Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications*

(Appeared at the Workshop on Compilers and Operating Systems for Low Power (COLP), Oct 2000)

Daniel Mossé, Hakan Aydın, Bruce Childers and Rami Melhem
Department of Computer Science, University of Pittsburgh,
Pittsburgh, PA 15260
{mosse, aydin, childers, melhem}@cs.pitt.edu

Abstract

Managing power consumption while simultaneously delivering acceptable levels of performance is becoming a critical issue in several application domains such as wireless computing. We integrate compiler-assisted techniques with power-aware operating system services and present scheduling techniques to reduce energy consumption of applications that have deadlines. We show by simulation that our dynamic power management schemes dramatically decrease energy consumption.

1 Introduction

The ubiquitous nature of mobile computing and more stringent demands for smaller devices, with longer battery life, has brought power management to the forefront of embedded systems research in recent years. Hardware manufacturers have introduced standards such as the ACPI (Advanced Configuration and Power Interface) [6] for power management of laptop computers that allows several power modes of operation; e.g., powering down some parts of the computer such as the disk drive, reducing the voltage of the CPU, and adjusting screen brightness. Handheld devices such as Personal Digital Assistants and cellular telephones will have even more stringent battery requirements in the future due to smaller size and increased functionality. Battery management is therefore one of the most important design issues since it determines to a large extent mission duration, quality of service (QoS), battery size, device weight, etc.

To increase the capabilities of power management, we enlist both compiler and power-aware operating system scheduling techniques. The problem to be solved is to extend battery life to enable system integrators to enhance the mix of jobs and/or the quality of the results of the current set of applications running in the device.

These applications will initially be a "fixed" set loaded on the processor for the lifetime of the system. The timings are also "fixed" for the set of applications. Examples include audio transmission, reception, and play-back, web-page loading, video transmission and display, among others. We use the compiler to annotate an application's source code with temporal information and to insert power management points in the program that invoke the operating system to adaptively alter processor supply voltage (and consequently, the clock speed is also reduced). Reducing the voltage to the CPU has a linear impact on the speed of the applications, but has a quadratic effect on energy savings [13].

*This work has been supported by the Defense Advanced Research Projects Agency through the PARTS (Power-Aware Real-Time Systems) project under Contract F33615-00-C-1736.

We show how to reuse time not used by other tasks (with information gathered from compiler annotations) to reduce processor clock speed. Such time reclaiming may either reduce average battery size or increase average mission life and/or QoS.

We first describe related work in the next section, and present our system model in Section 3. Our power-aware scheduling algorithms are presented in Section 4. Section 5 gives simulation results that show energy reductions possible with our techniques for synthetic workloads. We close the paper in Section 6 with concluding remarks and future work directions.

2 Related Work

In the last decade, there has been considerable research effort in low-power system design, as exemplified by new techniques in VLSI, Code/Algorithm/Compiler transformations, use of hierarchical memory systems and application-specific software modules. On-going research has had an important effect in embedded real-time systems design, simply because many of the applications running on power-limited systems have tight temporal constraints (e.g., real-time satellite communication and control tasks for unmanned autonomous vehicles–UAVs). As a simple approach, the predictive system shutdown technique [10] is usually used to turn off the power supply when the processor is idle and is likely to remain idle for a considerable amount of time. However, due to the convex dependence between supply voltage and power consumption, this technique is clearly sub-optimal, even for a system with perfect knowledge of idle intervals. On the other hand, the variable voltage scheduling (VVS) framework, which involves dynamically adjusting the voltage and frequency of CPUs (and hence the CPU speed), has recently become a major research area. Reducing the voltage usually results in considerable power savings, but also in a delay in response time. Thus, VVS research focuses on minimizing energy consumption of the system, while still meeting deadlines.

Although it is a relatively new research field, there is already a large body of literature on VVS, addressing various aspects of the problem such as aperiodic and/or periodic models, off-line or on-line scheduling, static or dynamic priority assignments [13, 4, 5, 8]. However, most of the scheduling schemes presented in these studies, while using exclusively worst-case execution time (WCET) to guarantee the timeliness of the system, lack the ability to dynamically take advantage of unused computation time. In fact, an application with timing constraints usually exhibits a large variation in *actual* execution times; for example [2] reports that the ratio of the best-case execution time to the worst-case execution time can be as low as 0.1.

Consequently, *dynamically reclaiming 'slack-time'* can be (and as we show later in this paper, is) a powerful approach to obtaining considerable power savings and to minimizing the effects of designing the system with WCET information, which is usually a very conservative prediction of *actual* execution time. It goes without saying that dynamic slack-management techniques should preserve the feasibility of the task system (i.e. no deadline should be missed), even under a worst-case scenario that may take place after *any* speed adjustment decision. The only practical approach in the seminal work presented in weiser:osdi94 only takes into consideration the past interval energy consumption in order to set the future interval.

A recent study [7] addressed slack reclaiming issues in power-aware scheduling, but only in the context of systems with two (discrete) voltage levels. Systems which are able to operate on a (more or less) continuous voltage spectrum are rapidly becoming a reality thanks to advances in power-supply electronics and CPU design [3, 9]. For example, the recently announced Crusoe processor

is able to dynamically adjust clock frequency in 33 MHz steps [11]. This trend is likely to have a major impact on the design of power-aware computer systems.

3 Model

Our techniques are targeted to embedded systems with a fixed set of applications, each of which is assigned a specific amount of time for completing its computations. This type of situation can occur in a system that has a single periodic application that must complete each iteration during its period, or a system that assigns an amount of time for each application (in the classical real-time systems literature, this is called the worst-case execution time of the application). Another example is an event-triggered system, in which an application request arrives with a fixed deadline. The system must carry out the computation in a non-preemptive way for that period of time. The schemes that we propose show how to slow down the processor and extend the time the application executes to reduce energy consumption.

For simplicity of presentation, in this paper we consider an embedded system with a single application with the following characteristics. An application is divided into n sections or tasks, $\tau_i, 1 \leq i \leq n$. We assume that each section (e.g., a loop or a procedure call) has an average execution time (c_avg_i), and a WCET (c_i). The compiler inserts power management points (PMPs) at the beginning and end of a section to test whether the processor clock speed can be changed. The check at the beginning of the section records the current time, and the check at the end of the section computes the actual execution time for the section. Based on how the actual execution time compares to the WCET, processor speed may be adjusted either up or down.

We are currently developing the hardware and software interfaces to support compiler-assisted adjustment of processor voltage. We are also developing the optimizations necessary to intelligently insert the PMPs in the code to minimize their overhead. Selecting the locations at which to place PMPs relates to how code sections are determined in a program. Loop boundaries and procedure call sites are natural locations to insert PMPs. Inserting a PMP also relates to how close the WCET is to the average execution time for a given piece of code. If the compiler can determine that a WCET is close to actual execution time (e.g., a tight loop with no cache misses and high branch prediction accuracy will have a WCET that closely corresponds to the actual execution time), then it may be possible to statically determine how fast to run a given code section.

We can also use profiling information to find the locations in a program's dynamic execution trace where the speed is most likely to change. For instance, assume that PMPs are inserted at loop boundaries and call sites. After profiling the application, we may discover that the processor's speed never changes at particular PMPs. It may be possible to remove such PMPs since they only contribute to performance overhead and do not affect processor speed. When removing a PMP, the code section controlled by that PMP needs to be merged with an adjacent section to ensure its deadlines will be met. In this paper, we assume fixed code sections and restrict ourselves to describing several schemes for determining the best execution speed (voltage) for a code section.

The processor has a maximum speed, S_{max}, at which it can execute; without loss of generality, we assume that $S_{max} = 1$. Further, it is possible to lower the speed of the processor by adjusting the voltage levels of the hardware (consequently slowing down the clock frequency that drives the processor). We will denote the CPU speed settings of each section by $s_i, 1 \leq i \leq n, s_i \leq S_{max}$. Note that the $\{c_i\}$ and $\{cavg_i\}$ values are given with respect to a system which uses the maximum speed (S_{max}).

Processor power consumption is inversely pro-

portional to the square of supply voltage. In CMOS technology, the dominant component of energy consumption is the *dynamic power dissipation* P_d, which is given by: $P_d = C_{ef} \cdot V_{dd}^2 \cdot f$, where V_{dd} is the supply voltage, C_{ef} is the *effective switched capacitance* and f is the frequency of the clock. The gate delay D is inversely related to the supply voltage V_{dd} ($D = k \cdot \frac{V_{dd}}{(V_{dd}-V_t)^2}$, where k is a constant, and V_t is the threshold voltage). Hence, the gate delay increases roughly linearly with the decrease in the supply voltage. From these equations, it is apparent that one can reduce *quadratically* or even *cubically* the power consumption, at the expense of *linearly* increased delay (reduced speed) [5].

Because every code section in a program has a WCET, it is possible to derive the nominal worst case load of the system. Moreover, the application has a deadline, d, by which it must complete execution. Given the deadline and application characteristics, we can compute a slack factor SF in the system, which represents the amount of free time there is in the system with respect to the computational requirements of an application. Formally, $SF = \frac{d}{\sum_{i=1}^{n} c_i}$, where n is the number of sections in the application. In other words, the application deadlines and the slack in the system are interdependent, since the amount of computation to be carried out is fixed. One can interpret the slack factor as being the *inverse of the load* imposed in the system; for example, if the application consumes 10ms at maximal speed, and the deadline is 50ms, then the load in the system is 20% and the slack factor is 5.

4 Voltage Adjustment Schemes

In this section, we describe techniques that change processor speed (i.e., slow down) based on the natural slack of the system and time reclaimed from code sections that finish before their deadlines. We assume that the CPU goes into *idle* mode and consumes no power when there is no application to execute. The schemes we study are:

- **NPM:** No Power Management. That is, all n sections execute at S_{max}.
- **SPM:** Static Power Management. This conservative scheme uses only WCET and deadline information (i.e., the existing slack factor or load of the system). This scheme distributes the slack proportionally among all program sections, prior to run time. In other words, each section is executed with speed $s_i = 1/SF$. For example, if the load on the system is 80%, then the slack factor is 1.25 and the speed of the processor is 0.8.

  This conservative scheme is based on the principle that in a minimization problem with quadratic objective functions, the best solution is to increase the time given to each section proportionally. This scheme also guarantees that all sections finish before their deadlines.

  On the other hand, SPM does not take advantage of the fact that often sections do not execute as much as their WCETs, and that the system can reclaim the unused resource/time. Below we describe **DPM** (Dynamic Power Management) schemes which reclaim unused CPU time whenever a section finishes earlier than its predicted WCET.
- **DPM-P:** Dynamic Power Management–Proportional. The reclaimed slack is distributed proportionally to each section in an application. In this scheme, every time a section finishes, the system computes the reclaimed time, and allows other sections to slow down proportionally. In other words, when the first section finishes execution at time t'_1, the processor speed will be set to $s'_2 = \frac{\sum_{i=2}^{n}(c_i)}{d-t'_1}$. If j sections have finished their execution at $t'_1, ..., t'_j$, the speed of the processor is set to $s'_{j+1} = \frac{\sum_{i=j+1}^{n} c_i}{d-t'_j}$.

- **DPM-G:** Dynamic Power Management–Greedy. This is a more aggressive technique, because it gives all the reclaimed slack to the next section. In addition, DPM-G allows the next section to use the maximum possible amount of natural slack in the system, while guaranteeing the feasibility of the remaining sections. In this case, if j sections have finished their execution at $t''_1, ..., t''_j$, the speed of the processor is set to $s''_{j+1} = \frac{c_{j+1}}{d - t''_j - \sum_{i=j+2}^{n} c_i}$.

- **DPM-S:** Dynamic Power Management–Statistical. A very aggressive scheme would distribute the reclaimed slack, and the natural slack, and the slack that would appear in the system if other sections were to finish early (base it on the c_avg). That is, if j sections have finished their execution at $t'''_1, ..., t'''_j$, the speed of the processor is set to $\frac{\sum_{i=j+1}^{n}(c\_avg_i)}{d - t'''_j}$.

  Clearly, this would not guarantee the deadlines of any sections, since it consumes time that will perhaps be needed for other sections after τ_j executes under worst-case scenario. To take advantage of the predictions embedded in the c_avg, while striking a balance between DPM-P and DPM-G, DPM-S limits the speed setting to a level that would enforce the guarantees needed in the frame-based system: that is $s'''_{j+1} = max\{s''_{j+1}, \frac{\sum_{i=j+1}^{n}(c\_avg_i)}{d - t'''_i}\}$. Note that DPM-S as described s''' is less aggressive than DMP-G, because it never slows the processor more than the latter.

In the next section, we evaluate the performance of each of these schemes, with respect to the energy saved when slowing down the process. We compare the energy consumed by each of the schemes, including the baseline scheme, NPM.

parameter	name	values assumed
WCET	c	$5, 10, ..., 25$
avg exec time	a_c	$\frac{c}{2}, \frac{6c}{10}, ..., \frac{9c}{10}$
slack factor	slack	$\frac{1}{0.05}, \frac{1}{0.1}, ..., 1$

Table 1: Parameters for Simulations

5 Performance Evaluation

Using our experimental set up, we carried out over one million simulations. Each result presented here is an average of 100 data points. The actual execution times of each section were drawn from either a uniform or a normal distribution, using a_c as the average and c as the maximum (therefore, the normal distribution is cut at the upper limit).

Let us represent the $\sum c_i$ by σ, and the slack factor is an input parameter ranging from 1 to 20 (i.e., $1.0 \geq load \geq 0.05$). The deadline of the system is defined as $d = \sigma/load$. For example, if .5, then $d = 2\sigma$, and if .1, then $d = 10\sigma$. The values for the variables in the simulations are shown in Table 1.

In Figure 1 we show the (normalized) energy consumed by each scheme as a function of the slack in the system. Schemes SPM, DPM-P, and DPM-S (which depend on the load) show a large advantage when the slack is maximal. Scheme Baseline, which does not depend on the slack in the system has a flat curve, since the work is performed at maximum speed for all sections (independent from the slack in the system).

On the other hand, scheme DPM-G, which also depends on the deadline, has very stable performance (it does comparatively bad when the slack is large, but better when slack is small). This is because when there is too much slack, the first few sections will execute extremely slowly, consuming most of the slack available. By the time the fourth or fifth sections starts executing, the slack all but vanished, forcing a dramatic increase

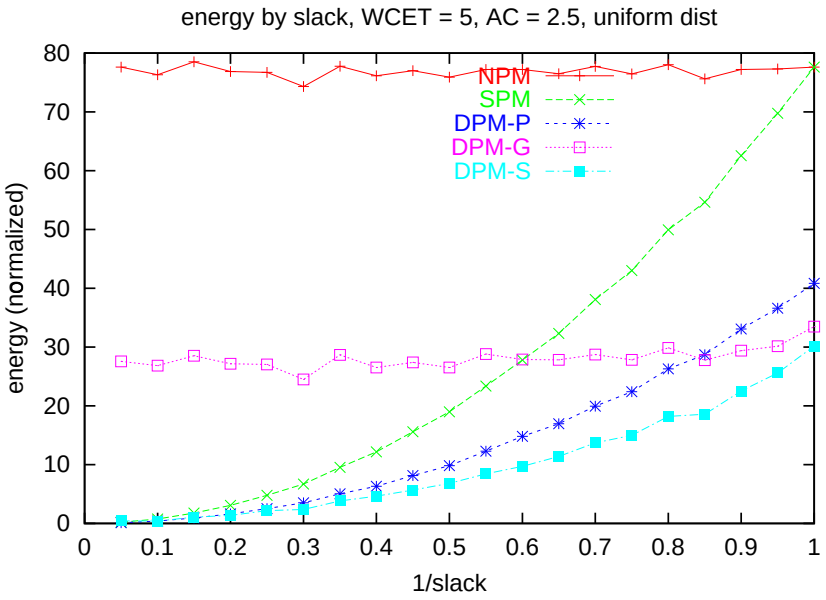

Figure 1: Energy consumed by various schemes as a function of the slack in the system

in processor speed up (even getting to the maximum speed at times). This phenomenon makes about 90% of the sections in the system execute with almost no slack, causing the energy to be at the same level for all slack values. The speed settings and the completion time of each section can be seen in Figure 2.

Note also that, were we to consider the overhead of speed changes in our results, DPM-G would fare even worse: if overhead were high, one would only consider changing the speeds when the savings were significant, consequently with larger speed differentials. Because DPM-S and DPM-P have vary stable speed settings, the overhead in these schemes is minimal.

Average computation time also plays an interesting role in the performance of the schemes. We have observed above that as the load increases, the performance of the schemes degrades, except for DPM-G and NPM. This is confirmed in the graphs shown in Figure 3, where DPM-G has approximately the same energy consumption, but the other schemes do not. The interesting point to observe is that only after slack factor values $SF > 2$ (not shown) does the performance of DMP-P and and SPM become strictly better than DPM-G. This graph also confirms that DPM-S is *always* better than the other schemes.

6 Conclusion

We have shown that when the compiler is used to assist the operating system in changing CPU speed, several schemes are possible. Among these, aggressive schemes usually do better than non-aggressive schemes. However, our simulation results indicate that the best scheme is an adaptive one that takes an aggressive approach (such as trusting average-case estimates for execution time) while providing safeguards that avoid violating application deadlines.

We are currently modifying our simulator to include speed-changing overheads and to consider lowest-speed settings. We note that a single application with sections as described here is equivalent to having a frame-based system with a set of applications, in which all sections must execute repeatedly with the same frequency. The deadline of this set of tasks is determined as the inverse of the frequency (see [1] for examples of applications

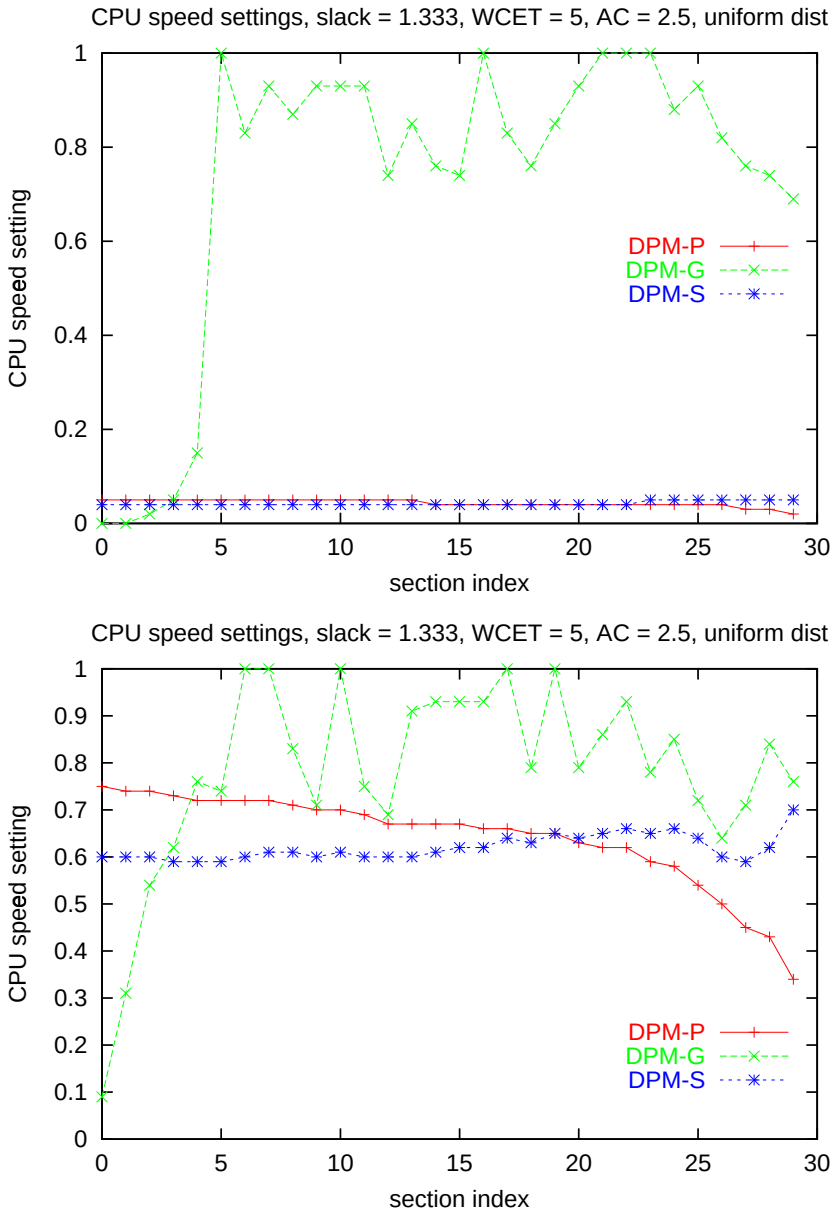

Figure 2: Speed settings for sections according to different schemes (for 2 representative slacks, 0.05 at the top and 0.75 at the bottom). Note that the DPM-G scheme has a high spike in speed settings, while other schemes are more uniform.

using this model).

References

[1] Anthony Egan, David Kutz, Dmitry Mikulin, Rami Melhem, and Daniel Mossé. Fault-Tolerant RT-Mach (FT-RT-Mach) and its Application to Real-Time Train Control. *Software Practice and Experience*, 29(4):379–395, 1999.

[2] R. Ernst and W. Ye. Embedded Program Timing Analysis based on Path Clustering and Architecture Classification. In *Computer-Aided Design (ICCAD)'97*. pp. 598-604.

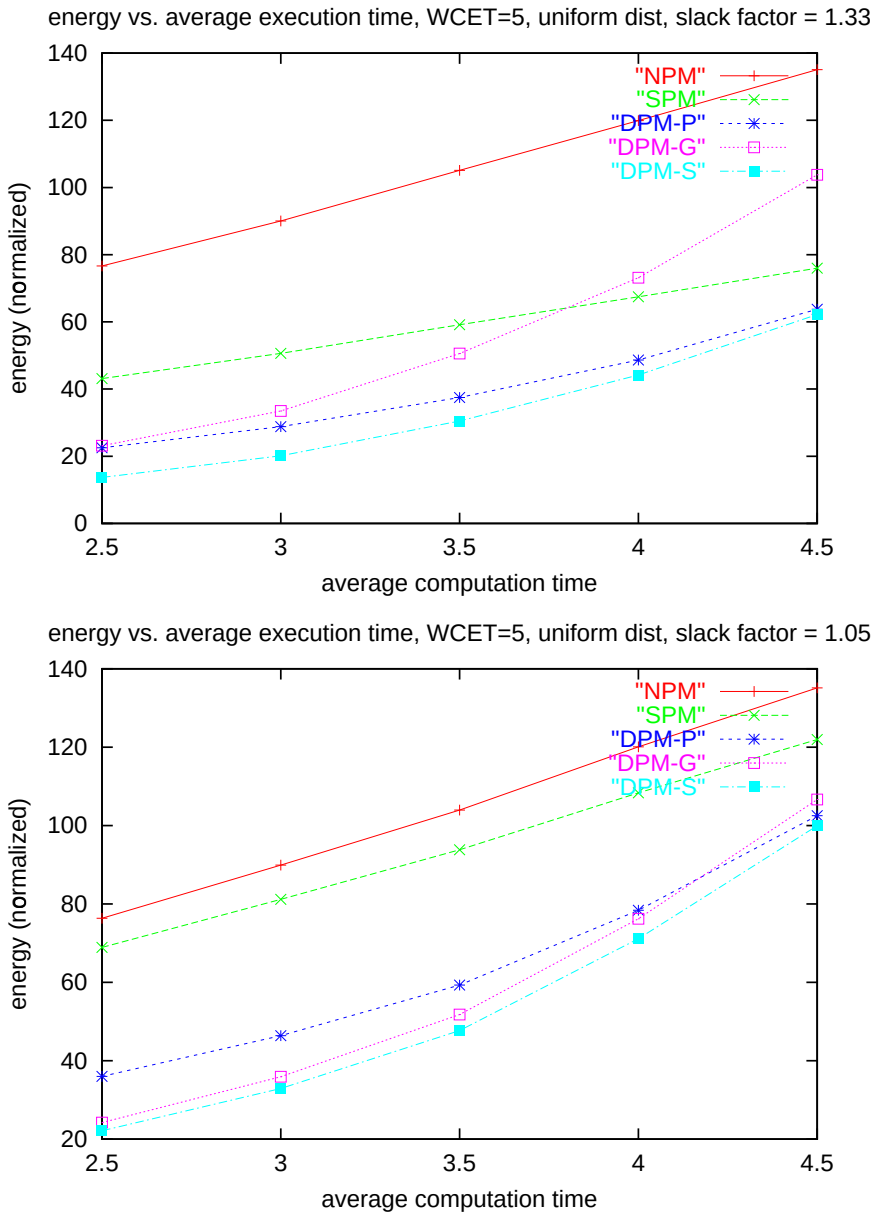

Figure 3: Energy as a function of the average computation time, medium load (at top) and high load (bottom)

[3] V. Gutnik and A. Chandrakasan. An Efficient Controller for Variable Supply Voltage Low Power Processing. *Symposium on VLSI Circuits*, pp.158-159, 1996.

[4] I. Hong, M. Potkonjak and M. B. Srivastava. On-line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor. In *Computer-Aided Design (ICCAD)'98*. pp. 653-656.

[5] I. Hong, G. Qu, M. Potkonjak and M. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors. In *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, December 1998.

[6] Intel, Microsoft, and Toshiba. Advanced Configuration and Power Man-

agement Interface (ACPI) Specification. http://www.intel.com/ial/WfM/design/pmdt/acpidesc.htm, 1999.

[7] C. M. Krishna and Y. H. Lee. Voltage Clock Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems. In *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, Washington D.C., May 2000.

[8] Y. H. Lee and C. M. Krishna. Voltage Clock Scaling for Low Energy Consumption in Real-Time Embedded Systems. In *The Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, December 1999.

[9] W. Namgoang, M. Yu and T. Meg. A High Efficiency Variable-Voltage CMOS Dynamic DC-DC Switching regulator. *IEEE International Solid-State Circuits Conference*, pp.380-391

[10] M. Srivastava, A. P. Chandrakasan and R. W. Brodersen. Predictive System Shutdown and other Architectural Techniques for Energy Efficient Programmable Computation. *IEEE Transactions on VLSI Systems*, 4(1): 42-55, 1996.

[11] Transmeta Corporation, Crusoe Processor Specification, http://www.transmeta.com

[12] Mark Weiser, Brent Welch, Alan Demers, Scott Shenker. Scheduling for Reduced CPU Energy. First Symposium on Operating Systems Design and Implementation (OSDI '94)

[13] F. Yao, A. Demers and S. Shankar. A Scheduling Model for Reduced CPU Energy. *IEEE Annual Foundations of Computer Science*, pp. 374 - 382, 1995.