

# Optimal Placement of Power Management Points In Real Time Applications

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

## Abstract

## 1 Introduction and Related Work

In the last decade, there has been considerable research effort in low-power system design. 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. Recently, the variable voltage-scheduling (VVS) framework, which involves dynamically adjusting the voltage and frequency of CPUs (and hence the CPU speed), has become a major research area. Reducing the voltage usually results in considerable power savings, but also in a delay in the response time, and extra energy consumption for each speed adjustment. Thus the problem is to optimize for the amount of energy consumed considering both; the savings achieved by the dynamic speed adjustment, and the extra work exerted for employing these adjustments. In [1] we presented three dynamic setting schemes used in periodic real time systems that take advantage of the time left over from previous program segments. These schemes were compared to the energy consumed in case of no power management is used. We showed a saving in the CPU energy up to 90% and 62% savings compared to a static setting scheme.

It is the compiler's task to insert power management points *PMPs* that holds the information about the execution of the program segments, as

well as taking the the decision of changing the speed of the processor. The desired speed is computed according to the speed setting scheme used.

Dynamic speed setting is very adequate to use at a frequent procedure call or for loops with relatively large index, where the speed is set at the start of iterations. To select the granularity of the program segment that is assigned a single speed, [20] does global program analyses to detect regions of sufficiently large granularity, then select a single region with highest predicted benefit.

A similar problem is the optimal insertion of checkpoints in a fault tolerance system as in [22, 23, 24, 25], where the placement can be static at fixed time intervals or dynamic based on the cost of the checkpoints[26].

Our contributions in this paper is extending the work in [1] showing the overhead effect of speed adjustments for different speed setting schemes. We also provide a theoretical solution for deciding the optimal number of *PMPs* to be placed in a program aiming to reach the minimum energy consumption. We compare our results from the theoretical solution with the simulated results , the theoretical shows a decision accuracy within  $\pm 2$  *PMPs*

We describe our model in the next section. The effect of different overheads are considered and shown for dynamic speed setting schemes in section 3. Section 4 gives a theoretical solution for best selection of the number of *PMPs*, as well as comparing these results with simulation results for each scheme. We close our paper in section 5 with

the concluding remarks.

## 2 Model

Our techniques are targeted for the embedded systems where applications typically repeat computations with a certain periodicity, this period is called deadline,  $d$ . We consider the sequential form of the program execution, where a program consists of  $n$  segments (or partitions). Each segment is characterized by its worst case execution time,  $wc_i$ , and average execution time  $avg_i$ . The actual executions of the segment  $i$ ,  $ac_i$ , is only known at run time, but is limited by the  $wc_i$  such that  $0 \leq ac_i \leq wc_i$ . These times describe the execution behavior of segment  $i$  in case the processor is running at its maximum speed. The quantity  $\alpha = avg_i/wc_i$  describes the expected variability in the execution time of the segment.

Given the deadline and the application characteristics, we can compute a slack factor  $SF$  in the system. This represents the natural amount of free time there is in the system with respect to the computational requirements of an application. Intuitively, *natural slack* related to the load parameter  $l$ , as the reciprocal of the  $SF$ . There is another source of slack that is generated whenever a program segment finishes its execution before the estimated worst-case time for this segment. This remaining time is called *reclaimed slack*.

Processor dynamic power consumption follows the *CMOS* technology; the power is directly proportional to the frequency and the square of the supply voltage.  $P = aCfV^2$ , where  $a$  is the activity factor in the processor,  $C$  is the effective switched capacitance,  $f$  is the operating frequency, and  $V$  is the supply voltage. We employed the Transmeta's TM 5400 processor [12] speed model. There is a 16-step frequency scale that ranges from 200MHz at 1.65V to 700 MHz at 1.1V. Each step is  $\simeq 33$ MHz. We consider the different overheads of each frequency and voltage change below.

### 2.1 Overhead Sources

When computing and changing the frequency and voltage of the target CPU, several sources of overhead may be encountered (from now on we denote changing/setting both voltage and frequency by changing/setting speed). The principal sources of overhead are computing the new speed, and the setting this speed by doing some voltage transition for the DC-DC switching converter, and hence a frequency change.

**Computing the new speed:** The dynamic decision of the desired frequency is taken by a speed selection algorithm which takes from the CPU time and consumes extra energy. The details of the different algorithms are discussed in Section 3. For each adjustment scheme considered, this overhead,  $F$ , is constant in terms of the number of cycles needed for execution. This includes the overhead of calling the library and performing the operations that compute the new speed. Since this might be executed at different frequencies, the time overhead,  $O_1$ , may vary as shown below.

$$O_1(S_i) = \frac{F}{S_i} \quad (1)$$

where  $S_i$  is the CPU speed executing segment  $i$ , and is normalized to  $S_{max}$  which is equal to 1. From our experiments on SimpleScalar 3.0 [19] (a micro architectural simulator)- where we implemented the speed setting schemes and inserted a set of *PMPs* in some real applications like an MPEG decoder - we observed an execution range between 285-320 cycles for such overhead. Later, in our experiments we fix this overhead for the presented schemes to 300 for this type of overhead.

**Setting the new speed:** The variable voltage generated by the DC-DC switching regulators in the power supply cannot instantly make a transition from one voltage to another [6]. This transition exerts delay time and extra energy consumption. The transition times typically range from 10 to 100 microsec/volt. For example, the *Strong Arm 100* core is capable of on-the-fly clock frequency changes in the range of 59MHz to 206MHz

where each frequency change incurs a latency of up to 150  $\mu\text{sec}$  [16], while the *1pARM* processor (an implementation of the *ARM8* architecture) takes 25  $\mu\text{sec}$  for the complete 10 MHz to 100 MHz. Some systems can continue operation while speed and voltage are changing[17][6], but the frequency will keep varying during the transition period. We will follow a conservative approach and assume that the processor can not serve the application during this period.

The other dimension of the speed change is the energy overhead, this overhead varies based on the difference between the current and the target frequencies . The voltage change creates an energy penalty on the external capacitance on the power supply. In the *1pARM* processor[17] this penalty is at most  $4\mu\text{J}$ , which is equivalent to 712 full-load cycles for the transition between 5 - 80 MHz[15]. For the sake of the simulation we will assume a constant overhead cycles,  $G$ , for each speed step transition. This overhead is assumed to be 320 cycles for every 33 MHz step(from [15], 712 cycles for 5-80 MHz transition  $\simeq$  320 cycles for 33MHz transition). The time overhead  $O_2$  would be computed as in equation (2)

$$O_2(S_i, S_{i+1}) = G \frac{d(S_{i-1}, S_i)}{S_{i-1}} - G \frac{d(S_{max}, S_i)}{S_i} \quad (2)$$

where  $d(S_m, S_n)$  is a function that returns the number of speed steps needed to make a transition between  $S_m$  and  $S_n$ . In the Transmeta model, this function returns how many many multiples of 33MHz is the difference between  $S_m$  and  $S_n$ .

We study the impact of varying this overhead on the selection of the optimal number of PMPs in section 4.

### 3 Speed Adjustment Schemes

In [1], we presented a set of power management schemes that dynamically sets the processor speed taking advantage of the reclaimed slack besides the natural slack. Here we take two of those schemes as examples to demonstrate how to include the

aforementioned overhead in the analysis and algorithms. Deadlines maybe violated only in cases where the processor needs to run at almost the maximum speed to meet the application’s deadline and there is not enough slack to accomodate the time overhead for a single transition. We choose to base our schemes on the static speed at full load;  $l=1$ , as it is proven to be an optimal static speed while meeting all the deadline [21].

#### 3.1 Proportional- Dynamic Power Management

In this scheme the slack (natural and reclaimed) is uniformly distributed to all the remaining segments proportional to their worst-case execution times. The time overheads for both computing and setting the new speed are subtracted from the time remaining to deadline. The processor’s speed for segment  $i$ ,  $S_i$  is equal to :

$$S_i = \frac{\sum_{j=i}^n wc_j}{d - t_{i-1} - O_1(S_{i-1}) - O_2(S_i, S_{i+1})} \quad (3)$$

where  $t_{i-1}$  is the accumulated execution times of all segments before reaching  $PMP_i$

Our main concern here is to compute the exact time left for the application to execute before the deadline. This remaining time should exclude the time taken to compute the new speed as well as setting the target speed. To address feasibility, we always spare time that suffices the processor at its new speed to be able to return to the maximum speed in case the application turns out to consume all its available slack. Given that the voltage setting overhead is dependent on the new frequency, we note that the  $S_i$  appears on both sides of the formula. It is solved iteratively and it converges very fast to the right solution.

#### 3.2 Dynamic Greedy Power Management

The Dynamic Greedy scheme presented here distributes the reclaimed slack to the segment right after the *PMP*. It is a combination of the static

scheme where the speed is constant throughout the program and is proportional to  $l$ , and the greedy scheme that assigns all the slack (natural and reclaimed) to only the next segment.

$$S_i = \frac{wc_i}{d - t_{i-1} - \frac{\sum_{j=i+1}^n wc_j}{S_{static}} - O_1(S_{i-1}) - O_2(S_i, S_{i+1})} \quad (4)$$

where  $S_{static}$  is the static speed. Addressing Feasibility allow to ensure enough time for a single later transition to the maximum speed.

### 3.3 Results

A simulation is performed for the two schemes with and without overhead included. Each result presented here is an average of 500 data points. The actual execution times  $ac_i$  of each section is drawn randomly from a normal distribution, using  $\alpha$  equals to 0.6. The used energy function is as shown in section 2

In Figure (1) we show the energy consumption for number of *PMPs*. As you can see the optimal *PMP* placement varies according to many factors, like  $\alpha$  or the speed adjustment scheme. For example for the Proportional and the Enhanced Greedy schemes, the optimal number lies between 10 and 20 *PMPs*.

Although counter intuitive, it is clear that energy consumption is not inversely proportional to the number of compiler inserted *PMPs*. The question here is what is the granularity of the program segments, where the compiler can perform the power management around?

## 4 Optimal PMP Placement

To achieve the minimum energy consumption for any scheme depends on the optimal placement of *PMP*; considering the energy overhead effect on the total energy consumed. It is equivocal that the more inserted management points, the less consumption is achieved. But with the overhead accounted, the energy consumed - at a point - increases with the increasing of the number of

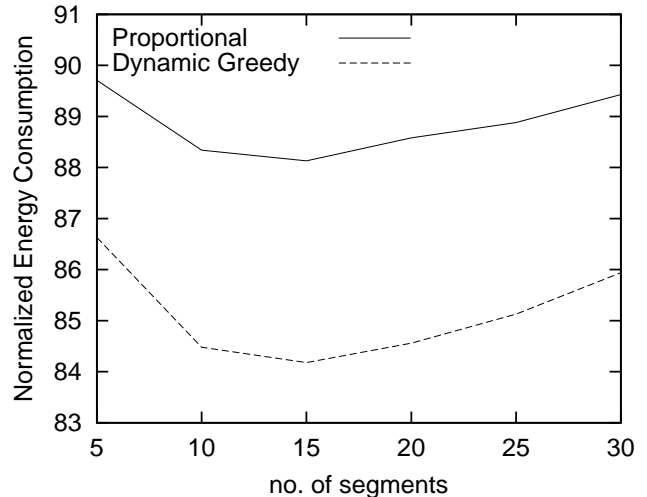


Figure 1: Total Energy Consumption for different schemes versus the number of *PMPs* at  $\alpha = 0.6$ .

*PMPs*, this part of what is illustrated in figure 1. We develop a theoretical framework for deciding on the number of *PMPs*. Under this framework, we assume sequential programs that can be partitioned at equal computational distances, that each segment has a perfect execution behavior (i.e.,  $ac_i = avg_i$  for all  $1 \leq i \leq p$ ), that there is a constant energy overhead,  $h$ , for the insertion of any *PMP*, and that the speed range is continuous.

The total consumed energy for  $p$  segments is equal to the energy consumed by the actual program plus the energy consumed in the different overheads.

$$E_p = \gamma_1 \sum_{i=1}^p \frac{avg_i}{\phi_i^3} + \gamma_2 \sum_{i=1}^p \frac{h}{\phi_i^2} + \gamma_3 \quad (5)$$

where,  $\phi_i = S_{max}/S_i$ . The first term in Equation (5) accounts for the energy consumed in the original program in execution. The second term represents the overhead energy. This number is based on a rough estimate of the number of step transitions per *PMP*.  $\gamma_1$ ,  $\gamma_2$  and  $\gamma_3$  are technology dependent; constants that scale the normalized simulated speed (0-1) to the actual processor's speed eg. (200-700)MHz and (1.1- 1.65) V

The following formulas are for computing  $\phi_i$  for each of the described schemes.

## Proportional

$$\phi_i = \frac{p}{p-i+1} \prod_{k=1}^{i-1} \left[ 1 - \frac{\alpha}{p+1} \right] \quad (6)$$

## Dynamic Greedy

$$\phi_i = \frac{(1-\alpha)^i - 1}{\alpha} \quad (7)$$

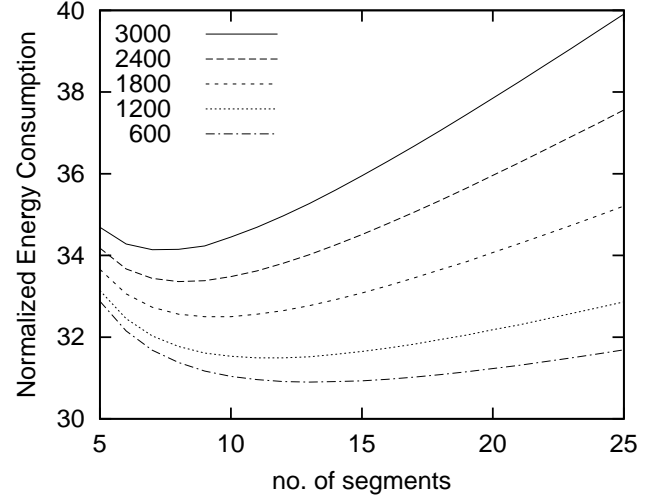
Due to the space limitation, the derivation of these formulas are not included in this paper.

## 4.1 Results

The optimal placement varies by varying several parameters in the execution behavior, such as variability of the execution  $\alpha$ , the amount of the imposed overhead for the speed changing. Figure (2) shows the effect of varying the overhead cycles,  $h$ , on the total consumed energy. The experiment runs at  $\alpha = 0.6$ . Optimal number of *PMPs* in the Proportional scheme lies in the range of 5-15, while the Dynamic Greedy from 10-30. As predicted, this optimal number decreases in case of increased overhead. This doesn't apply in case  $\alpha = 1$ , as when  $\alpha$  reaches 1 the desired optimum speed reaches the  $S_{max}$ , this implies that any extra work trying to run at lower speed will do the system worse energy wise, and eventually would force the application to miss its deadline. From now on, we will exclude this case from our experiments and interpretations.

We run some experiments for the Proportional and the Dynamic Greedy aiming to evaluate the theoretical results relative to the simulated ones.

Table (1) shows the best placement choice of the theoretical Proportional scheme as opposed to the simulation. The table shows the results for the same programs at different  $\alpha$  s and different overhead values that might describe different VVS processor architectures. The overhead values presented as a tuple of the theoretical overhead  $h$  and its corresponding simulated overhead  $F$  and  $G$ . For example (1000, 300) means we use  $h$  of 1000 cycles while  $F$  and  $G$  are both equal to 300 cycles. The correspondence between the two overheads (theoretical and simulated) comes from the



□

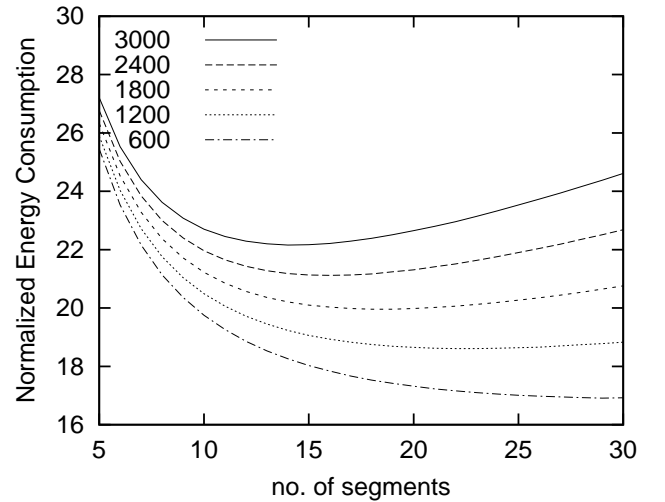


Figure 2: Total energy consumption for (a) the Proportional and (b) the Dynamic Greedy scheme versus the number of *PMPs* inserted in a program for different overhead cycles.

observation that in the Proportional scheme, the average number of transitions for the whole programs is nearly 2.2 ( $300 + 2.2 * 300 = 1000$ ). The table shows variations in the theoretical of  $\pm 2$  *PMPs* from the simulation. There is a strong matching in the  $\alpha$ 's middle range. The variations result from the assumption that the speed is continuous in the theoretical method while is it discrete in the simulation. Moreover, the simulation - as described by Transmeta [12] - is limited by a minimum speed to run at.

	(1000, 300)		(2000, 600)		(3000, 900)	
$\alpha$	T	S	T	S	T	S
0.2	10	12	7	7	6	7
0.4	12	12	9	9	7	6
0.6	12	12	9	9	7	6
0.8	11	9	8	6	7	5

Table 1: Theoretical (**T**) versus Simulation (**S**) choice of optimal number of *PMPs* for the Proportional scheme.

	(3000, 300)		(6000, 600)		(9000, 900)	
$\alpha$	T	S	T	S	T	S
0.2	29	28	20	20	16	15
0.4	22	21	14	14	11	12
0.6	14	12	10	11	8	9
0.8	9	9	7	6	5	6

Table 2: Theoretical (**T**) versus Simulation (**S**) choice of optimal number of *PMPs* for the Dynamic Greedy scheme.

During simulation, we noticed that on average the Dynamic Greedy performs step transitions three times more the Proportional scheme. From here comes the choice of  $h = 3000$  that corresponds to simulation overhead of  $F = G = 300$ . Table (2) shows that the optimal placement varies dramatically by varying  $\alpha$ . For example this variation at overhead (3000,300) ranges from 29-9 *PMPs* corresponding to  $\alpha$ 's range 0.2-0.8.

## 5 Conclusion

In VVS, accounting for overhead and the selection of a speed setting schemes are two faces for the same coin; there might be cases where the energy consumption exerted by the overhead of selecting and setting a new speed overshadows the energy savings of the speed setting algorithms, which imply that the system energy is harmed more than being benefited by employing such speed adjustments.

For relatively small program segments, it is essential to know the optimal number of adjustment

points or else we are not getting the best of the speed adjustment scheme. However for relatively large segments, it is effective to know the boundary for the minimum inserted *PMPs* as the energy curve will move flatter beyond the optimal number in large segments than for small segments.

## References

- [1] Daniel Mossé, Hakan Aydin, Bruce Childers and Rami Melhem. Compiler-Assisted Dynamic power-Aware Scheduling for Real-Time Applications. *Workshop on Compilers and Operating Systems for Low Power, COLP00*. 2000.
- [2] 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.
- [3] 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.
- [4] V. Gutnik and A. Chandrakasan. An Efficient Controller for Variable Supply Voltage Low Power Processing. *Symposium on VLSI Circuits*, pp.158-159, 1996.
- [5] 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.
- [6] 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.
- [7] Intel, Microsoft, and Toshiba. Advanced Configuration and Power Management Interface (ACPI) Specification. <http://www.intel.com/ial/WfM/design/pmdt/acpidesc.htm>, 1999.

- [8] 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.
- [9] 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.
- [10] 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
- [11] 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.
- [12] Transmeta Corporation, Crusoe Processor Specification, <http://www.transmeta.com>
- [13] Mark Weiser, Brent Welch, Alan Demers, Scott Shenker. Scheduling for Reduced CPU Energy. First Symposium on Operating Systems Design and Implementation (OSDI '94)
- [14] F. Yao, A. Demers and S. Shankar. A Scheduling Model for Reduced CPU Energy. *IEEE Annual Foundations of Computer Science*, pp. 374 - 382, 1995.
- [15] Thomas Burd, Trevor Pering, Anthony Stratakos, Robert Brodersen. Dynamic Voltage Scaled Microprocessor System. *International Solid-State Circuits Conference, ISSCC 2000*, pp.294-295, 2000.
- [16] Rex Min, Travis Furrer, and Anantha Chandrakasan. Dynamic Voltage Scaling Techniques for Distributed Microsensor Networks. *IEEE Computer Society VLSI Workshop*, 2000.
- [17] Trevor Pering, Thomas Burd, and Robert Brodersen. Voltage Scheduling in the IpARM Microprocessor System. *International Symposium on Low Power Electronics and Design 2000, ISLPED00*, pp.96-101, 2000.
- [18] Youngsoo Shin, Kiyoung Choi, and Takayasu Sakauri. Power Optimization of Real-time Embedded Systems on Variable Speed Processors. *IEEE/ACM International Conference on Computer Aided Design, ICCAD00*, pp.365-368, 2000.
- [19] Doug Burger, and Todd M. Austin. The SimpleScalar Tool Set, Ver 2.0 . *University of Wisconsin-Madison Computer Sciences Department Technical report no. 1342*, 1997.
- [20] Chung-Hsing Hsu, Ulrich Kremer, and Micheal Hsiac. Compiler-Directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessors. *International Symposium on Low Power Electronics and Design (ISLPED'01)*, August 2001.
- [21] H.Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Dynamic and Aggressive Power-Aware Scheduling Techniques for Hard Real-Time Systems. *To appear in 2001 IEEE Real-Time Systems Symposium (RTSS'01)*, December 2001.
- [22] E. G. Coffman and E. N. Gilbert. Optimal strategies for scheduling checkpoints and preventive maintenance. *IEEE Transactions on Reliability*, April 1990.
- [23] A. Duda. The effects of checkpointing on program execution time. *Information Processing Letters*, June 1983.
- [24] E. Gelenbe. On the optimum checkpoint interval. *Journal of the ACM*, April 1979.
- [25] C.-C. J. Li, E. M. Stewart, and W. K. Fuchs. Compiler-assisted full checkpointing. *Software Practice and Experience*, October 1994.
- [26] Avi Ziv, Jehoshua Bruck , An On-Line Algorithm for Checkpoint Placement. *IEEE Transactions on Computers*, 1997.