the absolute numbers differ for these utilizations, the trends are similar. Hence, we only depict the medium utilization case in Figure 7. Static DVS shows two levels of voltages (busy/idle time) whereas cycle-conserving DVS differentiates three levels on a dynamic base. Even lower voltage and current readings are given by look-ahead DVS, which not only distinguishes more levels but also exhibits much lower power levels during load. The lowest results were obtained by our feedback DVS, which defers execution even more aggressively than any of the other methods. However, our feed-

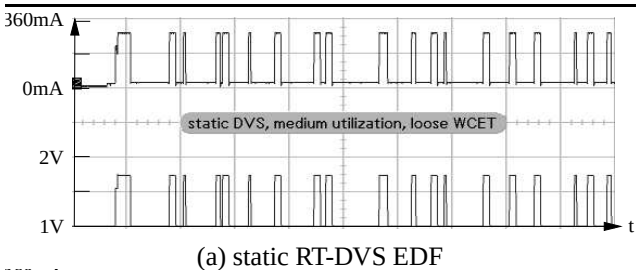
back scheme can only further reduce power consumption occasionally as sufficient slack exists to be recovered by the algorithms of the previous schemes. Dynamic slack is recovered in increasing levels by the latter three schemes.

8.2. Tight task sets

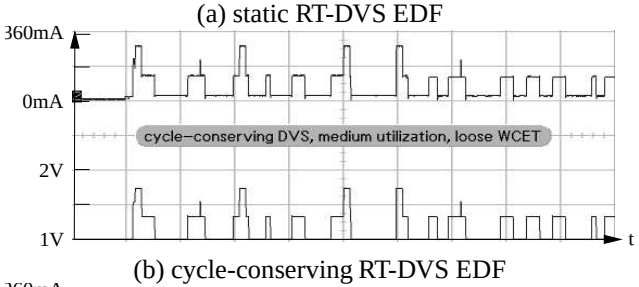
The actual execution times of these task sets are very close to the worst-case execution times we presented to the schedulability and DVS analysis. Figure 8 shows a medium utilization task set. Static voltage scaling will select one safe frequency based on worst-case execution times and retain this frequency throughout the execution. Static selects the minimum frequency/voltage pair (33 MHz @ 1 V) when there are no ready tasks to run. Figure 8 indicates that low frequencies are not often exhibited due to the high utilization of the system and the tight execution times.

Cycle-conserving DVS again shows a more refined voltage (and, hence, frequency) variation, albeit at only a few levels, under load. It cannot lower the frequency based on the actual execution times because these are very close to the worst-case execution times.

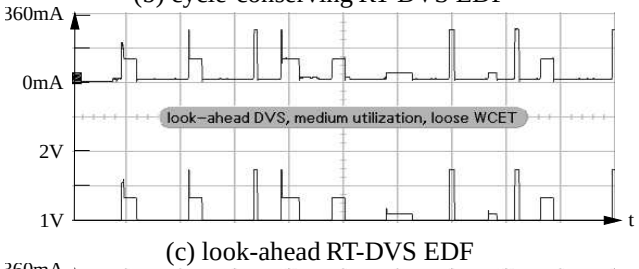
Look-ahead, however, due to its more aggressive nature, can take advantage of some amount of dynamic slack created and lower the frequency/voltages. There is not much of dynamic slack to recover. Infrequently, it even has to overcome the fact that the frequency was lowered too much in the past by raising the voltage and frequency to a level even higher than the safe frequency calculated by static.



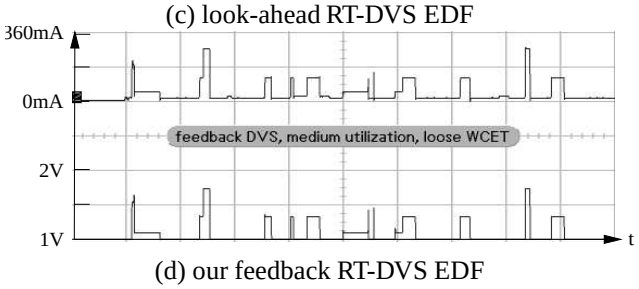
(a) static RT-DVS EDF



(b) cycle-conserving RT-DVS EDF

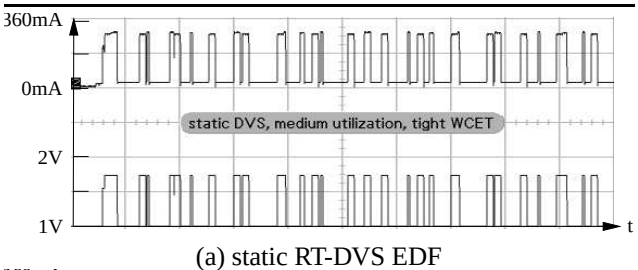


(c) look-ahead RT-DVS EDF

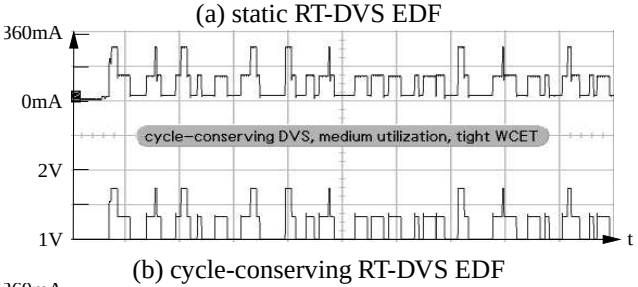


(d) our feedback RT-DVS EDF

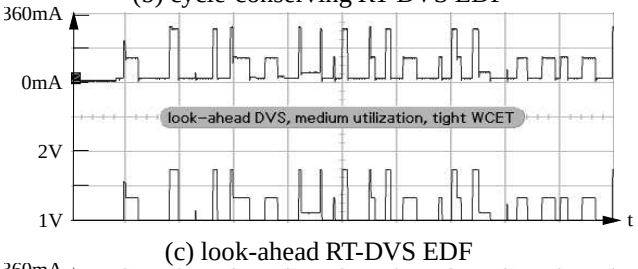
Figure 7. Voltage/current oscilloscope shot with loose task set: $WCET = 2 \times ActualExecTime$, $Utilization\, U = 0.5$



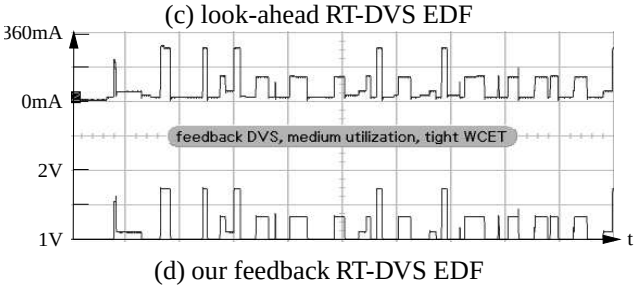
(a) static RT-DVS EDF



(b) cycle-conserving RT-DVS EDF



(c) look-ahead RT-DVS EDF



(d) our feedback RT-DVS EDF

Figure 8. Voltage/current oscilloscope shot with tight task set: $WCET = ActualExecTime$, $Utilization\, U = 0.5$

Nonetheless, the cumulative savings are higher for look-ahead.

Our feedback scheme shows even lower voltages/frequency than the look-ahead scheme, which illustrates again its ability to scale power even more aggressively than any of the previous schemes. The tighter execution times illustrate this much better than the looser ones in Figure 7. Hence, the potential for feedback is more significant in systems with actual execution times closer to their WCET.

The voltage and current graphs obtained from an oscilloscope clearly exhibit the expected behavior for the four implemented DVS algorithms. These graphs, together with the execution traces of long-running task sets under different scheduling algorithms, showed no missed deadlines of tasks while lowering power whenever possible, thus experimentally validating our implementation as well as the chosen scheduling algorithms.

9. Conclusion

We successfully created an infrastructure for investigating hard real-time DVS schemes on the IBM PowerPC 405LP with the task scheduling performed on an EDF basis. The results indicate benefits in energy reduction of up to 5% for fast DVS modulation without entering sleep modes, *i.e.*, by continuing with the execution of application code while switching between voltages/frequencies. We assessed the benefits quantitatively by determining the energy consumption over the hyperperiod of real-time tasks for different algorithms and found that aggressive scheduling algorithms that modulate processor voltage and frequency can achieve up to 54% reduction in energy consumption for periodic real-time task sets.

Acknowledgements

Bishop Brock provided most valuable support on hardware modifications, Linux patches and installation procedures as well as technical details on the overhead of DVS. IBM Research (Austin) donated the 405LP board. The work was supported in part by NSF grants CCR-0208581 and CCR-0310860.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

[2] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and agressive scheduling techniques for power-aware real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2001.

[3] B. Brock and K. Rajamani. Dynamic power management for embedded systems. In *IEEE International SOC Conference*, Sept. 2003.

[4] E. Cooper and R. Draves. C threads. TR CMU-CS-88-154, Carnegie Mellon University, Dept. of CS, 1988.

[5] A. Dudani, F. Mueller, and Y. Zhu. Energy-conserving feedback edf scheduling for embedded systems with real-time constraints. In *ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, pages 213–222, June 2002.

[6] R. Engelschall. Portable multithreading: the signal stack trick for user-space thread creation. In *Proceedings of the USENIX Conference*, June 2000.

[7] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *1st Int'l Conference on Mobile Computing and Networking*, Nov 1995.

[8] D. Grunwald, P. Levis, C. M. III, M. Neufeld, and K. Farkas. Policies for dynamic clock scheduling. In *Symp. on Operating Systems Design and Implementation*, Oct 2000.

[9] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, Dec. 1995.

[10] IBM and M. Software. Dynamic power management for embedded systems. white paper.

[11] D. Kang, S. Crago, and J. Suh. A fast resource synthesis technique for energy-effi cient real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2002.

[12] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, Jan. 1993.

[13] K. Nowka, G. Carpenter, and B. Brock. The design and application of the powerpc 405lp energy-effi cient system on chip. *IBM Journal of Research and Development*, 47(5/6), September/November 2003.

[14] T. Pering, T. Burd, and R. Brodersen. The simulation of dynamic voltage scaling algorithms. In *Symp. on Low Power Electronics*, 1995.

[15] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Symposium on Operating Systems Principles*, 2001.

[16] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *USENIX Conference*, pages 65–80, Winter 1991.

[17] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verifi cation of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.

[18] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *1st Symp. on Operating Systems Design and Implementation*, Nov 1994.

[19] Y. Zhu and F. Mueller. Preemption handling and scalability of feedback dvs-edf. In *Workshop on Compilers and Operating Systems for Low Power*, Sept. 2002.

[20] Y. Zhu and F. Mueller. Feedback edf scheduling exploiting dynamic voltage scaling. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 84–93, May 2004.