A Voltage Scheduling Heuristic for Real-Time Task Graphs

D. Roychowdhury[1], I. Koren[1], C.M. Krishna[1], and Y.-H. Lee[2]
[1]Department of Electrical and Computer Engineering
University of Massachusetts,Amherst, MA 01003
[2]Department of Computer Science
Arizona State University, Tempe, AZ 85287

Abstract

Energy constrained complex real-time systems are becoming increasingly important in defense, space, and consumer applications. In this paper, we present a sensible heuristic to address the problem of energy-efficient voltage scheduling of a hard real-time task graph with precedence constraints. Our algorithm uses only two voltage levels as it is a feature of many existing processors. We demonstrate that significant energy gains are possible even with only two voltage levels and our heuristic is as effective as those based on hypothetical systems having infinite voltage levels.

1 Introduction

In CMOS devices, the power consumption is proportional to the square of the voltage, while the circuit delay increases roughly linearly with the voltage. As a result, controlling the supply voltage allows the user to trade off workload execution time for energy consumption.

Over the past few years, many researchers have studied this tradeoff. For hard real-time systems, so-called because the workload is associated with a hard deadline, the tradeoff is particularly interesting. In such applications, the workload is characterized by analysis or profiling, so that its worst-case execution time can be bounded with reasonable certainty. In most cases, the workload is run periodically, with the period being known in advance. The task deadlines are also known in advance. Further, many real-time applications (e.g., spaceborne platforms) have constraints on their power or energy consumption. There is thus an increased need for power-management techniques for such systems; the *a priori* knowledge about the workload also provides an increased opportunity to use such techniques.

Most of the power-aware voltage-scheduling work for real-time systems has concentrated on independent tasks. By contrast, in this paper, we consider voltage scheduling while executing a task graph, which defines the precedence constraints between tasks.

The problem can be informally described as follows. We are given a task graph, the worst-case execution time of each task and the period at which the task graph is to be executed. This workload is to execute on a multiple-processor system, each processor of which has its own private memory. The problem is to allocate the tasks to processors, and to schedule the voltage of each processor in such a way that the energy consumption is kept low.

Our algorithm has both an offline and an online component. The offline component provides voltage scheduling based on the worst-case execution profiles. The online component adjusts the voltage schedule as tasks complete: in most cases, tasks consume less than their worst-case time, and this can be exploited to run the processors slower than might otherwise be required.

The remainder of this paper is organized as follows. In Section 2, we provide a brief survey of the relevant literature. In Section 3, we outline our algorithm and in Section 4, we provide some numerical results which serve to show its effectiveness. The paper concludes with a brief discussion in Section 5.

[0]This research has been supported in part by the National Science Foundation under grants EIA-0102696 and EIA-0102539.

2 Literature Survey

A good survey of system-level power optimization techniques, including voltage scheduling, can be found in [2]. A static voltage-control heuristic is presented in [8], and an integer programming approach is taken in [5]. In [5], the authors point out that two voltage levels are usually sufficient so long as these levels are properly chosen: not much is gained by having a larger number of levels.

Most papers in this area deal with independent tasks (see, for example, [1, 3, 4]) An initial study of tasks with precedence constraints has been made in [7]. In this paper, a static evaluation is carried out that defines the order in which the tasks are to be executed. This order is kept unchanged, even if the task execution times are much less than the worst-case. When tasks are completed ahead of their worst-case time, the slack that is thus released can be used by running the processor(s) at a lower voltage than would otherwise have been the case.

3 The Algorithm

The given task graph, henceforth referred to as the task precedence graph (*TPG*), is assumed to have a *hard deadline* associated with it. Therefore our algorithm tries to minimize energy expenditure by voltage scheduling in such a way that the deadline is always met.

3.1 System Model

Our system model consists of a multi-processor where each of the processors is independent and is connected by a low cost fast interconnection network. The processors can operate in 3 voltage levels - v_{HI}, v_{LO}, and v_{IDLE}. The v_{HI} and v_{LO} are the voltages where the processors can do useful computation whereas v_{IDLE} is the voltage necessary to sustain the system in idle state. From the energy perspective, when the processor is running at v_{HI} it can perform computation faster than when running at V_{LO} but consumes more power during its operation following these well known relationships:

The factor by which the processor is slower at voltage v relative to when at voltage v_{HI} is

$$slow(v) = \frac{v}{v_{HI}} \left(\frac{v_{HI} - v_T}{v - v_T} \right)^2 \quad (1)$$

where v_T is the threshold voltage.

We define one unit of computation as the computation performed by the processor at v_{HI} in unit time. Thus, one unit of computation will take $slow(v)$ units of time at voltage v. The ratio of power consumed by processor at voltage v relative to that of voltage v_{HI}

$$power\_ratio(v) = \left(\frac{v}{v_{HI}} \right)^2 \quad (2)$$

For the purpose of analysis we have neglected the energy cost of communication and idle processors. We have also considered the voltage switching cost as negligible both with respect to the time needed and the energy expended.This is justified by the fact that our algorithm has at most one voltage switch within the runtime of the task and at most one switch at the time of context switching of the tasks. Thus we can account for it by merging this effect into the worst case profile information.

3.2 The Algorithm

We follow a three-pronged approach to achieve our objective. The approach includes two offline components - effective assignment of the tasks to the finite number of processors available to us (this can modify the input TPG if the number of processors available is not enough to exploit the parallelism completely) and the voltage scheduling of the TPG based on the static worst case execution profile. We use the very pessimistic worst case execution profile approach because the system is a hard real-time and a deadline miss under any circumstances would be catastrophic. We follow with a online phase - the dynamic slack reclamation.

We will use the following terms for describing our algorithm. The *critical path* is a set of tasks from a source to a sink of the *TPG* that misses *deadline* under current voltage configuration. The *reverse slack* or *rslack* of a *critical path* is the difference between the *deadline* and the worst case execution time of that path with current voltage configuration. The *start-time* of the task is the time relative

to the beginning of the execution of the task set that a task must start, and *commit-time* is the time that a task must complete its execution under static scheduling.

The task assignment problem of a task graph on a finite number of processors is in general an NP-complete problem [9] and many heuristics have been put forward to address this issue. We follow a list scheduling heuristic that gives the highest preference to the tasks in the longest paths during the task assignment [10]. This will allow us to finish the execution of the entire task set as fast as we can if we ran everything in v_{HI}, and hence to exploit considerable *slack* before the *deadline* to slowdown substantial portion of tasks and run them in v_{LO} instead. In order to achieve this we had come up with a scheme for assigning *priorities* to the tasks by using the concept of the *top_level* and *bottom_level* for the tasks. We define *top_level* as the maximum of the sum of worst case execution units from any connected source of the *TPG* to the given task (excluding the execution units of the given task) and *bottom_level* as the maximum of the sum of the worst case execution units from the given task (including the execution units of the given task) to any connected leaf of the *TPG*. The *priority* of the task is the sum of *bottom_level* and *top_level*. Once we assign the priority we follow the greedy algorithm that whenever a processor is free and tasks are ready to run we assign the task with the highest priority to the processor.

After we get the modified *TPG* due to our resource constraints we are ready to apply our static voltage scheduling heuristics. In order to better understand this scheduling heuristic let us rephrase the issue as the following optimization problem. Let S_i denote the speedup associated with each task i. For each path P_k, we have to satisfy the constraint

$$\sum_{j \in P_k} S_j \geq t_k - D$$

where task j belongs to path P_k,t_k is the worst case execution time of the path P_k without any speedups and D is the *deadline* associated with the TPG. Our objective is to minimize $\sum_{i=1}^{n} S_i$ where n is total number of tasks in the task set.

There is a trivial solution for this problem if the *deadline* is met when all the tasks are run at v_{LO}; that is, we do not need any speedups. However, the problem becomes more interesting when some of the paths become *critical paths* and a decision has to be made about which task to speed up. Based on the equations above it appears that if we speed up a task that is present in more critical paths than any other task, then we would affect many paths while paying the energy price only once. Based on this intuition we formulate the following iterative algorithm to figure out which task needs to run at v_{HI} and for how many execution units. We start the algorithm by assigning all the tasks to run at v_{LO} and then speeding them up iteratively until there are no more *critical paths* left. The weight associated with each task is dependent on the membership of the task in the set of critical paths, every time we encounter a task in the critical path we increment its weight by 1. When we have to break a tie between tasks of equal weight we choose the task nearest to the leaf of the *TPG*, that is the one having lower value of *bottom_level*. The rationale behind this is that we would like to schedule a task to run at v_{HI} as late as we can because during dynamic resource reclamation, we could potentially re-acquire enough slack to avoid

Algorithm 1 Static Voltage Scheduling

while list of critical paths not empty **do**
  assign weights to the tasks
  taskId = choose task with maximum weight and if more than one task have the same maximum weight choose the one with minimum *bottomlevel* value.
  pathId = choose the path with minimum *rslack* among all the critical paths having *taskId* as a member task.
  speed up taskId using the following scheme:
  if *rslack* can be covered by changing units from v_{LO} to v_{HI} **then**
    change the appropriate units of *taskId* to run them at v_{HI} instead of v_{LO}
  else
    run the entire *taskId* at v_{HI} and mark the task so that its weight is never considered during subsequent iterations.
  end if
  update the path execution times and remove any path which now meets the *deadline* from the list of critical paths.
end while

having to run it at v_{HI} altogether.

Once we have the static scheduling of the paths we can assign the *start time* and the *commit time* of the individual tasks. Since the static analysis was based on the worst case execution profile, the tasks will always finish before or at the *commit time* during actual runtime. Thus, its successor can begin execution earlier if it has no other pending dependencies and we can use this extra *slack* between *start time* and the current time to slow down the processor further under the constraint that this task still finishes at its *commit time* even if running at worst case profile. This would result in further energy savings.

We now provide an illustration to demonstrate our algorithm. The example graph is shown in Figure 1. The number inside the circle represents the task identity while the two numbers on the side are the worst case execution units (in bold) and the actual execution units at runtime. For this case v_{HI} is chosen at 3.3V and v_{LO} at 2V.

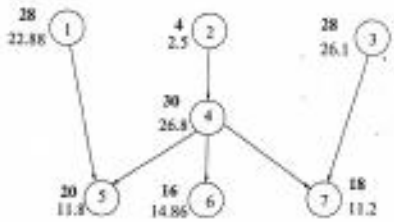

Figure 1: An example task graph with execution times in terms of v_{HI}.

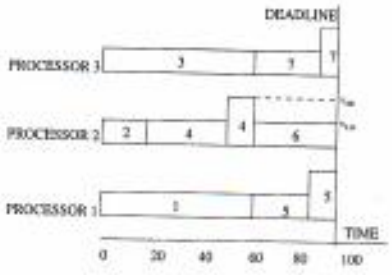

Figure 2: The Gantt chart showing static scheduling.

We run this task graph in a system with three processors. The processor assignment following our heuristic keeps the graph unchanged in this case. We then apply the static voltage heuristic to the graph. During the first iteration, both the tasks 2 and 4 have the maximum weight of 3, and we choose task 4 since it is nearer to the leaf of TPG and speed it up appropriately to make the path (with minimum *rslack*) consisting of tasks 2, 4 and 6 meet its deadline. We remove this path from the list of critical paths and proceed with our algorithm. In the next iteration the weights of tasks 2,4,5 and 7 are all 2. We then choose task 7 and speed it up such that the path consisting of tasks 3 and 7 meets its deadline. We continue this iterative procedure until finally we come up with the static schedule as shown in Figure 2. v_{HI} is represented by a greater height in this gantt chart. We then do dynamic resource reclamation to reclaim any slack that occurs in runtime. Figure 3 shows the effect of dynamic resource reclamation on our static algorithm.

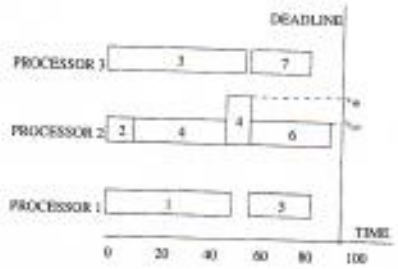

Figure 3: The Gantt chart showing actual behavior of tasks at run time.

4 Numerical Results

We have performed extensive experimentation with the algorithm described in the previous section and describe here, for brevity, only our experiences with two real-life applications. The first is a task graph for a Newton-Euler dynamic control calculation of a robotic application for six degrees of freedom, henceforth referred to as the *robot control*, while the other is a task graph for a random sparse matrix solver of electronic circuit simulation using symbolic generation technique, henceforth referred to as *sparse matrix*. The *robot control* has 88 tasks and the *sparse matrix* has 96 tasks. These task graphs have been published by Kasahara Lab [11], and the timings are based on actual profiling done on their *OSCAR* multiprocessor system.

We first compare the energy gain that we get when our scheduling method is followed with respect to a system where there is no voltage

scheduling: that is, all tasks have to run in a predefined v_{HI}. Analyzing Figure 4 we find that our algorithm gives considerable energy gains. All these graphs saturate at around 17 percent. This is because the voltage range has been chosen between 3.3V and 2V. It can be shown analytically that the maximum energy savings that we can achieve is around 17.2 percent under this voltage range. As the variance of the tasks increases, we see that we can get increasing gain from the algorithm due to the increasing *slack* that we can exploit at runtime. But even in the case of worst-case execution, the plots demonstrate that considerable energy gain can be made because of the novel static algorithm we are using. Similarly, when we vary the number of processors, we can exploit the parallelism more and hence have better performance with increasing number of processors (see Figure 5). However, once we have exploited the parallelism, there is little more gain to be had from more resources (see Figure 6) where 12 is the maximum number of processors which can be efficiently used.

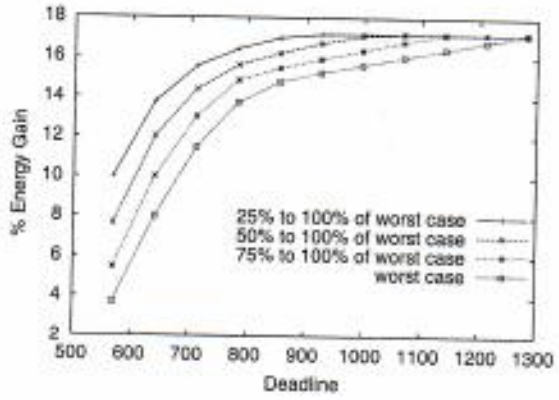

Figure 4: Energy Gain after runtime adjustments for the *robot control* with differing variance in execution time (for 12 processor system).

The plots in Figure 7 show the gain achieved by dynamic resource reclamation over the static scheduling. As predicted we can see that the higher the variance in execution time, the better is the performance. Since our adjustment is fast and happens only during context switch, we can have substantial gain with relatively little overhead.
Next we compare our algorithm with a dynamic voltage adjustment algorithm that chooses from infinite voltage levels [7] (see Figure 8). Here we

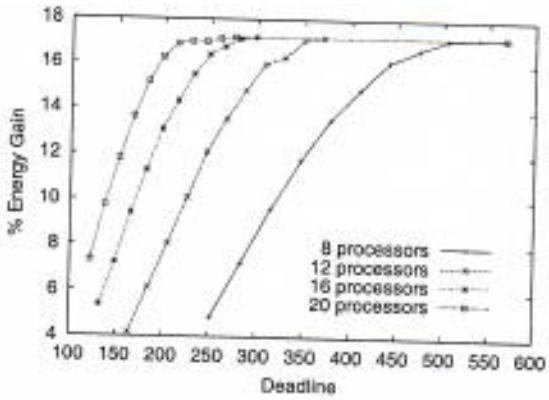

Figure 5: Energy Gain after runtime adjustments for the *sparse matrix* with varying number of processors.

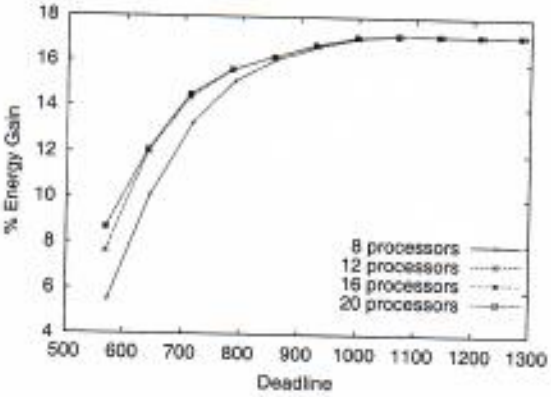

Figure 6: Energy Gain after runtime adjustments for the *robot control* varying the number of processors.

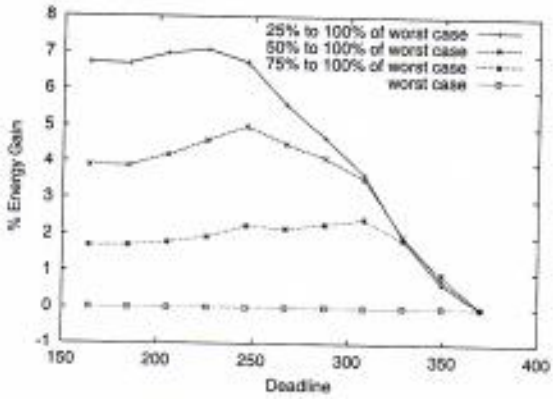

Figure 7: Energy Gain due to dynamic resource reclamation for *sparse matrix* (12 processor system).

relax the constraint that the v_{HI} has to be fixed to a particular value and instead allow it to have any value in the voltage range specified. v_{HI} for the subsequent experiments is chosen as the minimum uniform voltage in which the tasks can execute so that the longest path meets the *deadline*

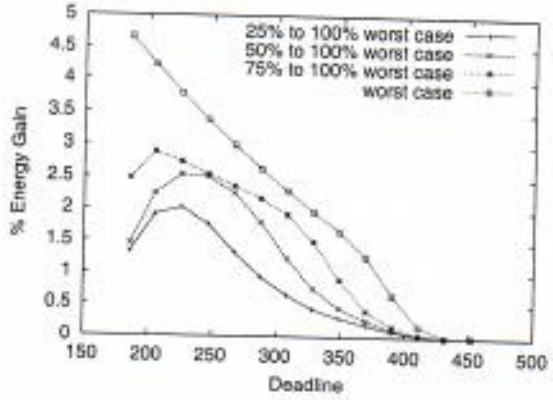

Figure 8: Comparison with Infinite Voltage Algorithm for *sparse matrix* (12 processor system).

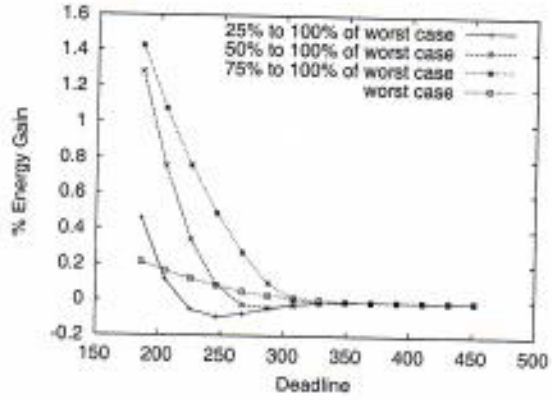

Figure 9: Comparison when infinite voltage level selection is used instead of dual voltage in our algorithm for *sparse matrix* (12 processor system).

under worst case scenario. Our two voltage algorithm actually outperforms the infinite-voltage algorithm in most cases. Finally, we measure the energy gain if we would have used a infinite voltage selection for our algorithm instead of two voltage levels (see Figure 9). We see that energy gain is less than 2% for most of our experiments. Thus, we claim that our algorithm can be overlaid in processors already existing using only software enhancements and would be at least as much energy efficient as the hypothetical systems with infinite voltage levels.

5 Discussion

In this paper we have considered the problem of an energy efficient voltage scheduling heuristic for task graphs having precedence constraints. We have described a three-pronged approach to solve this problem and have demonstrated that considerable energy savings can be achieved by considering the relationships among the tasks in the set. We have also considered a practical scenario where we have only two voltage levels instead of infinite voltage levels and demonstrated that our algorithm outperforms or is at least as good as the infinite voltage level algorithms in the majority of cases. We acknowledge that we have ignored the energy issues associated with the communication in the distributed framework and keep this as a future work to be taken up in due course.

References

[1] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez, "Determining optimal processor speeds for periodic real-time tasks with different power characteristics," *Euromicro Conference on Real-Time Systems*, 2001.

[2] L. Benini and G. De Micheli, "System-level power optimization: techniques and tools," *ACM Trans. Design Automation for Electronic Systems*, Vol. 5, April 2000, pp. 115–192.

[3] Y. Do, Y.-H. Lee, and C. M. Krishna, "EDF Scheduling Using Two-mode voltage-clock-scaling for hard real-time systems," *Int'l Conf on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

[4] I. Hong, G. Qu, M. Potkonjak, and M. Srivastava, "Synthesis techniques for low-power hard real-time systems on variable voltage processors," *Real-Time Systems Symposium*, 1998.

[5] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," *ACM ISLPED*, pp. 197–199, 1988.

[6] C. M. Krishna and Y.-H. Lee, "Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems," *Real-Time Applications Symposium*, 2000.

[7] D. Zhu, R. Melhem, and B. Childers, "Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems," *Real-Time Systems Symposium*, 2001.

[8] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," *Proc. 36th IEEE Symp. Foundations of Computer Science*, 1995, pp. 374–382.

[9] J.A.Hoogeveen, S.L.van de Velde, and B.Veltman, "Complexity of scheduling multiprocessor tasks with prespecified processor allocations," *CWI, Report BS-R9211, Netherlands*, 1992.

[10] T. Yang and A. Gerasoulis, "List Scheduling With and Without Communication Delays," *Parallel Computing 19(12)*, 1993, pp. 1321-1344

[11] http://www.kasahara.elec.waseda.ac.jp/schedule/old4/apply_pe.html