

A Voltage Scheduling Heuristic for Real-Time Task Graphs

D. Roychowdhury, I. Koren, C.M. Krishna
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
droychow,koren,krishna@ecs.umass.edu

Y.-H.Lee
Department of Computer Science
Arizona State University, Tempe, AZ 85287
lee@cs.asu.edu

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 for a multi-processor environment. We show that consideration of inter-relationships among the tasks in a holistic way can lead to an effective heuristic for reducing energy expenditure. We developed this algorithm for systems running with two voltage levels since this is currently supported by a majority of modern processors. We then extend the algorithm for processors that can support multiple voltage levels. The results show that substantial energy savings can be achieved by using our scheme. The algorithm is then compared with other relevant algorithms derived for hypothetical systems which can run on infinite voltage levels in a given range. Our two voltage systems, using the task dependencies effectively, can provide a comparable performance with those algorithms in the cases where continuous voltage switching is not allowed.

1. Introduction

In CMOS devices, energy consumption per cycle is proportional to the square of the voltage, while circuit delay decreases 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 challenging. In such applica-

tions, 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 and deadlines known in advance. Furthermore, 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 provides an added 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 task assignment in the processors and the order of the algorithm is also determined a priori and serves as an input parameter. The problem is to schedule the voltage of each processor in such a way that the energy consumption is kept low. Assuming the deadline of the task graph equals its period, there will be no more than one task iteration alive in the system at any time.

Our algorithm has both an offline and an online component. The offline component performs voltage scheduling based on the worst-case execution requirements. The online component adjusts the voltage schedule as tasks complete: in most cases, tasks consume less than their worst-case time, and this time can be reclaimed 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.

2. Literature Survey

A good survey of system-level power optimization techniques, including voltage scheduling, can be found in [5]. Initial work on dynamic voltage scheduling (DVS) [8, 23] was in the context of non-real-time systems where the average throughput is the performance metric. A static voltage-control heuristic is presented in [26], and an integer programming approach is taken in [14]. In [14], the authors point out that two voltage levels are usually sufficient as long as these levels are properly chosen; not much is gained by having a larger number of levels. Researchers have also proposed the concept of compiler directed DVS [17] where the compiler sets the processor frequency and voltage with the aim of minimizing energy under real-time constraints. There have been several other techniques proposed for energy efficient real-time task scheduling. Most papers in this area deal with independent tasks (see, for example, [6, 11, 12, 15, 18, 19, 22]).

In [10], the scheduling problem of independent hard real-time tasks with fixed priorities assigned in a rate monotonic or deadline monotonic manner is addressed. This method employs stochastic data to derive energy-efficient schedules taking the actual behavior of the real time systems into account. Several papers have also recognized the need for both offline and online approaches to address the issue of energy efficient scheduling of independent real-time tasks (see for example, [4, 20]).

Variable voltage scheduling as a low power design technique at the behavioral synthesis stage is discussed in [21]. Given as input an unscheduled data flow graph with a timing constraint, the goal of this paper is to establish a voltage value at which each of the operations of the data flow graph would be performed while meeting its timing constraint. The authors have used an iterative graph-theoretic approach to identify critical paths and assign nodes to a specific voltage level.

An interesting approach to power-conscious joint scheduling of periodic task graphs and aperiodic tasks in a distributed real-time embedded system has been proposed in [16]. Here the authors focus on the problem of effective scheduling of a mix of task graphs and independent tasks and an effective dynamic energy reduction heuristic for them. They use a slack-based list scheduling approach to perform static resource allocation, assignment and scheduling of the periodic task graphs. The emphasis of this work is to meet all hard real-time constraints, minimize

the response time of all soft aperiodic tasks and also to engage in dynamic voltage scaling and power management to reduce energy consumption. It is assumed in [16] that tasks always run to their worst-case execution times.

An initial study of tasks with precedence constraints has been made in [27]. 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 be required.

By contrast, in this paper, we recognize that considering precedence relationships among the tasks and the entire shape of the task graph might help us in deriving an algorithm that would achieve significant energy savings. We also believe that an effective technique should comprise both online and offline components and hence we design our offline heuristics with the online component in mind. We focus entirely on the computation aspect of this problem in a multiprocessor system and try to come up with a comprehensive solution using static scheduling and runtime strategies to achieve energy efficient scheduling of hard real-time task graphs.

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 reduce energy expenditure by voltage scheduling in such a way that the deadline is always met.

In CMOS devices, the power consumption is proportional to the square of the voltage [7, 14]:

$$P_{CMOS} = C_L N_{SW} v_{DD}^2 f \quad (1)$$

where C_L is the circuit output load capacitance, N_{SW} is the number of switches per clock cycle, f is the clock frequency and v_{DD} is the supply voltage. However, reduction of power supply voltage causes increase of the circuit delay denoted by δ [7, 14]:

$$\delta = \frac{C_L v_{DD}}{K(v_{DD} - v_T)^\alpha} \quad (2)$$

where K is a constant depending on the process and gate size, v_T is the threshold voltage, and α varies between 1 and 2; $\alpha = 2$ for long channel devices which have no velocity saturation. In this paper, we assume $\alpha = 2$ for all our numerical experiments.

3.1 System Model

Our model consists of a multi-processor system where each of the processors is independent and is connected by

a low cost fast interconnection network. The processors can operate in three voltage levels: v_{HI} , v_{LO} , and v_{IDLE} ($v_{HI} > v_{LO} > v_{IDLE}$). v_{HI} and v_{LO} are voltages at which the processors can do useful computation whereas v_{IDLE} is the voltage necessary to sustain the system in idle state.

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

$$\text{slow}(v) = \frac{v}{v_{HI}} \left(\frac{v_{HI} - v_T}{v - v_T} \right)^2 \quad (3)$$

where v_T is the threshold voltage.

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

$$\text{energy - ratio_per_cycle}(v) = \left(\frac{v}{v_{HI}} \right)^2 \quad (4)$$

For the task-sets we study, inter-task communication only happens after each task has finished its computation. Such communication consists of transfer of small amounts of data; the communication cost can be safely ignored as insignificant in the types of application under consideration. Since idle(sleep) power consumption is considerably lower than operational power [1], the energy cost when the processors are idle is also ignored. We should also note that our algorithm actually decreases the idle time in the processors compared with the single voltage algorithm. Hence this is a conservative assumption since the relative gain using our algorithm would be even higher if we had considered the energy cost associated with the 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. We have accounted for this cost by merging this effect into the worst case profile information.

3.2. The Details of the Algorithm

Given the tasks we can apply any static task assignment and ordering heuristic. Any generic multiprocessor static task assignment algorithm can be followed. The task graph under this assignment should be able to meet the deadline if running to their worst case under the highest available voltage. Any assignment that satisfies the above criterion can act as a valid input to our algorithm. Once we are given the assignment and task schedule order in the multiprocessor environment, we can apply our voltage scheduling heuristic to minimize the energy expenditure. We follow

a two-pronged approach to achieve our objective. The approach includes an offline component - the voltage scheduling of the TPG based on the static worst-case execution profile. We use the pessimistic worst case execution profile approach because the system has hard real-time requirements and a deadline miss under any circumstances would be catastrophic. We follow with an online, the dynamic slack reclamation, phase.

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 the *deadline* under the 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 the current voltage configuration. We define the amount of computation done in v_{HI} in unit time as one execution unit. The *start-time* of a task is the latest time, relative to the beginning of the execution of the task graph, at which the particular task must be invoked, and *commit-time* is the time by which the task must complete its execution.

3.2.1. Static Voltage Scheduling

We apply any static assignment heuristic, such as the one described in Section 3.2.3, to the TPG. In the cases where there are not enough processors available to exploit the parallelism inherent in the TPG, this assignment and task ordering may lead to new dependency relationships. We would, therefore, need to modify the original TPG by adding new edges to accommodate these dependencies. After the assignments and modifications are done, we apply our static voltage scheduling heuristics to the TPG. In order to better understand this scheduling heuristic let us rephrase the issue as the following optimization problem. Let S_i denote the speedup in time associated with each task i . Speedup can be explained as the difference in execution time of a task when running under our voltage schedule and running it entirely in v_{LO} . For each path P_k , we have to satisfy the constraint

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

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_k is the *deadline* associated with the path P_k . Our objective is to minimize $\sum_{i=1}^n S_i$ where n is the total number of tasks in the task set. This objective function implies that we are trying to minimize the amount of time needed for the tasks to run in v_{HI} which in turn means minimizing overall energy expenditure.

There is a trivial solution for this problem if the *deadline* is met when all the tasks are run at v_{LO} . 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. Analyzing the expressions above,

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 has the same maximum weight
  choose the one with minimum bottom Level 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 it-
    erations.
  end if
  Update the path execution times and remove any path
  which now meets the deadline from the list of critical
  paths.
end while

```

it appears that if we speed up a task that is part of a large number of critical paths, we affect many paths while paying the energy price only once. Based on this intuition we formulated the following iterative algorithm to determine which task needs to run at v_{HI} and for how many execution units. We start the procedure by assigning all the tasks to run at v_{LO} and then speed 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*. 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 having to run it entirely at v_{HI} . Note that we could formulate our problem as a linear programming optimization. This, however, would yield an optimal static scheduling which would not attempt to increase the opportunity for dynamic adjustments. We still have experimented with linear programming techniques and the overall results tend to match closely those of our static heuristic. Our static heuristic appears to give us near-optimal performance in most cases as well as facilitating the dynamic resource reclamation in the subsequent step.

3.2.2. Dynamic Resource Reclamation

Once we have completed the static scheduling of the paths, we can assign *start time* and *commit time* to the individual tasks. Since the static analysis was based on the worst case execution profile, each task will finish before or at its *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 it runs to its worst-case execution profile. The following equation calculates the amount of units of execution to be transferred from running at v_{HI} to running at v_{LO} (given that it takes unit time to execute one unit in v_{HI}):

$$\text{units_to_Lo} = \frac{\text{start_time} - \text{current_time}}{\text{slow}(v) - 1} \quad (5)$$

where *units_to_Lo* is the additional number of units of the task that should be transferred to the portion executed in v_{LO} , *current_time* is the current time and *start_time* is the start time predicted during the static voltage scheduling based on the worst case execution profile of the tasks. This transfer of certain units of execution from v_{HI} to v_{LO} results in further energy savings.

The next subsection first explains the algorithm, and then illustrates it through an example *TPG* on which the whole procedure is performed. For this example, we used a list scheduling heuristic as our static assignment algorithm.

3.2.3. An Example Static Assignment Scheme

The assignment problem of a task graph to a finite number of processors is, in general, an NP-complete problem [13] and many heuristics have been proposed to address it. Any of these assignments can be used in conjunction with our algorithm as long as the assignment makes the real time task graph feasible under the worst case profile. However, the task assignment heuristic that is followed would have an impact on the effectiveness of our voltage scheduling heuristic. The faster the entire task graph can be completed, the higher will be the effectiveness of our algorithm. We employ a list scheduling heuristic adapted from [25] as an example of how task assignment can be done. We should emphasize here that other assignment heuristics would also work with our algorithm. For example, assignments using genetic algorithms, such as [9], or assignment using simulated annealing, such as [24], can be combined with our algorithm.

The list scheduling heuristic we consider gives the highest priority to the tasks in the longest paths during the task assignment [25]. For a particular fixed voltage, this heuristic allows us to finish the execution of the entire task set in

the least amount of time for most cases. Hence, the scope for exploiting the slack is likely to be high if we use this assignment. If this algorithm does not meet the deadline criteria of the task graph under worst case, we have to use another task assignment heuristic.

The assignment heuristic is based on the concept of assigning *priorities* to 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 the 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 do an offline analysis using a greedy list-scheduling algorithm to assign tasks to each of the processors and to determine the order of their execution. The heuristic we follow is that whenever we find a free slot in a processor and tasks are ready to run, we assign the ready task with the highest priority to that processor.

3.2.4. An Example Taskgraph

We now provide an example to illustrate our algorithm. The example graph is shown in Figure 1. The number inside the circle represents the task number while the two numbers on the side are the worst case execution units (in bold) and the actual execution units at runtime for some execution instance, respectively. For this example v_{HI} is chosen at 3.3V and v_{LO} at 2V. The deadline of the execution of the task graph is chosen as 99.

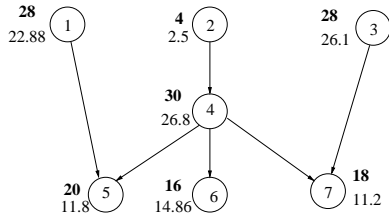


Figure 1. An example task graph with execution times in terms of v_{HI} .

We execute this task graph in a system with three processors. The processor assignment following our heuristic keeps the graph unchanged in this case. The priority calculation is demonstrated in Table 1 which shows the calculated values of the *top_Level* and *bottom_Level* of each task. The priority is determined as the sum of these two parameters. The task assignment and ordering are shown in Figure 2.

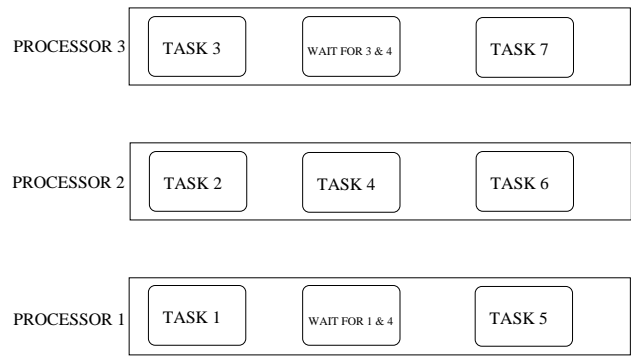


Figure 2. Static Task Assignment and Ordering.

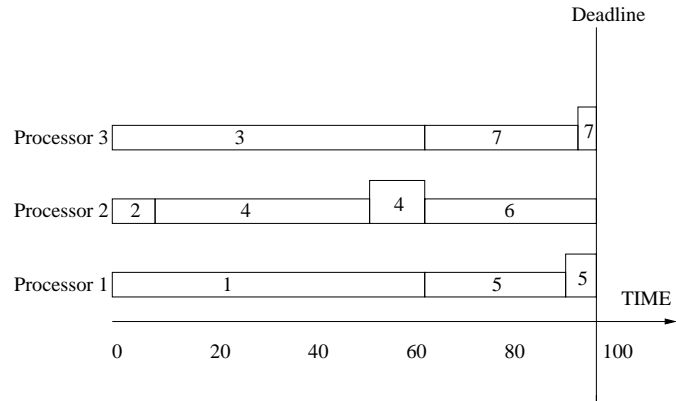


Figure 3. The Gantt chart showing static scheduling.

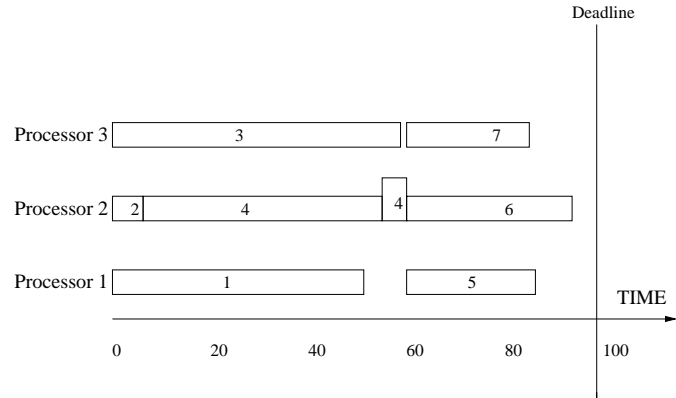


Figure 4. The Gantt chart showing actual behavior of tasks at run time.

We then apply the static voltage heuristic to the graph. During the first iteration, both tasks 2 and 4 have the maximum weight of 3. We choose task 4 since it is nearer to the leaf of the *TPG* and speed it up appropriately to make the path (with minimum *slack*) consisting of tasks 2, 4 and 6 meet its deadline. We remove this path from the list of

task ID	top_Level	bottom_Level	task ID	top_Level	bottom_Level
1	0	48	2	0	54
3	0	46	4	4	50
5	34	20	6	34	16
7	34	18			

Table 1. The priority parameters for the example *TPG*.

Step No.	Path chosen	weight of task1	weight of task2	weight of task3	weight of task4	weight of task5	weight of task6	weight of task7	task chosen
1	2-4-6	1	3	1	3	2	1	2	4
2	3-7	1	2	1	2	2	0	2	7
3	1-5	1	2	0	2	2	0	1	5
4	2-4-7	0	2	0	2	1	0	1	4
5	2-4-5	0	1	0	1	1	0	0	5
		0	0	0	0	0	0	0	

Table 2. The iterative steps of algorithm for the example *TPG*

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. Table 2 describes the individual iterations in detail. The second column depicts the path chosen for speeding up, the middle columns show the weights of the individual tasks in this step of the algorithm, and the last column shows the task that was selected for speeding up. We continue this iterative procedure until finally we obtain the static schedule shown in Figure 3. 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. Let us now look at task 4 and see how runtime variations affect the scheduling. After the static scheduling, the *start time* of task 4 is 9 and the *commit time* is 63 and it has been scheduled to execute 19.2 units in v_{LO} and 10.8 units in v_{HI} . However, its preceding task, task 2, did not take its worst case time to execute and finished instead at time 5.625. So now task 4 could be started at 5.625 instead of at 9 and we can use this extra time to slow it down further such that its worst-case commit time still remains at 63. Thus, at the time of invocation it is scheduled to execute 21.9 units in v_{LO} and 8.1 units in v_{HI} . Figure 4 shows the effect of dynamic resource reclamation on our static algorithm. In addition, we have used the simplex method to solve the corresponding linear programming optimization problem and found the speedups required by different tasks under their worst case profile. The simplex method yielded the same sum of speedups as our algorithm. However, the distribution of the speedups was quite different. For example, if we did static scheduling following the simplex method's solution we would schedule the entire task 2 at the highest voltage, v_{HI} . This would

have led to inefficient use of the slack resulting from actual execution time being less than the worst case.

Even though we have shown an example with a single task graph, the algorithm can be used for multiple task graphs as long as they have the same period. After their assignment, the multiple task graphs can be cast as a single task graph and the same algorithm can be applied to achieve our objective.

3.3. Extension to a Multi-Voltage System

The algorithm described above has been created keeping in mind processors supporting just 2 voltage algorithms. This can be easily extended to processors running under multiple voltage levels. The extension can be described as follows: We can use the same scheduling algorithm and assign *start time* and *commit time* to the individual tasks. Once we fix the interval, we can find an unique voltage level which can finish the task in that interval without any voltage switching. Once the voltage level is calculated, we can choose the two voltage levels that the processor supports between which the calculated voltage level lies. We then run the task in the two chosen voltage levels such that the task finishes exactly at *commit time* when running under worst case.

4. Numerical Results

We have performed extensive simulation experiments of the algorithm described in the previous section. We present results from one real-life application and from some random task graphs. The application is a task graph for a random

	Voltage (V)	Frequency (MHz)
1	1.75	1000
2	1.40	800
3	1.20	600
4	1.00	466

Table 3. The Voltages and corresponding Frequencies Intel xScale supports.

sparse matrix solver of electronic circuit simulation using the symbolic generation technique, henceforth referred to as *sparse matrix*. The *sparse matrix* has 96 tasks. These task graphs have been published by the Kasahara Lab [3], and the timings are based on actual profiling done on the OSCAR multiprocessor system. The processors have been modeled based on technology used for *Intel xScale* processors [2].

The parameters used in the simulations are $v_{HI} = 1.75V$, $v_{LO} = 1.0V$, and $v_T = 0.2V$ unless specified otherwise. Using these values in (3) and (4), the maximum energy savings possible is about 67.34% if everything can be run at v_{LO} . The execution units of the individual tasks are uniformly distributed in the range $[A, 100]\%$ of their worst case profile. We have varied A in the simulations and the results are presented below. We first compare the energy savings that we get when our scheduling method is followed, with a system where there is no voltage scheduling: that is, all tasks have to run in a predefined v_{HI} . The results are shown in Figure 5: our algorithm yields considerable energy savings. As the variance of the tasks' execution times increases, we see that we can get increasing savings from the algorithm due to the increasing *slack* that we can exploit at runtime. Yet, even in the case of worst-case execution ($A=100$), the plots demonstrate that significant energy savings can be achieved because of our static algorithm. Similarly, when we vary the number of processors, we can exploit the parallelism more and hence have better performance with an increasing number of processors (see Figure 6).

The plots in Figure 7 show the savings achieved by dynamic resource reclamation over the static scheduling. As predicted, we can see that the greater the variance in execution time, the better the performance. Since our adjustment is fast and happens only during context switch, we achieve substantial savings with relatively little overhead.

We now present results for random graphs to show the effectiveness of our algorithms. We chose 60 randomly generated graphs, each consisting of 50 tasks. For each task graph we calculated the deadline necessary to schedule the task graph under v_{HI} and D_{MIN} and to schedule it under v_{LO} and D_{MAX} . We varied the deadlines D from 0 to 10 such that $D = 2$ means $D = D_{MIN} + 2 * \frac{D_{MAX} - D_{MIN}}{10}$. For each deadline D we calculated the average relative gain and presented the results in Figure 8. The graph shows that the

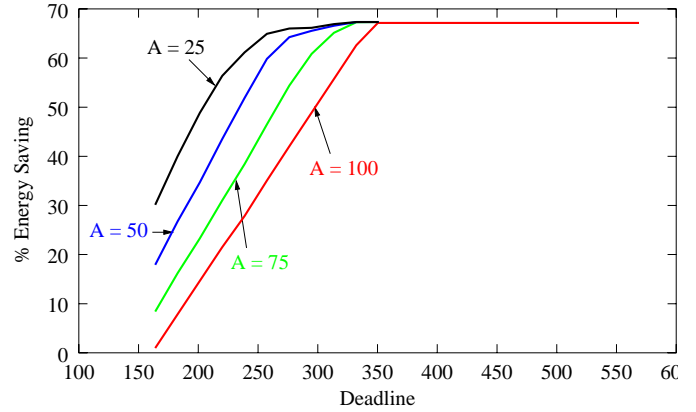


Figure 5. Energy savings after runtime adjustments for the *sparse matrix* with differing variance in execution time (for 12 processor system).

algorithm performs well for a large variety of task graphs.

Next we compared our algorithm with a dynamic voltage adjustment algorithm, referred to as LSSR-N [27], that chooses from an infinite number of voltage levels (see Figure 9). For this infinite level algorithm, voltage adjustments have been considered only at the time of context switches. Here we relaxed the constraint that the v_{HI} has to be fixed at a particular value and instead allowed it to have any value in the voltage range specified. v_{HI} for the subsequent experiments was chosen as the minimum uniform voltage at which the tasks can execute so that the longest path meets the deadline under the worst case scenario. Our two-voltage-level algorithm actually outperformed this infinite-voltage algorithm in most cases. This shows that considering the overall structure of the task graph at the time of voltage scheduling does provide substantial benefits over dynamic slack sharing heuristics.

Finally, we measured the energy savings if we had used multiple voltage levels for our algorithm instead of two voltage levels (see Figure 10). The system supporting multiple voltage levels, as expected, exhibited higher energy savings than the two-voltage level system. The processors were modeled after Intel xScale technology. The multiple voltage algorithm chose voltage from any voltage level supported, see Table 3, while the dual-voltage scheme used

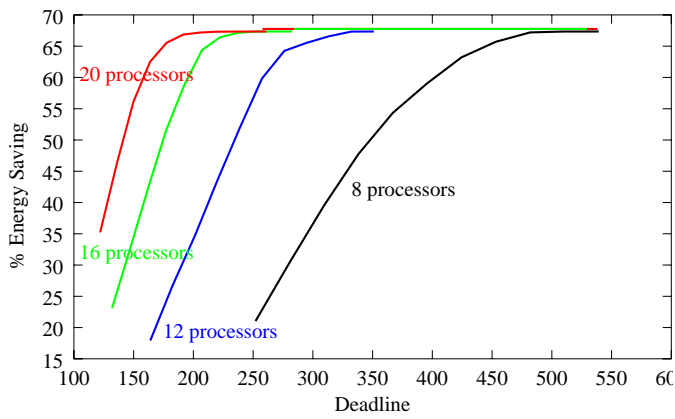


Figure 6. Energy savings after runtime adjustments for the *sparse matrix* with varying number of processors.

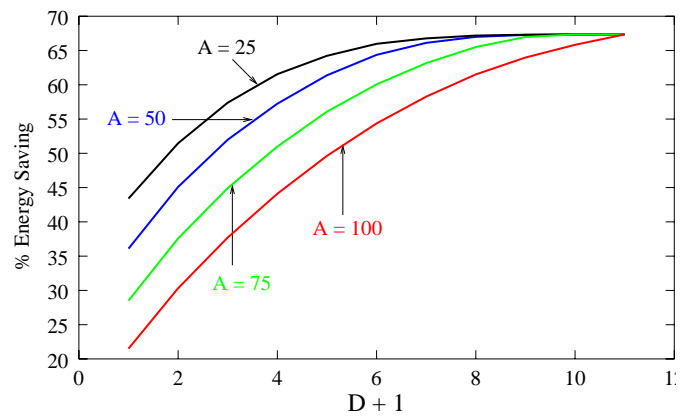


Figure 8. Average Energy savings after runtime adjustments for the random graphs with differing variance in execution time (for 12 processor system).

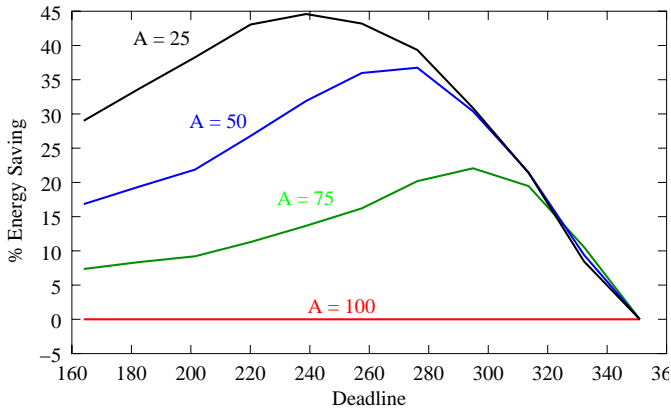


Figure 7. Energy savings due to dynamic resource reclamation for *sparse matrix* (12 processor system).

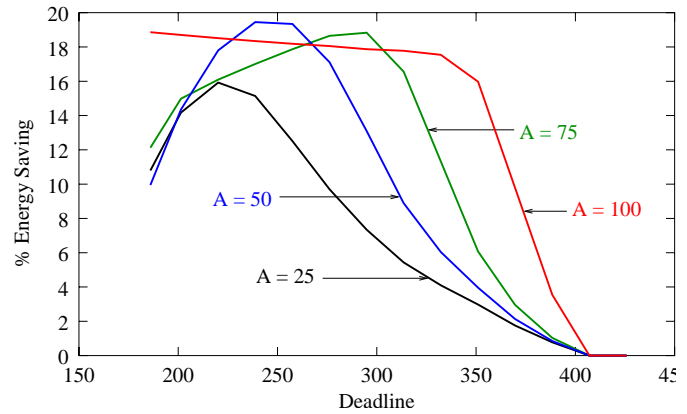


Figure 9. Comparison with Infinite Voltage Algorithm [27] for *sparse matrix* (12 processor system).

$v_{HI} = 1.75V$ and $v_{LO} = 1.0V$. The results demonstrate that our algorithm can be easily incorporated to satisfy more complicated system configurations to achieve superior performance. They also show that when deadlines are more relaxed, the effect of the multiple voltage levels diminishes significantly.

5. Discussion

In this paper we have considered the problem of an energy efficient voltage scheduling heuristic for task graphs with precedence constraints. We have described a two-pronged approach to solve this problem and have demonstrated that substantial energy savings can be achieved by considering the relationships among the tasks in the graph. The focus of this paper has been on exploiting the structure of task graphs for energy minimization purposes. The volt-

age scheduling proposed needs at most one voltage switching, and is therefore a relatively low overhead algorithm. We have presented a simple, low overhead voltage scheduling heuristic for executing task graphs in an energy efficient way.

References

- [1] <http://developer.intel.com/design/pca/>.
- [2] <http://www.intel.com/design/intelxscale/>.
- [3] <http://www.kasahara.elec.waseda.ac.jp/schedule/>.
- [4] Aydin H., Melhem R., Mosse D., and Alvarez P.M. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, June 2001.
- [5] Benini L. and De Micheli G. System-level power optimization: techniques and tools. In *ACM Trans. Design Automa-*

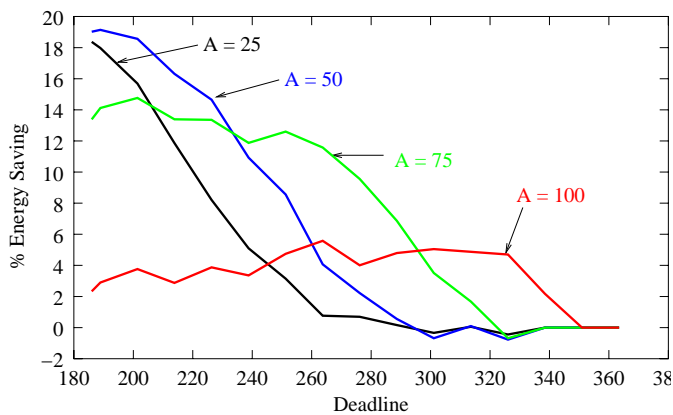


Figure 10. Comparison showing how much more energy savings is possible using multiple voltage level selection is used instead of dual voltage in our algorithm for *sparse matrix* (12 processor system).

tion for Electronic Systems, pages 115–192, Vol. 5, April 2000.

[6] Burd T.D., Pering T.A., Stratakos A.J., and Brodersen R.W. A dynamic voltage scaled microprocessor system. In *IEEE Journal of Solid-State Circuits*, pages 1571–1580, Vol. 35, Iss. 11, November 2000.

[7] Chandrakasan A., Sheng S., and Brodersen R.W. Low power cmos digital design. In *IEEE Journal Solid State Circuits*, pages 472–484, 1992.

[8] Govil K., Chan E., and Wasserman H. Comparing algorithms for dynamic speed-setting of a low power cpu. In *Proc. MOBICOM*, pages 13–25, November 1995.

[9] Greenwood G. W., Lang C., and Hurley S. Scheduling tasks in real-time systems using evolutionary strategies. In *Workshop on Parallel and Distributed Real-Time Systems*, 1995.

[10] Gruian F. Hard real-time scheduling for low energy using stochastic data and dvs processor. In *International Symp. on Low-Power Electronics and Design*, August 2001.

[11] Hong I., Potkonjak M., and Srivastava M. On-line scheduling of hard real-time tasks on variable voltage processor. In *International Conference on Computer Aided Design*, pages 653–656, 1998.

[12] Hong I., Qu G., Potkonjak M., and Srivastava M. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *19th IEEE Real-Time Systems Symposium.*, pages 178–187, December 1998.

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

[14] Ishihara T. and Yasuura H. Voltage scheduling problem for dynamically variable voltage processors. In *International Symp. on Low-Power Electronics and Design*, pages 197–201, 1998.

[15] Krishna C.M. and Lee Y.-H. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time

systems. In *Real-Time and Embedded Technology and Applications Symposium*, pages 156–165, May 2000.

[16] Luo J. and Jha N.K. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *International Conference on Computer Aided Design*, pages 357–364, 2000.

[17] Mosse D., Aydin H., Childers B., and Melhem R. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compiler and OS for Low Power*, 2000.

[18] Pering T., Burd T., and Brodersen R. The simulation and evaluation of dynamic voltage scaling algorithms. In *International Symp. on Low-Power Electronics and Design*, pages 76–81, 1998.

[19] Pering T., Burd T., and Brodersen R. Voltage scheduling in the Iparm microprocessor system. In *International Symp. on Low-Power Electronics and Design*, pages 96–101, 2000.

[20] Raghunathan V., Spanos P. and Srivastava M.B. Adaptive power fidelity in energy-aware wireless embedded systems. In *Real-Time Systems Symposium*, pages 106–115, 2001.

[21] Raje S. and Sarrafzadeh M. Variable voltage scheduling. In *Proc. Int. Symp. Low Power Design*, pages 9–14, 1995.

[22] Shin Y. and Choi K. Power-conscious fixed priority scheduling for hard real-time systems. In *Design Automation Conference*, pages 134–139, 1999.

[23] Weiser M., Welch B., Demers A. and Shenker S. Scheduling for reduced cpu energy. In *Proc. First Symp. on OSDI*, pages 13–23, November 1994.

[24] Wells B.E. A hard real-time static task allocation methodology for highly-constrained message passing environments. In *International Symposium of Computer Architecture*, 1995.

[25] Yang T. and Gerasoulis A. List scheduling with and without communication delays. In *Parallel Computing Journal*, pages 1321–1344, Vol. 19 1993.

[26] Yao F., Demers A., and Shenker S. A scheduling model for reduced cpu energy. In *Proc. 36th IEEE Symp. Foundations of Computer Science*, pages 374–382, 1995.

[27] Zhu D., Melhem R., and Childers B. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *22nd IEEE Real-Time Systems Symposium*, December 2001.