Power Aware Scheduling for AND/OR Graphs in Multi-Processor Real-Time Systems *

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

Abstract

Power aware computing has become popular recently and many techniques have been proposed to manage the energy consumption for traditional real-time applications. We have previously proposed two greedy slack sharing scheduling algorithms for such applications on multi-processor systems. In this paper, we are concerned mainly with real-time applications that have different execution paths consisting of different number of tasks. The AND/OR graph model is used to represent the application's data dependence and control flow. The contribution of this paper is twofold. First, we extend our greedy slack sharing algorithm for traditional applications to deal with applications represented by AND/OR graphs. Then, using the statistical information about the applications, we propose a few variations of speculative scheduling algorithms that intend to save energy by reducing the number of speed changes (and thus the overhead) while ensuring that the applications meet the timing constraints. The performance of the algorithms is analyzed with respect to energy savings. The results surprisingly show that the greedy scheme is better than some speculative schemes and that the greedy scheme is good enough when a reasonable minimal speed exists in the system.

1. Introduction

Power aware computing has recently become popular not only for general purpose systems but also for real time systems. For the traditional applications in real-time systems, where a task is *ready* to execute when all its predecessors complete execution, many techniques have been proposed to manage the energy consumption. Such applications are modeled by AND-graphs and the relationship over their tasks is known as AND-only precedence constraints [10]. But this traditional AND model cannot describe many applications encountered in practice, where a task is *ready* to execute when one *or more* of its predecessors finish execution, and one *or more* of its successors are ready to be executed after the task finishes execution. A real life example that falls within this AND/OR model is an automated target recognition (ATR) application, in which the number of regions of interests (ROI) in one frame varies substantially. For some frames, the number of detected ROIs may be maximum and all the tasks need to be executed, while in most cases, the number of detected ROIs in a frame is less than the maximum and part of the application can be skipped. The control flow of most practical applications also have OR structures, where execution of the sub-paths depends on the results of previous tasks. In some applications, the probability of the paths to be executed is also known a priori.

In this paper, we modify the greedy slack sharing algorithm developed in [20] to incorporate the AND/OR features and prove its correctness on meeting the timing constraints. While it achieves some energy savings, the greedy slack sharing algorithm may perform many voltage/speed changes. Considering the timing and energy overhead of voltage/speed adjustment, along with the statistical information about the application and the intuition that minimal energy can be obtained by running all tasks with the same speed, we study a few variations of the speculative scheduling algorithms that intend to save more energy by reducing the number of voltage/speed changes (and thus the overhead) while ensuring that the application's timing constraints will not be violated.

The performance, in terms of energy savings, is analyzed for all the schemes. The results surprisingly show that the greedy scheme is better than some speculative schemes especially when the system has a reasonable minimal speed. All the dynamic schemes perform the best with moderate $load$ and α (the ratio of the tasks' average case execution time over worst case execution time).

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

1.1. Related Work

For uniprocessor systems, based on dynamic voltage scaling (DVS) technique, Mossé et al. proposed and analyzed several schemes to dynamically adjust processor speed with slack reclamation, and statistical information about task's run-time was used to slow down the processor speed evenly and save more energy [14]. In [16], Shin et al. set the processor's speed at branches according to the ratio of the longest path to the taken paths from the branch statement to the end of the program. The granularity of the proposed schemes is the basic block, which will impose a very high overhead due to too frequent speed changes. Kumar et al. predict the execution time of the task based on the statistics gathered about execution time of previous instances of the same task [12]. Their algorithm is adequate for soft real time operating systems. We note that statistical schemes that predict execution times using history data are not eligible for hard real time systems where the deadlines must be guaranteed. The best scheme is an adaptive one that takes an aggressive approach while providing safeguards that avoid violating the application deadline [2, 13].

When considering the limited voltage/speed levels in the real processors, Chandrakasan et al. have shown that, for periodic tasks, a few voltage/speed levels are sufficient to achieve almost the same energy savings as infinite voltage/speed levels [6]. AbouGhazaleh et al. have studied the effect of the voltage/speed adjustment overhead on choosing the granularity of inserting power management points in a program [1].

For multi-processor systems, with AND-model applications that have fixed task sets and predictable execution times, static power management (SPM) can be accomplished by deciding beforehand the best voltage/speed for each processor [11]. For the system-on-chip (SOC) with two processors running at two different fixed voltage levels, Yang et al. proposed a two-phase scheduling scheme that minimizes the energy consumption while meeting the timing constraints by choosing different scheduling options determined at compile time [17]. Based on the idea of *slack sharing*, for AND-model applications, we have studied the dynamic voltage/speed adjustment schemes on multi-processor systems and proposed two dynamic management algorithms for independent tasks and dependent tasks, respectively [20].

In this paper, we consider the AND/OR model applications that have different execution paths with different task sets taking into account overhead and discrete voltage/speed levels. The paper is organized in the following way. The application model, power model and system model are described in Section 2. The greedy slack sharing algorithm is extended for applications represented by AND/OR graphs in Section 3. Section 4 proposes a few variations of speculative algorithms using the applications' statistical information. Simulation results are given and analyzed in Section 5 and Section 6 concludes the paper.

2. Models

2.1. Application Model: AND/OR Graph

In this paper, we use the AND/OR model [10], which is represented by a graph $G(V, E)$, where the vertices in V represent tasks or synchronization nodes, and the edges $E \subseteq V \times V$ represent the dependence between vertices. The graph represents both the control flow and data dependence between tasks. Only when v_i is the direct predecessor of v_j, is there an edge $e :: v_i \rightarrow v_j \subseteq E$, which means that v_j depends on v_i; in other words, only after v_i finishes execution can v_j become ready for execution. The application also has a deadline D.

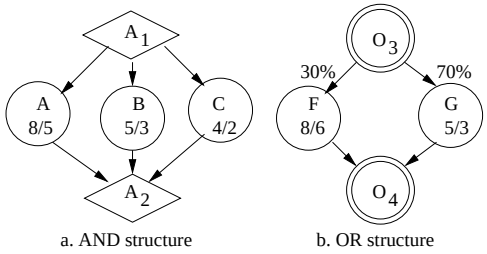


Figure 1. The AND/OR Structures

In the extended AND/OR model, there are three different kinds of vertices: computation nodes, AND nodes and OR nodes. A computation node T_i is represented by a *circle*, which has two attributes, c_i and a_i, where c_i is the worst case execution time (WCET) of T_i and a_i is the average case execution time of T_i, all based on maximum processor speed (f_{max}). An AND synchronization node is represented by a *diamond*, which depends on all its predecessors and all its successors depend on it. It is used to explore the parallelism in the applications as shown in Figure 1a. An OR synchronization node is represented by a *double circles*, which depends on only one of its predecessors and only one of its successors depends on it. It is used to explore the different execution paths in the applications as shown in Figure 1b. For simplicity, we only consider the case where an OR node cannot be processed concurrently with other paths. In other words, all the processors will synchronize at an OR node. The synchronization nodes are considered as *dummy* tasks with execution time as 0 ($c = a = 0$).

In the figure, the computation node is labeled by its name and (c_i/a_i). The AND/OR nodes are labeled correspondingly. To represent the probability of taking each execution path after the OR synchronization node v, a number is associated with each successor of v.

Since there is no back edges in our AND/OR model, for the loops in an application, we can treat a whole loop to be one task with the execution time of maximal iterations as c_i and average iterations as a_i. Alternatively, we can expand the loop as several tasks if we know the maximal number of iterations

and the corresponding probabilities to have specific number of iterations.

2.2. Power Management Points

In [14], the insertion of power management points (PMP) at the start of each program section is proposed. These points are inserted by the user, or set by the compiler. At each PMP, a new speed is computed based on the time taken so far and an estimation of the time for the future tasks. If the new speed is different from the current processor speed, the speed/voltage setting is invoked.

For the AND/OR model proposed above, there is a PMP before each node. Two values, Π_c and Π_a, are associated with the PMP before the first node in the graph. The values represent the worst case execution time and average case execution time of the application, respectively. For the PMP before an OR node, two values, Π_c^i and Π_a^i, are associated with each path p_i after the OR node. The values represent the worst case execution time and average case execution time for path p_i from the PMP to the end of the program, respectively. All these values can be obtained from profiling and will be used in speculation and computing the new speed. The details are discussed in Sections 3 and 4.

2.3. Power and System Models

We assume that processor power consumption is dominated by dynamic power dissipation P_d, which is given by: $P_d = C_{ef} \times V_{dd}^2 \times f$, where C_{ef} is the effective switch capacitance, V_{dd} is the supply voltage and f is the processor clock frequency. Processor speed, represented by f, is almost linearly related to the supply voltage: $f = k \times \frac{(V_{dd}-V_t)^2}{V_{dd}}$, where k is constant and V_t is the threshold voltage [4, 7]. The energy consumed by a specific task T_i can be given as $E_i = C_{ef} \times V_{dd}^2 \times C$, where C is the number of cycles needed to execute the task. When decreasing processor speed, we also reduce the supply voltage. This reduces processor power consumption cubically and reduces task energy consumption quadratically at the expense of linearly increasing the execution time of the task. For example, consider a task that, with maximum speed f_{max}, needs 10 time units to finish execution. If we have 20 time units allocated to this task, we can reduce the processor speed by half while still finishing the task on time. The new energy consumption would be: $E' = C_{ef} \times (\frac{V_{dd}}{2})^2 \times \frac{f_{max}}{2} \times 20 = \frac{1}{4} \times C_{ef} \times V_{dd}^2 \times f_{max} \times 10 = \frac{1}{4} \times E$, where E is the energy consumption with maximum processor speed. From now on, we refer to *speed change* as both changing the CPU voltage and frequency.

We consider systems that have multiple identical processors with shared memory. The application characteristics and state are kept in the shared memory. All the ready tasks are put into a global queue. Each processor executes the scheduler independently and fetches the tasks from the global queue as needed. We assume that the shared memory is accessed in a mutual exclusive way and access to the shared memory has no extra cost (actually, the cost is part of context switch that we do not consider in this paper).

Table 1. Speed & Voltages of Transmeta 5400

f(MHz)	700	666	633	600
V(V)	1.65	1.65	1.60	1.60
f(MHz)	566	533	500	466
V(V)	1.55	1.55	1.50	1.50
f(MHz)	433	400	366	333
V(V)	1.45	1.40	1.35	1.30
f(MHz)	300	266	233	200
V(V)	1.25	1.20	1.15	1.10

In this paper, we consider two different power configurations for the processors. First, in the Transmeta model, the voltage/speed setting is given as in Table 1 [18]. There are 16 voltage/speed settings between 700MHz (1.65V) and 200MHz (1.10V). The second power configuration is the Intel XScale model [19], with the voltage/speed setting as shown in Table 2. Note that the speed and voltage do not obey a linear relation in either model, which is different from the assumptions in many published papers.

Table 2. Speed & Voltages of Intel XScale

f(MHz)	1000	800	600	400	150
V(V)	1.80	1.60	1.30	1.00	0.75

3. Greedy Algorithm for AND/OR Graph

3.1. Scheduling for Multi-Processor Systems

Since list scheduling is a standard technique used to schedule tasks with precedence constraints [8], we will focus on list scheduling in this paper. List scheduling puts tasks into a ready queue as soon as they become ready and dispatches tasks from the front of the ready queue to processors. When more than one task is ready at the same time, finding the optimal order of the tasks that minimizes execution time is NP-hard [8]. In this paper, we use the same heuristic as in [20] and put into the ready queue first the longest (based on tasks' WCET) among the tasks that become ready simultaneously.

If there is some slack in the system and a task can be allocated more time than its WCET, the system can slow down the

CPU for the task appropriately save energy. Since tasks exhibit a large variation in actual execution time, and in many cases, only consume a small fraction of their worst case execution time [9], any unused time can be considered as *slack*. Furthermore, the execution does not always follow the longest path, and thus there may be some extra slack. For the AND-model applications, *greedy slack sharing* algorithms have been discussed for multi-processor systems, in which part of the slack on processor P_x will be shared with processor P_y if P_x's expected finish time[1] is later than P_y's but actually finishes earlier than P_y. The remaining slack is given to the next task to be run on P_x. See [20] for details.

In the following, we will explore the application's dynamic characteristics both at the task set level (different execution paths) and at the task level (different actual execution time of each task). We extend the greedy slack sharing algorithm for dependent tasks [20] to incorporate the characteristics of AND/OR model and show how it is correct with respect to meeting the timing constraints.

3.2. Greedy Algorithm

The algorithm consists of two phases: an *off-line phase* and an *on-line phase*. The off-line phase is used to collect the execution information about the application with processor speed as f_{max} and it is a two-round phase. In the first round, using list scheduling with longest task first (LTF) heuristics, a *canonical schedule* is generated for each program section separated by OR nodes, in which the tasks use their worst case execution time. The time to finish the application along the longest path (consisting of all the longest program section between OR nodes) is defined as Π_c, which is stored in the PMP at the very beginning. For the PMPs before the OR synchronization nodes, the worst case execution time for remaining tasks on path j is gathered and stored as Π_c^j. For the average case, Π_a and Π_a^j are also analogously collected. The execution order of task T_i is recorded as EO_i and we will maintain the same execution order of tasks in the on-line phase to meet the timing constraints. The execution order of an OR node is the maximal execution order of its predecessors plus 1. For tasks which are on different paths after an OR node and will be executed at the same time, they may have the same execution order.

If $\Pi_c > D$, the algorithm fails to guarantee the deadline; otherwise, the second round of the off-line phase shifts the canonical schedules for all program sections to make them finish exactly on time. Notice that the shifting is a recursive process when there are embedded OR nodes. The start time of T_i in the shifted schedules is called *latest start time* LST_i and is also recorded, it is the time T_i *must* start execution for the remaining tasks in the shifted schedules to meet the deadline. LST_i will be used to claim the slack for T_i at run time.

Given any heuristic, if the off-line phase does not fail, the following on-line phase can be applied under the same heuristic. The following algorithm will assume that the longest path in the worst case meets the deadline, that is, $\Pi_c \leq D$.

Before presenting the on-line phase of the algorithm, we give some definitions. As in [20], we define the *estimated end time (EET)* for a task executing on a processor as the time at which the task is expected to finish execution if it consumes all the time allocated for it. To determine the *readiness* of tasks, we define the number of *unfinished predecessors* (UP_i) for each task T_i. UP_i will decrease by 1 when any predecessor of task T_i finishes execution. Task T_i is *ready* when $UP_i = 0$.

The speed to execute task T_i using greedy slack sharing reclamation is denoted as f_g^i. To maintain the execution order of tasks as in the canonical schedules, the execution order of the next expected task is defined as EO_{NET}. The current time is represented by t.

Initially, all the root tasks are put into a *Ready-Q*. For all other tasks, UP_i is initialized as the number of predecessors of T_i if the corresponding vertex is not an OR node, and 1 otherwise. The current time t is set to 0 and the execution order of the next expected task EO_{NET} is set to 1.

1 T_k =*Head(Ready-Q)*;
2 If (T_k is OR node || EO_{NET}==EO_k) && ($UP_k == 0$))
  Goto Step 4;
3 $wait()$; Goto Step 1;
4 T_k =dequeue(*Ready-Q*);
  $EO_{NET} = EO_{NET} + 1$;
5 If (T_k is Computation node)
  $EET_k = LST_k + c_k$; /* Note that $LST_k \geq t$ */
  $f_g^k = f_{max} \times \frac{c_k}{(EET_k - t)}$; /*compute the speed for T_k*/
  If(P_y is sleep && Head(Ready-Q) is next expected and ready)
    signal(P_y);
  Execute T_k at speed f_g^k;
6 If (T_k is Computation node or AND node)
  For each successor T_j of T_k:
    $UP_j = UP_j - 1$;
    If ($UP_j == 0$) enqueue(T_j, *Read-Q*);
  Goto Step 1;
7 If (T_k is OR node)
  $EO_{NET} = EO_k + 1$; /*update the next expected task*/
  If selected path p_i /*the 1^{st} task of path p_i is denoted by T_i*/
    $UP_i = 0$;
    Put T_i into *Ready-Q*;
  Goto Step 1;

Figure 2. The GSS Algorithm invoked by P_{id}

The greedy slack sharing (GSS) algorithm for AND/OR applications is shown in Figure 2. Remember that the execution

[1] More accurately, it is the estimated end time (EET) of the task running on processor P_x, as defined later.

order of tasks will be kept the same as in the canonical schedules. From the algorithm, each idle processor tries to fetch the next ready task (Step 1 and 2). If the next expected task is not ready, the processor will go to sleep (Step 3; We use the function *wait()* to put an idle processor to sleep and another function *signal(*P*)* to wake processor P.). Otherwise, if the task is a computation task, the processor computes a new speed for the ready task, wakes up an idle processor if the task expected after the one the processor is handling is ready and changes the speed if necessary before executing the ready task (Steps 4 and 5). If the task is a dummy task (AND/OR node), the successors of the node are handled properly (Step 6 and 7). The function *enqueue(T,Q)* is used to put the ready task T into Q in the order of tasks' execution order. The shared memory holds the control information, such as *Ready-Q*, UP values, which must be updated within a critical section (not shown in the algorithm for simplicity).

The slack sharing is implicit in the algorithm. In Step 4 and 5, when a processor picks a task that has a earlier LST than the one it should pick, slack sharing happens implicitly.

3.3. Algorithm Analysis

Theorem 1 *For an application represented by an AND/OR graph with a deadline D and for a given heuristic for assigning priorities to tasks, the greedy slack sharing algorithm will finish by time D if the application can finish before D in the canonical schedules.*

From the off-line phase, after the canonical schedules are shifted, for any execution path if every task uses up the time that GSS algorithm allocates to it, the application will finish just in time. Define FT_i^w as the finish time for T_i in the worst case for the shifted schedule. For any execution path p, we will have $FT_i^w \leq D$ ($T_i \in p$). Notice that EET_i is the latest time T_i should finish, following the same idea as in [20], it is not hard to prove that $EET_i = FT_i^w$ ($T_i \in p$) for any path p. That is, if the application can finish before D in canonical schedules, the execution of all paths under GSS will finish in time. The proof is omitted for lack of space.

While the greedy slack sharing algorithm is guaranteed to meet the timing constraints, there may be many speed changes during the execution since it computes a new speed for each task. It is known that a clairvoyant algorithm can achieve minimal energy consumption for uniprocessor systems by running all tasks at a single speed setting if the actual running time of every task is known. Considering the speed adjustment overhead, the single speed setting is even more attractive. From this intuition, using the statistical information about application, we propose the following speculative algorithms.

4. Speculative Algorithms

Based on different strategies, we developed two speculative schemes. One is to statically speculate the speed for the whole application, the other is to speculate the speed *on-the-fly* to reflect actual behavior of the application, that is, dynamically taking into account the remaining work. For the first scheme, we can speculate a single speed if the speed levels are fine grained, or two speeds otherwise. For the second scheme, we speculate after each OR synchronization node since more slack can be expected from different paths after the branch.

4.1. Static Speculative Algorithms

For static speculative (SS) algorithm, the speed, f_{ss}, at which the application should run is decided at the very beginning based on the statistical information about the whole application, as follows:

$$f_{ss} = f_{max} \times \frac{\Pi_a}{D}$$

where Π_a is the average case finish time of the entire application, which is calculated as[2] $\Pi_a = \sum_p \Pi_a^p \times P_p$, where Π_a^p is the average case finish time for path p, and P_p is the probability of executing path p.

If the speculated speed falls between two speed levels ($f_l < f_{ss} < f_{l+1}$) and the speed levels are fine grained, single speed static speculation will set $f_{ss} = f_{l+1}$. But if the difference between two speed levels is large, two speeds will be speculated for the application. At the beginning, the speculation speed, f_{ss}, is set as the lower speed, f_l. After a certain time point (t_{tp}), f_{ss} is changed to the higher speed level, f_{l+1}. The value of t_{tp} can be statically computed as follows:

$$t_{tp} = \frac{f_{l+1} - f_{ss}}{f_{l+1} - f_l} \times D$$

After f_{ss} is calculated, we will execute the application at f_{ss}, but at the same time ensure that the application finishes on time. This means that we choose the maximum speed between f_{ss} and f_g^i for task T_i, where f_g^i is computed from GSS, guaranteeing temporal correctness.

It is easy to see that, when picking the speed to execute a task, because the SS algorithms never sets a speed below the speed determined by GSS, the SS algorithms will meet the timing constraints if GSS can finish on time. Therefore, from Theorem 1, the SS algorithms can meet the timing constraints if the longest path in canonical schedule under the same heuristics finishes on time.

[2] If path p contains OR nodes, the same formula is applied recursively.

4.2. Adaptive Speculative Algorithm

If the statistical characteristics of tasks in the application vary substantially, it may be better to speculate the speed on-the-fly based on the statistical information about the remaining tasks. Considering the speed adjustment overhead and expecting that different paths after the OR synchronization node may result in more slack, the adaptive speculative (AS) algorithm speculates the speed after each OR synchronization node based on statistical information about the remaining tasks, and sets the speculative speed as:

$$f_{as} = f_{max} \times \frac{\Pi_a^r}{D - t}$$

where t is the current time and Π_a^r can be calculated dynamically as the summation of the weighted average of the execution times of all the remaining tasks. Again, to guarantee the deadline, the speed f_i for T_i will be: $f_i = max(f_g^i, f_{as})$.

5. Evaluation and Analysis

In the simulation, we account for both the overhead and discrete speed levels. There are two kinds of overhead, the new speed computation overhead and the voltage/speed adjustment overhead. A detailed discussion about accounting for overheads and discrete speed levels can be found in [20]. The new speed computation overhead used is 300 cycles, obtained by running the code to compute the new speed on the SimpleScalar micro-architecture simulator [5]. With current technology, changing the voltage/speed needs between $5\mu s$ and $150\mu s$ [3, 15], but we expect this overhead to drop with technological advances in the near future. We assume that speed adjustment needs $5\mu s$ (the time needed to change the speed once). We assume that an idle processor consumes 5% of the maximal power level [2].

We consider an application of automated target recognition (ATR) (the dependence graph is not shown due to space limitation) and a synthetic application shown in Figure 3. In ATR, the regions of interest (ROI) in one frame are detected and each ROI is compared with all the templates. For the synthetic application, the time unit for c_i and a_i is in the order of 10^{-4} second. The loops in the graph can be expanded as discussed in Section 2. The numbers associated with each loop are the maximal number of iterations paired with the probabilities to have specific number of iterations. If there is only one number, it is the number of iterations during the execution. We vary the parameters: $load$ and α to see how they affect the energy consumption for each scheme. The *load* is defined as the length of the canonical schedule for the longest path p^l over the deadline. The variability of the tasks' execution time α is defined as the average case execution time over worst case execution time for the tasks in the application, which indicates how much dynamic slack there is during tasks' execution. The

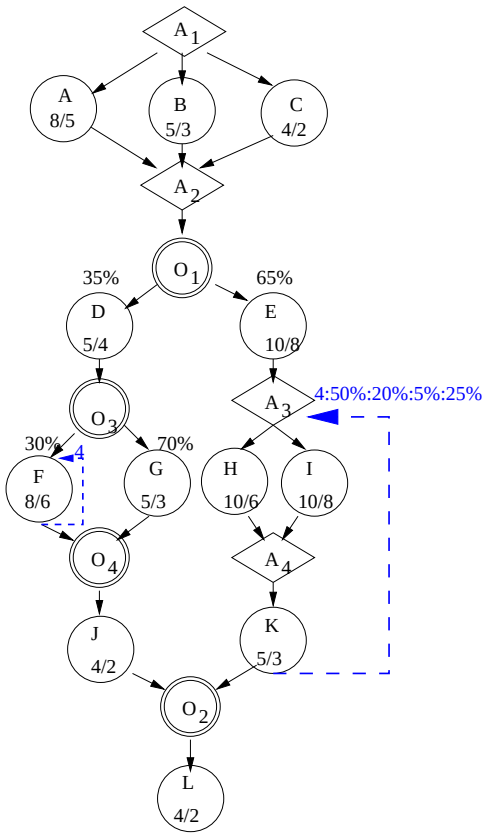


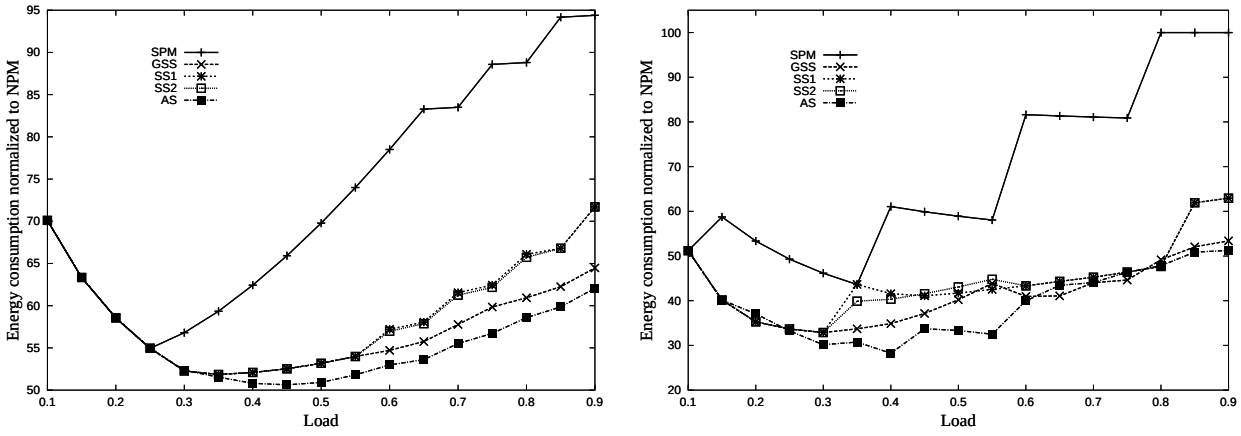
Figure 3. An Example of AND/OR Graph

value of α_i for task T_i is generated from a normal distribution around α and the actual execution time of T_i follows a normal distribution around α_i. Each point in each graph is an average of 1000 runs.

We show results for the following schemes in the graph: static power management (SPM), greedy slack sharing (GSS), static speculation with single speed (SS1), static speculation with two speeds (SS2) and adaptive speculation at each OR node (AS). The energy consumption of each scheme is normalized to the energy consumed by no power management (NPM) where every task runs at f_{max} with idle state energy set as 5% of the maximum power level.

5.1. The Effect of $Load$

When load increases, there is less static slack in the system and therefore energy consumption should increase for all the schemes (since the slowdown capability is smaller) except NPM (since it will consume less idle energy). The results in Figure 4 show the normalized energy consumption for ATR running on dual-processor systems ($\alpha \approx 0.95$, which was measured and means that there is little slack from task's run-time behavior). Note that the normalized energy consumption starts by decreasing with $load$. This is counter-



a. Transmeta Model b. Intel XScale Model

Figure 4. Energy vs. $Load$ for ATR running on Dual-Processor systems; $\alpha \approx 0.95$, $overhead = 5\mu s$.

intuitive since, without accounting for the idle energy, normalized energy consumption would increase proportionally with the $load$ (since the slowdown capability is smaller). But, at lower $load$, the idle energy consumption has a significant effect: when the desired speed is less than f_{min}, the CPU speed is set to run at f_{min}; the normalized energy consumption curves go down with increased $load$ (i.e. decrease in idle time) and starts increasing with $load$ when speed is set above f_{min}. When the processors are simulated following the Intel XScale model (Figure 4b), where there are fewer speed levels but wider speed range between levels, the normalized energy for SPM incurs sharp changes. These changes correspond to the upgrade of speed from one level to the next level. For example, when $load = 0.35$, SPM runs at 400MHz and when $load = 0.4$ SPM needs to run at 600MHz (rather than 400MHz) because of speed adjustment overhead. While when $load = 0.5$, SPM still runs at 600MHz and the energy consumption decreases because of less idle energy consumed. For static speculation, when $load$ changes from 0.3 to 0.35 or from 0.8 to 0.85, the normalized energy for SS1 and SS2 has a jump, the reason is that the speculative speed is upgraded from one level to the next higher level[3]. Note that the figures show the normalized energy and the energy consumption by NPM decreases with load increasing since less idle energy is consumed.

When ATR is executed on 4 or 6 processor systems, similar results are obtained with more energy consumed by each scheme and more sharp changes because of the idle time forced by the scheduler between tasks for the sake of synchronization. Due to the similarity of results, we only show in Figure 5 the results for 6 processors.

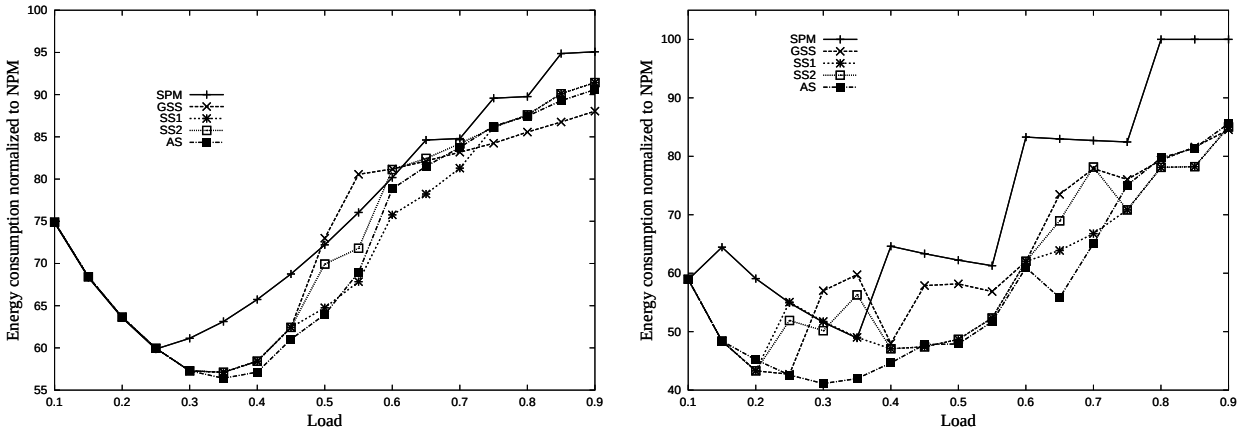
We expected the speculative schemes to perform better than the greedy scheme. The reason is that, typically, the greedy behavior tends to run at the least possible speed to use up all the slack for the current task, and consequently the future tasks must run at very high speed [1, 20]. However, the minimum speed is bounded by f_{min}, preventing the greedy scheme from using all the available slack at the very beginning and forcing some slack to be saved for future use. Fewer speed levels also prevents the greedy scheme from using the slack by decreasing the probability of speed changes. As a result, the greedy scheme benefits from the presence of f_{min} and speed levels. The greedy scheme is better than some speculative algorithms when f_{min} is rather high or there are fewer speed levels.

To see how f_{min} and the speed levels affects the performance of the schemes on energy savings, in our future work we plan to experiment with different values of $\frac{f_{max}}{f_{min}}$ and different number of speed levels between f_{max} and f_{min}.

5.2. The Effect of α

For the synthetic application running on a dual-processor system with $load$ set at 0.8 and $overhead$ at $5\mu s$, when changing α, the normalized energy consumed by each scheme is shown in Figure 6. Since both changing $load$ and α have the same effect on the available slack in the system, the shapes of the curves for dynamic schemes are similar to when $load$ was changed (the curves for SPM are quite different since SPM can only use static slack that related to $load$ only). Notice that, for the Intel XScale model, with $load = 0.8$, SPM runs

[3] Note that $f_{ss} = f_{max} \times \frac{\Pi_a}{D} = f_{max} \times \frac{\Pi_a}{\Pi_c} \times \frac{\Pi_c}{D}$ and we defined $load = \frac{\Pi_c}{D}$, so $f_{ss} = f_{max} \times load \times \frac{\Pi_a}{\Pi_c}$. For ATR running on dual-processor system, $\frac{\Pi_a}{\Pi_c} \approx 0.48$, when $load = 0.3$ the speculative speed is 150MHz, while when $load = 0.35$ the speculative speed will increase one level and be 400MHz. It is the same for $load$ change from 0.8(400MHz) to 0.85(600MHz).



a. Transmeta Model b. Intel XScale Model

Figure 5. Energy vs. $Load$ for ATR running on 6-Processor systems; $\alpha \approx 0.95$, overhead=5μs.

at $f_{max} = 1GHz$ rather than $800MHz$ and consumes the same energy as NPM since SPM does not take into account the actual execution time behavior.

6. Conclusion

In this paper, we extend the AND/OR model by adding probabilities to each execution path after each OR node. This extended model can be used for applications, where a task is ready to execute when one *or more* of its predecessors finish execution and one *or more* of its successors will be ready after the task finishes execution. With the extended AND/OR model, we modify the greedy slack sharing algorithm for dependent tasks on multi-processor systems developed in [20]. Then, using statistical information about the application, we proposed a few variations of speculative algorithms that intend to save more energy by reducing the number of speed change (and thus the overhead) while ensuring that the application meet the timing constraints.

The performance of all the algorithms in terms of energy savings is analyzed through simulations. The greedy algorithm is surprisingly better than some speculative algorithms. The reasons come from two points: one is the minimal speed limitation that prevents the greedy algorithm from using up the slack very aggressively; the other is fewer speed levels that prevents the greedy algorithm from changing the speed frequently. The greedy scheme is good enough when the system has a reasonable minimal speed. The energy consumption for all the power management schemes decreases unexpectedly when the $load$ increases at low $load$ because of the minimal speed limitation and the idle energy consumption. The dynamic schemes become worse relative to static power management (SPM) when $load$ becomes higher and α becomes larger, since most of the slack will be used to cover the speed adjustment overhead. All the dynamic algorithms perform the best with moderate $load$ and α. When the number of processors increases, the performance of the dynamic schemes decreases due to the limited parallelism and the frequent idleness of the processors.

References

[1] N. AbouGhazaleh, D. Mossé, B. Childers and R. Melhem. Toward the Placement of Power Management Points in Real Time Applications. *Workshop on Compilers and Operating Systems for Low Power (COLP)*, Barcelona, Spain, 2001.

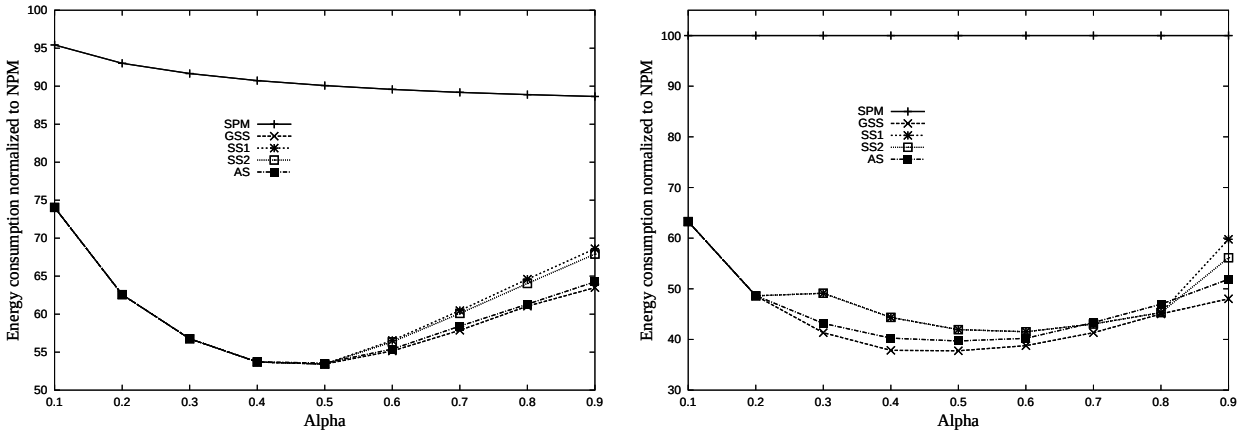
[2] H. Aydin, R. Melhem, D. Mossé and P. M. Alvarez. Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. *Proc. of the 22^{th} IEEE Real-Time Systems Symposium*, London, UK, Dec. 2001.

[3] T. D. Burd, T. A. Pering, A. J. Stratakos and R. W. Brodersen. A Dynamic Voltage Scaled Microprocessor System, *IEEE Journal of Solid-State Circuits*, vol.35, no.11, Nov. 2000.

[4] T. D. Burd and R. W. Brodersen. Energy Efficient CMOS Microprocessor Design. *Proc. HICSS Conference*, pp. 288-297, Maui, Hawaii, January 1995.

[5] D. Burger and T. M. Austin. The SimpleScalar Tool Set, version 2.0. Tech. Report 1342, Computer Science Department, University of Wisconsin-Madison, Jun. 1997.

[6] A. Chandrakasan, V. Gutnik and T. Xanthopoulos. Data Driven Signal Processing: An Approach for Energy Ef-



a. Transmeta Model

b. Intel XScale Model

Figure 6. Energy vs. α for the synthetic application of running on 2-Processor systems; $load = 0.8$, $overhead = 5\mu s$.

ficient Computing, *Proc. Int'lSymp. Low-Power Electronic Devicess*, Monterey, CA 1996.

[7] A. Chandrakasan, S. Sheng and R. Brodersen. Low-power CMOS Digital Design. *IEEE Journal of Solid-state circuit*, pp. 473-484, April 1992.

[8] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Trans. On Software Engineering*, SE-15 (12): 1497-1505, 1989.

[9] R. Ernst and W. Ye. Embedded Program Timing Analysis based on Path Clustering and Architecture Classification. *In Computer-Aided Design (ICCAD)97*. pp. 58-604. San Jose, CA, November 1997.

[10] D. W. Gillies and W.-S. Liu, Scheduling Tasks With AND/OR Precedence Constraints. *SIAM J. Compu.*, 24(4): 797-810, 1995.

[11] F. Gruian. System-Level Design Methods for Low-Energy Architectures Containing Variable Voltage Processors. *The Power-Aware Computing Systems 2000 Workshop at ASPLOS 2000*, Cambridge, MA, November 2000.

[12] P. Kumar and M. Srivastava, Predictive Strategies for Low-Power RTOS Scheduling, *Proceedings of the 2000 IEEE International Conference on Computer Design: VLSI in Computers and Processors*

[13] R. Melhem, N. AbouGhazaleh, H. Aydin and D. Mossé. Power Management Points in Power-Aware Real-Time Systems. In *Power Aware Computing*, Ed. by R.Graybill and R.Melhem, Plenum/Kluwer Publishers, 2002.

[14] D. Mossé, H. Aydin, B. Childers and R. Melhem. Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications, *Workshop on Compiler and OS for Low Power*, Philadelphia , PA, October 2000

[15] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems, 18^{th} *ACM Symposium on Operating Systems Principles (SOSP'01)*, Banff, Canada, Oct. 2001

[16] D. Shin, J. Kim and S. Lee, Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications, *IEEE Design and Test of Computers*, March 2001.

[17] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Kerkest and R. Lauwereins. Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SOCs, *IEEE Design and Test of Computers*, vol. 18, no. 5, Sep. 2001.

[18] http://www.transmeta.com

[19] http://developer.intel.com/design/intelxscale/

[20] D. Zhu, R. Melhem and B. Childers. Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems. *Submitted to IEEE Trans. on Parallel and Distributed Systems*, Nov. 2001. *A preliminary version appeared in the* 22^{th} *IEEE Real-Time System Symposium, 2001.*