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

Dakai Zhu, Daniel Mossé and Rami Melhem

Abstract

Power aware computing has become popular recently and many techniques have been proposed to manage processor energy consumption for traditional real-time applications. In this paper, we are concerned mainly with the AND/OR model of real-time applications that have different execution paths consisting of different tasks. The contribution of this paper is twofold. First, we propose a greedy slack stealing algorithm to deal with applications represented by AND/OR graphs and prove its correctness in terms of meeting the timing constraints. Then, using 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 application meets its timing constraints. Some practical issues are also considered, such as shared memory access contention and idle energy consumption. The performance of the algorithms is analyzed with respect to processor energy savings. The results surprisingly show that the greedy slack stealing scheme is better than some speculative schemes and that the greedy scheme is good enough when a reasonable minimal speed exists in the system or when there are only a few (4-6) voltage/speed levels.

Index Terms: Power-Aware Scheduling; AND/OR; Real-Time Systems

1 Introduction

Power aware computing has recently become popular not only for hand-held devices that have limited energy supply but also for large systems consisting of multiple processors (e.g., complex satellite and surveillance systems, data warehouses or web server farms), where the cost of energy consumption and cooling is substantial. Since processors consume substantial energy in most systems, many techniques have been proposed to reduce the processor energy consumption, such as low-power processor design [3] and dynamic voltage scaling (DVS) [14, 15]. Based on DVS

techniques, many works have been proposed to manage the processor energy consumption when executing the traditional AND-model applications in real-time systems [26, 28], where a task is *ready* to execute when all its predecessors complete execution [11]. 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 become 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, which is widely used and requires real-time processing [24]. The control flow of most practical applications also has 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 apriori or can be obtained from profiling. To represent all these features, we extend the AND/OR model developed in [11].

In this paper, we extend the work in [28] and consider the AND/OR model applications. We propose the *greedy slack stealing* algorithm that incorporates the AND/OR features and prove its correctness with respect to meeting applications' timing constraints when executing on a shared memory N-processor system. While achieving some energy savings, the greedy algorithm carries out many voltage/speed changes. Considering the timing and energy overhead of voltage/speed adjustment, along with the statistical information about the applications, we study a few variations of 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 applications' timing constraints will not be violated. We take into account shared memory access contention, energy spent in idle periods, speed adjustment overhead and the effects of discrete voltage/speed levels.

The performance, with respect to processor energy savings, is evaluated through simulations. The results surprisingly show that the greedy scheme is better than some speculative schemes, especially when the system has a reasonable minimal speed or when there are only a few (4-6) voltage/speed levels. Less surprisingly, the effects of voltage/speed adjustment overhead increase when the size of application decreases or the amount of available slack decreases.

1.1 Related Work

For uniprocessor systems, Yao *et al.* described an off-line scheduling algorithm for independent tasks running with variable speed, assuming worst-case execution time [27]. Based on dynamic

voltage scaling (DVS) technique, Mossé *et al.* proposed and analyzed several schemes to dynamically adjust processor speeds using slack reclamation, where statistical information about task's run-time was used to slow down the processor speed evenly and save more energy [21]. In [25], Shin *et al.* set the processor's speed at branches according to the ratio of the longest path to the taken path from the branch statement to the end of the program, but the fine granularity of invoking speed changes (at each the basic block) incurs high overheads. Kumar *et al.* predict the execution time of tasks based on the statistics gathered about execution times of previous instances of the same task [17]. Their algorithm is adequate for soft real time systems with several task priority levels. Using stochastic data while taking into account the actual run-time behavior about tasks, Gruian proposed a two-phase (offline and online) algorithm for hard real-time systems with fixed priority assignment for tasks [13]. The best scheme is an adaptive one that takes an aggressive approach while providing safeguards that avoid violating the application deadlines [2, 19].

When considering the limited voltage/speed levels in real-life uniprocessors, 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]. Pillai *et al.* also proposed a set of scheduling algorithms (static and dynamic) for periodic tasks based on EDF/RM scheduling policy. The simulation (assuming 4 speed levels) as well as the prototype implementation show that the algorithms effectively reduce energy consumption upto 40% compared to no power management [22]. AbouGhazaleh *et al.* have studied the effect of the voltage/speed adjustment overhead on choosing the granularity of inserting power management points in an application [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 [12]. For system-on-chips (SOCs) 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 [26]. Although they considered the application's dynamic behavior at the task set level, they did not consider tasks' run-time behavior. For AND-model applications, we previously studied the dynamic voltage/speed adjustment schemes on multi-processor systems and proposed two dynamic management algorithms (called *slack sharing*) for independent tasks and dependent tasks [28]. The effect of discrete voltage/speed

levels and voltage/speed adjustment overhead on the performance in terms of energy savings of the dynamic management algorithms is also studied in [28], where we show that the algorithms can save up to 50% energy compared with SPM, even considering overhead.

The paper is organized in the following way. The application model, power model and system model are described in Section 2. The greedy slack stealing algorithm is proposed for applications represented by AND/OR graph and its correctness is proved in Section 3. Section 4 proposes a few variations of speculative algorithms using the applications' statistical information. Practical considerations, such as shared memory access contention and processor energy consumption during idle period, are discussed in Section 5. Simulation results are given and analyzed in Section 6 and Section 7 concludes the paper.

2 Models

2.1 Application Model: AND/OR Graph

The AND/OR model is an extension of the model presented in [11] and is represented by a directed acyclic graph $G(V, E)$, where the vertices in V represent tasks or synchronization nodes, and the edges $E \subseteq V \times V$ represent the dependencies between vertices. The graph represents both the control flow and data dependence between tasks. If v_i is the immediate predecessor of v_j, there is an edge $e :: v_i \rightarrow v_j \subseteq E$, which means that, in general, v_j depends on v_i. In other words, v_j becomes *ready* for execution only after v_i finishes execution.

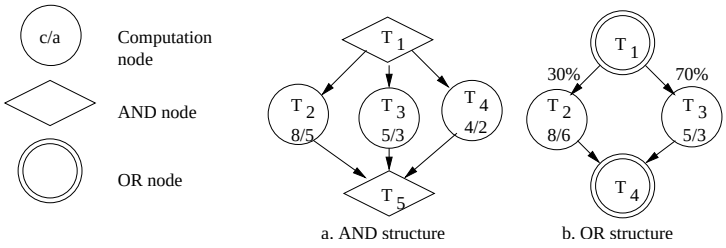


Figure 1: The AND and OR Structures; with the maximum and average length for each computation node.

There are three different kinds of vertices in this model: computation nodes, AND nodes, and OR nodes. As shown in Figure 1, a computation node, T_i, is represented by a *circle* and labeled by its two attributes, c_i and a_i, which are the maximum and average computation requirement (in

terms of number of cycles[1]), respectively, of T_i. An AND node is represented by a *diamond*; an AND node depends on all its predecessors (that is, it can be executed only after all its predecessors finish execution) and all its successors depend on it (that is, all its successors are executed after it finishes execution). It is used to explore the parallelism in the application as shown in Figure 1a. An OR synchronization node is represented by a *double circle*, which depends on only one of its predecessors (that is, when any one of its predecessors finishes execution, it is ready for execution) and only one of its successors depends on it (that is, exactly one of its successors is executed after its execution). It is used to explore the different execution paths in the application as shown in Figure 1b. To represent the probability of taking each branch after an OR synchronization node, a number is associated with each successor of an OR node. The AND/OR nodes are considered as *dummy* tasks with the number of required cycles being 0. For a synchronization node with non-zero computation requirement, it is easy to transform it to a synchronization node and a computation node. The application represented by the AND/OR graph has a deadline D.

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. We define *segment* as a set of tasks that are separated by two adjacent OR nodes. There are several segments, one for each of the branches, between two adjacent OR nodes.

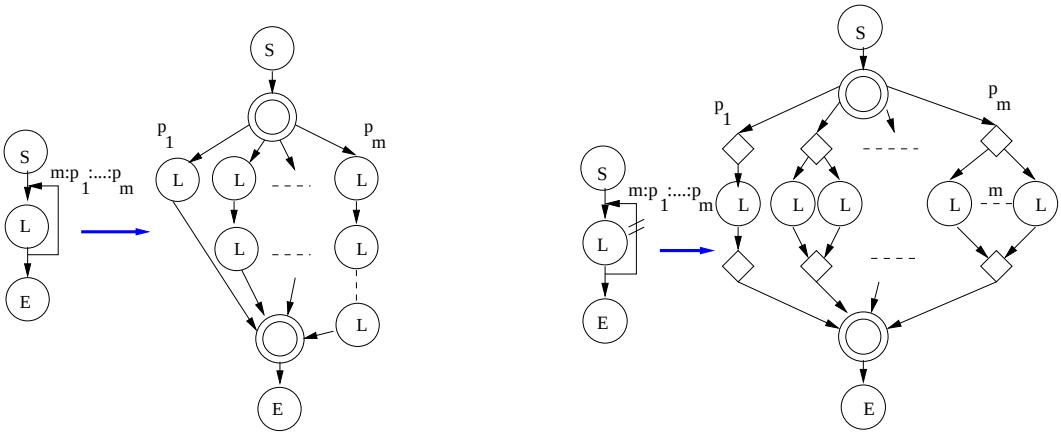


Figure 2: Expanding Loops: a. Serially (left); b. In parallel (right).

For loops in the AND/OR model, assuming that we know the maximum number of iterations, m, and the corresponding probabilities to execute specific number of iterations, we can either treat

[1]For a specific architecture, we assume that the number of cycles to execute a task is independent of the processor speeds. This assumption was justified experimentally in [19].

the whole loop as a single task that has c_i and a_i (computed as the maximum and average number of cycles needed to execute the entire loop, respectively) or we can expand the loop as several tasks. Based on the dependency between the iterations in a loop, there are two ways to expand the loop. If there is a dependency between iterations (i.e., iteration $i+1$ depends on iteration i, $i = 1, \cdots, m-1$), the loop can only be expanded serially (Figure 2 left), where each branch represents the case of a specific number of iterations to be executed. If there is no dependency between iterations (two parallel lines across the back edge of the loop are used to depict this property), to explore the parallelism, the loop can be expanded in parallel (Figure 2 right). In Figure 2, the probability p_j of having j iterations is associated with the corresponding successor of the OR node.

2.2 Power Management Points

In [21], we proposed that the user or the compiler inserts a power management point (PMP) at the beginning of each program segment. At each PMP, a new speed is computed based on the time taken so far and an estimation of the time for future tasks. If the new speed is different from the current processor speed, the speed/voltage setting procedure is invoked.

For the AND/OR model proposed above, we assume that there is a PMP before each node. Π_c is the maximum schedule length (in cycles) when all tasks in the application run for c_i on a specific system. The average schedule length (in cycles), Π_a, is defined analogously. Both Π_c and Π_a are associated with the PMP before the first node in the graph. For example, in Figure 1a, $\Pi_c = 8$ if the number of processors (N) equals 3, while $\Pi_c = 9$ if $N = 2$ due to a smaller degree of parallelism. Assuming that there are m branches after an OR node, for the PMP before the OR node, two values, Π_c^i and Π_a^i, are associated with each branch b_i ($i = 1, \ldots, m$). These values represent the remaining maximum and average schedule length (in cycles), respectively, when branch b_i is taken. We assume that all these values are obtained from profiling.

2.3 Power and System Models

The power consumption for CMOS processors is dominated by dynamic power dissipation P_d, which is given by: $P_d = C_{ef} \cdot V_{dd}^2 \cdot f$, where C_{ef} is the effective switch capacitance, V_{dd} is the supply voltage and f is the processor clock frequency. Processor speed, f, is almost linearly related

to the supply voltage: $f = k \cdot \frac{(V_{dd}-V_t)^2}{V_{dd}}$, where k is constant and V_t is the threshold voltage [3, 7]. In this paper, we do not consider processor static power dissipation. The processor energy consumed to execute task T_i is given as $E_i = C_{ef} \cdot V_{dd}^2 \cdot C_i$, where C_i is the actual number of cycles needed to execute T_i. When decreasing the processor speed, we also reduce the supply voltage. This reduces processor power consumption cubically and energy consumption quadratically at the expense of linearly decreasing the processor speed and linearly increasing the execution time of a task.

For example, consider a task that executes 10^4 cycles. At the maximum speed f_{max}, it will take $\frac{10^4}{f_{max}}$ time units. If we have $2 \cdot \frac{10^4}{f_{max}}$ time units allocated to this task, we can reduce the processor speed and supply voltage by half while still finishing the task on time (recall that the number of cycles needed to execute a task does not vary for different processor speeds). The processor energy consumption after reducing the speed would be: $E' = P' \cdot t' = C_{ef} \cdot (\frac{V_{dd}}{2})^2 \cdot \frac{f_{max}}{2} \cdot (2 \cdot \frac{10^4}{f_{max}}) = \frac{1}{4} \cdot C_{ef} \cdot V_{dd}^2 \cdot f_{max} \cdot (\frac{10^4}{f_{max}}) = \frac{1}{4} \cdot E$, where E is the processor energy consumption with f_{max}. From now on, we refer to *speed adjustment* as both changing the processor supply voltage and frequency.

Table 1: Speed and Supply Voltage of Transmeta 5400

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

Table 2: Speed and Supply Voltage of Intel XScale

$f(MHz)$	1000	800	600	400	150
$V_{dd}(V)$	1.80	1.60	1.30	1.00	0.75

For real processors, the supply voltage may not decrease linearly with reduced processor speed, which is different from the assumptions in many published papers. In this paper, we consider two real processor models. First, in the Transmeta model [15], the voltage/speed settings are given as in Table 1. There are 16 voltage/speed settings between $700MHz(1.65V)$ and $200MHz(1.10V)$. The second power configuration is the Intel XScale model [14], with the voltage/speed settings as shown in Table 2. Note that Intel XScale has a wider voltage/speed range than the Transmeta model but fewer voltage/speed levels. If the 10^4-cycle task is executed on an Intel XScale processor and is allocated $20\mu s$ instead of $10\mu s$, we can run the task at the speed of $600MHz$ (notice that the

speed of $500MHz$ does not exist in the Intel XScale model and the speed is rounded to the next higher level) instead of $1GHz$. Thus, at the reduced speed, the processor energy consumption is $\frac{1.3^2}{1.8^2} = 52\%$ of that at $1GHz$. Compared with ideal processor model, less energy saving is obtained.

When a processor is not executing a task (i.e., the processor is *idle*), we can put it into a lower power state (e.g., *halt* or *sleep*) to save more energy. For the processors considered in this paper, we assume that they have a *power-saving* state for each different speed, which consumes 15% of the corresponding power, and the transition time out of this power-saving state is very short (several cycles). We also assume a *sleep* state which consumes 1% of the maximal power. However, returning to an operating state from the sleep state takes thousands of cycles [10, 15].

We consider systems that have N identical processors with shared memory. The application characteristics and state are kept in the shared memory. All tasks are put into a global queue when they are ready. Each processor executes the scheduler independently and fetches the tasks from the global queue as needed. For simplicity, no cache hierarchy is considered as we concentrate on processor energy consumption. The shared memory must be accessed in a mutual exclusive way; we consider memory access contention in Section 5.

3 Greedy Algorithm for AND/OR Graph

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 *longest-task-first (LTF)* heuristic [28] which adds tasks to the ready queue in the order of their maximum computation requirement when the tasks become ready at the same time. Note that the LTF heuristic is not optimal. That is, the algorithm may reject an application due to potential deadline miss even when the application is schedulable.

Since tasks exhibit a large variation in actual computation requirement, and in many cases, only need a small fraction of their maximum number of cycles [9], unused cycles (i.e., unused time) can be considered as *slack*. We say that there is *static slack* in the system if an application executes for its maximum number of cycles with f_{max} but still finishes before its deadline. *Dynamic slack*

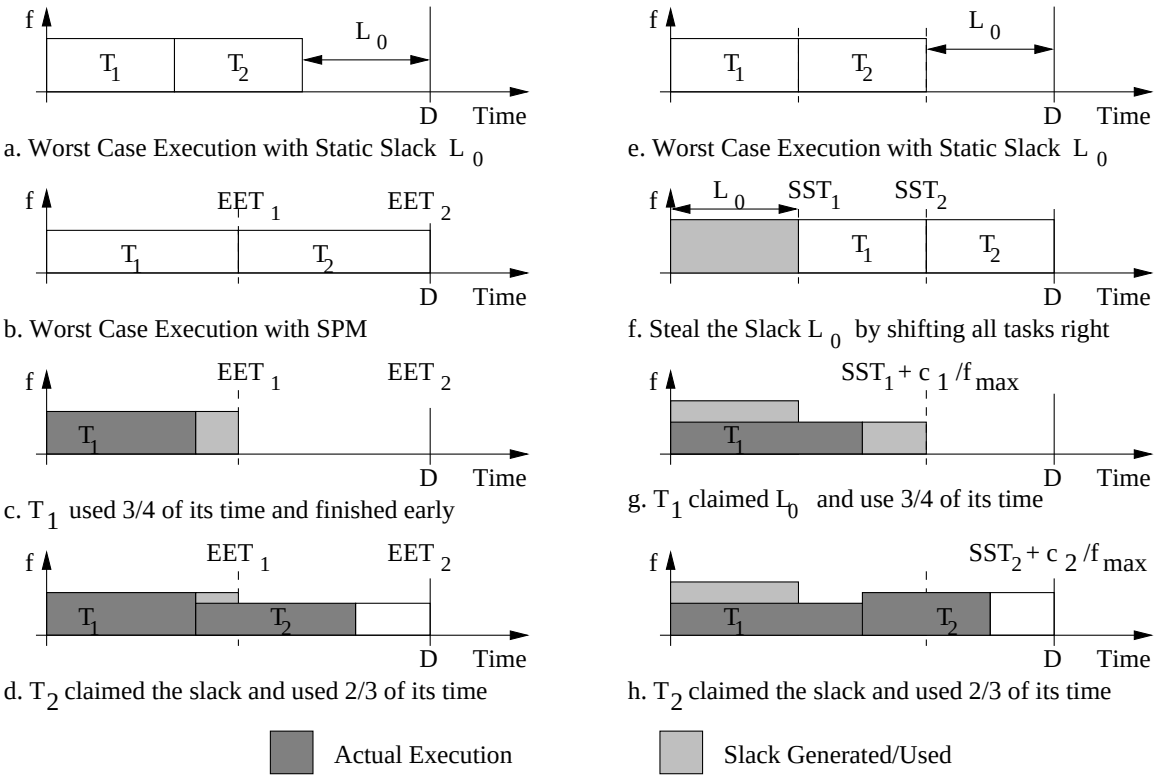


Figure 3: Greedy Algorithms with Static Slack: (a-d) SPM+Greedy; (e-h) Greedy Slack Stealing.

is generated when a task of the application executes less than its maximum number of cycles or the execution of the program does not follow the longest branch after an OR node. If there is some slack in the system and a task T_k can use the processor for longer than its worst case timing requirement (i.e., $\frac{c_k}{f_{max}}$), the system can appropriately slow down the processor while executing the task to save energy.

3.1 SPM + Greedy vs. Greedy Slack Stealing

In static power management (SPM), all tasks share the static slack proportional to their maximum computation requirement; during execution, a dynamic greedy algorithm is applied [28] to reclaim the dynamic slack due to less-than-maximum executions.

In this paper, we propose a *greedy slack stealing* (GSS) technique composed of two parts, namely, *slack stealing* and *dynamic greedy*. In *slack stealing*, the static slack is reclaimed by the first task on each processor while shifting all other tasks toward the deadline. During execution, a dynamic greedy algorithm is applied to reclaim the dynamic slack both due to less-than-maximum

executions and shorter-branch executions. In Figure 3, we illustrate the difference between SPM + Greedy (Figures 3a-d) and Greedy Slack Stealing (Figures 3e-h). Note that this simple example assumes a single processor and an application with no branches (see Figures 4 and 5).

In the figure, the X-axis represents time, the Y-axis represents processor speed (in cycles per time unit), and the area of the box represents the number of cycles needed to execute the task. First, SPM uses the static slack L_0 to uniformly slow down all the tasks and makes the application finish just in time (Figure 3b). The expected end time (EET_i) for T_i in the schedule after applying SPM is also labeled. Figures 3c-d show how the dynamic greedy scheme works over SPM. In contrast, slack stealing shifts all tasks toward the deadline and uses the static slack L_0 for slowing down the processor when executing T_1; the shifted start time (SST_i) for T_i is labeled (Figure 3f). The dynamic greedy after the slack stealing is shown in Figures 3g-h.

In this paper, we will focus on the slack stealing scheme. First, it is easy to claim L_0 and the slack from different branches after the OR nodes using the same idea of slack stealing. Second, SPM can only claim L_0 while missing the slack from different branches after the OR nodes.

3.2 Details of Greedy Slack Stealing (GSS) Algorithm

In the following, we will further explore the application's dynamic characteristics both at the task set level (different execution paths) and at the task level (less-than-maximum execution). GSS algorithm considers the characteristics of the AND/OR model; we show how it is correct with respect to meeting the timing constraints when executing on an N-processor shared memory system. The GSS algorithm consists of two phases: an *off-line phase* and an *on-line phase*.

3.2.1 Off-line Phase of GSS

The off-line phase is used to collect and compute the timing information about an application running on a specific system with processor speed set at f_{max}. It is a two-pass process. To illustrate the concept and show how the off-line phase works, we use the example shown in Figure 4, where each node depicts a task (recall that each synchronization node is considered as a dummy task). The computation nodes are labeled by c_i/a_i (in units of 10^6 cycles).

In the first pass, using list scheduling with LTF heuristic, we generate the *canonical schedule* for all the segments, where all tasks execute for their maximum number of cycles. The maximum

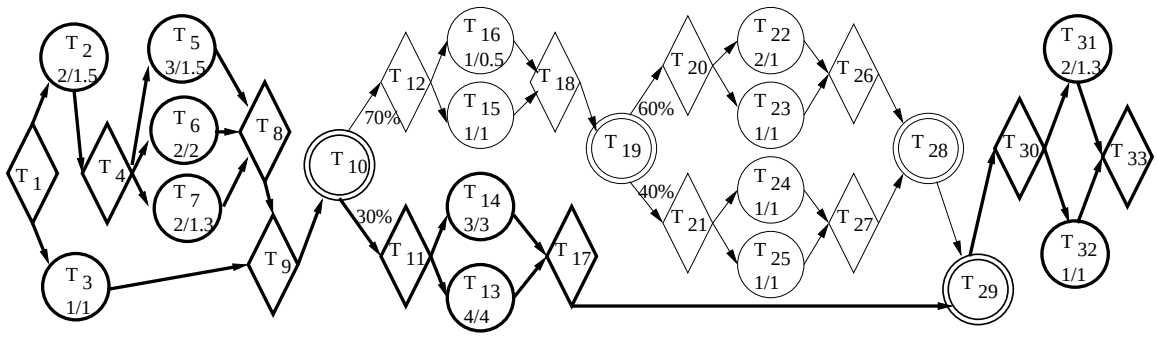


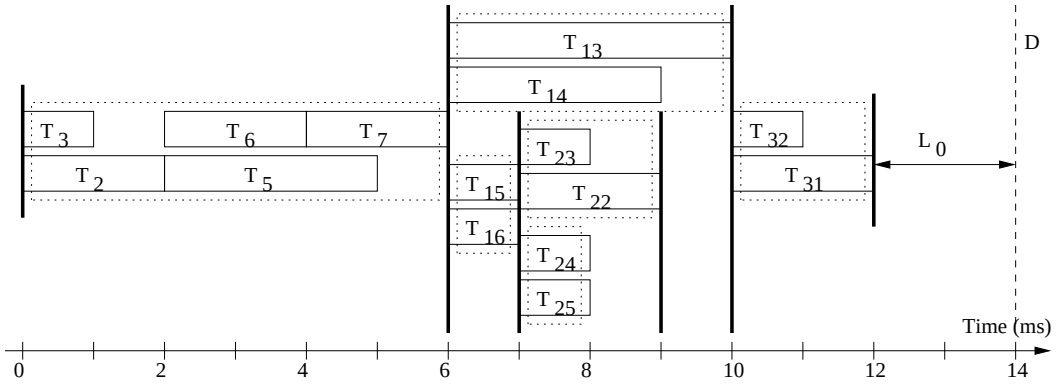
Figure 4: The AND/OR Graph for an Example Application.

schedule length (in cycles) for the application is computed as Π_c. For the branches after an OR node, the maximum remaining schedule length (in cycles) needed along branch b_i is computed as Π_c^i. Further, Π_a and Π_a^i are computed analogously as the weighted average schedule length needed. The *start time* of task T_i in the canonical schedule is recorded as ST_i^c. The *execution order* of task T_i in the canonical schedule is recorded as EO_i. We 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 maximum execution order of its predecessors plus one. Tasks on different branches after an OR node have the same execution order since they are not executed at the same time.

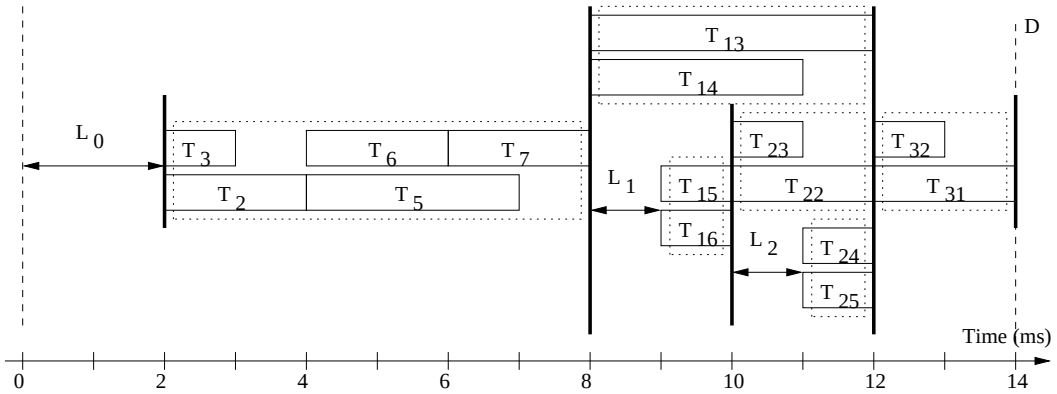
For the example running on a dual-processor system with $f_{max} = 1GHz$, the canonical schedule is shown in Figure 5a. Notice that the synchronization nodes are considered as dummy tasks and are shown as bold vertical lines. The dotted rectangles represent the program segments that are, as a whole, executed or not executed (*integrated segment*). The longest path of the application is shown in bold in Figure 4.

EO_i and ST_i^c for T_i are shown in Table 3. For example, the execution order of T_{29} (an OR node) is larger than the maximum execution order of its predecessors, T_{17} (with $EO_{17} = 14$) and T_{28} (with $EO_{28} = 20$), that is, $EO_{29} = max\{EO_{17}, EO_{28}\} + 1 = max\{14, 20\} + 1 = 21$. T_{11} and T_{12} are on different branches after T_{10} (an OR synchronization node) and will not be executed at the same time; they have the same execution order. The values of ST_i^c are recorded as the time T_i starts execution.

For the example, it takes $\frac{\Pi_c}{f_{max}} = \frac{12 \cdot 10^6}{10^9} = 12ms$. If the deadline $D < 12ms$, the algorithm fails; otherwise, assuming $D = 14ms > 12ms$ (see Figure 5a), the second pass of the off-line phase prepares to steal the slack by shifting the canonical schedule as late as possible toward the deadline.



a. Canonical Schedule



b. Shifted Canonical Schedule

Figure 5: The Schedules for The Example in Figure 4

Table 3: Off-Line Variables of The Example

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}	T_{17}
EO_i	1	2	3	4	5	6	7	8	9	10	11	11	12	13	12	13	14
ST_i^c	0	0	0	2	2	2	4	6	6	6	6	6	6	6	6	6	10
SST_i^c	2	2	2	4	4	4	6	8	8	8	8	9	8	8	9	9	12

	T_{18}	T_{19}	T_{20}	T_{21}	T_{22}	T_{23}	T_{24}	T_{25}	T_{26}	T_{27}	T_{28}	T_{29}	T_{30}	T_{31}	T_{32}	T_{33}	
EO_i	14	15	16	16	17	18	17	18	19	19	20	21	22	23	24	25	
ST_i^c	7	7	7	7	7	7	7	7	9	8	9	10	10	10	10	12	
SST_i^c	10	10	10	11	10	10	11	11	12	12	12	12	12	12	12	14	

For each task T_i, we define the *canonical shifted start time* (SST_i^c) as the time T_i starts execution in the shifted canonical schedule. It is the time by which T_i *must* start execution for the remaining tasks to meet the deadline, providing that the tasks in the same integrated segment have the same shifting factor; that is, if task T_j is in the same segment as T_i, $SST_j^c - ST_j^c = SST_i^c - ST_i^c$. Notice that the shifting is a recursive process when there are nested OR nodes. SST_i^c is used to claim the

slack for T_i at run time. The *canonical shifted end time* for task T_i in the shifted canonical schedule is defined as $SET_i^c = SST_i^c + \frac{c_i}{f_{max}}$.

The shifted canonical schedule for the example is shown in Figure 5b, where L_0, L_1 and L_2 depict the slack stolen by the algorithm. Notice that the tasks in one segment are shifted together and have the same shifting factor. When there are nested OR nodes, different segments may have different shifting factors. We did not consider shifting tasks individually in a segment. For example, for the segment consisting of tasks T_2, T_3, T_5, T_6 and T_7, we may shift T_5 $1ms$ more without violating the timing requirement. But shifting single tasks increases the complexity of the off-line algorithm and we do not expect too much gain from it since that $1ms$ could be claimed by the subsequent tasks in the on-line phase. We leave the examination of this point for future work. After shifting the schedule, SST_i^c is computed for all T_i. SST_i^c values for the example are shown in Table 3.

Given any off-line scheduling heuristic, if the off-line phase does not fail (i.e., $\frac{\Pi_c}{f_{max}} \leq D$ holds), the on-line phase (see next section) can be applied. In the rest of the paper, we assume that $\frac{\Pi_c}{f_{max}} \leq D$ (i.e., we do not consider overload).

3.2.2 On-Line Phase of GSS

Before presenting the on-line phase of the algorithm, we give some definitions. To determine whether a task is *ready* or not, we define the number of *unfinished immediate predecessors (UIP_i)* for each task T_i. UIP_i decreases by one when any predecessor of task T_i finishes execution. Task T_i is *ready* when $UIP_i = 0$. The speed to execute task T_i using GSS is denoted as f_g^i. To maintain the execution order of tasks as in the canonical schedule, the execution order of the next expected task (T_{NET}) is denoted by NET_EO, that is, $EO_{NET} = NET\_EO$. The current time is represented by t.

Suppose that task T_i starts execution at time t_i (i.e., the time t at which one processor fetches T_i and starts to execute it), we define the *estimated end time (EET_i)* as the time at which the task is expected to finish execution if it consumes all the time allocated for it, we have $EET_i = t_i + \frac{c_i}{f_g^i}$.

Initially, the tasks that have no predecessor (root tasks) are put into a *Ready-Q*. For other task T_i, UIP_i is initialized to 1 if T_i is an OR node; otherwise, UIP_i is initialized to the number of predecessors of T_i. The current time t is set to 0 and NET_EO is set to 1.

Algorithm 1 The GSS Algorithm invoked by P_{id}

```
1:  while (1) do
2:    T_k =Head(Ready-Q); /*Just examine the head; do not dequeue.*/
3:    if ( (T_k is an OR node || EO_k == NET_EO) && (UIP_k == 0) ) then
4:      T_k =dequeue(Ready-Q);
5:      NET_EO = NET_EO + 1 ;
6:      if (T_k is a Computation node) then
7:        /*T_k reclaims the slack SST_k^c - t; as proven in Section 3.3, t <= SST_k^c;*/
8:        /*total time allocated to T_k is SST_k^c - t + c_k/f_max = SET_k^c - t*/
9:        f_g^k = c_k/(SET_k^c - t); /*compute the speed for T_k */
10:       /*Let T_next = Head(Ready-Q); */
11:       if (P_y is asleep && EO_next == NET_EO) then
12:         signal(P_y); /*Wake up one sleeping processor*/
13:       end if
14:       Execute T_k at speed f_g^k;
15:     end if
16:     if (T_k is a Computation node or an AND node) then
17:       for ( each successor T_j of T_k) do
18:         UIP_j = UIP_j - 1; /*update successor's variable*/
19:         if (UIP_j == 0) then
20:           /*put T_j into ready-queue if it is ready*/
21:           enqueue(T_j , Read-Q);
22:         end if
23:       end for
24:     else if (T_k is an OR node ) then
25:       NET_EO = EO_k + 1; /*update the next expected task*/
          /*Let T_i be the first node in the path selected at the OR node;*/
26:       UIP_i = 0;
27:       enqueue(T_i , Read-Q); /*put T_i into ready-queue*/
28:     end if
29:   else
30:     wait();
31:   end if
32:   /*go back to fetch a new task to execute*/
33: end while
```

The GSS algorithm is shown in Algorithm 1. In the algorithm, each idle processor tries to fetch the next ready task (line 2 and 3). If the next expected task (T_{NET}) is not ready, the processor goes to sleep (line 30); we use function $wait()$ to put an idle processor to sleep and function $signal(P)$ to wake up processor P. If T_{NET} is ready and is a computation task, the processor computes its new speed, wakes up an idle processor if the task expected after T_{NET} is ready and changes the speed if necessary before executing T_{NET} (from line 6 to 15). The slack reclamation takes place in line 9 when handling a computation task. Suppose T_k starts execution at t_k in the on-line phase, the amount of slack available to T_k is $SST_k^c - t_k$ (note that $t_k \leq SST_k^c$ as proved in Section 3.3).

For the successors of the computation tasks and the AND synchronization nodes, their UIPs are

updated properly and a successor task T_j will be put into the ready queue when it is ready, that is, $UIP_j = 0$ (from line 16 to 23). When entering the ready-queue (line 21), tasks are ordered in their canonical execution order. For an OR synchronization node, the first task in the chosen branch is put into the *Ready-Q* (line 26 to 27). Then the processor will go back and try to fetch another task to execute. The shared memory holds the control information, such as *Ready-Q* and UIP values, which must be updated within a critical section (not shown in the algorithm for simplicity).

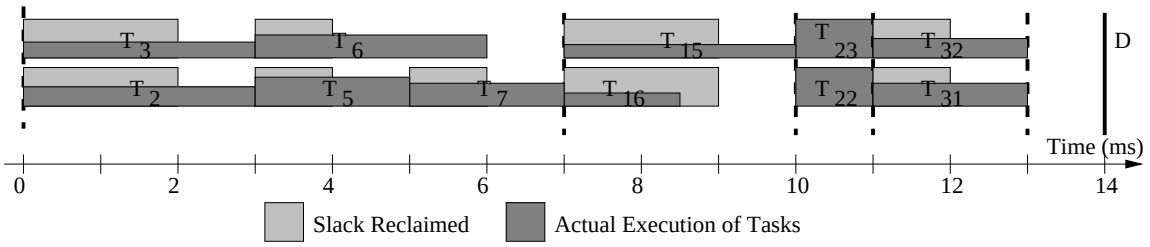


Figure 6: An execution trace for the example of Figure 4

For the example in Figure 4, if the execution follows the lower branch at T_{10} and upper branch at T_{19} and the tasks on this path use their average number of cycles, the single-instance execution trace is shown in Figure 6. Initially, both T_2 and T_3 claim $2ms$ of static slack (the difference between SST_i^c and the time they begin execution). From algorithm GSS line 9, the speed for T_2 and T_3 will be $\frac{f_{max}}{2}$ and $\frac{f_{max}}{3}$, respectively, and we have $EET_2 = 4ms$ and $EET_3 = 3ms$. Since T_2 only uses $\frac{3}{4}$ of the time allocated to it and T_3 uses all the time allocated to it, both T_2 and T_3 actually finish at time $3ms$. Both T_5 and T_6 get $1ms$ of slack (again, the difference between SST_i^c and the time they begin execution) and are supposed to finish at time $7ms$ and $6ms$, at speeds $\frac{3}{4}f_{max}$ and $\frac{2}{3}f_{max}$, respectively. T_7 is supposed to follow T_6 on the upper processor. Since T_5 only uses $\frac{1}{2}$ of the time allocated to it and T_6 uses all its time, T_5 actually finishes earlier than T_6, and T_7 follows T_5 on the lower processor and claims $1ms$ of slack. The algorithm continues and the execution of the example application finishes at time $13ms$, $1ms$ before its deadline.

3.3 Algorithm Analysis

From the above example, we can see that every task T_i starts execution no later than SST_i^c with available slack $SST_i^c - t_i \geq 0$, where t_i is the execution start time for T_i. Thus, the speed for T_i is $f_g^i = \frac{c_i}{SET_i^c - t_i} = \frac{c_i}{SST_i^c + \frac{c_i}{f_{max}} - t_i} \leq f_{max}$ (line 9 in Algorithm 1). Since f_g^i is the speed which

guarantees that task T_i finishes no later than SET_i^c, the application will meet the deadline if the canonical execution meets the deadline.

Hence, for proving the correctness of Algorithm 1, we need to show that any task T_i starts its execution no later than its canonical shifted start time SST_i^c.

Lemma 1 *The execution start time t_i for any task T_i in an AND/OR application under Algorithm 1 is less than or equal to its canonical shifted start time, that is, $t_i \leq SST_i^c$.*

Proof

Define an *execution trace* as a set of tasks that are executed during one running of an application. Notice that *any* task in an application will be in *at least* one execution trace. For *any* execution trace, $TRACE_e$, we will prove that for any T_i in $TRACE_e$, there is a time $t_i (\leq SST_i^c)$ at which the following two conditions are satisfied: (a) T_i is the head of the task queue and is ready (i.e., all its predecessors finished execution); and (b) a free processor is available for T_i.

We first relabel the tasks in $TRACE_e$, starting from 1 to n_e (the number of tasks in $TRACE_e$), in the order of their dispatch in the canonical execution. This step compacts the task numbering to include only the tasks that are in $TRACE_e$. Notice that the execution order of tasks in Algorithm 1 is kept the same as in canonical execution (lines 3, 11 and 26 of Algorithm 1), therefore, for tasks T_i and T_j in $TRACE_e$, we have $SST_i^c \leq SST_j^c$ if $1 \leq i < j \leq n_e$.

The proof then proceeds by induction on i, for task T_i in $TRACE_e$.

Base case: Assume that the number of root tasks that begin execution at time 0 is $m \leq N$, where N is the number of processors. Hence, $t_i = 0 \leq SST_i^c$ for $i = 1, \ldots, m$.

Induction step: Assume $t_i \leq SST_i^c$ for $i = 1, \ldots, k-1$.

Given that the execution order of tasks in Algorithm 1 is kept the same as in the canonical execution (lines 3, 11 and 26 of Algorithm 1), task T_k is the header of the task queue after the first $k-1$ tasks are dispatched. For any predecessor, T_q, of task T_k (i.e., $T_q \rightarrow T_k \in E$), we have[2] $1 \leq q \leq k-1$. Hence, in the shifted canonical schedule, task T_q finishes no later than SST_k^c, that is, $SET_q^c \leq SST_k^c$. During the on-line phase, task T_q starts execution at $t_q \leq SST_q^c$ with speed $f_g^q = \frac{c_q}{SET_q^c - t_q} = \frac{c_q}{SST_q^c + \frac{c_q}{f_{max}} - t_q} \leq f_{max}$. Thus T_q will finish no later than $SET_q^c \leq SST_k^c$. Therefore, task T_k is ready no later than SST_k^c.

[2]Recall that there is no back edge in our AND/OR graph (the back edges are expanded as discussed in Section 2).

Next, for the shifted canonical schedule, before task T_k starts at time SST_k^c, there are at most $r = N - 1$ tasks (among the first $k - 1$ tasks) that are running and will finish later than SST_k^c (since at least one processor is free and fetches T_k at time SST_k^c). For task T_a ($1 \leq a \leq k - 1$) that finishes no later than SST_k^c in the shifted canonical schedule, we have $SET_a^c \leq SST_k^c$. At run time, we have $f_g^a = \frac{c_a}{SET_a^c - t_a} = \frac{c_a}{SST_a^c + \frac{c_a}{f_{max}} - t_a} \leq f_{max}$, that is, T_a will finish no later than $SET_a^c \leq SST_k^c$. Thus, there are at most $r = N - 1$ tasks that could finish later than SST_k^c. That is, at or before time SST_k^c, at least $N - r = 1$ processors are idle and free.

Therefore, one free processor will fetch task T_k no later than SST_k^c and task T_k starts execution at time $t_k \leq SST_k^c$.

Therefore, for any task T_i in an AND/OR application under Algorithm 1, $t_i \leq SST_i^c$.

$\diamondsuit$

4 Speculative Algorithms

While the GSS algorithm is guaranteed to meet the timing constraints, there may be many speed changes during execution since a new speed is computed for each task. It is known that if all tasks execute at the same speed on uniprocessor systems, the minimum energy consumption can be achieved [16]. Considering the speed adjustment overhead, the single speed setting is even more attractive. From this intuition, using the statistical information about an application, we propose the following speculative algorithms.

Based on different strategies, we developed two speculative schemes. One is to *statically predict* an optimal speed (or at most 2 discrete speeds [16]). The second strategy is to *dynamically adjust* the speed while speculating about the remaining work. The point in the application at which we attempt speed changes leads to two sub-schemes. One is to speculate before each task[3], the other is to speculate only after the OR synchronization nodes since different amounts of slack can be expected from different branches after an OR synchronization node.

[3]It is hard to implement this sub-scheme for multi-processor systems because it is difficult to compute the remaining work for the processors. One reason is that when a task ends on one processor, we do not know on which processor other tasks will run and some tasks may be in the middle of execution on other processors. Another reason is the gaps between tasks' execution because the dependencies between tasks are unpredictable for different task sets. So, we consider the speculation before each task only for uniprocessor systems.

4.1 Static Speculation Algorithms

For static speculative algorithms, the speed at which an application should run is decided at the very beginning of the application based on the statistical information about the whole application as: $\frac{\Pi_a}{D}$, where Π_a is the average schedule length (in cycles) needed to execute the application.

If the speculated speed falls between two speed levels ($f_l < \frac{\Pi_a}{D} \leq f_{l+1}$), the static speculation with a single speed (method SS1) will set $f_{ss1} = f_{l+1}$. Alternatively, two speeds can be speculated for the application (method SS2). At the beginning, the speculation speed, f_{ss2}, is set as the lower speed, f_l. After a certain time point (t_{tp}), f_{ss2} is changed to the higher speed level, f_{l+1}. The value of t_{tp} can be statically computed as: $f_l \cdot t_{tp} + f_{l+1} \cdot (D - t_{tp}) = \Pi_a \Rightarrow t_{tp} = \frac{f_{l+1} \cdot D - \Pi_a}{f_{l+1} - f_l}$.

Even after f_{ss1} (or f_{ss2}) is calculated, we choose the maximum speed between f_{ss1} (or f_{ss2}) and f_g^i for task T_i, where f_g^i is computed from GSS. This is to guarantee temporal correctness, since the speculative speed is optimistic and does not take into consideration the worst-case behaviors.

4.2 Adaptive Speculative Algorithm

If the statistical characteristics of tasks in an application vary substantially (e.g., the tasks at the beginning of one application have the average/maximal cycle requirement ratio as 0.9 while tasks at the end of the application the ratio is 0.1), it may be better to re-speculate the speed while the application execution progresses based on the statistical information about the remaining tasks. Here we consider two sub-schemes. First, for uniprocessor systems, we can speculate a new speed before each task begins execution as: $f_{as1} = \frac{\Pi_a^r}{D-t}$, where t is the current time when a new task begins execution and Π_a^r is the average remaining schedule length (in cycles) considering the possibilities to execute different paths. Initially, $\Pi_a^r = \Pi_a$. Note that Π_a is the average schedule length (in cycles) needed to execute the application. After T_i finishes, Π_a^r can be calculated as $\Pi_a^r = \Pi_a^r - a_i$ (recall that this is for uniprocessor systems). When branch b_i is taken after an OR node, Π_a^r will be reset as Π_a^i, where Π_a^i is the average remaining schedule length (in cycles) needed to finish the application after b_i is taken. To guarantee the deadline, the speed f_i for task T_i will be: $f_i = max(f_g^i, f_{as1})$.

Considering the speed adjustment overhead and expecting different amounts of slack from different paths after an OR node, the second scheme speculates the speed only after each OR node. Therefore, the speculative speed would be set as $f_{as2} = \frac{\Pi_a^i}{D-t}$, where t is the current time (when the

OR node is processed) and Π_a^i is the average number of cycles needed when branch b_i is taken after that OR node. Again, to guarantee the deadline, the speed f_i for T_i will be: $f_i = max(f_g^i, f_{as2})$.

Because the speculative algorithms never set a speed below the speed determined by GSS, they will meet the timing constraints if GSS can finish on time. Therefore, from the discussion in Section 3, the speculative algorithms meet the timing constraints if the canonical schedule under the same heuristics finishes on time.

5 Practical Considerations

Issues of speed adjustment overhead and discrete speed levels presented first in [28] can be incorporated into the algorithms described in this paper. In the following, we will discuss two other practical issues: shared memory access contention and the energy consumed during idle period.

Shared Memory Access Contention In shared memory architectures, the data shared among processors (e.g., the UIP structure in Algorithm 1) must be updated in a critical section every time a task is dispatched. There will be additional waiting time due to the shared memory access contention as part of the context switch.

We found that the scheduling algorithm takes approximately 600 cycles without power management. For power management, an additional 500 cycles are needed for claiming the slack and computing the new speed. These values[4] are obtained by running the speed adjustment algorithms on the SimpleScalar micro-architecture simulator using its default configuration [5]. For the experiments in the next section, we assume that the algorithm takes 1100 cycles. Note that, the exact number depends on the number of processors in the system and the number of successors each task can have in the application.

In the worst case, one processor needs to wait until all other processors finish accessing the shared data structures if they start to execute the algorithm at the same time. To account for this waiting time, in the worst case analysis, we need to assume that each processor incurs the longest waiting time for shared memory contention. However, during execution, if one processor does not wait for the longest contention time, the extra time is reclaimed as slack and used to slow

[4]The values were obtained by using up to 6 processors and at most 3 successors for each task.

down the processor. Thus, the algorithm that considers shared memory access contention is more conservative and, on average, more slack is available for the power management schemes.

Idle Period Without considering speed adjustment overhead and/or processor response time, the best strategy is to put any idle processor into the *sleep* state. This unrealistic analysis yields the least processor energy consumption for the idle period but may result in deadline misses because of the transition time from *sleep* to an operating state. To ensure that future tasks finish on time, a processor needs to run at f_{max} when it enters the idle period. This is because it is possible that the next task to be dispatched must run at f_{max} and can spare no slack, not even for changing speeds.

For the simulation presented in Section 6, we assume that the *power-saving* state is used for the idle period in the middle of execution and *sleep* state is used for the idle period appearing at the very end of the schedule (see Section 2.3).

6 Evaluation and Analysis

We implement a shared memory multiprocessor simulator using *C++*. It emulates the execution of an application by simulating the execution at the task level. As mentioned in Section 2, for simplicity, we assume the maximum number of cycles to execute task T_i (i.e., c_i) is independent of the processor speed for a given system architecture [19].

We vary a number of parameters in our experiments: *the number of processors*, *laxity over deadline ratio (LDR)*, *execution time variability*, f_{min} and *overhead of speed adjustment* to see how they affect the processor energy consumption for each scheme. The *laxity* is defined as the difference between application's worst case execution time and its deadline, and *laxity over deadline ratio* is defined as $LDR = \frac{laxity}{deadline}$. It indicates the amount of static slack in specific systems. The *execution time variability*, denoted by α, is defined as the average over the maximum number of cycles needed to execute the application, which indicates the amount of dynamic slack the application will get on average during execution. The value of α_i for task T_i in the application is generated from a discretized normal distribution with average α and standard deviation $0.48 \cdot (1 - \alpha)$ (if $\alpha > 0.5$) or $0.48 \cdot \alpha$ (if $\alpha \leq 0.5$); the 0.48 value comes from discretizing the values of the normal distribution. The actual execution time of T_i follows a similar discretized normal distribution with average $\alpha_i \cdot c_i$. Each point in the presented graphs is an average of 1000 runs. We show results for

both the Transmeta model and the Intel XScale model.

Our simulation study considers the following schemes: static power management (SPM), greedy slack stealing (GSS), adaptive speculation before each task (AS1), adaptive speculation at OR nodes (AS2), static speculation with a single speed (SS1), static speculation with two speeds (SS2) and clairvoyant (CLV). CLV is the ideal case achievable only via post-mortem analysis by running all tasks with single speed (or two discrete speeds) computed from tasks' actual run time, using list scheduling but following the canonical execution order. For each scheme, the processor energy consumption is compared to that of no power management (NPM) where every task runs at f_{max}. Recall that the *power-saving* and *sleep* states are used for different idle periods in the schedule.

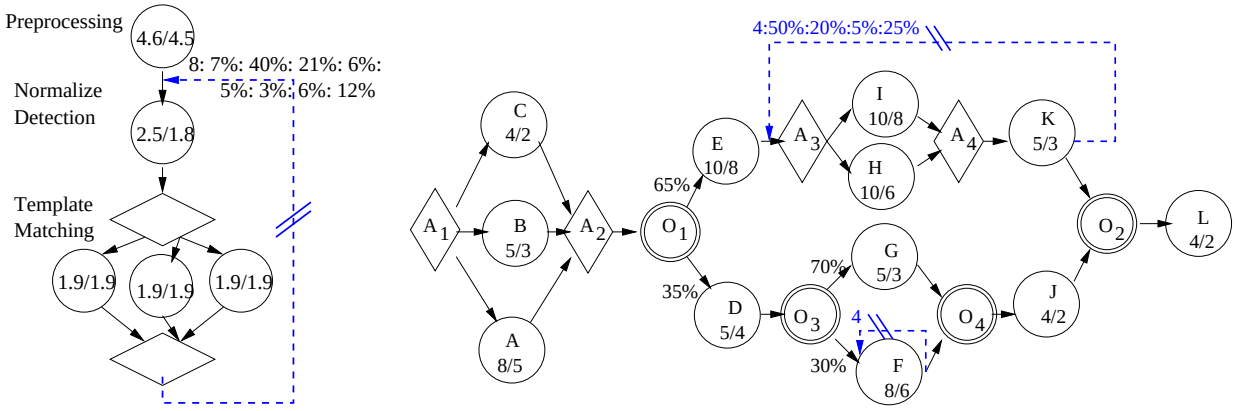


Figure 7: Dependence Graph for: ATR (left) and a Synthetic Application (right)

We consider an application of automated target recognition (ATR) (left part of Figure 7) and a synthetic application (Figure 7, right), with c_i and a_i in units of 10^5 cycles. ATR is a real life application provided to us by BAE systems. ATR detects regions of interest (ROIs) in one frame and compares the ROIs with certain templates. The number of templates that each ROI should be matched to is three, the maximum number of ROIs in one frame is eight and p_i $(i = 1, \ldots, 8)$ is the probability of having i ROIs in one frame (the values in the graphs are from processing 180 successive frames). The loops in the dependence graphs can be expanded as discussed in Section 2. The numbers associated with each loop are the maximum number of iterations paired with the probabilities of having a specific number of iterations. If there is only one number, it is the exact number of iterations during execution.

6.1 The Effect of *Laxity over Deadline Ratio (LDR)*

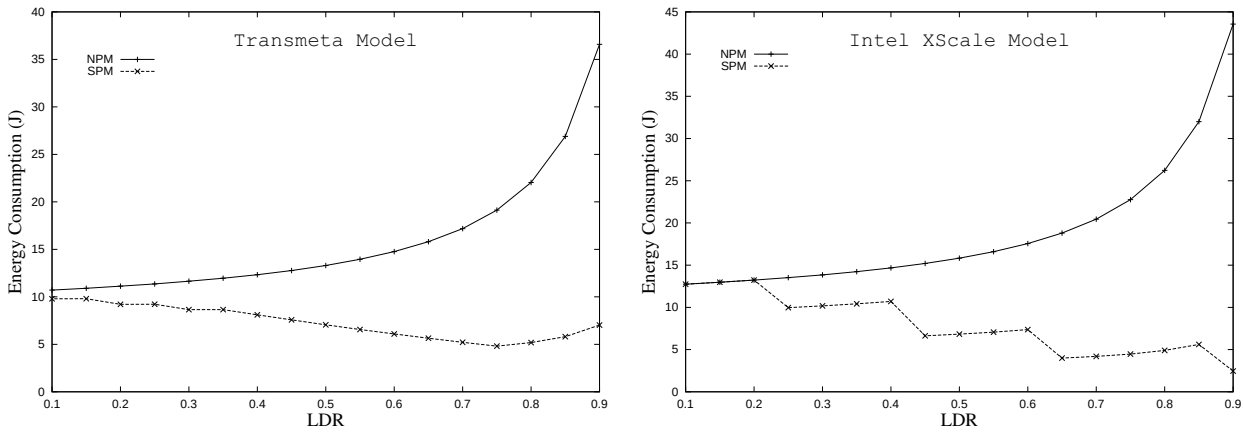


Figure 8: The Absolute Energy Consumption vs. LDR for ATR Running on a Uniprocessor System with $\alpha \approx 0.95$ and $overhead = 5\mu s$. Assuming that $C_{ef} = 10^{-6}$.

We start by discussing the effect of LDR on energy savings for uniprocessor systems. The reason is that we want to compare the effectiveness of the schemes on different architectures (uniprocessor and multiprocessor). As the maximum schedule length (in cycles) of an application is assumed to be fixed for a specific system, LDR will be changed by varying the application's deadline. Further, as mentioned above, the AS1 scheme is only used in uniprocessor systems.

First, we run ATR on a uniprocessor system with $\alpha \approx 0.95$ (the value was measured and means that there is little dynamic slack) and $overhead = 5\mu s$. For different LDR values (i.e., different deadlines), Figure 8 shows the absolute energy consumed by NPM and SPM. The Intel XScale model (Figure 8 right) has small number of discrete speeds and thus, when $LDR \leq 0.2$, SPM runs at the same speed ($1GHz$) as NPM and consumes the same amount of energy. When LDR increases, there is more static slack in the system and therefore energy consumption should decrease for SPM (since the slowdown capability is bigger). In the Transmeta model (Figure 8 left), when $LDR \geq 0.75$, the energy consumption for SPM increases with increased LDR. The reason is that every task runs at f_{min} when $LDR \geq 0.75$ and the idle time increases (and thus the idle energy consumed increases) when LDR increases (recall that we change LDR by varying the application's deadline). For NPM, the energy consumption increases with increased LDR since it will consume more idle energy.

Figure 9 shows the *normalized* energy consumption for SPM and all other dynamic schemes

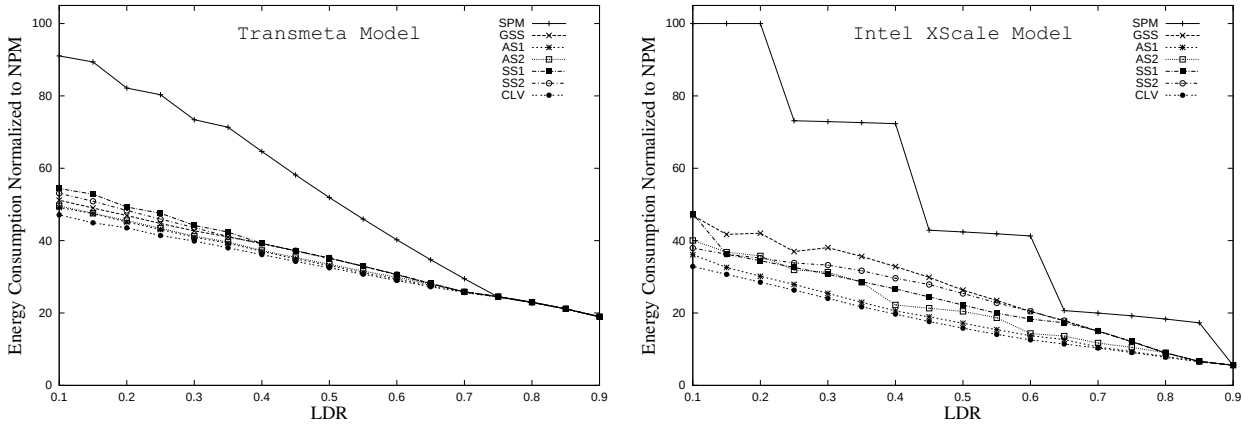


Figure 9: Energy Normalized to NPM vs. LDR for ATR Running on a Uniprocessor System with $\alpha \approx 0.95$ and $overhead = 5\mu s$.

with NPM as the baseline. It can be seen that the processor energy consumed is approximately the same for all dynamic schemes. The reason is that the maximum number of regions of interest (ROIs) in one frame is 8 and the average number of ROIs is three. On average, the speed for most of the tasks is around f_{min} by dynamically reclaiming the slack from branching even when no static slack is considered. When the processor is simulated following the Intel XScale model (Figure 9 right), where there are fewer speed levels but a wider speed range between levels, the normalized energy for SPM incurs sharp changes. These changes correspond to the downgrade of speed from one level to the next level. For example, when $LDR \leq 0.2$, SPM needs to run at $1GHz$ while when $0.2 < LDR \leq 0.4$ SPM only runs at $800MHz$.

From the results, we can also see that the greedy scheme is better than static speculation schemes and worse than adaptive speculation schemes when the processor is modeled as Transmeta (Figure 9 left). When using Intel XScale model (Figure 9 right), the greedy scheme is worse than all speculative schemes. The reason is that Transmeta has a relatively higher f_{min} ($200MHz$ over $700MHz$) than Intel XScale ($150MHz$ over $1GHz$), which prevents the greedy scheme from using all the slack at the very beginning. For all the dynamic schemes, adaptive speculation after every new task (AS1) performs the best and is very close to the clairvoyant scheme (CLV) even accounting for overheads. The adaptive speculation schemes are better than static speculation schemes because they take into account the remaining tasks.

We executed ATR on 2, 4 and 6 processors and obtained similar results; we show only the

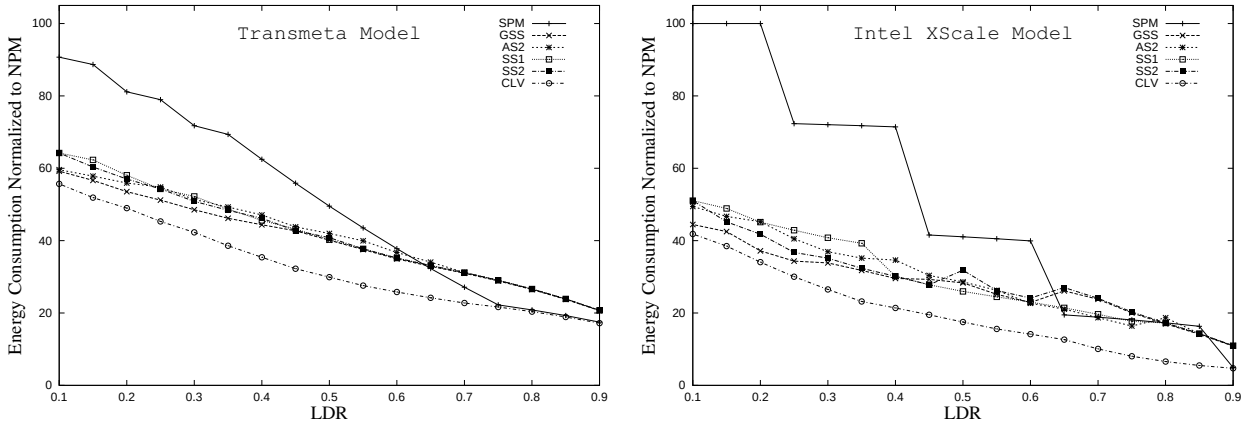


Figure 10: Energy Normalized to NPM vs. LDR for ATR Running on a 6-Processor System with $\alpha \approx 0.95$ and $overhead = 5\mu s$.

normalized energy consumption for 6 processors in Figure 10. The main difference observed for different number of processors is that the total energy consumption is higher for more processors. This is because the more processors there are, the more synchronization between processors and therefore, the larger amount of idle time in the schedule. A surprising result, caused by the energy consumption during idle period can be seen: at high LDR, SPM is better than dynamic schemes. The reason is that the idle period in the schedule consumes 15% of the maximum power in dynamic schemes while it only consumes around 15% of the minimum power (i.e., around 1% of maximum power) in SPM.

6.2 The Effect of f_{min} and Speed Levels

We expected that the speculative schemes 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 [21, 28]. However, when the minimum speed is bounded by f_{min}, it prevents the greedy scheme from using all the available slack at the very beginning and forces some slack to be saved for future use. Fewer speed levels also prevent the greedy scheme from using slack early by decreasing the probability of speed changes: the closer the speeds are to each other, the higher the probability that a small amount of slack will cause a speed change. As a result, the greediness of the greedy scheme is moderated with a higher f_{min} and fewer speed levels. To see how f_{min} and the number of speed levels affect

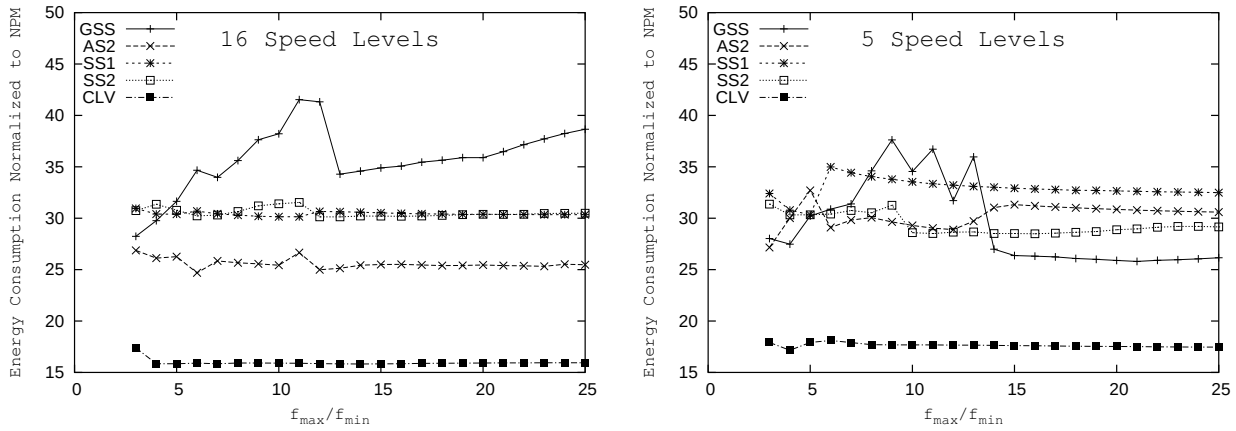


Figure 11: Energy Normalized to NMP vs. $\frac{f_{max}}{f_{min}}$ for ATR running on Dual-Processor Systems with $LDR = 0.2$, $overhead = 5\mu s$.

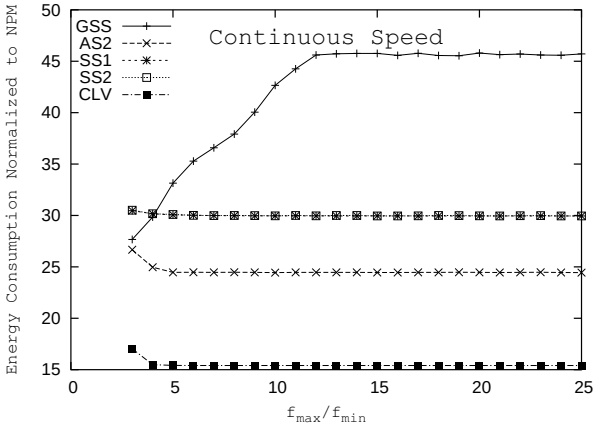


Figure 12: Energy Normalized to NPM vs. $\frac{f_{max}}{f_{min}}$ for ATR Running on a Dual-Processor System with $LDR = 0.2$, $overhead = 5\mu s$ and Continuous Speed.

performance, in Figure 11 we plot energy savings for different values of $\frac{f_{max}}{f_{min}}$ and different number of speed levels between f_{max} and f_{min}.

With fixed $f_{max} = 1GHz$, we study the effect of f_{min} by setting the factor of $\frac{f_{max}}{f_{min}}$ with different values. Assuming 5 or 16 speed levels equally distributed between f_{max} and f_{min}, Figure 11 shows the results for ATR running on dual-processor systems with $LDR = 0.2$ and $overhead = 5\mu s$. In this experiment, we do not have corresponding voltage value for each arbitrary speed level, for simplicity, we assume $P_d \approx f^3$ [18]. When there are 5 speed levels (Figure 11 right), the greedy scheme is almost the same or better than all the speculative schemes even when $\frac{f_{max}}{f_{min}} = 25$ (i.e.,

$f_{min} = 40MHz$), which coincides with our previous observation that a few speed levels (4-6) is good enough for greedy power management [28]. The reason is that a few speed levels will decrease the probability that greedy scheme changes speed and thus some slack is saved for future tasks. For 16 speed levels (Figure 11 left), the greedy scheme becomes worse than the speculative schemes as expected when $\frac{f_{max}}{f_{min}} \geq 5$ (i.e., $f_{min} \leq 200MHz$).

From Figure 11, notice that the greedy scheme does not always become worse with increased $\frac{f_{max}}{f_{min}}$ (that is, decreased f_{min}). The non-monotonic change in the performance of GSS is a direct result of the quantized speed levels. When we use continuous speeds between f_{max} and f_{min} as shown in Figure 12, the energy consumption of GSS increases monotonically with decreased f_{min}. Overall, we can see that the performance of the speculative schemes is 10% to 15% worse than the clairvoyant scheme. For higher values of LDR, similiar results are obtained.

6.3 The Effect of *Execution Time Variability (α)*

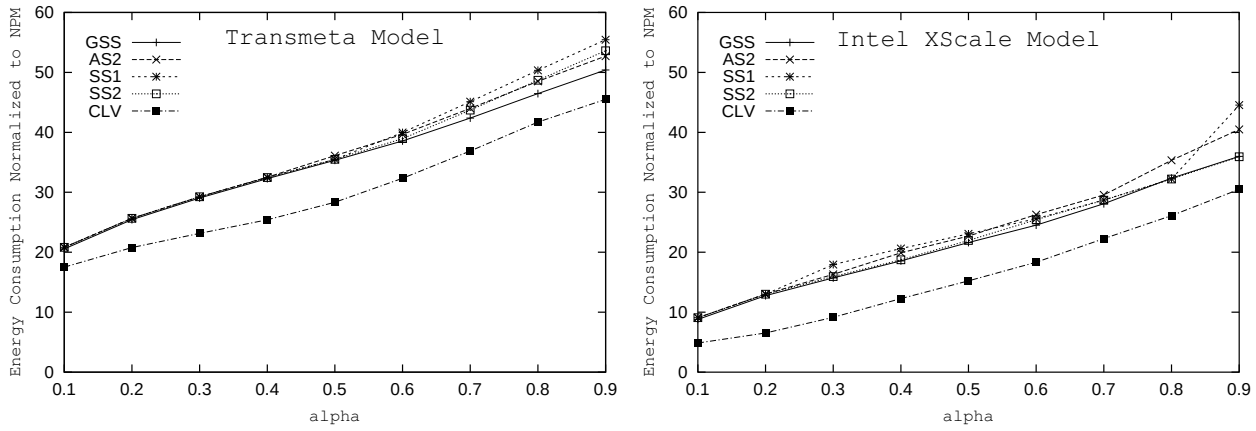


Figure 13: Energy Normalized to NPM vs. α for Synthetic Application Running on a Dual-Processor System with $LDR = 0.2$ and $overhead = 5\mu s$.

For the synthetic application running on a dual-processor system with $LDR = 0.2$ and $overhead = 5\mu s$, the normalized processor energy for each scheme is shown in Figure 13 as a function of α. Since increasing LDR and decreasing α have the same effect on the available slack in a system, the shapes of the curves for dynamic schemes correspond to when LDR was changed. Again, the greedy scheme is as good as the speculation schemes.

6.4 The Effect of the *Overhead*

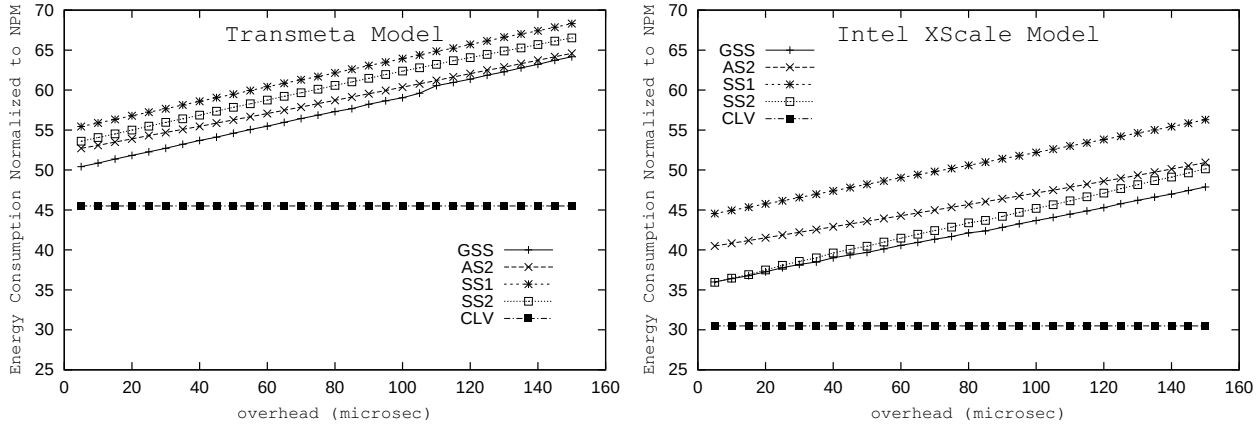


Figure 14: Energy Normalized to NPM vs. $overhead$ for Synthetic Application Running on a Dual-Processor System with $LDR = 0.2$ and $\alpha \approx 0.9$.

The time overhead of speed adjustment varies a lot based on different architectures. For example, an AMD K6-2+ was measured to have an overhead of $400\mu s$ for changing voltage and $40\mu s$ for the frequency [22]. The lpARM processor needs $70\mu s$ to change voltage/speed [4]. For Intel XScale, the maximal overhead for changing both voltage and speed was measured as $30\mu s$ while the StrongARM SA-1110 needs $150\mu s$ to change the speed [23]. With new technology, the overhead of voltage/speed adjustment is expected to decrease in the future. In this paper, the range of overhead considered is from $5\mu s$ to $150\mu s$.

The results in Figure 14 show the normalized processor energy consumption of each scheme for the synthetic application running on a dual-processor system with $\alpha = 0.9$ and $LDR = 0.2$ (for smaller α or bigger LDR, the schemes will set the speed close to f_{min} and the effect of the overhead will decrease). In contrast with the Transmeta model, static speculation with 2 speeds (SS2) is much better than static speculation with single speed (SS1) when using Intel XScale model due to the wider range between adjacent levels in the Intel XScale model.

If the application is smaller, such as removing the loops in the synthetic application (Figure 7), the effect of overhead increases. Figure 15 shows the effect of overhead on energy consumption for the application without loops by setting $\alpha = 0.9$ and $LDR = 0.2$. From these figures, we can see similar behavior for all the schemes, but all of them perform worse. A surprising result is that, when processors are configured as the Intel XScale model, the normalized energy consumption for

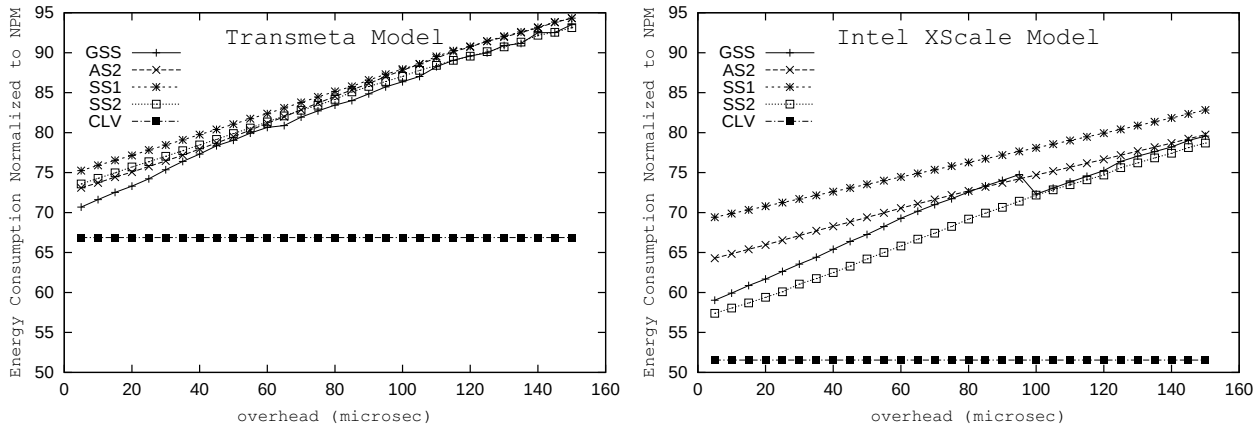


Figure 15: Energy Normalized to NPM vs. $overhead$ for Smaller Application Running on a Dual-Processor System with $LDR = 0.2$ and $\alpha \approx 0.9$.

GSS drops around $overhead = 95\mu s$. The reason is that, with increased overhead ($= 100\mu s$), the static slack is not enough for GSS to run the first few tasks at a lower speed. The speed for the first few tasks then increases one level and saves some slack for future tasks. Subsequently, the speed for all tasks becomes smoother, consequently consuming less energy.

7 Conclusion

In this paper, we extend the AND/OR model by adding probabilities to each branch after the OR nodes. 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. We proposed the *greedy slack stealing* algorithm for the AND/OR model applications executing on N-processor shared memory systems and proved its correctness in meeting the timing constraints. Then, using statistical information about applications, we proposed a few speculative algorithms to save more energy by reducing the number of speed changes (and thus the overhead) while ensuring that the applications meet their timing constraints. Some practical problems were also addressed, such as shared memory access contention, energy consumed during idle state as well as speed adjustment overhead and discrete speed levels.

The performance of all the algorithms in terms of processor energy savings is analyzed through simulations. The greedy slack stealing algorithm performs surprisingly better than some speculative algorithms in two situations: one is when the minimum speed prevents the greedy algorithm

from using up the slack very aggressively; the other is when there are only a few speed levels which prevent the greedy algorithm from changing speeds very frequently. The greedy scheme is good enough when the system has a reasonable minimum speed or there are only a few (4-6) speed levels for the processors. The dynamic schemes become worse relative to static power management (SPM) when the size of the application becomes smaller, *laxity over deadline ratio (LDR)* becomes smaller and *execution time variability* (α) does not have wide fluctuations, since most of the slack will be used to cover the speed adjustment overhead. When the number of processors increases, the performance of the dynamic schemes decreases due to limited parallelism and frequent processor idleness forced by synchronization among tasks.

The algorithms described here for shared memory systems cannot be directly applied to distributed systems, in which the communication time plays an important role. Energy efficient scheduling algorithms for distributed systems are deserve further exploration, such as [20].

Acknowledgments

This work has been supported by the Defense Advanced Research Projects Agency through the PARTS project (Contract F33615-00-C-1736). The authors would like to thank Nevine AbouGhazaleh for providing the trace data of ATR application and the referees' criticisms and suggestions that helped us in rewriting the paper in a better form.

References

[1] N. AbouGhazaleh, D. Mossé, B. R. Childers, and R. Melhem. Toward the placement of power management points in real time applications. In *Proc. of Workshop on Compilers and Operating Systems for Low Power*, 2001.

[2] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proc. of The 22^{th} IEEE Real-Time Systems Symposium*, Dec. 2001.

[3] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proc. of The HICSS Conference*, pages 288–297, Jan. 1995.

[4] 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*, 35(11):1571–1580, 2000.

[5] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Department of Computer Science, University of Wisconsin-Madison, Jun. 1997.

[6] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos. Data driven signal processing: An approach for energy efficient computing. In *Proc. Int'l Symposium on Low-Power Electronic Devices*, 1996.

[7] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuit*, 27(4):473–484, 1992.

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

[9] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. of The International Conference on Computer-Aided Design*, pages 598–604, Nov. 1997.

[10] S. Gary, P. Ippolito, G. Gerosa, C. Dietz, J. Eno, and H. Sanchez. Powerpc 603^{TM}, a microprocessor for portable computers. *IEEE Design & Test of Computers*, 11(4):14–23, 1994.

[11] D. W. Gillies and J. W.-S. Liu. Scheduling tasks with and/or precedence constraints. *SIAM J. Compu.*, 24(4):797–810, 1995.

[12] F. Gruian. System-level design methods for low-energy architectures containing variable voltage processors. In *Proc. of The Workshop on Power-Aware Computing Systems*, Nov. 2000.

[13] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *Proc. of the 2001 International Symposium on Low Power Electronics and Design*, Aug. 2001.

[14] http://developer.intel.com/design/intelxscale/benchmarks.htm.

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

[16] T. Ishihara and H. Yauura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of The 1998 International Symposium on Low Power Electronics and Design*, pages 197–202, Aug. 1998.

[17] P. Kumar and M. Srivastava. Predictive strategies for low-power rtos scheduling. In *Proc. of the 2000 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Sep. 2000.

[18] P. Kumar and M. Srivastava. Power-aware multimedia systems using run-time prediction. In *Proc. of the* 14^{th} *International Conference on VLSI Design*, pages 64 – 69, Bangalore, India, Jan. 2001.

[19] R. Melhem, N. AbouGhazaleh, H. Aydin, and Daniel Mossé. *Power Management Points in Power-Aware Real-Time Systems*, chapter 7, pages 127–152. Power Aware Computing. Plenum/Kluwer Publishers, 2002.

[20] R. Mishra, N. Rastogi, D. Zhu, D. Mossé, and R. Melhem. Energy aware scheduling for distributed real-time systems. In *Proc. of International Parallel and Distributed Processing Symposium*, Nice, France, Apr. 2003.

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

[22] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of* 18^{th} *ACM Symposium on Operating Systems Principles (SOSP'01)*, Oct. 2001.

[23] V. Raghunathan, P. Spanos, and M. B. Srivastava. Adaptive power-fidelity in energy aware wireless embedded systems. In *Proc. of The* 21^{th} *IEEE Real-Time Systems Symposium*, Orlando, FL, Nov. 2000.

[24] J. A. Ratches, C. P. Walters, R. G. Buser, and B. D. Guenther. Aided and automatic target recognition based upon sensory inputs from image forming systems. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 19(9):1004–1019, 1997.

[25] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design & Test of Computers*, 18(2):20–30, 2001.

[26] 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 & Test of Computers*, 18(5):46–58, 2001.

[27] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proc. of The* 36^{th} *Annual Symposium on Foundations of Computer Science*, pages 374–382, Oct. 1995.

[28] D. Zhu, R. Melhem, and B. R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):686–700, 2003.

Dakai Zhu received a B.E. in Computer Science and Engineering from Xi'an Jiaotong University in 1996, an M.E. degree in Computer Science and Technology from Tsinghua University in 1999 and an M.S. degree in Computer Science from University of Pittsburgh in 2001. Currently he is a Ph.D. student in University of Pittsburgh and is researching on power and fault tolerance management for parallel real-time systems. He is a student member of IEEE.

Daniel Mossé received the BS degree in mathematics from the University of Brasilia in 1986 and the MS and PhD degrees in computer science from the University of Maryland in 1990 and 1993, respectively. He joined the faculty of the University of Pittsburgh in 1992, where he is currently an associate processor. His research interests include fault-tolerant and real-time systems, as well as networking. The major thrust of his research in the new millenium is power-aware computing and security. He has served on program committees for all major IEEE-sponsored real-time related conferences and as program and general chairs for RTAS and RT Education Workshop. Typically funded by the US National Science Foundation and US Defense Advanced Research Projects Agency, his projects combine theoretical results and implementations. He is on the editorial board of the *IEEE Transactions on Comuputers* and is a member of the IEEE Computer Society and of the ACM.

Rami Melhem received a B.E. in Electrical Engineering from Cairo University in 1976, an M.A. degree in Mathematics and an M.S. degree in Computer Science from the University of Pittsburgh in 1981, and a Ph.D. degree in Computer Science from the University of Pittsburgh in 1983. He was an Assistant Professor at Purdue University prior to joining the faculty of The University of Pittsburgh in 1986, where he is currently a Professor of Computer Science and Electrical Engineering and the Chair of the Computer Science Department. His research interests include Real-Time and Fault-Tolerant Systems, Optical Interconnection Networks, High Performance Computing and Parallel Computer Architectures. Dr. Melhem served on program committees of numerous conferences and workshops and was the general chair for the third International Conference on Massively Parallel Processing Using Optical Interconnections. He was on the editorial board of the IEEE Transactions on Computers and the IEEE Transactions on Parallel and Distributed systems. He is serving on the advisory boards of the IEEE technical committees on Parallel Processing and on Computer Architecture. He is the editor for the Kluwer/Plenum Book Series in Computer Science and is on the editorial board of the Computer Architecture Letters. Dr. Melhem is a fellow of IEEE and a member of the ACM.