Optimization of Soft Real-Time Systems with Deadline Miss Ratio Constraints

Sorin Manolache, Petru Eles, Zebo Peng
Dept. of Computer and Information Science, Linkoping University, Sweden
{sorma, petel, zebpe}@ida.liu.se

Abstract

Both analysis and design optimization of real-time systems has predominantly concentrated on considering hard real-time constraints. For a large class of applications, however, this is both unrealistic and leads to unnecessarily expensive implementations. This paper addresses the problem of task priority assignment and task mapping in the context of multiprocessor applications with stochastic execution times and in the presence of constraints on the percentage of missed deadlines. We propose a design space exploration strategy based on Tabu Search together with a fast method for system performance analysis. Experiments emphasize the efficiency of the proposed analysis method and optimization heuristic in generating high quality implementations of soft real-time systems with stochastic task execution times and constraints on deadline miss ratios.

1. Introduction

For the large majority of cases, if not all, the task execution times are variable. This variability may be caused by application dependent factors (data dependent loops and branches), architectural factors (unpredictable cache and pipeline behavior), or environment dependent factors (network load, for example). In the case of safety critical applications (avionics, automotive, medicine, nuclear plant control systems), the designers have to deal with the worst case scenario, in particular with worst case task execution times (WCET). An impressive amount of research results, both for analyzing and designing these systems has been published [2].

Designing based on the worst-case execution time (WCET) model guarantees that no timing requirement is broken. However, for large classes of applications, the soft real-time systems, breaking a timing requirement, though not desirable, is tolerated provided that this happens with a sufficiently low probability. Whereas in the case of safety critical systems, the designers stress safety at the expense of product cost, in the case of soft real-time systems, cost reduction is a strong incentive for using cheap architectures.

Let us consider a cheap processor and a task which runs on it. The probability density function of the task execution time (ETPDF) on the cheap processor is depicted in Figure 1a. If the imposed deadline of the task is *t* as shown in the figure, then the cheap processor cannot guarantee that the task will always meet its deadline, as the WCET of the task exceeds the deadline. If no deadline misses were tolerated, a faster and more expensive processor would be needed. The ETPDF of the task on the faster processor is depicted in Figure 1b. In this case, the more expensive processor guarantees that no deadlines are missed. However, if a miss deadline ratio of at most 15% is tolerated, then even the cheaper processor would suffice.

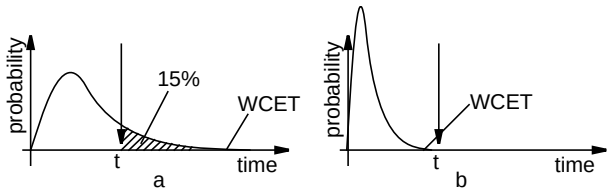

Figure 1: Execution time probability density

The problem of finding the deadline miss ratios, given a hardware platform and an application, is not trivial and has attracted relatively recent research work both for monoprocessor [5], [9], [12] and for multiprocessor systems [6], [10].

This work, for the first time to our knowledge, addresses the complementary problem: given a multiprocessor hardware architecture and a functionality as a set of task graphs, find a task mapping and priority assignment such that the deadline miss ratios satisfy imposed constraints.

A naïve approach to this problem would be to use fixed execution time models (average, median, worst case execution time, etc.) and to hope that the resulting designs would be optimal or close to optimal also from the point of view of the percentage of missed deadlines. The following example illustrates the pitfalls of such an approach and emphasizes the need for an optimization technique which considers the stochastic execution times. Let us consider the application in Figure 2a. The circles denote the tasks, their shades denote the processors they are mapped onto. The solid disks show the inter-processor communication. The arrows between the tasks indicate their data dependencies. All the tasks have period 20 and the deadline of the task graph is 18. Tasks A, B, C, and D have constant execution times of 1, 6, 7, and 8 respectively. Task E has a variable execution time whose probability is uniformly distributed between 0 and 12. Hence, the average (expected) execution time of E is 6.

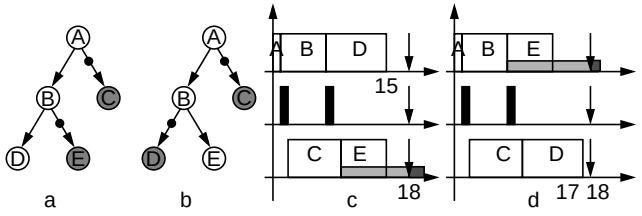

Figure 2: Motivational example

The inter-processor communication takes 1 time unit per message. Let us consider the two mapping alternatives depicted in Figure 2a and b. The two Gantt diagrams in Figure 2c and d depict the execution scenarios corresponding to the two considered mappings if the execution of task E took the expected amount of time, that is 6. The shaded rectangles depict the probabilistic execution of E. A mapping strategy based on the average execution times would select the mapping in Figure 2a as it leads to a shorter response time (15 compared to 17). However, in this case, the worst case execution time of the task graph is 21. The deadline miss ratio of the task graph is 3/12 = 25%. If we took into consideration the stochastic nature of the execution time of task E, we would prefer the second mapping alternative, because of the better deadline miss ratio of 1/12=8.33%. If we considered worst case response times instead of average ones, then we would chose the second mapping alternative, like a stochastic approach. However, approaches based on worst case execution times can be dismissed by means of very simple counter-examples.

Let us consider a task τ which can be mapped on processor P_1 or on processor P_2. P_1 is a fast processor with a very deep pipeline. Because of its pipeline depth, mispredictions of target addresses of conditional jumps, though rare, are severely penalized. If τ is mapped on P_1, its ETPDF is shown in Figure 3a. The long and flat density tail corresponds to the rare but expensive jump target address misprediction. If τ is mapped on processor P_2, its ETPDF is shown in Figure 3b. Processor P_2 is slower with a shorter pipeline. The WCET of task τ on processor P_2 is smaller than the WCET if τ ran on processor P_1. Therefore, a design space exploration tool based on the WCET would map task τ on P_2. However, as Figure 3 shows, the deadline miss ratio in this case is larger than if task τ was mapped on processor P_1.

The remainder of the paper is structured as follows. The next section presents our system model and gives the problem formulation. Section 3 presents the design space exploration strategy detailing on the neighborhood restriction heuristic. Section 4 describes the fast approximate method for system analysis. Section 5 presents a set of experiments we conducted in order to evaluate and demonstrate the efficiency of the proposed heuristic. Finally, Section 6 draws the conclusions.

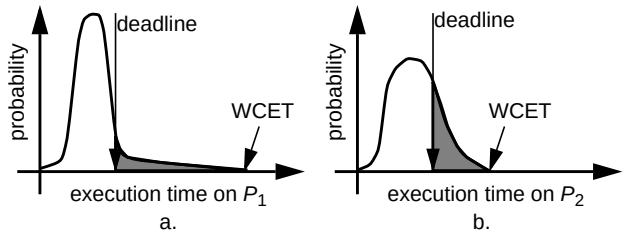

Figure 3: Motivational example

2. System model and problem formulation

This section presents the platform and functionality models and gives the problem formulation. It concludes by identifying the implied subproblems which will be discussed in the following sections.

2.1. System model

2.1.1. Hardware platform. The hardware platform is composed of a set of *processors* which are interconnected by means of *buses*. Processors and buses will be both referred to as *processing elements* in the sequel of this paper.

2.1.2. Application model. The application consists of a set of *N processing tasks*. Processing tasks are depicted as circles, as exemplified in Figure 2. *Data dependencies* between pairs of tasks are captured as arrows connecting the dependent tasks, as seen in Figure 2. Data dependencies define precedence relationships between tasks. For example, in Figure 2, there exists a data dependency between task A and task B and A is a predecessor of B. The predecessors of a task form its predecessor set. The transitive and symmetric closure of the precedence relationship partitions the task set into *task graphs*. For the example in Figure 2, the application consists of one task graph.

A job is an instantiation of a task and (τ, j) denotes the j^{th} job of task τ. Jobs of any task τ_i, $1 \le i \le N$, are periodically released with a *task-specific period* π_i. The period π_i of task τ_i has to be a common integer multiple of the periods of the tasks in the predecessor set of τ_i. Let $k_{ij}=\pi_j/\pi_i$, where τ_j depends on τ_i, with the interpretation that each job of τ_j receives k_{ij} data items from k_{ij} jobs of τ_i. A job of task τ is ready for execution only after it receives all the needed data items, i.e. only after the corresponding jobs of all predecessor tasks of τ have finished their execution. The least common multiple of the periods of the tasks belonging to a task graph G is the *task graph period* of G.

The execution of jobs is considered to be *non-preemptive*. If preemption is desired, the designer can define preemption points by splitting a task in a chain of subtasks.

If a task τ is *mapped* on a processor *PE*, then all jobs of τ execute on *PE*. If two communicating tasks are mapped on the same processor, then the communication time is not explicitly specified but it is considered as part of the sender task's execution time. If they are mapped on different pro-

cessors, the sender has to send a message on the bus linking the two processors. Message transmission is modelled by means of *communication tasks* mapped on buses. Communication tasks are depicted by means of solid disks in Figure 2. Unless explicitly stated, processing and communication tasks will not be differently treated and will be referred to as *tasks* in the sequel of this paper.

Each task τ can potentially be mapped on a set of processing elements PE_τ. The set PE_τ represents the set of allowed mappings of τ.

The execution (communication) times of tasks are assumed to be varying. Therefore, they are specified by means of *execution time probability density functions (ET-PDFs)*. There are no restrictions whatsoever on the class of those functions. The task execution times are considered to be statistically independent.

If several tasks (messages) compete for the processor (bus), the task with the highest *priority* is chosen by a real-time scheduler (or bus arbiter) to execute. The task priorities are assumed to be static, i.e. invariant during the lifetime of the application. Based on these assumptions, we are able to model any priority based bus arbitration protocol as, for instance, CAN [1].

The real-time requirements on tasks are specified by means of deadlines of tasks and task graphs. A job misses its deadline if its completion time exceeds its deadline. The long-term percentage of missed deadlines defines the *deadline miss ratio (DMR)* of a task. A task graph instance misses its deadline if not all its jobs complete at the time of the task graph deadline. These not completed jobs are said to be *late* and they are immediately removed (discarded) from the system. Similarly, the deadline miss ratio of a task graph is defined as the long-term percentage of deadlines missed by the task graph.

The designer can impose *thresholds* for deadline miss ratios and can designate certain tasks or task graphs as being *critical*. A task or a task graph is critical if its DMR is not allowed to exceed the corresponding specified threshold. We define the *deviation* (denoted *dev*) from the miss threshold as follows:

$$dev_\tau = \begin{cases} \infty, DMR > threshold, \tau \text{ critical} \\ DMR - threshold, DMR > threshold, \tau \text{ non-critical} \\ 0, DMR \le threshold \end{cases}$$

2.2. Problem formulation

The problem addressed in this paper is formulated as follows.

The problem input consists of

- the set of processors, the set of buses, and their connection to processors,
- the set of task graphs,
- the set of task periods,
- the set of task deadlines,
- the set of task graph deadlines,
- the set PE_τ of allowed mappings of τ for all tasks τ,
- the set of execution (communication) time probability density functions corresponding to each processing element $p \in PE_\tau$ for each task τ,
- the set of miss thresholds, and
- the set of critical tasks and task graphs.

The problem output consists of a *mapping* and *priority assignment* such that the cost function

$$\sum dev,$$

giving the sum of miss deviations is minimized. If a mapping and priority assignment is found such that Σ *dev* is finite, it is guaranteed that the DMRs of all critical tasks and task graphs are below their imposed thresholds.

Because the defined problem is of NP-hard complexity (see the complexity of the classical mapping problem [3]), we have to rely on heuristic techniques for solving the formulated problem. An accurate estimation of the miss deviation, which is used as a cost function for the optimization process, is in itself a complex and time consuming task [10]. Therefore, a fast approximation of the cost function value is needed to guide the design space exploration. Hence, the following subproblems have to be solved:

- find an efficient design space exploration strategy, and
- develop a fast and sufficiently accurate analysis, providing the needed indicators.

Section 3 discusses the first subproblems while Section 4 focuses on the system analysis we propose. Before that, we will refresh and define some notions to be used in the following sections.

2.3. Definition of various random variables

We refresh the notions of job arrival time, starting time, and finishing time. The finishing time of the j^{th} job of task τ is the time moment when (τ, j) finishes its execution. We denote it with $F_{\tau,j}$. The deadline miss ratio of a job is the probability that its finishing time exceeds its deadline:

$$DMR_{\tau,j} = 1 - P(F_{\tau,j} \le deadline_{\tau,j})$$

The arrival time or ready time of (τ, j) is the time moment when (τ, j) is ready to execute, i.e. the maximum of the finishing times of jobs in its predecessor set. We denote the arrival time with $A_{\tau,j}$ and we write

$$A_{\tau,j} = max_{\sigma \in Pred(\tau)} F_{\sigma,j}$$

The starting time of (τ, j) is the time moment when (τ, j) starts executing. We denote it with $S_{\tau,j}$. Obviously, the relation $F_{\tau,j}=S_{\tau,j}+E_{\tau,j}$ holds between the starting and finishing times of (τ, j), where $E_{\tau,j}$ denotes the execution time of (τ, j). The arrival time and starting times of a job may differ because the processor might be busy at the time the job arrives. The arrival, starting and finishing times are all random variables.

3. Mapping and priority assignment heuristic

In this section, we propose a design space exploration strategy which is based on the Tabu Search (TS) [4] metaheuristic.

Before presenting the details of the exploration strategies, we define some notions we will use in the sequel. We define the design space as the space of all allowed task mappings and priority assignments. Each task is characterized by two *attributes*: its mapping and its priority. A *move* in the design space is equivalent to changing one or both attributes of *one single task*. Two points in the design space are *neighbors* if one of them can be reached from the other by means of one move, i.e. by remapping and/or reassigning the priority of one task. After a move M is performed, the reverse move $\overline{M}$ is labelled as *tabu* for the next k iterations, where k is the *tabu tenure* of $\overline{M}$. Tabu moves may not be performed, with few exceptional cases. Thus, it is ensured that the effect of a move M is not reversed and the exploration does not stuck at local minima.

The exploration algorithm is shown in Figure 4. The exploration starts from a random initial solution, labelled also as the current solution and considered the globally best solution so far. The cost function Σ *dev* is evaluated for the current solution. The list *TM* of tabu moves is initially empty. A set S of candidate moves is defined and let *N(S)* be the set of solutions which can be reached from the current solution by means of a move in S. The cost function is evaluated for each solution in *N(S)*. A move $m \in S$ is selected if

- it is non-tabu and leads to the solution with the lowest cost among the solutions in *N(S)*, or
- it is tabu but improves the globally best solution so far, or
- all moves in S are tabu and m leads to the solution with the lowest cost among those solutions in *N(S)*.

The new solution is obtained by applying the chosen move m on the current solution. The reverse of move m is marked as tabu such that m will not be reversed in the next few iterations. If it is the case, the new solution becomes also the globally best solution reached so far. However, it should be noted that the new solution could have a larger cost than the current solution. This could happen if there are no moves which would improve on the current solution or all such moves would be tabu. The list *TM* of tabu moves ensures that the heuristic does not get stuck in local minima. If no global improvement has been noted for the past W iterations, the loop is interrupted. In this case, a diversification phase follows in which a rarely used move is performed in order to force the heuristic to explore different regions in the design space. The whole procedure is repeated until the heuristic iterated for a specified maximum number of iterations.

The cost function is evaluated $|S|$ times at each iteration, where $|S|$ is the cardinality of the set of candidate moves. Let us consider that task τ, mapped on processor P_j, is moved on processor P_i and there are q_i tasks on processor P_i. Task τ can take one of q_i+1 priorities on processor P_i. If task τ is not moved on a different processor, but only its priority is changed on processor P_j, then there are q_j-1 possible new priorities. If we consider all processors, there are $N+P-2$ possible moves for each task τ, as shown in the equation below, where N is the number of tasks and P is the number

$$q_j - 1 + \sum_{i \neq j} (q_i + 1) = P - 2 + \sum_i q_i = P - 2 + N$$

of processors. Hence, if all possible moves are candidate moves, $N \cdot (N+P-2)$ moves are possible at each iteration. Therefore, a key to the efficiency of the algorithm is the intelligent selection of the set S of candidate moves. If S contained only those moves which had a high chance to drive the search towards good solutions, then fewer points would be probed, leading to a speed up of the algorithm.

In our approach, the set S of candidate moves is composed of all moves which operate on a subset of tasks. Tasks are assigned scores and the chosen subset of tasks is composed of the first K tasks with respect to their score.

We illustrate the way the scores are assigned to tasks based on the example in Figure 2a. As a first step, we identify the critical paths and the non-critical branches of the application. In general, we consider a path to be an ordered sequence of tasks $(\tau_1, \tau_2, \ldots, \tau_n)$ such that τ_{i+1} is data dependent on τ_i. The average execution time of a path is given by the sum of the average execution times of the tasks belonging to the path. A path is critical if its average execution

```
TM=∅
crt_sol = init_sol
global_best_cost = evaluate(crt_sol)
global_best_sol = crt_sol
since_last_improvement = 0
iteration_count = 0
S = set_of_candidate_moves(TM, crt_sol)
(chosen_move, next_cost) = choose_move(S)
while iteration_count < max_iterations do
    while since_last_improvement < W do
        next_sol = move(crt_sol, chosen_move)
        TM = TM ∪ {chosen_move}
        since_last_improvement++
        iteration_count++
        crt_sol = next_sol
        if next_cost < global_best_cost then
            global_best_cost = next_cost
            global_best_sol = crt_sol
            since_last_improvement = 0
        endif
        S = set_of_candidate_moves(TM, crt_sol)
        (chosen_move, next_cost) = choose_move(S)
    done
    since_last_improvement = 0
    (chosen_move, next_cost) = diversify(TM, crt_sol)
    iteration_count++
done
return best_sol
```

Figure 4: Design space exploration strategy

time is the largest among the paths belonging to the same task graph. For the example in Figure 2a, the critical path is $A \rightarrow B \rightarrow D$, with an average execution time of 1+6+8=15. In general, non-critical branches are those paths starting with a root node or a task on a critical path, ending with a leaf node or a task on a critical path and containing only tasks which do not belong to any critical path. For the example in Figure 2a, non-critical branches are $A \rightarrow C$ and $B \rightarrow E$. For each critical path or non-critical branch, a *path mapping vector* is computed. The mapping vector is a P-dimensional integer vector, where P is the number of processors. The modulus of its projection along dimension p_i is equal to the number of tasks which are mapped on processor p_i and which belong to the considered path. For the example in Figure 2a, the vectors corresponding to the paths $A \rightarrow B \rightarrow D$, $A \rightarrow C$ and $B \rightarrow E$ are 3**i**+0**j**, 1**i**+1**j**, and 1**i**+1**j** respectively, where **i** and **j** are the versors along the two dimensions. Each task is characterized by its *task mapping vector*, which has a modulus of 1 and is directed along the dimension corresponding to the processor on which the task is mapped. For example, the task mapping vectors of *A*, *B*, *C*, *D*, and *E* are 1**i**, 1**i**, 1**j**, 1**i**, and 1**j** respectively. Next, for each path and for each task belonging to that path, the angle between the path and the task mapping vectors is computed. For example, the task mapping vectors of tasks *A*, *B*, and *D* form an angle of 0° with the path mapping vector of critical path $A \rightarrow B \rightarrow D$ and the task mapping vectors of task *A* and *C* form an angle of 45° with the path mapping vector of the non-critical branch $A \rightarrow C$. The score assigned to each task is a weighted sum of angles between the task's mapping vector and the mapping vectors of the paths to whom the task belongs. The weights are proportional to the relative criticality of the path. Intuitively, this approach attempts to map the tasks which belong to critical paths on the same processor. In order to avoid processor overload, the scores are penalized if the task is intended to be moved on highly loaded processors.

Once scores have been assigned to tasks, the first $K=N/c$ tasks are selected according to their scores. In our experiments, we use c=2. In order to further reduce the search neighborhood, only 2 processors are considered as target processors for each task. The selection of those two processors is made based on scores assigned to processors. These scores are a weighted sum of potential reduction of interprocessor communication and processor load. The processor load is weighted with a negative weight, in order to penalize overload. For example, if we moved task *C* from the shaded processor to the white processor, we would reduce the interprocessor communication with 100%. However, as the white processor has to cope with an average work load of 15 units (the average execution times of tasks *A*, *B*, and *D*), the 100% reduction would be penalized with an amount proportional to 15.

On average, there will be N/P tasks on each processor. Hence, if a task is moved on a processor, it may take $N/P+1$ possible priorities on its new processor. By considering only $N/2$ tasks and only 2 processors for each task, we restrict the neighborhood to $N/2 \cdot 2 \cdot (1+N/P) = N \cdot (1+N/P)$ candidate moves on average, i.e. approximately $N \cdot (N+P-2) / (N \cdot (1+N/P)) \approx P$ times. We will denote this method as the *restricted neighborhood search* and we will compare it with an exhaustive exploration of the neighborhood of design space points in Section 5.

4. Analysis

The cost function which is driving the design space exploration is $\sum dev$, where *dev* is the miss deviation as defined in Section 2. Given the input data listed in Section 2.2 and given a task mapping and priority assignment, the analysis procedure computes an approximation of the miss deviation for each task.

In previous work [10], we proposed a performance analysis method for multiprocessor applications with stochastic task executions times. The method is based on the Markovian analysis of the underlying stochastic process. As the latter captures all possible behaviors of the system, the method gives great insight regarding the system's internals and bottlenecks. However, its large analysis time makes its use prohibitive inside an optimization loop setting as presented in the previous section. Therefore, we propose an *approximate* analysis method of polynomial complexity.

Let *LCM* denote the least common multiple of task periods and *GCD* their greatest common divisor. Because late tasks are removed from the system, each *LCM* time units the system returns to its initial state. Therefore, it is sufficient to analyses the system over the time span [0, *LCM*). We define the discrete set of time moments T as the set

$$T = \{t_n, t_n = n \cdot h, 0 \le n < LCM/h\}$$

where h is the disc re tis at ion resolution of the interval [0, *LCM*). The disc re tis at ion resolution is experimentally chosen such that it leads to a satisfactory trade-off between analysis time and accuracy. In order to speed up the analysis, we approximate the starting and finishing time of jobs with values in the discrete set T.

If the events that the predecessor tasks complete their execution were independent, the probability that τ arrives before or at time moment t_n would be equal to the product of the probabilities that the predecessor tasks complete their execution prior or at time moment t_n:

$$P(A_{\tau,j} \le n) = \prod_{\sigma \in Pred(\tau)} P(F_{t,j} \le n) \qquad (1)$$

The independence assumption stated above generally does not hold. However, the dependence is usually weak, as the task completion events are influenced by so many different aspects. Therefore, in practice, Eq(1) is a good ap-

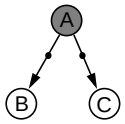

Figure 5: Example application

proximation, as shown by Kleinrock [7] for computer networks and by Li [8] for multiprocessor applications.

Let $L_{\tau,j}(n)$ be the event of (τ, j) running at time moment t_n. Its probability represents the instantaneous processor load caused by (τ, j).

We depart from the observation that it is easy to determine the probabilities $P(F_{\tau,j}=n)$ and $P(L_{\tau,j}(n))$ if we know the starting time of (τ, j). In this case, the following equations hold:

$$P(F_{\tau,j} = n \mid S_{\tau,j} = k) = P(E_{\tau,j} = n-k)$$
$$P(L_{\tau,j}(n) \mid S_{\tau,j} = k) = P(E_{\tau,j} > n-k) \quad (2)$$

It follows that the probabilities $P(F_{\tau,j}=n)$ and $P(L_{\tau,j}(n))$ at time moment t_n could be determined if $P(S_{\tau,j}=k)$ was known for all moments t_k prior to t_n.

Let $M(\tau)$ denote the processor on which task τ is mapped and let MT denote the set of all tasks which are mapped on $M(\tau)$. The probability $P(S_{\tau,j}=n)$ that job j of task τ starts executing at time moment t_n is given by the equation

$$P(S_{\tau,j} = n) = P(S_{\tau,j} \geq n \cap I_{\tau,j}(n)) - P(A_{\tau,j} > n \cap I_{\tau,j}(n))$$

where $I_{\tau,j}(n)$ denotes the event that the processor $M(\tau)$ is idle and no task of higher priority than τ is ready to execute. Intuitively, if task τ starts in the present ($S_{\tau,j}=n$) or if it arrives in the future ($A_{\tau,j}>n$) *and* the processor is free in the present moment ($I_{\tau,j}(n)$) then it follows that task τ starts in the present or in the future and that the processor is free in the present. Because $P(S_{\tau,j} \geq n \cap I_{\tau,j}(n)) = P(I_{\tau,j}(n)) - P(S_{\tau,j} < n \cap I_{\tau,j}(n))$, the equation above can be rewritten as

$$P(S_{\tau,j} = n) = P(A_{\tau,j} \leq n \cap I_{\tau,j}(n)) - P(S_{\tau,j} < n \cap I_{\tau,j}(n)) \quad (3)$$

Eq(3) is rewritten as

$$P(S_{\tau,j} = n) = P(I_{\tau,j}(n) \mid A_{\tau,j} \leq n) \cdot P(A_{\tau,j} \leq n) - P(I_{\tau,j}(n) \mid S_{\tau,j} < n) \cdot P(S_{\tau,j} < n) \quad (4)$$

The complement of the event $I_{\tau,j}(n)$ is the event that processor $M(\tau)$ is busy. Therefore

$$P(I_{\tau,j}(n) \mid A_{\tau,j} \leq n) = 1 - \sum_{\sigma \in MT} P(L_{\sigma}(n) \mid A_{\tau,j} \leq n) \quad (5)$$

The event that a task $\sigma \in MT$ is running is not independent of the event that (τ, j) arrived by a time moment t_n, as it can be seen in the following example. Let us consider the application depicted in Figure 5. Task A has an average execution time of 1/3s and its execution time probability is exponentially distributed. Task A is mapped on the shaded processor while tasks B and C are mapped on the white processor. Task B has a fixed execution time of 4s. Interprocessor message transmission take 0.5s. The message from task

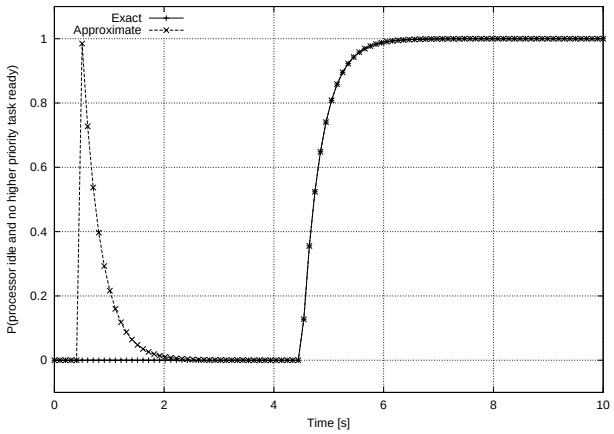

Figure 6: Approximation of idle probability

A to task *B* is always sent before the message to task *C*. Therefore, when task *C* becomes ready to run, it will always find the processor already busy executing task *B*. Hence, task *C* runs only after task *B* completes its execution. We are interested in the strength of the dependence between arrival of task *C* before time moment *t* and the event that task *B* is running at time moment *t*. Differently stated, we would like to know how well does $P(L_B(t))$ approximate $P(L_B(t)|A_C \leq t)$. Task *B* is running at time moment *t* if task *A* completed its execution at a time moment in the interval (*t*-4.5, *t*-0.5]. Thus,

$$P(L_B(t)) = \begin{cases} 1 - e^{-3 \cdot (t-0.5)} & 0.5 < t \leq 4.5 \\ e^{-3 \cdot (t-4.5)} - e^{-3 \cdot (t-0.5)} & t > 4.5 \end{cases}$$

On the other hand, if we knew that task *C* arrived before time moment *t* the probability of task *B* running at time moment *t* would be modified as below, without giving the actual deduction.

$$P(L_B(t) \mid A_C \leq t) = \begin{cases} 1 & 1 < t \leq 4.5 \\ \dfrac{e^{-3 \cdot (t-4.5)} - e^{-3 \cdot (t-1)}}{1 - e^{-3 \cdot (t-1)}} & t > 4.5 \end{cases}$$

We compute two values for the $P(I_C(t)|A_C \leq t)$: the exact value using $P(L_B(t)|A_C \leq t)$ and an approximation using $P(L_B(t))$ instead of $P(L_B(t)|A_C \leq t)$ in the right hand side of Eq(5). The two resulting functions are plotted in Figure 6. We observe that the two curves almost overlap for $t>4.5$. However, we notice a "spike" at time 0.5 in the approximating curve. In the vicinity of this time moment, there is a rather low probability that task *B* already started. This also implies that the processor load is rather low. Hence, the approximate analysis assigns a high probability to the event that the processor is free and no tasks of higher priority than the priority of task *C* are ready to run. It ignores the fact that no matter how unlikely the running of task *B* is, task *C* still has to wait for task *B* to complete. The importance of the spike artificially introduced by approximation is diminished

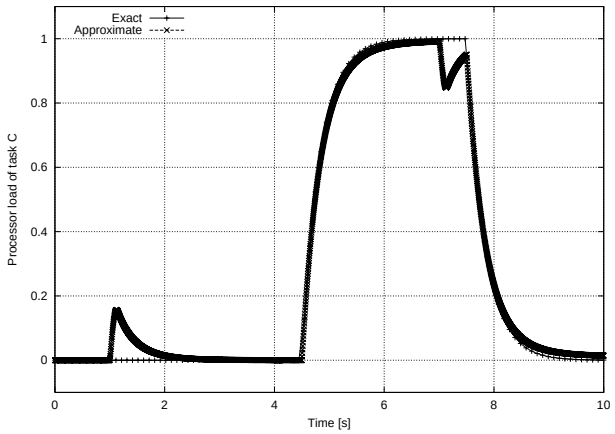

Figure 7: Approximation processor load

as task *B* "gains momentum". In order to illustrate how the approximation of $P(I_C(t)|A_C \le t)$ influences the computed value of the instantaneous processor load of task *C*, we plotted the real processor load and the one reported by the approximate analysis in Figure 7. We observe that they are close, especially towards the tail. This is important, as the deadline miss ratio is a function of the load at the time of the deadline.

In general, the dependence between arrival times of tasks and the event that a task is running at a certain time is the weaker the larger the number of tasks on a processor. Hence, we can approximate Eq(5) with

$$P(I_{\tau,j}(n) \,|\, A_{\tau,j} \le n) \;=\; 1 - \sum_{\sigma \in MT} P(L_\sigma(n)) \qquad (6)$$

Eq(6) is used also for approximating $P(I_{\tau,j}(n)|S_{\tau,j}<n)$. It follows that the probability that (τ, j) starts its execution at time moment t_n is approximated by

$$P(S_{\tau,j} = n) \;=\; (P(A_{t,j} \le n) - P(S_{\tau,j} < n)) \cdot \left(1 - \sum_{\sigma \in MT} P(L_\sigma(n))\right) \qquad (7)$$

Considering the previous observation that $P(A_{\tau,j} \le n)$ and $P(L_{\sigma,j}(n))$ can be determined if $P(S_{\tau,j}=k)$, $k<n$, are known (Eq(1) and Eq(2)), and considering Eq(7), it follows that the probability that (τ, j) starts its execution at time moment t_n can be determined if the probabilities that it started in the past are known. Hence, the algorithm for fast and approximate system analysis can be written as shown in Figure 8. The innermost loop is executed $|T| \cdot N$ times, where $|T|$ denotes the cardinality of the discrete set of time moments

```
for t_n in T do
  for p∈ set of processors do
    for τ∈MT in decreasing priority order do
      compute P(S_τ,j(n)) using Eq(7)
      update P(F_τ,j(u)) and P(L_τ,j(u)) for u≥n using Eq(2)
    done
  done
done
```

Figure 8: Approximate system analysis algorithm

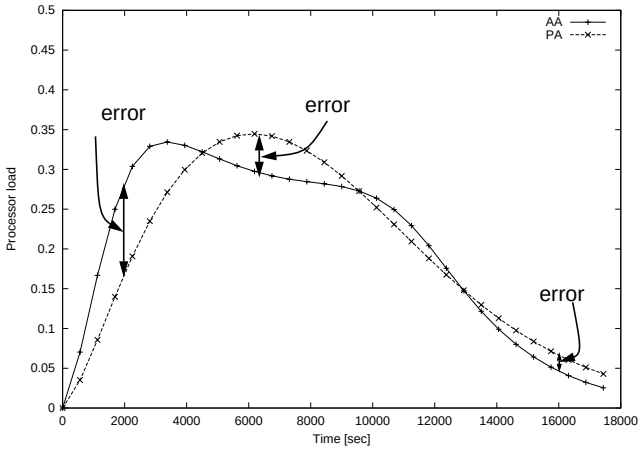

Figure 9: Approximation accuracy

T. The *update* function adjusts the probabilities of $F_{\tau,j}$ and $L_{\tau,j}$ at $|T|$ time moments in the worst case. Thus, the approximate system analysis algorithm is of complexity class $O(|T|^2 \cdot N)$.

In order to assess the accuracy of the proposed approximate analysis (AA), we compared the processor load curves obtained by AA with processor load curves obtained by our previously developed performance analysis (PA) [10]. The benchmark application consists of 20 processing tasks mapped on 2 processors and 3 communication tasks mapped on a bus connecting the two processors. Figure 9 gives a qualitative measure of the approximation. It depicts the two processor load curves for a task in the benchmark application. One of the curves was obtained with PA and the other with the approximate analysis. The figure also shows three error samples. A quantitative measure of the approximation is given in Table 1. We present only the extreme values for the average errors and standard deviations. Thus, row 1 in the table, corresponding to task 19, shows the largest obtained average error, while row 2, corresponding to task 13, shows the smallest obtained average error. Row 3, corresponding to task 5, shows the worst obtained standard deviation, while row 4, corresponding to task 9, shows the smallest obtained standard deviation. The average of standard deviations of errors over all tasks is around 0.065. Thus, we can say with 95% confidence that AA approximates the processor load curves with an error of ±0.13.

Table 1: Approximation accuracy

Task	Average error	Standard deviation of errors
19	0.056351194	0.040168796
13	0.001688039	0.102346107
5	0.029250265	0.178292338
9	0.016695770	0.008793487

5. Experimental results

The proposed Tabu Search based method for task mapping and priority assignment has been experimentally evaluated on randomly generated benchmarks and on a real-life example. This section presents the experimental setup and comments on the obtained results. The experiments were run on a desktop PC with one AMD Athlon processor clocked at 1533MHz.

The benchmark set consisted of 396 applications. The applications contained *t* tasks, clustered in *g* task graphs and mapped on *p* processors, where $t \in \{20, 22, \ldots, 40\}$, $g \in \{3, 4, 5\}$, and $p \in \{3, 4, \ldots, 8\}$. For each combination of *t*, *g*, and *p*, two applications were randomly generated. Three mapping and priority assignment methods were run on each application. All three implement a Tabu Search algorithm with the same tabu tenure, termination criterion and number of iterations after which a diversification phase occurs. In each iteration, the first method selects the next point in the design space while considering the *entire* neighborhood of design space points. Therefore, we denote it *ENS*, exhaustive neighborhood search. The second method considers only a restricted neighborhood of design space points when selecting the next design transformation. The restricted neighborhood is defined as explained in Section 3. We call the second method *RNS*, restricted neighborhood search. Both ENS and RNS use the same cost function, defined in Section 2 and calculated according to the approximate analysis described in Section 4. The third method considers only fixed task execution times, equal to the average task execution times. It uses an exhaustive neighborhood search and minimizes the value of the cost function $\sum lax_\tau$, where lax_τ is defined as follows

$$lax_\tau = \begin{cases} \infty & F_\tau > D_\tau \wedge \tau \text{ is critical} \\ F_\tau - D_\tau & otherwise \end{cases}$$

The third method is abbreviated *LO-AET*, laxity optimization based on average execution times. Once LO-AET has produced a solution, the cost function defined in Section 2 is calculated and reported for the produced mapping and priority assignment.

The first issue we look at is the quality of results obtained with RNS compared to those produced by ENS. The deviation of the cost function obtained from RNS relative to the cost function obtained by ENS is defined as $(cost_{RNS}-cost_{ENS})/cost_{ENS}$. Figure 10 depicts the histogram of the deviation over the 396 benchmark applications. The relative deviation of the cost function appears on the x-axis. The value on the y-axis corresponding to a value x on the x-axis indicates the percentage of the 396 benchmarks which have a cost function deviation equal to x. On average, RNS is only 1.65% worse than ENS. In 19% of the cases, the obtained deviation was between 0 and 0.1%. Note that RNS can obtain better results than ENS (negative deviation).

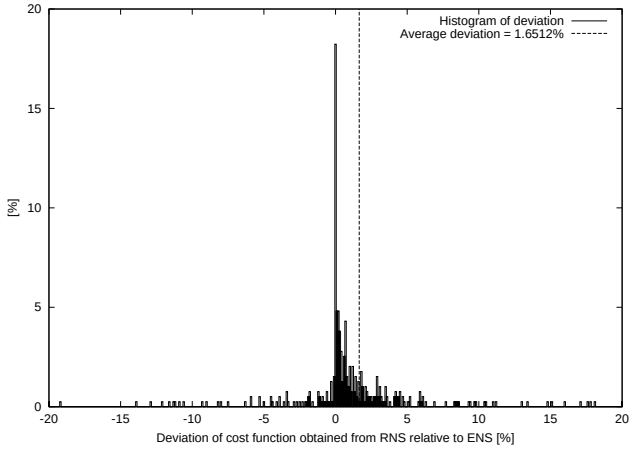

Figure 10: Cost obtained by RNS vs. ENS

This is due to the intrinsically heuristic nature of Tabu Search.

As a second issue, we compared the run times of RNS, ENS, and LO-AET. Figure 11 shows the average times needed to perform one iteration in the design space exploration algorithms RNS, ENS, and LO-AET. It can be seen that RNS runs on average 5.16÷5.6 times faster than ENS. This corresponds to the theoretical prediction, made at the end of Section 3, stating that the neighborhood size of RNS is *P* times smaller than the one of ENS, where *P* is the number of processors. In our benchmark suite, *P* is between 3 and 8 averaging to 5.5. We also observe that the analysis time is close to quadratic in the number of tasks, which again corresponds to the theoretical result that the size of the search neighborhood is quadratic in *N*, the number of tasks.

We finish the Tabu Search when 40*N* iterations have executed, where *N* is the number of tasks. In order to obtain the execution times of the three algorithms, one needs to multiply the numbers on the ordinate in Figure 11 with 40*N*. For example, for 40 tasks, RNS takes circa 26 minutes while ENS takes roughly 2h12'.

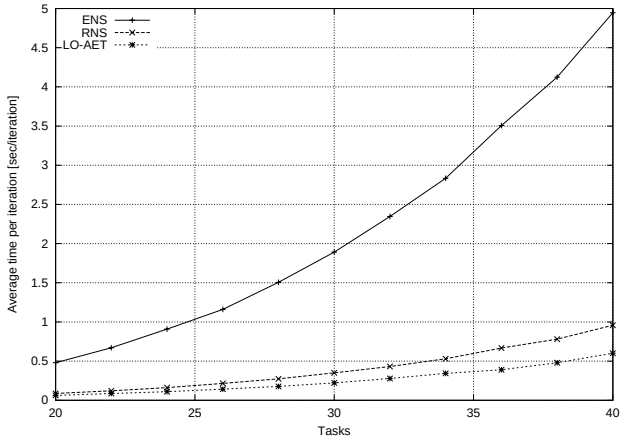

Figure 11: Run times of RNS vs. ENS

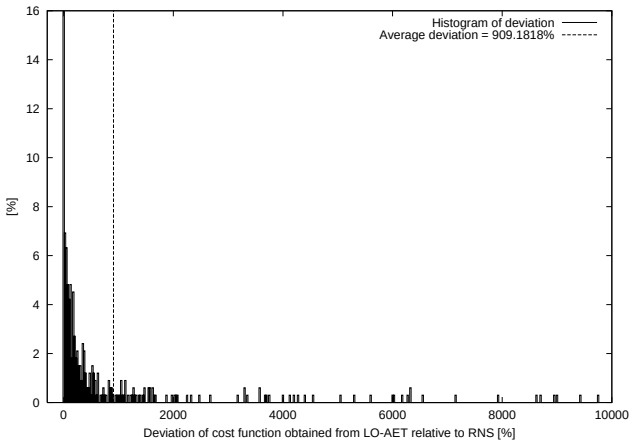

Figure 12: Cost obtained by LO-AET vs. RNS

The LO-AET method is marginally faster than RNS. However, as shown in Figure 12, the value of the cost function obtained by LO-AET is on average almost an order of magnitude worse (9.09 times) than the one obtained by RNS. This supports one of the main messages of this paper, namely that considering a fixed execution time model for optimization of systems is completely unsuitable if deadline miss ratios are to be improved. Although LO-AET is able to find a good implementation in terms of average execution times, it turns out that this implementation is very poor from the point of view of deadline miss ratios. What is needed is a heuristic like RNS, which is explicitly driven by deadline miss ratios during design space exploration.

Last, we considered an industrial-scale real-life example from the telecommunication area, namely a smart GSM cellular phone [11], containing voice encoder and decoder, an MP3 decoder, as well as a JPEG encoder and decoder. We focused on the voice decoding part, consisting of one task graph of 34 tasks mapped on two processors. They have been profiled to extract the execution characteristics of each task [11]. All tasks have the same period. Two tasks are critical. The restricted neighborhood search method improved the cost function from an initial value of 0.42 to 0.0255 (16 times), probing 729,662 potential task mappings and priority assignments in 1h31'.

6. Conclusions

In this paper, we addressed the problem of design optimization of soft real-time systems with stochastic task execution times under deadline miss ratio constraints. We have shown that methods considering fixed execution time models are unsuited for this problem. Therefore, we proposed a design space exploration heuristic guided by a fast approximate analysis. Experiments demonstrated the efficiency of the proposed approach.

References

[1] CAN Specification, Bosch, Robert Bosch GmbH, Postfach 50, D-7000 Stuttgart 1, Germany, 1991.

[2] G. Buttazzo, "Hard Real-Time Computing Systems", Kluwer 97

[3] M.R. Garey, D.S. Johnson, "Computers and Intractability, A Guide to the Theory of NP-Completeness", W.H. Freeman and Co., 2003

[4] F. Glover, "Tabu search—Part I", ORSA J. Comput., 1989

[5] X. S. Hu, T. Zhou, and E. H.-M. Sha, "Estimating probabilistic timing performance for real-time embedded systems", IEEE Trans. on VLSI Systems, vol. 9, no. 6, 2001.

[6] A. Kalavade, P. Moghe, "A tool for performance estimation of networked embedded end-systems", DAC, 1998

[7] L. Kleinrock, "Communication Nets: Stochastic Message Flow and Delay", McGraw-Hill, 1964

[8] Y.A. Li, J.K. Antonio, "Estimating the execution time distribution for a task graph in a heterogeneous computing system", HCW 97

[9] S. Manolache, P. Eles, Z. Peng, "Memory and Time-Efficient Schedulability Analysis of Task Graphs with Stochastic Execution Time", ECRTS 2001

[10] S. Manolache, P. Eles, Z. Peng, "Schedulability Analysis of Multiprocessor Real-Time Applications with Stochastic Task Execution Times", ICCAD 2002

[11] M. Schmitz, "Energy Minimisation Techniques for Distributed Embedded Systems", Ph.D. thesis 2003, Dept. of Comp. and Electrical Eng., Univ. of Southampton, UK

[12] T. Zhou, X. S. Hu, and E. H.-M. Sha, "A probabilistic performance metric for real-time system design", Intl. Wsh. on HW/SW Co-Design, 1999