Scheduling imprecise task graphs for real-time applications

R.C. Ravindran*, C. Mani Krishna, Israel Koren and Zahava Koren

Department of Electrical and Computer Engineering,
University of Massachusetts,
Amherst, MA 01003, USA
E-mail: rajeswarancr@gmail.com
E-mail: krishna@ecs.umass.edu
E-mail: koren@ecs.umass.edu
E-mail: zkoren@ecs.umass.edu
*Corresponding author

Abstract: Many of the real-time tasks within embedded real-time control applications fall into the imprecise category. Such tasks are iterative in nature, with output precision improving as execution time increases (up to a point). These tasks can be terminated early at the cost of poorer quality output. Many imprecise tasks in CPS are dependent, with one task feeding other tasks in a task precedence graph (TPG). A task output quality depends on the quality of its input data as well as on the execution time that is allotted to it. In this paper, we study the allocation/scheduling of imprecise TPGs on multiprocessors to maximise output quality where resources (time and energy) are limited. Our heuristic algorithms can effectively reclaim resources when tasks finish earlier than their estimated worst-case execution time. Dynamic voltage scaling is used to manage energy consumption and keep it under a specified bound.

Keywords: imprecise tasks; dynamic voltage scaling; real-time systems.

Reference to this paper should be made as follows: Ravindran, R.C., Krishna, C.M., Koren I. and Koren, Z. (2014) 'Scheduling imprecise task graphs for real-time applications', *Int. J. Embedded Systems*, Vol. 6, No. 1, pp.73–85.

Biographical notes: R.C. Ravindran received his MS degree from the ECE Department at the University of Massachusetts (UMass) in 2012. His research interests are in real time systems and wireless networks. He is currently working in Qualcomm Research Center, San Diego.

C. Mani Krishna is a Professor of Electrical and Computer Engineering at the University of Massachusetts. He received his PhD in Electrical Engineering at the University of Michigan and has been at UMass ever since. His principal interests are in real-time, fault-tolerant and distributed systems, as well as sensor networks. He has co-authored two texts on real-time systems and fault-tolerant computing. His recent research includes projects in secure sensor networks, resource provisioning in modern deeply-pipelined processors, power-aware issues in real-time systems, parameter variation problems in nanotechnology, and the efficient design of cyber-physical systems. He is a Fellow of the IEEE.

Israel Koren is a Professor of Electrical and Computer Engineering at the University of Massachusetts and a Fellow of the IEEE. He is an Associate Editor of the *VLSI Design Journal*, and *IEEE Computer Architecture Letters*. He served as the General/Programme Chair/committee member for numerous conferences. He is the author of *Computer Arithmetic Algorithms*, second edition, A.K. Peters, 2002, and a co-author of *Fault Tolerant Systems*, Morgan-Kaufman, 2007. His research interests include fault-tolerant systems, VLSI yield and reliability, secure cryptographic systems and computer arithmetic. He publishes extensively and has over 200 publications.

Zahava Koren is currently a Senior Research Fellow at the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst. She received her MA in Mathematics and Statistics from the Hebrew University, Jerusalem, and DSc in Operations Research from the Technion – Israel Institute of Technology. Previously, she has held positions with the Department of Industrial Engineering at the University of Massachusetts, the Department of Statistics, University of Haifa, Departments of Industrial Engineering and Computer Science at the Technion, and the Department of Business and Economics, California State University in Los Angeles. Her main interests are stochastic analysis of computer networks, yield of integrated circuits and reliability of computer systems.

1 Introduction

Imprecise tasks form an important category of workloads in cyber-physical and other real-time applications. An imprecise task [also known as an *increased reward with increased service* (IRIS) task] is one which can be terminated prematurely and still produce usable (albeit of poorer quality) output. Such tasks usually consist of a *mandatory* part that must be completed in order to produce any usable output, and an *optional* part which yields increasingly accurate results (up to a point) the longer it executes. The *value* of the output accuracy to the application is obviously application specific, and is captured by means of a cost or reward function. There is a category of imprecise tasks called *anytime* tasks, which are characterised by extremely small mandatory portions.

We address the following problem. We are given a set of imprecise tasks whose precedence conditions are specified by a directed acyclic task graph. Tasks consume results from their parents in the task graph (the root of the task graph has the system as its parent). The output to the application is from the leaves of the graph. Our aim is to map tasks to processors and then schedule these tasks so that the quality of output to the application is maximised, subject to the execution completing by a specified deadline while staying under some given energy limit. As proxy to the quality of the output, we use the weighted sum of the output errors of the graph's leaves, which we attempt to minimise. To this end, we present both offline and online heuristic algorithms; the offline heuristic carries out allocation of tasks to processors and produces an offline schedule, under the assumption that tasks run to their estimated worst-case execution times (WCET). The online heuristic reclaims execution times released by tasks which consume less than their WCETs.

This paper makes the following contributions. It considers imprecise task workloads consisting of arbitrary task precedence graphs (TPGs). It accounts for the fact that tasks seldom take their estimated WCET, but in fact, often complete much earlier, by providing means to reclaim resources released early by completing tasks. It accounts for the decrease in output quality that results when a task receives imprecise input from one of its parents in the task flow graph. Finally, it allows for energy management using dynamic voltage scaling.

2 Examples of IRIS tasks

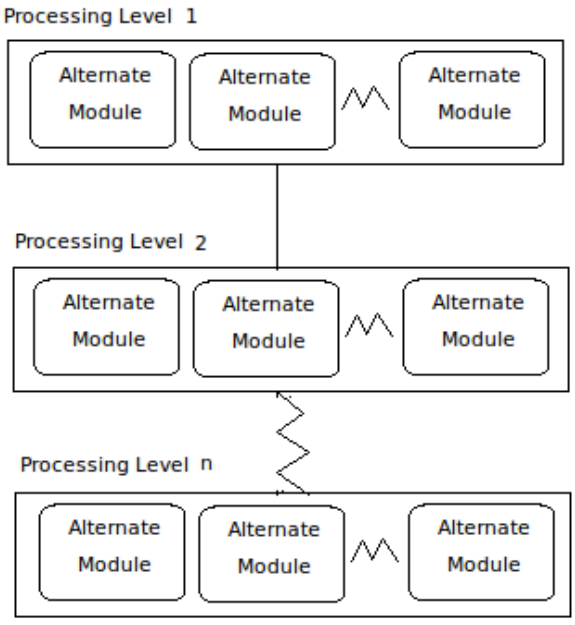
Imprecise tasks can be found for a wide variety of applications. Consider the open source video codec tool, xvid ('Xvid tool', http://www.xvid.org/). This tool has a two-pass option for video encoding. The first pass analyses the video clip; the second pass uses the results of that analysis to obtain a high-quality encoding. Algorithm settings allow one to control the time spent in first-pass analysis; one can trade off the precision of the motion search against the time taken. The second pass takes the first pass results to efficiently encode the video clip. Controlling the allowed bitrate allows us here to trade off the quality against the computational work of this step. The quality of the first pass affects the range of possibilities for the second pass; the quality of both passes depends on the length of time devoted to them.

A second example is path planning in robotics (Zilberstein and Russell, 1993). Path planning includes sensing and planning modules, both of which have the imprecise property. The sensing module builds up an awareness of the environment; this is then used by the planning module to complete path planning.

A third example is developing control inputs for cyber-physical systems. Suppose a linear control system has multiple control variables. One approach is to calculate these variables one at a time in order of their perceived impact on the quality of control provided; when control input k is calculated, the values of control inputs 1, ⋯, $k-1$ are already available. Depending on the amount of time available, we may only calculate the first N control inputs, leaving the others at 0. Gupta (2009) has shown this to be a viable strategy in an environment where the amount of time available for computation is variable.

Our final example is the task structure for the control of a planetary rover (Zilberstein et al., 2002) (see Figure 1). The task is composed of a sequence of processing levels l_i and each level contains alternative modules $m_i^1, m_i^2, \cdots$. Each alternative module has a different resource requirement in return for which it provides a certain quality output. By selecting the modules appropriately, we can trade off the quality of control provided against the resources (e.g., time) consumed.

Figure 1 Task structure in planetary rover



Source: Zilberstein et al. (2002)

3 Related work

While there is a vast literature on scheduling traditional real-time tasks, much less work has been reported on scheduling imprecise real-time workloads. We can classify prior work on imprecise task scheduling according to the following criteria: whether the:

a tasks are independent

b computational platform is a uniprocessor or multiprocessor

c execution time is fixed or variable.

Table 1 provides a summary of some representative papers from the literature. Most work in this area deals with independent tasks; only a handful of papers assume any inter-task precedence relationship. Similarly, most work assumes a fixed execution time, known in advance, for both the mandatory and optional portions; only in a few cases is the possibility of variable execution time considered. Only in rare cases (e.g., Cortes et al., 2006) are the possibility considered of dynamic voltage scaling to reduce energy consumption (at the price of slowing down the computation of the imprecise workload).

Table 1 Classification of some imprecise task scheduling algorithms

Reference	*Workload*	*Platform*	*Exec. time*
This paper	Dep	Multi	Variable
Chishiro and Yamasaki (2011)	Indep	Multi	Variable
Chishiro et al. (2010)	Indep	Multi	Variable
Tchamgoue et al. (2010)	Indep	Uni	Fixed
Li et al. (2009)	Indep	Multi	N/A
Gupta (2009)	One task	Uni	Fixed
Cortes et al. (2006)	Dep	Uni	Variable
Cheng and Wang (2004)	Dep	Uni	Fixed
de Oliveira et al. (2001)	Dep	Multi	Variable
Shin et al. (2000)	Indep	Uni	Fixed
Feng and Liu (1997)	Dep	Uni	Fixed
Dey et al. (1996)	Indep	Uni	Fixed
Khemka et al. (1993)	Indep	Muti	Fixed
Shih and Liu (1992)	Indep	Uni	Fixed
Liu et al. (1991)	Indep	Uni	Fixed
Chung et al. (1990)	Indep	Multi	Fixed
Shin et al. (1989)	Dep	Uni	Fixed

Notes: Indep = independent tasks; Dep = dependent tasks (task graph); Uni = uniprocessor; Multi = multiprocessor.

3.1 Independent tasks

Chung et al. (1990) consider periodic task sets running on multiprocessors; the task set is known ahead of time and a schedule can be setup offline. A first-fit approach is taken to allocating tasks to processors; following this, uniprocessor scheduling is carried out on each processor. The rate monotonic algorithm (Liu and Layland, 1973) is used to assign static priorities to the mandatory portions of each task based. The optional portions of all tasks have lower priority than the mandatory portion of any task. Various simple heuristics have been studied for scheduling the optional portions, including static priorities inversely related to the task utilisation and dynamic priorities favouring the optional portion with the least execution time provided or the one with the least slack time. It is assumed that the error associated with premature termination of an optional portion is proportional to some positive power of the fraction of uncompleted work.

An online approach is discussed in Shih and Liu (1992). The workload consists of a set of tasks known ahead of time together with tasks that arrive during system operation. The error model is linear, the output error being equal to the amount of unfinished work. As tasks arrive, time is reserved for their mandatory portions using the latest-ready-time-first order. Optional tasks can execute as long as there is enough time.

Dey et al. (1993, 1996) presented three heuristic scheduling algorithms for online scheduling of aperiodic workloads. Their reward function is a concave non-decreasing function of the execution time. Two of the algorithms take a two-level approach. The top level is executed whenever a new task arrives and is responsible for deciding the allocation of service time to that task such that the reward is maximised. The lower-level algorithms decide the order in which tasks execute. The third algorithm takes a greedy approach. The two metrics used for evaluating performance are the reward rate and average number of task preemptions using each scheduling policy. They have developed an analytical model for an imprecise task system and obtained the upper-bounds on the reward rate that is achievable by any scheduling policy adopted. This work concludes that with the appropriate lower-level scheduling policy, the performance of their algorithm approaches quite close to its upper bound. The average number of preemptions is very small when the earliest deadline first (EDF) scheduling algorithm is used at the lower level.

A hierarchical approach to scheduling is taken by Tchamgoue et al. (2010). The overall workload is divided into components; each component is guaranteed to obtain a certain minimum amount of resources over every specified period. Each component can then be scheduled with this guarantee in mind. A hierarchical approach allows the scheduling of one component to be decoupled from the scheduling of another.

Chishiro and Yamasaki (2011) and Chishiro et al. (2010) present a global semi-fixed-priority scheduling approach. Independent tasks arrive at a multiprocessor, and consist of three parts: mandatory, optional, and wind-up. The wind-up part is responsible for organising the dispatch of the output and terminating the task execution. Wind-up obviously exists implicitly in all other imprecise task models as well; however, here it is assumed to be non-negligible. Mandatory and wind-up segments are in a

real-time queue while optional segments are in a non-real-time queue and are only executed if the real-time queue is empty. The discipline is called semi-fixed priority because the task priority drops when the task moves from mandatory to the optional part and increases again for the windup part.

Li et al. (2009) discuss the problem of deciding when to terminate the execution of an anytime task. They focus on an air-defence case-study and suggest the use of three factors in making this decision: the chances of improvement in the solution quality, the cost of delaying action (i.e., in launching a missile), and the impact of the operating environment (e.g., the number and variety of other targets).

3.2 Dependent tasks

The above-mentioned works all deal with independent tasks. By contrast, Feng and Liu (1993, 1997) consider *composite tasks*, each of which consists of linearly dependent tasks. That is, each task (except for the first and last) in a composite task has exactly one parent and one child; a task receives input from its parent, carries out some processing, and then forwards the output to its child. The first task receives inputs from the application; the final task produces output to the application. The quality of output of a task depends both on the quality of its input as well as on the amount of time it executes for. An interesting assumption is that inaccuracies in the input can cause the mandatory and optional portions to require more time to execute.

Feng and Liu (1997) introduce a two-level scheduler. The first level schedules the composite tasks using a modified EDF approach which treats the entire composite task as optional and cuts off tasks at the deadline, even if they have not been given their full execution time. If it manages to find full execution time for each composite task, we are done. If not, it augments the execution time allocation to composite tasks with relatively small optional parts. In the second level, the time allocated for each composite task at the first level is distributed to its subtasks such that the output error of the composite task is minimised. They have developed, and compared the performance of, five second-level heuristic scheduling algorithms.

Cortes et al. (2006) consider a task graph being scheduled on a single processor in a non-preemptive fashion. Their reward model assumes that the quality of the output depends as the sum of the rewards of the individual tasks in the task graph; these rewards, in turn, depend only on the amount of optional time given to the task in question.

Gupta (2009) considers modifying the workload for a cyber-physical system responsible for computing control inputs for some control plant. His approach, which can be employed on either a uniprocessor or a multiprocessor platform, is to calculate control inputs in the presumed order of their importance to the controlled plant, stopping when time runs out. For example, if we have a plant for which three inputs u_1, u_2, u_3 have to be computed in that order of importance, we have the option of just calculating u_1 and setting $u_2 = u_3 = 0$, or of calculating u_1 and u_2 and setting $u_3 = 0$ or of calculating all three inputs. The quality of control and the computational workload will obviously improve in this order.

de Oliveira et al. (2001) consider a workload consisting of arbitrary acyclic task graphs, with the same task being entitled to belong to multiple task graphs. If a task has been executed imprecisely (i.e., its optional portion has not been calculated to completion), then the value to the system of a precise calculation of the next iteration of that task is increased. Input data of better quality is held to potentially reduce the execution time of a task; the deterioration of the quality of a child task output due to imprecise input from a parent task output is not explicitly accounted for. They have a four-algorithm suite in their approach:

a to allocate tasks to individual processors

b to verify that the allocation in (a) allows for feasible execution of at least the mandatory portion of each task

c to perform admission control of optional portions based on their perceived current value to the application

d determining whether a given optional part will, if executed, risk causing a mandatory portion to miss its deadline.

The work reported in this paper differs from prior work in scheduling imprecise workloads in the following respects:

- The variability inherent in task execution times is accounted for when deciding when to prematurely terminate optional portions.
- Arbitrary task graphs are allowed, with inaccurate output from a parent task contributing to the error of a child task.
- The algorithms allow for dynamic voltage scaling of both the mandatory and optional portions.

4 Model and problem statement

4.1 Task model

We are given a TPG indicating the dependence between tasks.[1] A task is assumed to require inputs from all its parents before it starts executing; it delivers output only at the end of its execution. Since the output of an imprecise task can be inaccurate (due to premature termination), and an imprecise task can provide input to another task, we have to account for input errors. Let $\vec{\sigma}_i$ denote the vector of inputs and input errors applied to task T_i, and ϕ_i the fraction of its optional portion that has been executed. Its output error is given by $E_i(\vec{\sigma}_i, \phi_i)$. As a practical matter, unless we instrument the code to monitor and output the progress of the execution, ϕ_i is never known exactly except when the optional portion finishes, i.e., when $\phi_i = 1$. At all other times, we must use our best estimate of this value based on profiling and on the number of cycles consumed so far in its execution.

4.2 Processor model

The task workload runs on a set of processors which use dynamic voltage scaling (Pillai and Shin, 2001) to trade off clock frequency (and hence rate of execution progress) and energy consumed. Due to the highly non-linear dependence of energy on processor speed, voltage scaling has emerged as a principal way by which real-time systems can reduce their energy consumption while still ensuring that all task deadlines are met.

In this paper, we assume that there are two discrete voltage levels, V_{high} and V_{low}. It is quite easy to extend this algorithm to account for a larger number of voltage levels; however, with maximum supply voltages dropping every semiconductor generation, the range of voltages over which the supply can be switched keeps shrinking, and it is increasingly unlikely that more than two voltage levels will be useful in the future. We assume that voltage switching costs are negligible; this is reasonable, given that each task in our algorithms undergoes at most one voltage switch. The overhead of voltage switching is typically a few tens of microseconds (Park et al., 2010), which is very small in comparison to the execution time of complex control algorithms and the task periods in cyberphysical systems. For this reason, it is common to ignore such overheads in real-time voltage scaling. The processor consumes e_{high} and e_{low} energies per clock cycle at V_{high} and V_{low}, respectively, and the corresponding frequencies are f_{high} and f_{low}. The energy spent in communication is folded into the cost of execution and is not accounted for separately. Also, while the number of cycles required to execute a task is assumed independent of voltage, the time taken is obviously scaled according to the clock frequency.

Table 2 Key notations

Notation	*Explanation*
d_i	Deadline of leaf task i
F_i	Finish time of leaf task i
B_e	Energy bound for the TPG
c_i^{high}	Number of high voltage cycles spent executing task i
c_i^{low}	Number of low voltage cycles spent executing task i
c_i^{wm}	Mandatory worst case cycles of task i
c_i^{wo}	Optional worst case cycles of task i
n_i^m	Number of cycles used by mandatory part of task i
n_i^o	Number of cycles used by optional part of task i
e_{high}	Energy consumed by one high voltage cycle
e_{low}	Energy consumed by one low voltage cycle
χ	A mapping of tasks to processors $(1..n) \rightarrow (1..m)$
$\vec{\sigma}_i$	Input vector to task i concatenated with input error vector
$E_i(\cdot)$	Output error function of task i
Γ	Final error of task graph
$F(\cdot)$	Recursive application of $E_i(\cdot)$

As mentioned earlier, this algorithm runs on multiple-processor systems. The algorithm places no restrictions on the structure of the underlying hardware system; it is not material whether it is a shared-memory or message-passing system, for example. This underlying structure will obviously have performance implications, which will be taken into account by the algorithm when making its decisions.

The key notation used is summarised in Table 2.

4.3 Optimisation objective and constraints

Denote by $T_1, \ldots, T_n$ the set of all tasks, by $\mathcal{L}$ the set of leaves of the TPG, by d_i and F_i the deadline and finishing time of task T_i, and by c_i^{low}, c_i^{high} the number of low-voltage and high-voltage clock cycles spent executing task T_i ($i = 1, \ldots, n$. Let B_e be the upper bound of the energy consumption (set to ∞ if no such bound exists).

The only output that is visible to the application is that from the leaves of the TPG; therefore, our aim is to minimise the weighted sum of the leaf errors, where κ_j is the weight given to the error in the output of leaf task T_j and reflects the scale of values of the application. The optimisation problem, stated formally, is to minimise

$$\left\{\Gamma = \sum_{i \in \mathcal{L}} \kappa_i E_i\left(\vec{\sigma}_i, \phi\right)\right\} \tag{1}$$

subject to the following constraints

$$F_j \le d_j \,(j \in \mathcal{L}) \tag{2}$$

$$\sum_{i=1}^{n}\left(c_i^{high} \cdot e_{high} + c_i^{low} \cdot e_{low}\right) \le B_e \tag{3}$$

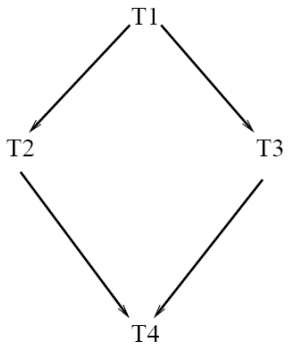
There are two sources for a task input: the external world and the other tasks. We have no control over the former; we focus instead on the latter. We assume that any expected errors from the external world input are factored into the error functions. Applying the error function E_i recursively, we can write the overall error as a function of the number of clock cycles consumed by each task. That is, if $c_i = c_i^{low} + c_i^{high}$ is the number of clock cycles consumed by task T_i, we can write

$$\Gamma = F\left(c_1, c_2, \cdots, c_n\right) \tag{4}$$

where $F(\cdot)$ can be obtained by recursive application of the $E_i(\cdot)$ functions.

As a simple example, consider the task graph shown in Figure 2. Tasks T_2 and T_3 receive inputs from T_1, and T_4 receives inputs from both T_2 and T_3. We wish to derive $F(\cdot)$ from the error functions, $_{Ei}(\cdot, \cdot)$, $i = 1, \cdots, 4$. Based on our profiling of these tasks, suppose our best estimate of the mandatory and optional numbers of cycles used by these tasks are given by μ_i, ω_i, $i = 1, \cdots, 4$.

Figure 2 Task graph example



Therefore, if c_i is the number of cycles allocated to task T_i, our best estimate of the fraction of the optional portions completed is $\phi_i = \max\left\{0, \frac{c_i - \mu_i}{\omega_i}\right\}$.

Hence, we can write

$$\vec{\sigma}_2 = \vec{\sigma}_3 = \left(E_1(0, \phi_1)\right)$$

$$\vec{\sigma}_4 = \left(E_2(\vec{\sigma}_2, \phi_2), E_3(\vec{\sigma}_3, \phi_3)\right)$$

The output error, which is the error in the output of T_4, is $E_4(\vec{\sigma}_4, \phi_4)$. Based on the above expressions, we can obviously express E_4 in terms of c_i, $i = 1, \cdots, 4$.

In this paper, we explicitly account for the fact that the actual total number of execution cycles required to finish a task is not known precisely (except when the task finishes). At best, we only know its probability distribution based on workload profiling. We therefore have to use an *estimate* of ϕ_i as a function of c_i, based on the information available. We do know the worst-case number of cycles, c_i^{wm}, c_i^{wo}, required for the mandatory and optional portions, respectively, of task T_i.

5 Offline allocation and scheduling heuristic

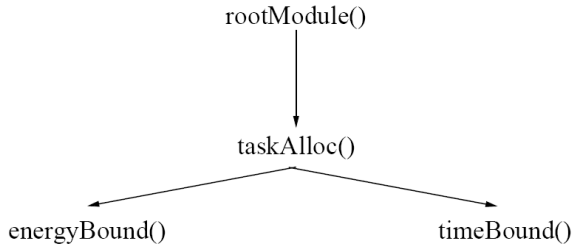
Minimising Γ in equation (1) is an NP-complete problem, and in addition, we do not have the exact value of ϕ_i. We therefore must use a minimisation heuristic.

Our heuristic exploits the fact that in cyber-physical systems (our target application area), the computational tasks are known in advance, and can be profiled extensively before the system starts operation. Such advance information can be exploited by having separate offline and online phases in the scheduling process. In the offline phase, which is executed once before the system is put into operation, tasks are assigned to processors and a schedule is generated making assumptions about the tasks' running times. The offline phase allocates the mandatory part's worst case requirement to all the tasks, thereby ensuring meaningful output of each task in the system.

In the online phase, as tasks finish, we update our knowledge of their actual running time and reclaim whatever resources are released by early task completion. For obvious reasons, the online heuristic must be lightweight, while the same constraint does not apply to the offline part.

The offline heuristic call structure is shown in Figure 3. In our algorithm, we start with a candidate allocation of tasks to processors. This allocation is assessed for its ability to meet time and energy constraints as will be described later. Simulated annealing is used to navigate through various allocations in search of one which offers good performance.

Figure 3 Offline heuristic call structure



5.1 Root module

The basic elements of the root module (Figure 4) are:

1 A finite set, S, of all possible configurations, where each configuration is a mapping of the entire task set to the processor set.

2 A step function STEP() which returns a configuration after moving a random task from one processor to another or swaps two random tasks on two different processors based on P_{swap} (the probability with which two tasks assigned to different processors are swapped). The greater the P_{swap}, the more chances of tasks getting exchanged between different processors.

3 A cooling schedule with an initial temperature $Temp_{initial}$ and a final temperature $Temp_{final}$, a depreciation factor df, and number of tries N_{tries} of the greedy algorithm at each temperature value.

4 An acceptance criterion. If δ is the difference between the new final error and the best final error, k is the Boltzmann constant and T is the current temperature, the new configuration is accepted with probability p.

$$p = \begin{cases} 1 & \text{if } \delta < 0 \\ \exp\left(-\delta / (kT)\right) & \text{if } \delta \geq 0 \end{cases} \quad (5)$$

5 An arbitrarily generated initial configuration $\chi_{initial}$ with a random mapping of $\{1..n\} \rightarrow \{1..m\}$.

The worst-case mandatory and optional execution cycles of task T_i are denoted by c_i^{wm} and c_i^{wo}, respectively. We assign cycles to tasks in steps where necessary: the step size at high and low voltage levels is denoted by v_{high}, v_{low}, respectively. These are chosen so as to take the same time, i.e., such that $v_{high} \cdot f_{high} = v_{low} \cdot f_{low}$.

Figure 4 Root module

```
function rootModule(T_init, T_final, df, N_tries, χ_init)
    temp = T_init;
    χ = χ_init;
    Π_final = invalid;
    χ_final = invalid;
    while (temp > T_final) do
        for (i = 1 … N_tries step 1) do
            χ_new = STEP(χ);
            Π = taskAlloc(χ_new);
            if (δ < 0 OR RND(0, 1)> e^{-δ/(k·temp)}) then
                χ = χ_new;
                if (F(Π) < Γ^{final}_{offline}) then
                    Γ^{final}_{offline} = F(Π);
                    Π_final = Π;
                    χ_final = χ;
                end if
            end if
        end for
        temp = temp/df;
    end while
    return Π_final;
end function
```

5.2 Task allocator module

The task allocator (Figure 5) returns a schedule based on which one can estimate the offline final error, $\Gamma_{offline}$ for the specified task assignment. The schedule is marked invalid if it is unable to find one which satisfies the time and energy constraints. It first generates a time allocation taking only the deadlines into account and disregarding the energy constraint, if any. If an energy constraint is specified, it then modifies this schedule by swapping high-voltage and low-voltage cycles if this is needed to meet the bound. If no such swap can be found, it declares failure and returns an invalid result.

Figure 5 Task allocator module

```
function taskAlloc(Configuration χ)
    Π = timeBound(χ);
    if (Π! = invalid) then
        if (∑_{i=1}^{n} c_i^{high} e_{high} ≤ B_e) then
            return Π
        else
            return energyBound(Π);
        end if
    end if
end function
```

Figure 6 Time-bound module

```
function timeBound(Configuration χ)
    Π = invalid;
    c_i = c_i^{wm}, i = 1, ⋯, n
    if (a deadline is violated) then
        return Π = invalid
    else
        t^i_af = 1, i = 1, ⋯, n
    end if
    while (∃i s.t. t^i_af == 1) do
        for (each such i) do
            for (j = 1..n step 1) do
                c'_j = c_j + δ_{i,j} v^{high}
            end for
            B_i = F(c_1, ⋯, c_n) − F(c'_i, ⋯, c'_n)
        end for
        Define i_max = argmax_{1≤i≤n} B_i
        c_{i_max} + = v^{high}
        if (a deadline is missed) then
            Set t^{i_high}_af = 0
            Revert allocation c_{i_max} − = v^{high}
        else
            update Π
        end if
    end while
    return Π
end function
```

5.3 Time-bound module

The time-bound module (Figure 6) generates a static offline schedule for the given configuration. The input to the algorithm is a configuration χ passed in by the root module. The algorithm starts by assigning high voltage cycles sufficient to meet the worst case mandatory requirement of each task. Next, a check is done to analyse whether the schedule generated after this step violates the deadline. If this happens, the search heuristic is informed that this is an invalid configuration. If the deadline is not violated then the algorithm proceeds with the allocation of high cycles for the optional part of all tasks. The allocation is given to tasks in slices of v_{high}. This algorithm allocates this slice greedily to the tasks. The slice is given to the task where it will have the greatest improvement in final error at that instant. If by allocating the slice to a task, the path on which it is placed becomes critical (the TPG violates end-to-end time deadline) or if it exceeds the total worst case requirement of this task, the allocation is retracted and the task is marked for no allocations in the future. This allocation continues until all the tasks are marked as unallocatable, at which point the valid schedule is returned to the task allocator

module (δ_{ij} is the Kronecker delta, i.e., $\delta_{ij} = 1$ if $i = j$ and 0 otherwise.).

Figure 7 Energy bound module

function energyBound((Configuration χ))
  for $(1 \le i \le n)$ **do**
    $c_i^{low} = \lfloor c_i^{high} f_{low} / f_{high} \rfloor$
    $c_i^{high} = 0$
  end for
  for $(1 \le i \le n)$ **do**
    while $\left(c_i < c_i^{wm}\right)$ **do**
      $c_i^{low} -= v^{low}$
      $c_i^{high} += v^{high}$
    end while
  end for
  Calculate energy consumed, C_e
  while $(C_e > B_e)$ **do**
    $task_{found}^{low} = False$
    for $(1 \le i \le n)$ **do**
      $\Delta_i = \infty$
      if $\left(c_i \ge c_i^{wm} + v^{low}\right)$ **then**
        $task_{found}^{low} = True$
        for $(1 \le j \le n)$ **do**
          $c'_j = c_j - \delta_{i,j} v^{low}$
          $\Delta_i = F\left(c'_1, \cdots, c'_n\right) - F\left(c_1, \cdots, c_n\right)$
        end for
      end if
    end for
    if $\left(task_{found}^{low} == False\right)$ **then**
      return invalid
    else
      $i_{\min} = \min \arg_{1 \le i \le n} \Delta_i$
      $c_{i_{\min}}^{low} -= v^{low}$
      Recalculate C_e
    end if
  end while
  while $(C_e < B_e)$ **do**
    for $(1 \le i \le n)$ **do**
      for $(1 \le j \le n)$ **do**
        $c'_j = c_j + \delta_{i,j}\left(v^{high} - v^{low}\right)$
      end for
      $B_i = F\left(c_1, \cdots, c_n\right) - F\left(c'_1, \cdots, c'_n\right)$
    end for
    if $\left(\forall i \in \{1, \cdots, n\} c_i^{low} < v_i^{low}\right)$ **then**
      return schedule Π
    end if
    if $\left(\forall i \in \{1, \cdots, n\} c_i \ge c_i^{wo} + c_i^{wm}\right)$ **then**
      return schedule Π
    else
      Define $i_{\max} = \text{argmax}_{1 \le i \le n} B_i$.
      $c_{i_{\max}}^{high} += v^{high}$
      $c_{i_{\max}}^{low} -= v^{low}$
      Recalculate C_e
    end if
  end while
end function

5.4 Energy-bound module

The energy bound phase (Figure 7) starts after the time-bound phase arrives at a valid schedule with respect to time. The offline energy-bound phase starts by assigning low-voltage cycles to all the tasks in the time frame allocated by the time-bound phase. Then it makes sure that all the tasks have enough cycles to satisfy their required worst-case mandatory workload by converting low-voltage cycles to high-voltage cycles. After this stage, if the schedule has violated the energy constraint, then low cycles are removed from the tasks which least affect the final error without violating their worst case mandatory workload requirement. If the algorithm runs out of tasks to remove low cycles and the energy deadline is still violated, an invalid schedule is returned. If we are still under the energy deadline, after completing the mandatory workload of the task, the low cycles of the tasks are converted into high cycles greedily until the energy barrier is hit or we run out of low cycles. When this condition is reached a valid schedule is returned.

6 Online algorithm

As mentioned earlier, the actual execution times vary considerably. The actual demand of a task is not known unless and until the task completes execution. At this point, we know that the entire optional part has been executed. Once task T_i completes execution, we know that $\phi_i = 1$, meaning that the T_i output error will be given by $E_i(\vec{\sigma}_i, 1)$. This then affects all tasks that are downstream from it and allows the error function $F(\cdot)$ to be updated appropriately. If a task (say T_f) completes before its assigned time has been spent, additional time is released for other tasks to use. The job of the online algorithm is to reclaim this released time to improve on the offline schedule.

The online algorithm (Figure 8) makes sure that the tasks do not exceed their static finish times assigned by the offline algorithm while distributing the energy, thereby

respecting the end-to-end deadline. The two parameters to control the amount of time the online scheduler has for distributing the released energy are the granularity of allocation Δ_{online} and the set of tasks considered for distribution. The coarser the granularity, the longer the time for calculating benefit for the tasks and distributing the released energy.

The input parameter *LEV* controls the set of tasks considered for energy distribution when a task finishes: it can be regarded as a means to limit look-ahead in an effort to reduce the algorithm overhead. We only consider tasks which are *LEV* levels away from T_f in the task graph. *depth*(T_i) gives the shortest distance of task T_i from the root of the TPG and *oncriticalpath*(T_i) returns true if allocation of additional energy to the task violates the deadline or energy constraint and *finished*(T_i) returns true if the entire optional portion has finished.

Note that there is an asymmetry in information availability between tasks close to the top of the task graph and those towards the bottom. The former are executed in the face of very little information about actual execution times. By contrast, by the time the later tasks start executing, the actual execution times of the earlier tasks are known and therefore better-quality decisions can be taken with respect to these. On the other hand, note that the tasks high in the task graph provide output consumed by a large number of other tasks, and they get enhanced optional time allocation because of this.

Figure 8 Online module

function online((Δ_{online}, *LEV*))
  Calculate $t_{reclaim}$, time reclaimed on T_f completion.
  if ($t_{reclaim} == 0$) **then**
    return
  else
    while ($t_{reclaim} > 0$) **do**
      $n_{low}^{temp} = t_{reclaim} \cdot f_{low}$
      $n_{slice} = \Delta_{online} \cdot f_{low}$
      if $\left(n_{low}^{temp} < \Delta_{online}\right)$ **then**
        return
      end if
      $n_{allocated} = 0$
      Find task set OTS of T_x such that:
        $depth(T_x) - depth(T_f) \le LEV$
        $finished(T_x) = false$
        T_x is not on a critical path to a leaf
      if (OTS is empty) **then**
        return
      end if
      for ($T_i \in OTS$) **do**
        if $\left(c_i + n_{slice} > c_i^{wo} + c_i^{wm}\right)$ **then**
          Remove T_i from OTS
        end if
        if (OTS is empty) **then**
          return
        end if
      end for
      Assign n_{slice} cycles at v_{low} to T_k in OTS
      which yields the greatest improvement in error:
      c_k+ = n_{slice}
      $t_{reclaim}$ – = Δ_{online}
      if (T_k finishes later than in offline schedule) then
        (reverse this)
        c_k – = n_{slice}
        Remove T_k from OTS
        $t_{reclaim}$ + = Δ_{online}
      end if
    end while
  end if
end function

7 Numerical results

7.1 Experimental setup

7.1.1 Task graph modelling

Our numerical results are based on simulating 1,000 random directed acyclic directed TPGs, each of which was run 500 times with different random on-line run-times. Each TPG was generated based on an edge probability P, $P \in (0, 1)$, which specifies the probability of an edge between two nodes in the TPG, and a maximum out degree D specifying the maximum number of children a node can have. Low values of P and D will generate leaner TPGs with fewer dependencies, and vice versa. The worst case mandatory and optional parts of each task were selected at random out of {5, 10, 15}. The deadline for each TPG was selected as no lower than the sum of the worst case mandatory parts of the longest directional path in the graph. During allocation of time or energy to tasks, a critical path violation (a path in the TPG which violates the deadline) is identified using standard algorithms mentioned in Kwok and Ahmad (1999).

We assume that the error generated by an incomplete task is a convex function of the fraction of the uncompleted optional part out of the total optional part. We used as error function the function x^8 unless stated otherwise. In addition, each task has sensitivity values, which denote the sensitivity of its output error to its input errors. We selected these sensitivities at random for each task out of the interval (0, 2.0]. A high sensitivity value will lead to high increases in output error for small input errors and vice versa. A linear error propagation model is assumed for all the experiments conducted; the output error is convex with respect to the fraction of unexecuted optional part whereas it is linear with respect to the input errors.

The run-time characteristics of the tasks are modelled as follows. The actual run-time follows the normal distribution, conditioned on falling between specified minimum and maximum values, with the mean midway between them. The minimum value is given by a fraction (mf) of the worst case requirement whereas the maximum is the worst case itself: $t_i^m \in \{[\mathrm{mf}, 1.0] * t_i^{wm}\}$ and $t_i^o \in \{[\mathrm{mf}, 1.0] * t_i^{wo}\}$, where t_i^m, t_i^o are the actual mandatory and optional high-voltage execution times, respectively. The online phase is sampled 500 times and the average of 1,000 successfully scheduled TPGs is used for analysis.

7.2 Results

Figure 9 shows the average online error as a function of mf – the fraction of the minimum and the worst case run-time of tasks. The different curves pertain to different values of edge probability, *P*. As the value of *P* increases, the dependency between the tasks increases. From the error model it can be noted that both precedence and quality dependency increase as *P* increases. This results in the sharp rise of curves pertaining to a higher *P* value. Very high values of *P* result in too many TPGs with cycles and cannot produce meaningful results. When mf is 1.0, the tasks run to their worst case rendering the offline error (worst possible error for the given allocation) during the online phase.

Figure 9 Effect of minimum run-time

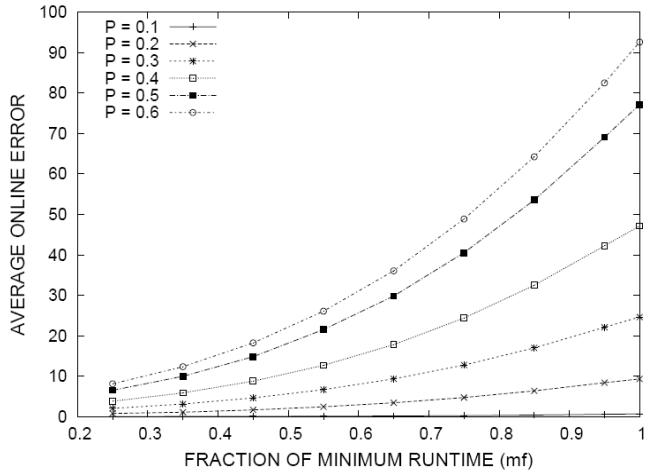


Figure 10 shows the effect of varying the standard deviation of the run-time distribution for different deadlines. As the standard deviation increases, the information available to the offline algorithm about the actual execution times decreases, and its expected error in estimating the actual execution time tends to increase. This leads to an increase in final error, as seen in the figure.

Figure 11 demonstrates the advantage obtained by online reclamation, by showing the ratio of the average online error with and without reclamation. We first observe that for a very tight deadline, not much reclamation is done even for increasing energy constraints. This can be due to the fact that cycles released by tasks are not effectively used by other tasks, because doing so would violate the TPG's deadline. Secondly, for medium time deadlines, reclamation is more effective for relatively tighter energy constraints than for loose energy constraints. This is due to the fact that for stricter energy constraints, even a small amount of energy released can be distributed much more effectively than for looser energy constraints. Thirdly, for very relaxed deadlines, there were not many chances to reclaim as tasks were allocated with ample resources in the offline stage.

Figure 10 Effect of standard deviation of run-times

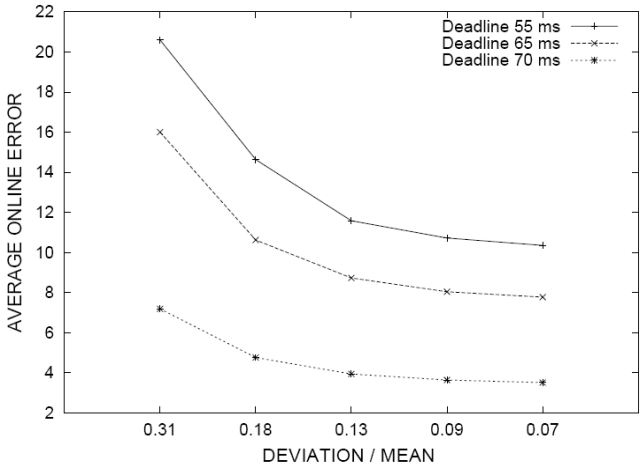


Figure 11 Effect of online reclamation

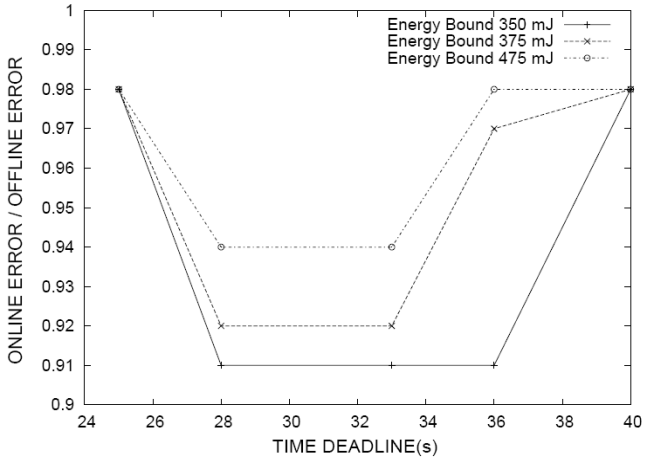
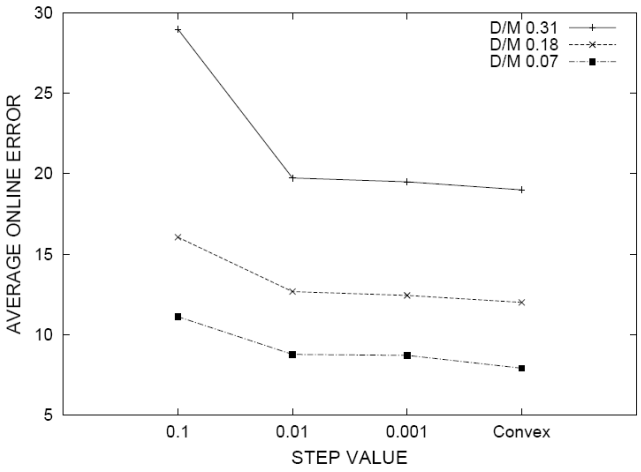


Figure 12 compares the average error for different error functions. We used a convex error function $f(x) = x^2$ and a series of step functions $g(x, steps) = (\lfloor (x * steps) \rfloor)/steps$ (where x is the fraction of the unexecuted/unallocated optional part and *steps* is a power of 10). As the value of *steps* increases, the step error function behaves much like the convex error function but is bounded below by it. The plot shows that for larger values of *step* (indicating a more abrupt but less frequent change in error value), the output error increases.

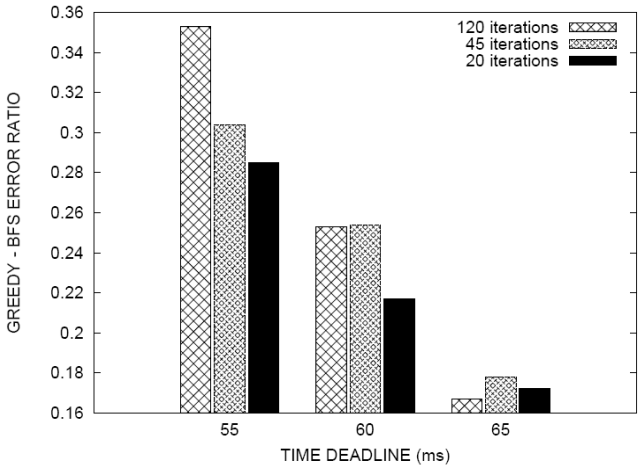
Figure 12 Effect of step error function



7.3 Comparison with the BFA algorithm

We are not aware of any other scheduling algorithm designed for arbitrary real-time imprecise task graphs running on an energy-bounded multiprocessor. Hence, our comparison is with a standard allocation approach: breadth first allocation (BFA): see Figure 13. Our justification for picking BFA as a baseline is that it is generally used as an initial (round robin) task allocation step, following which a uniprocessor scheduling step is undertaken. The standard approach to scheduling real-time tasks in multiprocessors is to first allocate them to processors and then use uniprocessor scheduling algorithms for tasks allocated to each processor.

Figure 13 Greedy vs. BFS allocation

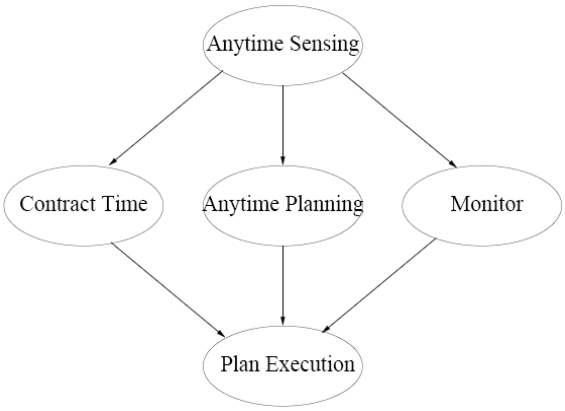


The BFA offline and online algorithms have no knowledge of the error functions associated with the tasks. This algorithm prioritises the tasks based on their appearance in a breadth first search and allocates time to them as per their priorities while taking care not to violate the deadline. The plot shows the ratio of the final average error of our algorithm to BFA as a function of the deadline, for three values of the number of iterations of the offline algorithm, which depends on the time allotted to the offline scheduler. Our algorithm performs much better than BFA when the scheduler has less time and a very tight deadline for the TPG. As expected, our algorithm beats BFA by larger margins as the deadline gets loose. This is mainly because the optional part for the tasks is allocated more wisely based on the benefit in final error. The rate of improvement as a function of the number of iterations is different for different algorithms; the performance ratio is therefore not monotonic.

7.4 A robot application

This experiment was conducted on a mathematical model of a robot implementing anytime sensing, planning and action as shown in Figure 14 (Zilberstein and Russell, 1993). Our analysis concentrates on determining the resource allocation for each of the tasks by using our scheduling algorithm for minimising the error in the final output. Figure 15 shows the effect of simultaneous time deadline and energy constraint on the system. The error functions we used were obtained by curve fitting to the performance profiles found in Zilberstein and Russell (1993) as shown in Figure 16. The curves in Figure 16 also show the effect of input error on the performance profiles. The tasks that do not have an optional part do not have an effect on the final output quality. This figure quantifies the improvement possible with looser deadline and energy constraints. Such improvements can be taken into account when selecting such deadlines in the process of designing the overall system. In particular, such a selection would constitute a trade-off between the quality of control output against its timeliness, as it affects the control of the plant.

Figure 14 Anytime robot



Source: Zilberstein and Russell (1993)

Figure 15 Error variation with time and energy constraints

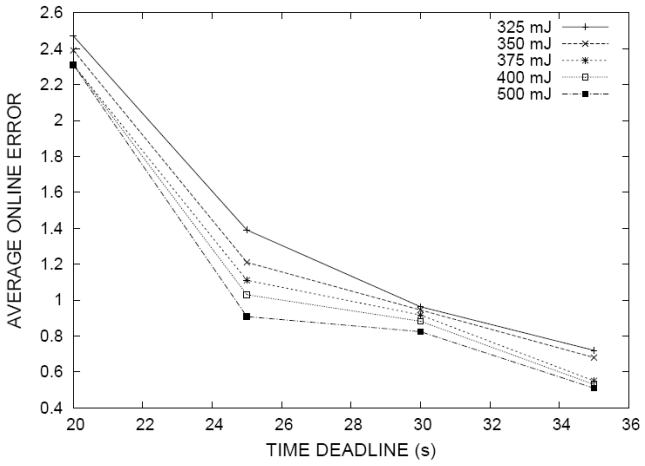
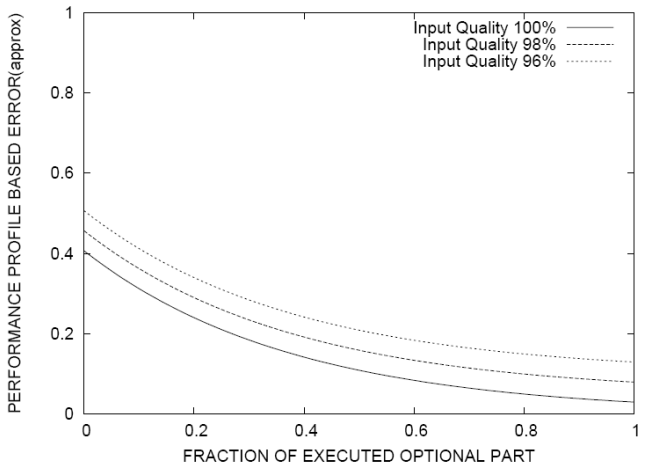


Figure 16 Error functions based on performance profiles



8 Discussion

We have presented offline and online heuristics for scheduling imprecise task graphs on multiprocessors. The task graphs may be arbitrary; the error output of a task is a function of both the input error and the premature termination (if any) of the optional portion of the task. We include in our model the fact that task run-times are not known precisely in advance, but only statistically.

Future work includes the use of hierarchical scheduling methods to allow imprecise tasks to coexist with traditional 0-1 task sets. Another promising area for study is the instrumenting of imprecise code, which would allow one to determine its execution progress and thereby provide additional run-time information to the system without imposing too great an overhead.

Acknowledgements

The authors would like to thank the referees for their careful reading of the draft manuscript and their perceptive comments. This paper was supported in part by the National Science Foundation under grant CNS-0931035.

References

'Xvid tool' [online] http://www.xvid.org/ (accessed 01/02/2011).

Cheng, A. and Wang, R. (2004) 'A new scheduling algorithm and a compensation strategy for imprecise computation', in *Proceedings of the Computer Software and Applications Conference (COMPSAC)*, Vol. 1, pp.167–172.

Chishiro, H. and Yamasaki, N. (2011) 'Global semi-fixed-priority scheduling on multiprocessors', in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, IEEE, pp.218–223.

Chishiro, H., Takeda, A., Funaoka, K. and Yamasaki, N. (2010) 'Semi-fixed-priority scheduling: new priority assignment policy for practical imprecise computation', in *IEEE International Conference on Embedded and Real-time Computing Systems and Applications, ser. RTCSA '10*, pp.339–348.

Chung, J-Y., Liu, J.W.S. and Lin, K-J. (1990) 'Scheduling periodic jobs that allow imprecise results', *IEEE Trans. Comput.*, September, Vol. 39, pp.1156–1174 [online] http://dx.doi.org/10.1109/12.57057 (accessed 05/07/2011).

Cortes, L., Eles, P. and Peng, Z. (2006) 'Quasi-static assignment of voltages and optional cycles in imprecise-computation systems with energy considerations', *IEEE Transactions on VLSI*, Vol. 14, No. 10, pp.1117–1129.

de Oliveira, R., Fraga, J. and Farines, J-M. (2001) 'Scheduling imprecise tasks in real-time distributed systems', in *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE, pp.319–326.

Dey, J.K., Kurose, J. and Towsley, D. (1996) 'On-line scheduling policies for a class of iris (increasing reward with increasing service) real-time tasks', in *IEEE Transactions on Computers*, July, pp.802–813.

Dey, J.K., Kurose, J.F., Towsley, D.F., Krishna, C.M. and Girkar, M. (1993) 'Efficient on-line processor scheduling for a class of iris (increasing reward with increasing service) real-time tasks', in *SIGMETRICS*, pp.217–228.

Feng, W. and Liu, J. (1993) 'An extended imprecise computation model for time-constrained speech processing and generation', in *Proceedings of the IEEE Workshop on Real-Time Applications*, May, pp.76–80.

Feng, W. and Liu, J.W.S. (1997) 'Algorithms for scheduling real-time tasks with input error and end-to-end deadlines', in *IEEE Transactions on Software Engineering*, February, pp.93–106.

Gupta, V. (2009) 'On an anytime algorithm for control', *Proceedings of the 48h IEEE Conference on Decision and Control CDC held jointly with 2009 28th Chinese Control Conference*, pp.6218–6223 [online] http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5400834 (accessed 12/08/2011).

Khemka, A., Shyamasundar, R. and Subrahmanyam, K. (1993) 'Multiprocessors scheduling for imprecise computations in a hard real-time environment', in *International Parallel Processing Symposium*, pp.374–378.

Kwok, Y-K. and Ahmad, I. (1999) 'Static scheduling algorithms for allocating directed task graphs to multiprocessors', *ACM Computing Surveys*, Vol. 31, No. 4, pp.406–471.

Li, P., Wu, W. and Lu, F. (2009) 'Analysis on influential factors for meta-level control of the anytime algorithm for dynamic WTA problem', in *International Workshop on Intelligent Systems and Applications (ISA)*, pp.1–4.

Liu, C.L. and Layland, J.W. (1973) 'Scheduling algorithms for multiprogramming in a hard-real-time environment', *J. ACM*, January, Vol. 20, pp.46–61 [online] http://doi.acm.org/10.1145/321738.321743.

Liu, J., Lin, K-J., Shih, W-K., Yu, A., Chung, J-Y. and Zhao, W. (1991) 'Algorithms for scheduling imprecise computations', *IEEE Computer*, May, Vol. 24, No. 5, pp.58–68.

Park, J., Shin, D., Chang, N. and Pedram, M. (2010) 'Accurate moeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors', in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp.419–424.

Pillai, P. and Shin, K. (2001) 'Real-time dynamic voltage scaling for low-power embedded operating systems', in *Symposium on Operating System Principles*, pp.89–102.

Shih, W. and Liu, J. (1992) 'On-line scheduling of imprecise computations to minimize error', in *IEEE RealTime Systems Symposium*, December.

Shin, W-K., Lee, C-R. and Tang, C-H. (2000) 'A fast algorithm for scheduling imprecise computational with timing constraints to minimize weighted error', in *IEEE Real-Time Systems Symposium*, pp.305–310.

Shin, W-K., Liu, J. and Chung, J-Y. (1989) 'Fast algorithms for scheduling imprecise computations', in *IEEE Real-Time Systems Symposium (RTSS)*, pp.12–19.

Tchamgoue, G.M., Kim, K.H., Jun, Y-K. and Lee, W.Y. (2010) 'Hierarchical real-time scheduling framework for imprecise computations', in *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pp.273–280.

Zilberstein, S. and Russell, S.J. (1993) 'Anytime sensing, planning and action: a practical model for robot control', in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, pp.1402–1407. [online] http://rbr.cs.umass.edu/shlomo/papers/ZRijcai93.html (accessed 10/12/2011).

Zilberstein, S., Washington, R., Bernstein, D.S. and Mouaddib, A-I. (2002) 'Decision-theoretic control of planetary rovers', in *Revised Papers from the International Seminar on Advances in Plan-Based Control of Robotic Agents*, Springer-Verlag, London, UK pp.270–289 [online] http://rbr.cs.umass.edu/shlomo/papers/ZWBMlnai02.html (accessed 03/01/2012).

Notes

1 If the workload consists of multiple, independent, TPGs, this can be handled within our framework by introducing a virtual root node whose children are the roots of these independent TPGs.