Leakage-Aware Dynamic Scheduling for Real-Time Adaptive Applications on Multiprocessor Systems

Heng Yu
National University of Singapore
g0600105@nus.edu.sg

Bharadwaj Veeravalli
National University of Singapore
elebv@nus.edu.sg

Yajun Ha
National University of Singapore
elehy@nus.edu.sg

ABSTRACT

While performance-adaptable applications are gaining increased popularity on embedded systems (especially multimedia applications), efficient scheduling methods are necessary to explore such feature to achieve the most performance outcome. In addition to conventional scheduling requirements such as real-time and dynamic power, emerging challenges such as leakage power and multiprocessors further complicate the formulation and solution of adaptive application scheduling problems. In this paper, we propose a runtime adaptive application scheduling scheme that efficiently distributes the runtime slack in a task graph, to achieve maximized performance under timing and dynamic/leakage energy constraints. A guided-search heuristics is proposed to select the best-fit frequency levels that maximize the additional program cycles of adaptive tasks. Moreover, we devise a two-stage receiver task selection method that runs efficiently at runtime, in order to quickly find the slack distribution targets. Experiments on synthesized tasks and a JPEG2000 decoder are conducted to justify our approach. Results show that our method achieves at least 25% runtime performance increase compared to contemporary approaches, incurring negligible runtime overhead.

Categories and Subject Descriptors: C.3 [Special-purpose and application-based systems]: Real-time and embedded systems

General Terms: Algorithms, Design
Keywords: Dynamic scheduling, Adaptive applications

1. INTRODUCTION

1.1 Background

Adaptive applications are receiving growing attentions owing to their capability to provide scalable performance quality in reaction to the execution environment. The more program cycles and/or energy budget assigned to an adaptive application, the higher performance quality it can achieve. One example is the Scalable Video Coding (SVC) scheme in H.264/MPEG-4 standard, which provides customized service quality to accommodate various network and device conditions [12]. The other example is JPEG2000 codec supporting multiple playback resolutions [11]. Rather than simply completing or failing the execution, adaptive applications usually define multiple execution granularities such that a finer-grained version results in better performance, at the price of increased program cycles and energy.

The strategy of scheduling adaptive tasks on embedded systems aims at maximizing program execution cycles (hence the performance quality), while meeting real-time deadlines and not neglecting program energy budgets. A normal Dynamic Voltage Scheduling (DVS) [6] technique can effectively reduce system energy by scaling down processor frequency; however, it gains no program quality improvement with unchanged execution cycles. Improved DVS techniques [2]–[4] utilize the energy saved by scaling down processor frequency to generate extra execution cycles, thus achieve the maximum quality under the same energy budget. A common scenario to apply this strategy is in dynamic scheduling, where tasks finish earlier than worst case execution times (WCETs) [5]. The slack time is then used by its successors to generate extra execution cycles budgeted on the associated slack energy. Since the actual execution time of a task is usually a fraction of the WCET, the runtime scheduling gain can be significant [3], [4].

Semiconductor technology trends further complicates the dynamic adaptive application scheduling. First of all, the significance of leakage power necessitates combining both dynamic and leakage energy consumption into the scheduling framework. Moreover, multiprocessor platforms allow multiple successor tasks (referred to as *slack receivers*) to share a slack, thus careful slack distribution methods are required to achieve optimal adaptive gain. Further, multiprocessors enforce complex precedence constraints to its mapped task graph, thus a slack receiver selection methodology should be developed, such that tasks with the most cycle increase potential are selected for the dynamic algorithm for slack distribution.

According to [4], on a single processor system, both slack time and energy can be fully consumed simultaneously if the slack receiver executes the increased cycles at the same frequency as the slack generator. A naive yet under-optimized approach for multiprocessor systems can be distributing the slack time and energy to the direct slack receiver on the same processor, such that system timing and energy constraints are not violated. However, with frequency scaling capability, other parallel slack receivers can take the slack time and scale down frequency to save energy, while the saved energy can be used to further increase cycles under timing constraints.

1.2 Illustrative Example

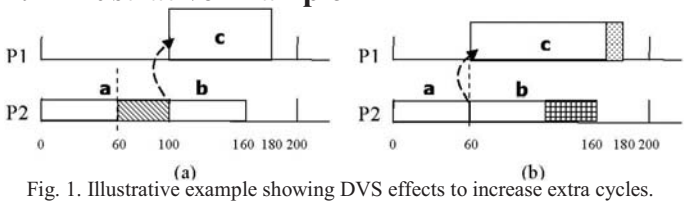

Fig. 1. Illustrative example showing DVS effects to increase extra cycles.

TABLE I: LIST OF FREQUENCIES AND THE CORRESPONDING ENERGY-PER-CYCLE

Freq.(MHz)	400	300	200	100
Ecyc(nJ)	15	12	10	5

Fig. 1 shows a set of three adaptive tasks *a*, *b*, and *c*. Task *a* has WCET 100μs and generates 40μs slack time (Fig. 1(a)). We refer to the successor tasks that qualify to receive the slack simply as *slack*

DAC2010, June13-18, 2010, Anaheim, California, USA.

receivers. Thus, the slack receivers *b* and *c* execute for 60μs and 80μs, respectively, before slack distribution. *a* and *b* run at 100MHz and *c* runs at 400MHz. The energy consumption of each task can be calculated from TABLE-I, which lists the available processor frequencies and the corresponding per-cycle energy consumption *Ecyc* at each frequency. Since task *a* has slack time of 40μs and runs at 100MHz, its slack energy is calculated as 20*nJ*. According to [4], *b* takes the full slack energy while not violating timing constraints, and generates 4000 extra cycles as shown in the grid area in Fig. 1(b). Moreover, task *c* also acquires 40μs slack time but no slack energy left. We observe that *c* can use the slack time to scale down its frequency to 300MHz, and save some energy for extra cycle generation. The execution time of c after scaling down becomes 106.7*ns*, and energy saved is 400*MHz** 80*ns** (15*nJ*-12*nJ*) = 96*nJ*. The saved energy is enough for $c\psi$to run till its deadline at $180ns$, generating 3990 cycles as shown in dotted area in Fig. 1(b). In total, the slack energy is capable of generating 4000 extra cycles for *b*, while DVS is capable of generating another 3990 extra cycles for *c*.

The above example identifies the multiprocessor challenges that we address in our paper. Firstly, slack energy is shared amongst slack receivers but slack time is duplicated, thus creating opportunities during runtime to avail additional program execution cycles. This results in considerable gain in energy savings, especially when the actual execution time is shorter. Secondly, DVS can be used to leverage on the amount of slack time and energy for tapping additional cycles. Thus, it is desired for a methodology that can fully utilize the slack times and energy to maximize the number of cycles by cleverly adjusting the task frequencies if there exists an imbalance between the required slack times and energy.

1.3 Scope of the Paper

In this paper, we focus on the design of a dynamic scheduling algorithm for adaptive tasks on multiprocessor systems with leakage-aware energy model. With the objective of achieving maximum additional cycles from the runtime slack, our algorithm tackles the following two fundamental questions: (1) *how to determine the frequencies of a given set of slack receivers, so that slack time and energy are fully utilized to generate maximum extra cycles*; (2) *how to select the best subset of receivers that provide the most cycles*.

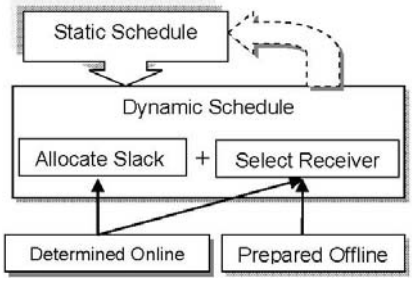

Fig. 2. Framework of our methodology.

This dynamic scheduling is a part of the framework in Fig. 2. The framework consists of the Static Scheduling and Dynamic Scheduling blocks. We focus on the Dynamic Scheduling block in this paper, and assume the Static Scheduling block generates a static schedule to the Dynamic Scheduling block. The static schedule decides the task-processor mapping and the initial task frequencies. Although our dynamic scheduling algorithm can be applied to arbitrary static schedule results, well designed static scheduling can improve our algorithm performance. In Section V, we suggest the rules to build static schedules that favor our dynamic scheduling algorithm. The dynamic scheduling algorithm takes the following steps. First of all, we derive an online heuristic that performs guided search on the largest increase amount of cycles, by selectively adjusting the receiver frequencies based on the runtime conditions. In addition, we select the best receiver candidates based on a graph decomposition scheme, which can be performed offline to reduce runtime overhead. At runtime, the picked candidates can be efficiently narrowed down to generate the slack receiver set.

To prove the viability of our algorithm, we conducted experiments on both synthesized and a JPEG2000 applications to examine the scheduling gain and runtime efficiency. Results show that our method achieves at least 25% runtime performance increase compared to contemporary approaches, and incurs very small runtime overhead.

The remainder of the paper is organized as follows. Section II presents the related work. Section III describes task models used. Section IV elaborates on the frequency selection process. Section V presents the receiver selection procedure. Section VI discusses our experimental results, and Section VII concludes the paper.

2. RELATED WORK

Representations of adaptive tasks are found differently modeled, for example multi-version tasks [2], and Imprecise-Computation tasks [1]. Scheduling adaptive tasks have been proposed by several pioneering works. [2] proposes a MV-Pack algorithm that selects proper version for each task instance in order to maximize rewards under a rechargeable energy budget model. The system has a single processor with DVS capability. Aydin *et al* [9] provide an optimal static solution for the Imprecise-Computation task scheduling problem using convex programming. Recently, a quasi-static approach [3] has been proposed for a DVS-supported uniprocessor system, to maximize IC functions under a fixed energy budget. The models considered are not hybrid, thus less practical. All of the above works focus on single processor mapping.

3. MODEL DEFINITIONS

In this paper, we consider a task graph containing a set of adaptive tasks mapped to a homogeneous multiprocessor system. The task set is represented by a Direct Acyclic Graph $G = (V, E)$. We denote an adaptive task $i \in V$ as T_i. Each edge $e(i, j) \in E$ directs from T_i to T_j, indicating that T_j should start after T_i finishes. The *adaptiveness* of T_i is defined by its ability to extend the execution cycles from c_i to $c_i + \Delta c_i$ where c_i represents the program cycles before dynamic scheduling for T_i and Δc_i is the scheduling gain. The maximum allowed program cycle for T_i is defined as $C_i \geq c_i + \Delta c_i$.

The platform we consider is a homogeneous multiprocessor system comprising *N* processors, each with identical frequency ranges and power characteristics. With device feature scaling down to nano-levels, the processor power consumption is not only dominated by the dynamic power but also by the leakage power. The dynamic power consumption is directly related to processor clock frequency *f* and supply voltage V_{dd}, expressed as $P_{dyn} = C_{eff} V_{dd}^2 f$, where C_{eff} is the effective switching capacitance. The leakage power consumption is static and not directly related the processor behaviors. Martin *et al* [10] propose an adjustable reverse bias voltage V_{bs} that flexibly changes the CMOS device threshold voltage, in order to achieve exponential leakage current reduction. We adopt the formulations in [10] to model the leakage power consumption, defined as $P_{sta} = V_{dd} K_3 e^{K_4 V_{dd} + K_5 V_{bs}} + |V_{bs}| I_j$. The constants K_3, K_4, and K_5 are dependent on process technology and I_j is the approximately constant

junction leakage current. Hence, the total power consumption becomes,

$$P = C_{eff} V_{dd}^2 f + V_{dd} K_3 e^{K_4 V_{DD}} e^{K_5 V_{BS}} + |V_{bs}| I_j. \quad (1)$$

and the energy consumed per cycle is then given by,

$$E_{cyc} = C_{eff} V_{dd}^2 + L_g f^{-1} (V_{dd} K_3 e^{K_4 V_{dd}} e^{K_5 V_{bs}} + |V_{bs}| I_j), \quad (2)$$

where L_g is logic path length of the circuit. The frequency f is determined by

$$f = (L_d K_6)^{-1} ((1+K_1) V_{dd} + K_2 V_{bs} - V_{th1})^{\alpha} \quad (3)$$

where L_d, K_1, K_2, and V_{th1} are process dependent constants [10]. As observed from (2) and (3), under a fixed f, there is a range of E_{cyc} due to varying (V_{dd}, V_{bs}) pairs. By properly adjusting (V_{dd}, V_{bs}) values E_{cyc} can be minimized at f. We denote it as E_{cyc}^{f}. According to [10], as f increases, E_{cyc}^{f} monotonically increases. In our model, we assume that the processors are able to perform frequency scaling using J discrete levels. Each processor can vary its frequency in $\{f_0,...,f_{J-1}\}$. The corresponding minimal per-cycle energy consumption is $\{E_{cyc}^{f_0},...,E_{cyc}^{f_{J-1}}\}$.

Our dynamic scheduling algorithm is invoked at runtime when a task (slack generator) finishes earlier and leaves t_s for slack receivers to execute in advance. In our work, dynamic scheduling is defined as the process of deciding the starting time of slack receivers, execution speed (frequency), and the execution cycles (hence the execution time). Our work focuses on hard real-time applications, thus if t_s is generated, then at most t_s can be distributed to receivers to guarantee per-task deadline requirement. Besides, the slack energy E_s saved along with t_s is available to slack receivers. Formally, t_s is computed as the difference between the actual execution time and the pre-scheduled execution time: $t_s = t_{sheduled} - t_{actual}$. E_s is defined as $E_s = E_{cyc,f} \times f \times t_s$, where f is processor frequency.

4. SLACK DISTRIBUTION WITH FREQUENCY SCALING

The slack distribution process is invoked at runtime, when a slack generator generates slack time t_s and the associated E_s. We assume a given set of n slack receivers $T = \{T_0,...,T_{n-1}\}$ that utilize the slack to generate additional cycles. As defined in Section III, for a task $T_i \in T$, its execution cycles before slack distribution is c_i, executing under frequency $f_{i,old}$. After the slack distribution the total cycles are denoted as $c_i + \Delta c_i$, executing under frequency $f_{i,new}$.

Each T_i receives its share of the slack resources $\{t_{s,i}, E_{s,i}\}$ for extra cycle generation. Since t_s is duplicated in parallel, $t_{s,i} = t_s$ for direct successors. $T_{s,i} < t_s$ if T_i is blocked by the other predecessors. $t_{s,i}$ is unrelated to $t_{s,j}$ of any other $T_j \in T$ and serves as the timing constraint of T_i. $E_{s,i}$ is related to $E_{s,j}$ since they constitute E_s.

The maximum runtime performance gain, i.e., the maximum extra adaptive cycles, is represented as the sum of all Δc_i for T. As shown in the illustrative example, DVS can be properly used to derive more runtime cycles than being confined by E_s. The rest of this section shows that optimally adjusting DVS under the timing and energy constraints is NP-hard, as well as our guided-search heuristic that efficiently finds the best possible extra cycles

4.1 Problem Formulation

The scheduling gain maximization problem can be formulated below with (4) as the objective and Δc_i, β_{ij} as the decision variables to be optimized.

Maximize

$$\sum_{T \in T_i} \Delta c_i \quad (4)$$

Subject to

$$\sum_{j \in J} (\beta_{ij} (\frac{c_i + \Delta c_i}{f_i} - \frac{c_i}{f_{i,old}})) \le t_{s,i} + \Delta t_{oh}, \forall T_i \in T \quad (5)$$

$$\sum_{T_i \in T} \sum_{j \in J} (\beta_{ij} (c_i + \Delta c_i) E_{cyc}^{f_j}) \le E_s + \sum_{T_i \in T} (c_i E_{cyc}^{f_j}) + \Delta E_{oh} \quad (6)$$

$$\sum_{j \in J} \beta_{ij} = 1, \beta_{ij} \in \{1,0\}, \forall T_i \in T \quad (7)$$

$$c_i + \Delta c_i \le C_i, \forall T_i \in T \quad (8)$$

Timing and energy constraints are enforced by (5) and (6), respectively. Δt_{oh} and ΔE_{oh} are defined as overhead differences due to altered frequencies of T_i and its preceding task T_p on the same processor. We use the worst case values of Δt_{oh} and ΔE_{oh} and treat them as constants. Since the available frequency range is finite and discrete, we use a boolean variable β_{ij} as an indicator to reflect which frequency level j is used for T_i. Constraints (5) and (6) sum all j for each T_i, and leave the optimization solver to decide optimum j for T_i. Thus selection of frequency is transformed into β value determination. Since β_{ij} requires integer value, the above formulation is an integer linear programming program. Added that the constraints are nonlinear (product of β_{ij} and c_i), the above optimization is an integer nonlinear programming problem with NP-hardness.

4.2 Guided-Search Heuristics

For description clarity, let us release the constraints of discrete frequency levels and the maximum cycle, and re-formulate the above optimization problem:

Maximize

$$\sum_{T_i \in T} \Delta c_i \quad (9)$$

Subject to

$$\frac{c_i + \Delta c_i}{f_{i,new}} \le \frac{c_i}{f_{i,old}} + t_{s,i} + \Delta t_{oh}, \forall T_i \in T \quad (10)$$

$$\sum_{T_i \in T} ((c_i + \Delta c_i) E_{cyc}^{f_{j,new}}) \le E_s + \sum_{T_i \in T} (c_i E_{cyc}^{f_{j,old}}) + \Delta E_{oh} \quad (11)$$

where the variables of interest are c_i and $f_{i,new}$. Assume we select a specific $f_{i,new}$ for every T_i in the above formulation, thus the corresponding $E_{i,new}$ is fixed. Then, the question becomes a linear programming problem that derive the maximal $\Sigma \Delta c_i$ under the specific frequencies $f_{i,new}$. However, we still need to decide the best-fit $f_{i,new}$ for each i. A non-ideal $f_{i,new}$ fails to fully utilize E_s and t_s, and can have either (10) or (11) equalized but not both.

Situation 1: If (11) is equal, it implies no enough energy to supply slack time for T_i. Then scaling down $f_{i,new}$ can lead to reduced $E_{cyc}^{f_{i,new}}$, hence increased Δc_i according to equalized (11) with all other values constant. According to (10), increased Δc_i and decreased $f_{i,new}$ use more slack time as long as not exceeding the deadline.

Situation 2: If branches in (10) are equalized it implies no enough slack time to fully consume the slack energy. Then scaling up $f_{i,new}$ would cost more energy while leave extra slack time for Δc_i increase.

The above observation reveals the frequency scaling directions to increase Δc_i in both situations. Thus, $\Sigma \Delta c_i$ can be steadily increased in a guided search process. We set the starting Δc_i to 0 since our approach keeps increasing Δc_i. Initially, the highest frequency is used

for all T_i in (10), such that largest energy consumption is made in (11). The reason to start from the most energy consumption is due to the fact that t_s is duplicated to receivers but not E_s. Thus it is more possible to equalize (11), enforced by using highest frequency. Note that the zero Δc_i and highest $f_{i,new}$ causes all branches in (10) unequal.

As discussed above, it is more probable that the l.h.s of (11) is larger than the r.h.s after setting all $f_{i,new}$ highest. Then we slow down $\Delta f_{i,new}$s in (10) to reduce the l.h.s of (11). Note that in this process all c_is remain their initial value, 0, to avoid increasing the l.h.s. value. The T_i chosen is in the order of increasing residual cycles C_i-c_i, such that larger residual tasks reserve higher frequency slowing down chances for cycle increase.

This process stops when the l.h.s. becomes smaller than the r.h.s. of (11). Otherwise the process terminates since no frequency can be scaled down further. Then we increase Δc_i in the l.h.s. of (11) to make situation 1 happen. Tasks with small residuals are selected to increase Δc_i, because they have less chance to fully exploit the benefit of frequency scaling due to the limited residual cycles.

Under situation 1, the frequency of a T_i in (10) is scaled down by one level, while its Δc_i should be increased to maintain equality of (11). The criteria to choose T_i depend on three aspects: the more residual cycles C_i-c_i available, the larger chance to generate more cycles; as $f_{i,new}$ is reduced to $f_{i-1,new}$, needs to be small to avoid radical cycle increase given the residual cycle constraint; it should have the largest laxity to timing constraint in (10). Combining the factors, we choose the tasks having the largest $\frac{C_i - c_i}{E_{cyc}^{f_{i,new}} - E_{cyc}^{f_{i-1,new}}} * \Delta t_i$ as frequency scaling down target, where Δt_i is the time left to violate timing constraint in (10). The selection process repeats if (11) remain equalized.

It may happen that after several iterations, all branches in (10) are equalized due to increased Δc_i, while there is still unused power quota in (11). In this case, situation 2 occurs and we choose to increase $f_{i,new}$. Similar criteria apply to T_i selection as in situation 1, except that the timing constraint is invalid. The T_i with the largest $\frac{C_i - c_i}{E_{cyc}^{f_{i,new}} - E_{cyc}^{f_{i-1,new}}}$ is selected.

This frequency scaling process terminates under three conditions:

- $c_i + \Delta c_i = C_i, \forall T_i \in T$;
- when increasing $f_{i,new}$ is required, all $f_{i,new} = f_{J-1}$;
- when decreasing $f_{i,new}$ is required, all $f_{i,new} = f_0$;

Finally, if initially after setting all $f_{i,new}$ to highest and Δc_i to 0, l.h.s. of (11) is still smaller than the r.h.s., we directly increase the $\not\in c_i$ from the smallest residual T_i, until (10) or (11) happens for further frequency scaling.

The complexity of our approach is confined by the number of frequency levels J and the number of T_is, n. Consider the scenario that starting from the f_j, all n tasks are scaled down to f_0 to reach situation 2, then all n tasks scale up, to f_j to deal with situation 2. This is a non-repeatable scenario, because if there was a second round scaling down from the highest frequency, it would already have been done in the first round. Hence, the loose upper bound of the complexity is $O(nJ)$.

5. SLACK RECEIVER SELECTION

A straightforward receiver selection process can be greedy-based, i.e., choosing the direct descendent tasks of the slack generator. There can be two limitations in this approach. Firstly, the direct receivers may not fully utilize the slack time, as illustrated in Fig. 3(a). Secondly, in the task graph there can be additional parallel candidates beyond the direct descendent tasks for slack distribution, as shown in Fig. 3(b). Hence deliberated receiver selection is essential to assist achieving maximized runtime cycles, while the searching process requires minimal runtime overheads. We present a two-stage receiver selection method for each task node, so that when it generates slacks, offline picked receiver candidates are immediately available for efficient online selection. The simplest graph for slack distribution is a *tree*. E.g., in Fig. 4(a), when T_1 generates slack time, candidate receiver sets are $\{T_2, T_3\}$, $\{T_2, T_6\}$, $\{T_4, T_5, T_6\}$, and $\{T_4, T_5, T_3\}$.

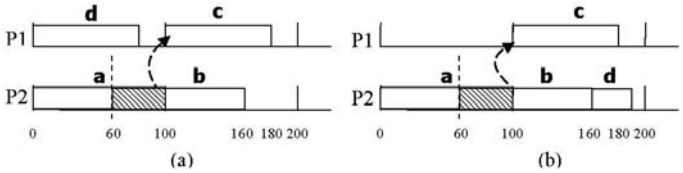

Fig. 3. (a) Task *d* hinders *c* from receiving the full slack. (b) *b* and *d* compete for the slack time.

Definition I: A *candidate set* (abbr. CS) of a slack generator is a set of slack receiver tasks that fully adopts the slack time.

Collection of all the CSs of a task can be performed offline. In the online stage we can directly choose the best CS based on proper priority. Note that in the previous section, every frequency scaling step aims at increasing Δc_i, thus the criterion of prioritization is the sum of residual cycles of all tasks in the set, and they are sorted offline for runtime usage. At runtime if T_1 generates t_s, the scheduler chooses the set with the most total residual cycles. At runtime, the sets at the sorted queue head may reduce their residual cycles due to slack allocation by previous slack generators. Re-sorting the queue reduces runtime efficiency, so we use the following heuristics:

```
Initial: i=0, Q=cand. set queue
for two cand. sets A=Q[i], B=Q[i+1]
if RESIDUAL(A) > RESIDUAL(B): return A
else if i==Q_size-2: return B
else: i++, repeat.
```

It is observed that every task in a tree has *at most* one predecessor, while a common task graph contains nodes with *more than* one predecessor, which can be either innate attribute of the task graph (concrete link from T_2 to T_6 in Fig. 4(c)) or enforced by being assigned on the same processor (dashed link from T_3 to T_6 in Fig. 4(c)). Such tasks can be classified into two types:

Definition II: A *Type-I* (abbr. *T-I*) task has multiple predecessors, each of which generates slacks independent of others. E.g. in Fig. 4(c), when T_4 and T_5 run concurrently, T_8 is a *T-I* task from the viewpoints of both T_4 and T_5.

Definition III: A *Type-II* (abbr. *T-II*) task has multiple predecessors, but receives single slack originating from the single slack generator. E.g. in Fig. 4(c), T_8 is a *T-II* task from the viewpoints of T_1. Because when T_1 runs, it is the only slack generator to T_8.

From the above definition, a multiple precedent task is *T-I* or *T-II* to different tasks, hence treated in respective when different predecessors generate slacks.

For *T-II* tasks, mutual exclusiveness exists on tasks residing on different processors. E.g., in Fig. 4(b), slack time given to T_{10} cannot be replicated to T_{11}. To resolve the conflict, we define sibling tasks below that are free to duplicate the slack time. Hence it is safe to add sibling tasks into the CSs of a slack generator. E.g., in Fig. 4(b) T_1's CS can be $\{T_2, T_3, T_4, T_5\}$ or $\{T_2, T_3, T_4, T_8, T_{10}\}$, while T_5's CS can be $\{T_8, T_9\}$, $\{T_8, T_{10}\}$, or $\{T_{11}\}$. We can thus combine the two types of

tasks in one framework. The runtime complexity for a slack generator to decide its tree and *T-II* receivers hence depends on when to find a larger residual CS in the above code snippet. The worst case is number of CSs.

Definition IV: T_m is a *sibling task* of T_n if T_m and T_n share common precedent nodes, while T_m is neither precedent nor descendant of T_n.

For *T-I* tasks, its availability can only be determined dynamically. Because when one predecessor generates slack, the task can take it fully, partially, or none, dependent on the execution condition of other predecessors. This implies *T-I* tasks cannot be placed into offline determined candidate receiver sets, but considered explicitly online. Fortunately, the number of *T-I* receivers is limited by the number of processors *N* since the slack time duplicates in parallel at most *N* times. Note that when the slack, full or partial, is given to a *T-I* task, it is actually distributed to its CS in which tasks can fully adopt the given slack. Moreover, when distributing slack to CS of *T-I*, processor overlapping with slack generator's CS may happen. In this case, slack generator's CS prioritizes since receivers in it receive full slack time. Finally, if *T-I*'s CS is empty, the slack is given to the *T-I* task itself.

There is maximally *N* *T-I* for a slack generator, and each CS contains at most *N* parallel receivers. Assuming *T-I* finds its CS immediately, the complexity of considering a slack generator's *T-I* candidates is $O(N_2)$.

The three categories of tasks, namely tree task, *T-I*, and *T-II* tasks, are classified based on their number of predecessors. It is a complete classification without missing types, thus the method can be applied to any task graphs. TABLE-II summarizes the above discussion.

TABLE II: TASK TYPES WITH APPLICABLE SCENARIO AND RUNTIME COMPLEXITY.

Types	app.offline	app.online	dyn.compl.
Tree	yes	yes	dep. CS no.
T-I	no	yes	$O(N^2)$
T-II	yes	yes	dep. CS no.

Implication to static scheduling

The actual slack time adopted by a *T-I* task is dependent on its latest finished direct predecessor. To maximally eliminate predecessor impact, the favored static schedule should: (1) choose a processor with the earliest available time, e.g. in Fig. 4(c), T_6 prefers *P*0 to *P*2. (2) amongst the processors with identical earliest available time, allocate the task to a processor that leads to least precedence constraints, e.g. in Fig. 4(c), T_8 prefers *P*4 to *P*3, assuming T_4 and T_5 complete at the same time. We could say that compared to Fig. 4(c), Fig. 4(b) provides a favorable static schedule for dynamic scheduling.

6. EXPERIMENTAL RESULTS

The performance of our dynamic algorithm shall be reflected in two aspects: scheduling gain measured by the generated cycles from runtime slack, and runtime efficiency measured by the actual algorithm execution time. For comparison: (1) We design an altered version of our methodology that adopts the same slack receiver selection process but evenly distributes energy to receivers, and individually apply DVS to receivers for cycle generation. (2) To evaluate the effects of receiver selection, we adopt the greedy-based dynamic algorithm from [7] which efficiently decides frequencies of immediate successors for energy minimization, and we apply our algorithm initially starting from those frequencies. (3) The performance is also measured under different static schedule inputs. We implement two list scheduling algorithms with the difference in processor selection criteria, one with dynamic scheduling awareness as described in the previous section, and the other with randomly chosen processor.

The simulation platform is based on the cycle-accurate SESC [13] ISA simulator. We develop a built-in scheduler which executes the abovementioned algorithms to regulate the runtime task execution, simulate energy consumption, and capture the runtime cycle gain. We adopt the energy parameters from the Intel XScale PXA270 CPU which is able to adjust execution frequency. The applications used in our experiment consist of a JPEG2000 decoder with adaptive feature, as well as synthesized task graphs for more extensive performance tests.

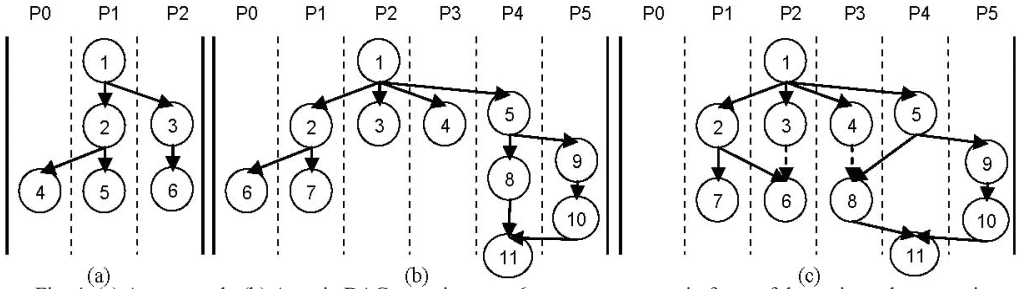

Fig. 4. (a) A tree graph. (b) A static DAG mapping on a 6-processor system in favor of dynamic cycle generation. (c) A static mapping creating *T-I* nodes not preferred for dynamic scheduling

6.1 Synthesized Task Simulation

We generate synthesized task graphs containing 100-200 tasks. The tasks are synthesized by the task graph generator TGFF [8], in which tasks have mean execution time as 15*ms*. We test the task graphs on our platform with 8, 32, and 64 processors respectively. The results are shown in Fig. 5 in which average gained execution cycles using the above three algorithms are normalized. We implement the three algorithms under different static schedules in which the dynamic scheduling aware static schedule gains as large as 57% on the 32-processor platform, compared to the list scheduling algorithm based on earliest available processor time. Under either static schedule, our algorithm can achieve at least 25% cycle increase compared to the even-distribution energy approach, and at leat 33% compared to the greedy approach. Note that interestingly, the greedy approach has better performance with less number of processors while the even-energy approach leads to better performance when processor number is large. These can reflect the feature of the greedy approach which does not scale well to the number of processors, and our algorithm can lead to better performance by considering both DVS scaling and slack receiver selection. Fig. 5(d) shows the execution cycle of our algorithm. Compared to the extremely fast greedy approach, our method runs with larger number of cycles, about tens of times. However, compared to the typical task size, the runtime overhead is still extremely small, near 0.3%. Results also show that our algorithm scales with the number of processors, which is predicted in the complexity analysis in the previous section.

6.2 The JPEG2000 Decoder

We use the JPEG2000 decoder example to show the applicability of our methodology. The JPEG2000 decoder is known as the adaptive application that allows reconstruction of images in a progressive manner. This is possible by the use of Discrete Wavelet Transform (DWT), which encodes the images into multiple subbands so that the lower frequency subband contains finer frequency resolution and coarser time resolution. At the decoder, as more data are received, higher resolution images can be decoded making use of subsequent higher frequency information.

In our experiment, we decode the "Ceveness" sample j2k file with the size of 14.4MB. The application is divided into three branches each of which decodes a colour component. Each branch contains a DWT block which is able to decode in multiple resolution levels. At the encoder side we have encoded six levels of resolution. In the decoder, we statically set DWT to decode only level 1, leaving all other five levels for online decision. We have profiled the execution cycles of DWT to perform the six levels of transformation respectively, as shown in TABLE-III. Note that, since the additional cycles are discrete, we round the derived optimal cycles to the next largest value as our result. Since there are three application branches we assume a platform with three processors.

TABLE III: DWT CYCLES TO TRANSFORM DIFFERENT LEVELS OF RESOLUTION.

no. resol.	1	2	3	4	5	6
million cycles	2.67E-3	0.38	1.86	9.74	49.17	134.45

TABLE IV: PERFORMANCE FROM SCHEDULING A JPEG2000 DECODER.

	Avg. cyc. inc.	Impr (%)	Exe. cyc.
Dyn. DVS	25.8E+3	141.1	1029
Egr.div	10.7E+3	0	897
Greedy	19.6E+3	83.1	200

The results are shown in TABLE-IV, in which the performance of our algorithm is around 2.5 times over the even-energy approach, and 31.6% better than the greedy approach. The greedy approach outperforms the even-energy approach since only 3 processors are used in this set of experiments. This eliminates the disadvantage that greedy-approach is weak in receiver candidate selection. Note that our algorithm runs fast in this example. The main reason is the small number of processors.

7. CONCLUSION

In this paper, we proposed a novel framework for leakage-aware multiprocessor dynamic scheduling on adaptive applications. Compared with contemporary approaches, it achieves dramatic performance gain without significant runtime overhead.

8. REFERENCES

[1] J. Y. Chung, J. W. S. Liu, and K. J. Lin, "Scheduling Periodic Jobs that Allow Imprecise Results," *IEEE Trans. on Computers*, vol. 39(9), pp.1156-1174, Sep. 1990.

[2] C. Rusu, R. Melhem, and D. Mosse, "Maximizing rewards for realtime applications with energy constraints," *ACM Trans. on Embedded Computing Systems*, vol. 2(4), pp. 537-559, Nov. 2003.

[3] L. A. Cortes, P. Eles, and Z. Peng, "Quasi-Static Assignment of Voltages and Optional Cycles in Imprecise-Computation Systems with Energy Considerations," *IEEE TVLSI*, 14(10), 2006.

[4] H. Yu, B. Veeravalli, and Y. Ha, "Dynamic scheduling of imprecise-computation tasks in maximizing QoS under energy constraints for embedded systems," *ASP-DAC'08*, pp. 452-455, 2008.

[5] R. Ernst and W. Ye, "Embedded Program Timing Analysis based on Path Clustering and Architecture Classification," *IEEE Int'l Conf. on Computer-Aided Design (ICCAD)*, pp 598-604, 1997.

[6] F. Gruian, "System-Level Design Methods for Low-energy Architectures Containing Variable Voltage Processors," *Proc. 1st Int'l Workshop on PACS*, pp. 1-12, Nov. 2000.

[7] D. Zhu, R. Melhem, and B. Childers, "Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems," *IEEE TPDS*, vol. 14(7), 2003.

[8] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," *CODES'98*, pp. 97-101, 1998.

[9] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Optimal reward-based scheduling for periodic real-time tasks," *IEEE Trans. on Computers*, vol. 50(2), pp. 111-130, Feb. 2001.

[10] S. M. Martin, *et al*, "Combined dynamic voltage scaling and adaptive body biasing for low power micropossers under dynamic work loads," *Proc. ICCAD*, pp. 721-725, 2002.

[11] T. Acharya and P. S. Tsai, JPEG2000 Standard for Image Compression: Concepts, Wiley 2004.

[12] H. Schwarz, et al,, "Overview of the Scalable Video Coding Extension of the H.264/AVC Standard," *IEEE Trans. on Circuits Syst. Video Techn.*, vol. 17(9), pp. 1103-1120, Sep. 2007.

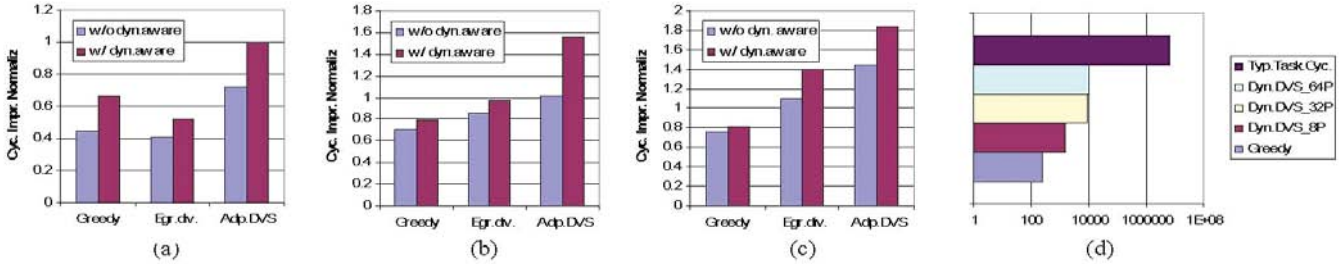

Fig. 5. (a)(b)(c) Normalized cycle gain on 8, 32, 64 processors using three methods. (d) Scheduler cycles compared with a typical synthesized task.