A Genetic Algorithm based Approach to Maximizing Real-Time System Value under Resource Constraints

Li Wang, Zheng Li, Miao Song, Shangping Ren
Department of Computer Science
Illinois Institute of Technology
email:{lwang64, zli80, msong8, ren}@iit.edu

***Abstract*—For many embedded systems, different real-time applications are consolidated to the same hardware platform to meet the growing demand for diverse functionalities. Due to functionality differences, the values that different applications contribute to the system may not be the same. When system resources are limited and not all applications can be executed with guaranteed QoS, decisions have to be made as to which applications should be selected and how their tasks are deployed on available processors so that the system value is maximized and all the selected applications meet their deadlines. However, making the optimal decision for the application selection and task deployment (ASTD) problem is NP-hard. In this paper, we present a genetic algorithm (GA) based approach for the ASTD problem. We experimentally compare the performance of GA-based approach with the optimal approach chosen by enumerating all possible choices on a small scale, and with other heuristic approaches existed in the literature on a large scale. The results show that the system value obtained by the GA-based approach is close to the optimal value and can be twice as large as the value obtained by other heuristic approaches.**

***Index Terms*—Application Selection, Task Deployment, Genetic Algorithm, Real-time System, Heterogeneous Processors.**

I. INTRODUCTION

For many embedded systems, different applications are consolidated to the same hardware platform to meet the growing demand for diverse functionalities. Although the functionalities of different applications may vary, their composing tasks can be shared. To illustrate, consider a simplified version of the example [1] given below.

Example 1: Assume there are two applications in a parking garage surveillance system. One is to photograph all vehicles moving faster than 25mph in a specific section, the other is to collect magnetic field signatures whenever there is a moving object (human or vehicles) passing through the same section. To facilitate the first application, we need to perform the following tasks: (1) signal receiving, (2) signal pre-processing, (3) vehicle detection, (4) speed checking, and (3) camera shooting. The second application consists of the following tasks: (1) signal receiving, (2) signal pre-processing, (3) moving object detection, (4) magnetic field signatures saving. The two applications and their composing tasks are shown in Fig. 1.

From Fig. 1, we can see that although the functionalities of

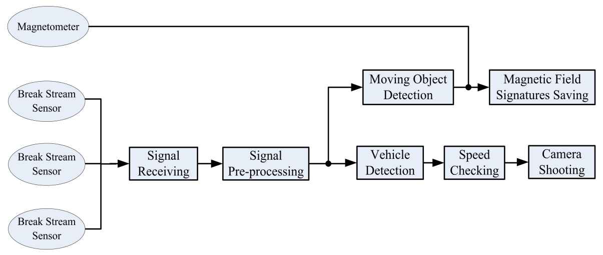


Fig. 1: Task sharing between two applications [1]

these two applications are different, *signal receiving* task and *signal pre-processing* task are shared by the two applications. In other words, the signal receiving task and signal pre-processing task are executed only once and the results are used by both applications. □

Due to functionality differences, the value that applications contribute to the system may not be the same [2], [3], [4]. For instance, in above example (Example 1), the application which photographs speeding vehicle contributes more value than application which collects magnetic field signatures of human or vehicles to system value. Hence, when not all applications can be executed with guaranteed QoS due to resource constraints, decisions have to be made as to which applications should be selected for execution so that the system value is maximized. Making application selection and its associated task deployment decisions on heterogeneous processors is a NP-hard problem [5]. Hence, a heuristic approach is needed when the size of the problem is not sufficiently small and exhaustive search becomes prohibitively expensive.

For illustrative purpose, consider an example given below.

Example 2: Assume a real-time system consists of two heterogeneous processors, i.e., π_1 and π_2. There are three applications α_1, α_2, and α_3 need to be executed on these two processors. The value that each application contributes to system is 40, 50, and 80, respectively.

Furthermore, assume α_1 has two tasks, τ_1 and τ_2; α_2 has task τ_2 and τ_3; and α_3 has task τ_1, τ_3, and τ_4. All these tasks are periodic and the periods for task τ_1, τ_2, τ_3 and τ_4 are 10, 15, 10, and 6, respectively. Their deadlines are at the end of their periods. The task execution time on different processors are given in Table I, where $+\infty$ indicates that the tasks can not be executed on the processor.

TABLE I: Task Execution Time

	τ_1	τ_2	τ_3	τ_4
π_1	8	9	6	3
π_2	4	10.5	$+\infty$	4.2

Given a task's execution time (e) and its period (p), the task utilization demand (e/p) on each processor is shown in Table II.

TABLE II: Task Utilization Demand

	τ_1	τ_2	τ_3	τ_4
π_1	0.8	0.6	0.6	0.5
π_2	0.4	0.7	$+\infty$	0.7

If all three applications are chosen, all tasks have to be deployed on the given two processors. Based on Table I, task τ_3 can only be executed on the π_1. The other three tasks have options to be either deployed on π_1 or π_2. Clearly, no matter which processor task τ_1, τ_2, and τ_4 are assigned to, the total utilization demand on at least one of π_1 and π_2 exceeds its capacity of 1 which indicates that some tasks will miss their deadlines. Therefore, in order to guarantee all deadlines are met, only a subset of the competing applications can be selected. It is not difficult to see that the number of selection options is combinatorial to the number of applications, tasks, and processors. □

Genetic Algorithm (GA) is a bio-inspired heuristic search algorithm which mimics the process of natural evolution [6], [7]. During the course of evolution, the populations (candidate solutions) with high fitness are given more chances to survive and reproduce, and the populations with low fitness are removed. Therefore, the quality of the populations improves after every evolution. In this paper, we develop a GA-based approach to solving the application selection and task deployment problem, i.e., the ASTD problem.

The rest of this paper is organized as follows. We discuss related work in Section II. In Section III, we define system model and formulate the problem. A brief introduction of genetic algorithm is given in Section IV. In Section V, we present a GA-based approach for the application selection and task deployment problem. The experimental results are discussed in Section VI. Finally, we conclude our work and point out the future work in Section VII.

II. Related Work

Research regarding task deployment has been intensively studied from different perspectives. For instance, in [8], [9], [10], [11], [12], [13], [14], [15], [16], researchers have focused on how to deploy real-time tasks on a set of homogeneous processors; while in [17], Anderson et al. take a step further to partition processors into two categories, i.e, type-1 and type-2 processors, and study the task deployment problem on 'semi-hetrogeneous' processors.

Baruah in [5] proposed a polynomial time algorithm to decide whether a set of tasks can be deployed to a collection of heterogeneous processors. The algorithm first transforms the task deployment problem into an Integer Linear Programming (ILP) problem, and then applies the Linear Programming (LP) relaxation technique to solve this problem. The algorithm works well only when the number of processors is small as the time complexity of the linear programming relaxation algorithm is $O(m^m)$ where m is the number of processors. In [18], Gopalakrishnan et al. also addressed the task deployment problem on heterogeneous multiprocessor systems. In their work, they proposed a utility balancing (or UB) algorithm to deploy tasks in a way that the maximum utilization of all processors is minimized. Armstrong et al. presented a Minimum Execution Time (MET) algorithm [19] to minimize the makespan when scheduling a collection of tasks among a set of heterogeneous processors. Unlike [18], MET algorithm assigns a task to the processor on which the task's execution time is minimized.

The major differences between the work mentioned above and ours are the objective differences and model differences. The objective of the work discussed above is to meet *all* task deadlines; while ours is to maximize accrued system value with the constraint that all *selected* tasks must meet their deadlines; and the model of the work mentioned above is based on a set of tasks; while our model is based on a set of applications which may share tasks.

It is also worth pointing out that although the problem we are to address in the paper is similar to the multidimensional knapsack problem (MKP) [20], [21] on surface, the essence of these two problems are different. In MKP, the amount of resources required by different items on different knapsacks is fixed and known *a priori*. While for the ASTD problem, the utilization demand of different applications is not known until the decision is made as to which processors tasks are assigned to.

In [22], a Utility Accrual (or UA) real-time scheduling algorithm is proposed to schedule a set of tasks. Each task is associated with a time-utility function (TUF) and the scheduling goal is to maximize total accrual utility. Our work has similar concept in that we also aim to maximize system value. The differences lie in that the UA model assumes task values are independent; while in our model, values are applied to applications and as applications may share tasks, hence, we cannot use simple value-monotonic approach to choose applications.

Genetic Algorithm (GA) is a search heuristic which simulates the process of natural evolution [6], [7]. It has been widely used to solve problems in various fields, such as reassembly line balancing problem [23], [24], [25] and timetable problems [26], etc. In [27], [28], [29], [30], genetic algorithm is used to solve the task allocation and scheduling problem on both homogeneous and heterogeneous multiprocessor systems with the goal of minimizing makespan when scheduling a set of tasks. In their application of GA-based approach, the task deadlines are not considered; while we need to meet all selected tasks' deadlines when applying GA-based approach to maximize accrued system value.

III. Problem Formulation

We assume that a real-time system consists of a set of heterogeneous processors which can support a set of applications. Each application consists of a set of periodic tasks, and tasks may be shared among applications [1], [31]. For each task, its deadline is at the end of its period and it can only be deployed to at most one processor. For each processor, preemptive rate monotonic scheduling policy [32] is used when there are multiple tasks.

Before we formally define the ASTD problem, we first introduce the following notations and definitions that will be used throughout the paper.

Processor Set Π: $\Pi = \{\pi_1, \pi_2, \cdots, \pi_k\}$, where π_i represents processor i and k is the total number of processors in the system.

Application Set $\mathcal{A}$: $\mathcal{A} = \{\alpha_1, \alpha_2, \cdots, \alpha_m\}$, where α_i represents application i, and m is the total number of applications which compete for execution.

Application Value Vector $\overrightarrow{V}_m$: $\overrightarrow{V}_m = [v_1, v_2, \cdots, v_m]$, where v_i represents the value application α_i may contribute to the system. if all of its tasks are completed before their deadlines.

Task Set Γ: $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$, where τ_i represents real-time task i, and n is the total number of tasks the application set α has.

Real-time Task τ: $\tau = (e, p)$, where e and p represent the task execution time and period, respectively. We assume all tasks are released at the beginning of each period.

Application-Task Matrix $\mathbf{A}_{m\times n}$: $\mathbf{A}_{m\times n} = (b_{i,j})_{m\times n}$, where $b_{i,j} \in \{0, 1\}$ and $b_{i,j} = 1$ indicates that application α_i contains task τ_j, and task τ_j does not belong to application α_i if $b_{i,j} = 0$.

Task-Execution-Time Matrix $\mathbf{E}_{n\times k}$: $\mathbf{E}_{n\times k} = (e_{i,j})_{n\times k}$, where $e_{i,j} \in \Re^+ \cup \{+\infty\}$ represents the execution time when processor π_j only executes task τ_i. We use $e_{i,j} = +\infty$ to indicate that processor π_j cannot perform task τ_i.

Application Selection Vector $\overrightarrow{A}_m$: $\overrightarrow{A}_m = [a_1, a_2, \cdots, a_m]$, where $a_i \in \{0, 1\}$. Application α_i is selected if $a_i = 1$, and $a_i = 0$, otherwise.

Task-Deployment Matrix $\mathbf{D}_{n\times k}$: $\mathbf{D}_{n\times k} = (d_{i,j})_{n\times k}$, where $d_{i,j} \in \{0, 1\}$ and $d_{i,j} = 1$ indicates that task τ_i is deployed on processor π_j, and $d_{i,j} = 0$ indicates the opposite.

Example 3: Under these notations, Example 2 given in Section I can be represented as:

- processor set $\Pi = \{\pi_1, \pi_2\}$;
- Application set $\mathcal{A} = \{\alpha_1, \alpha_2, \alpha_3\}$;
- Application value vector $\overrightarrow{V}_3 = [40, 50, 80]$;
- Task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$;
- Task periods p = 10, 15, 10, and 6 for task τ_1, τ_2, τ_3, and τ_4, respectively;
- Task-Execution-Time matrix $\mathbf{E}_{4\times 2}$

$$\mathbf{E}_{4\times 2} = \begin{pmatrix} 8 & 4 \\ 9 & 10.5 \\ 6 & +\infty \\ 3 & 4.2 \end{pmatrix}$$

- Application-Task matrix $\mathbf{A}_{3\times 4}$

$$\mathbf{A}_{3\times 4} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

□

For a given task-deployment matrix $\mathbf{D}_{n\times k}$, as each task can only be deployed to at most one processor, and tasks that belong to selected applications must be deployed, hence we have the following constraints.

$$\sum_{j=1}^{k} d_{i,j} \le 1 \qquad i = 1, 2, \ldots, n \tag{1}$$

$$a_i \times b_{i,j} \le \sum_{l=1}^{k} d_{j,l} \qquad j = 1, 2, \ldots, n; \;\; i = 1, 2, \ldots, m \tag{2}$$

where (1) ensures that each task can only be deployed to at most one processor (*uni-deployment constraint*), and constraint (2) guarantees that each task of the selected application is deployed to a processor (*completeness constraint*).

Furthermore, as preemptive rate-monotonic scheduling policy is used on all processors in the system. Therefore, in order to guarantee all deployed tasks meeting their deadlines, we have to ensure that the utilization demand on each processor is within the bound given by Liu et al. [32], i.e., *processor utilization bound constraint* given by (3)

$$\sum_{i=1}^{\beta} \frac{e_i}{p_i} \le \beta(2^{1/\beta} - 1) \tag{3}$$

where e_i and p_i are task τ_i's execution time and period, respectively, and β is the total number of tasks deployed on a processor.

Given the above assumptions, the ASTD problem can be mathematically defined below.

Problem 1 (The ASTD Problem): Given k heterogeneous processors, m applications with the value vector ($\overrightarrow{V}_m$), an Application-Task matrix ($\mathbf{A}_{m\times n} = (b_{i,j})_{m\times n}$), and a Task-Execution-time matrix ($\mathbf{E}_{n\times k} = (e_{i,j})_{n\times k}$), decide the application selection vector $\overrightarrow{A}_m$ and the Task-Deployment matrix $\mathbf{D}_{n\times k} = (d_{i,j})_{n\times k}$, with

Objective:

$$\text{maximize} \qquad \nu = \overrightarrow{A}_m \times (\overrightarrow{V}_m)^T \tag{4}$$

Subject to:

$$\sum_{j=1}^{k} d_{i,j} \le 1 \quad i = 1, 2, \ldots, n \tag{5}$$

$$a_i \times b_{i,j} \le \sum_{l=1}^{k} d_{j,l} \quad j = 1, 2, \ldots, n; \; i = 1, 2, \ldots, m \tag{6}$$

$$\sum_{i=1}^{n} \frac{e_{i,j}}{p_i} \times d_{i,j} \le \beta_j (2^{1/\beta_j} - 1) \quad j = 1, 2, \ldots, k. \tag{7}$$

where $a_i, d_{i,j}, b_{i,j} \in \{0, 1\}$. β_j is number of tasks deployed on processor j, i.e., $\beta_j = \sum_{i=1}^{n} d_{i,j}$.

□

Clearly, optimization problem defined above falls into the category of nonlinear integer programming (NIP) problem which has been proved to be NP-hard [33]. Next section, we present a GA-based heuristic approach to find a near optimal solution to the problem.

IV. Brief Introduction to Genetic Algorithm

Genetic Algorithm (GA) is a search heuristic which simulates the process of natural evolution [6], [7]. It starts with an initial population of chromosomes which represent candidate solutions to a problem. Chromosomes, or solutions, are evaluated with respect to their fitness. For example, in our work, the fitness of a solution is defined as the total accrued value of supported applications.

After the initial population of chromosomes are generated and evaluated, the next step of GA is to select the parent chromosomes for reproduction. Chromosomes with higher fitness value are often given more opportunities to be selected as parents. When the parent chromosomes are selected, crossover operator is applied to the parent chromosomes to generate new offsprings. This is done by exchanging certain parts of the parent chromosomes. Mutation operator transforms a chromosome into another by altering some pieces of genes, it often takes place after crossover operation.

After crossover and mutation, we evaluate the fitness of the new offsprings. The new offspring will either be used to replace an existing chromosome if it has higher fitness value than the existing chromosomes, or be simply removed. Therefore, the size of the populations remains unchanged. The evaluation-selection-reproduction procedure is repeated until predefined stopping conditions are met. The steps of the GA are shown in Algorithm 1.

Algorithm 1 Genetic Algorithm

```
1:  Generate an initial population of chromosomes
2:  Evaluate the fitness of the chromosomes
3:  while stopping conditions not met do
4:      Select parent chromosomes from the population.
5:      Generate children chromosomes by applying crossover
        operator.
6:      Mutate the children chromosomes.
7:      Evaluate the fitness of children chromosomes.
8:      Replace the exiting chromosomes with the children
        chromosomes if the offsprings have larger fitness values.
9:  end while
10: return the chromosome with highest fitness value
```

V. A GA-based Approach to Maximizing Accrued System Value

In this section, we give the detailed design of a GA-based approach for solving the problem defined in Section III.

A. Chromosome Representation

In order to maximize system value from supported applications, we need to select applications and deploy associated tasks to processors. Therefore, we use a pair of vectors to represent a candidate solution to the problem, i.e, $S = (\overrightarrow{T}_n, \overrightarrow{N}_n)$, where the 0-1 binary vector $\overrightarrow{T}_n = [t_1, t_2, \cdots, t_n]$ represents the selection of tasks, and vector $\overrightarrow{N}_n = [d_1, d_2, \cdots, d_n]$ denotes the processor index where the tasks are deployed.

To better illustrate the chromosome representation, consider the following example.

Example 4: Assume a real-time system consists of three processors $\Pi = \{\pi_1, \pi_2, \pi_3\}$. There are three applications $\mathcal{A} = \{\alpha_1, \alpha_2, \alpha_3\}$, and the value that each application contributes to the system is 20, 50, and 60, respectively.

Each application consists of multiple periodic tasks, and the task set of the these three applications is $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_7\}$. The relationship between applications and their composing tasks is specified in Application-Task matrix $\mathbf{A}_{3\times7}$.

$$\mathbf{A}_{3\times7} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Fig. 2 gives a candidate solution to the problem. The solution representation provides the following information: all tasks except τ_6 are deployed. In addition, task τ_1, τ_3, and τ_5 are deployed to processor π_1, task τ_2 and τ_4 to processor π_3, and task τ_7 to processor π_2.

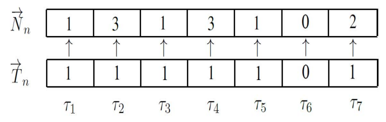


Fig. 2: Representation of candidate solution

Based on the deployed tasks and Application-Task matrix $\mathbf{A}_{3\times7}$, we know application α_1 and α_2 are selected. □

B. Initial Population Generation

In order to create the initial population, we generate the candidate solutions randomly, i.e., the value of t_i in $\overrightarrow{T}_n$ is uniformly selected from 0 and 1, and the value of d_i in $\overrightarrow{N}_n$ is uniformly selected from 1 to k, where k refers to the total number of processors. Clearly, such randomly generated solution may not meet all the constraints and hence may not be valid. To illustrate this, consider the following example.

Example 5: Assume the number of processors, applications, tasks, and the relationship between applications and tasks are the same as given in Example 4. The execution time of each task τ_i $(1 \le i \le 7)$ on processor π_j $(1 \le j \le 3)$ is shown in Task-Execution-Time matrix $\mathbf{E}_{7\times3}$.

$$\mathbf{E}_{7\times3} = \begin{pmatrix} 2 & 5 & 4 \\ 3 & 2 & 4 \\ 1 & +\infty & +\infty \\ 1.8 & +\infty & 1.8 \\ +\infty & 3 & 4 \\ 8 & 3 & +\infty \\ +\infty & 10.5 & +\infty \end{pmatrix}$$

Clearly, a randomly generated candidate solution as shown in Fig. 2 is not valid. It is because based on the solution, task τ_5 is deployed to processor π_1 on which the task execution time is $+\infty$, which indicates that processor π_1 cannot execute task τ_5. $\square$

In the literature, there are many approaches which can be applied to deal with the infeasible solutions, i.e., separate the evaluation of fitness and infeasibility [34], apply a penalty function to penalise the fitness of any infeasible solution [35], or design a heuristic repair operator to transform the infeasible solution into feasible solution [36]. In this paper, we design a heuristic repair operator to deal with infeasible solutions. The detailed description of our heuristics for the repair operator in given in Section V-F.

C. Fitness Evaluation

As our goal is to maximize system value from supported applications, we hence define the fitness of a solution as the total accrued value provided by the solution. We use vector $\overrightarrow{A}^i_m$: $\overrightarrow{A}^i_m = [a_1, \cdots, a_j, \cdots, a_m]$ to denote whether the applications are supported or not with solution S_i, where $a_j \in \{0, 1\}$. This can be achieved by checking whether each task that belongs to application α_i $(1 \le i \le m)$ exists in the vector $\overrightarrow{T}_n$ of solution S_i. Algorithm 2 below gives the details.

A brief explanation of Algorithm 2 is as follows: In Line 1, we initialize the supported application vector $\overrightarrow{A}^i_m$, and set the value of each element to 0. Line 4 to Line 8 check whether all the tasks of each application α_i are selected.

For a given application selection vector $\overrightarrow{A}^i_m = [a_1, a_2, \cdots, a_m]$ and value vector $\overrightarrow{V}_m = [v_1, v_2, \cdots, v_m]$, the fitness value of solution S_i can be calculated as

$$\nu(S_i) = \overrightarrow{A}^i_m \times (\overrightarrow{V}_m)^T \tag{8}$$

Example 6 (Example 5 Revisited): Assume the execution time of task $\tau_i \in \Gamma$ is the same as given in Example 5, and the period p_i of task $\tau_i \in \Gamma$ is given in vector $\overrightarrow{P}_7 = [p_1, p_2, \cdots, p_7] = [10, 20, 4, 6, 10, 15, 15]$.

Based on the Task-Execution-Time matrix $\mathbf{E}_{7\times3}$ and the period vector $\overrightarrow{P}_7$, the utilization demand $u_{i,j} = e_{i,j}/p_i$ of task τ_i on processor π_j is shown in the Task-Utilization matrix $\mathbf{U}_{7\times3} = (u_{i,j})_{7\times3}$ as below.

Algorithm 2 OBTAIN SUPPORTED APPLICATIONS

```
Input: A candidate solution $S_i = (\overrightarrow{T}_n, \overrightarrow{N}_n)$; Application-Task matrix $\mathbf{A}_{m\times n} = (b_{i,j})_{m\times n}$.
Output: Application Selection Vector $\overrightarrow{A}^i_m$ given by candidate solution $S_i$.
1: Initialize $\overrightarrow{A}^i_m = [a_1, a_2, \cdots, a_m]$, and set the value of all the elements to 0.
2: for $i \leftarrow 1$ to $m$ do
3:     $all\_selected \leftarrow true$
4:     for $j \leftarrow 1$ to $n$ do
5:         if $b_{i,j} > t_j$ then
6:             $all\_selected \leftarrow false$
7:         end if
8:     end for
9:     if $all\_selected$ is $true$ then
10:        $a_i \leftarrow 1$
11:    end if
12: end for
```

$$\mathbf{U}_{7\times3} = \begin{pmatrix} 0.2 & 0.5 & 0.4 \\ 0.15 & 0.1 & 0.2 \\ 0.25 & +\infty & +\infty \\ 0.3 & +\infty & 0.3 \\ +\infty & 0.3 & 0.4 \\ 0.6 & 0.2 & +\infty \\ +\infty & 0.7 & +\infty \end{pmatrix}$$

Consider a candidate solution shown in Fig. 3, according to this solution, the utilization demands on processor π_1, π_2, and π_3 are 0.8, 0.7, and 0.5, respectively. Clearly, all processors satisfy the utilization constraint (3). In addition, based on the Application-Task matrix $\mathbf{A}_{3\times7}$ and vector T_n shown in Fig. 3, we know application α_1 and α_2 are not supported because task τ_3 is not selected in the candidate solution. For application α_3, as all its tasks, i.e., task τ_6 and τ_7, are deployed, and hence, α_3 is selected. Therefore, the supported application vector is $\overrightarrow{A}_3 = [0, 0, 1]$.

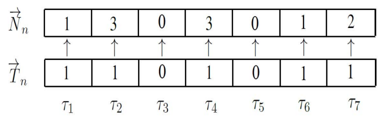


Fig. 3: A candidate solution

According to the supported application vector $\overrightarrow{A}_3 = [0, 0, 1]$ and application value vector $\overrightarrow{V}_3 = [v_1, v_2, v_3] = [20, 50, 60]$, the fitness value of the candidate solution is $\nu(S) = \overrightarrow{A}_3 \times (\overrightarrow{V}_3)^T = [0, 0, 1] \times [20, 50, 60]^T = 60$. $\square$

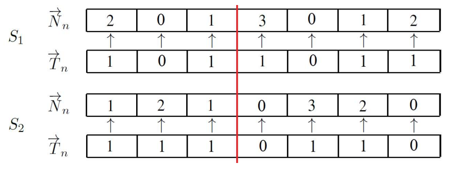
D. Parent Selection

Traditionally, parent solution is either decided by a rank-based roulette wheel selection scheme [37] or a value-based roulette wheel selection scheme [7]. In the rank-based roulette wheel selection scheme, the probability of a solution being selected is decided by its rank. While, in the value-based roulette wheel selection scheme, the angle of the sector is proportional to the fitness of the solution. In our implementation, we choose neither of the approaches but follow the way given in [23], [24] to randomly select the parents from the populations without any preference. The reason for randomly selecting the parents is to decrease the similarity of candidate solutions in the population and prevent solutions from converging too early and resulting in suboptimal [26]. Therefore, the probability that optimal solution to be found increases.

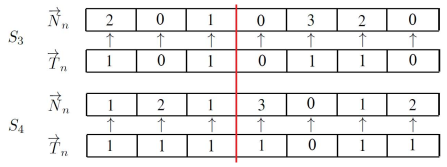
E. Crossover and Mutation

Different crossover operators are available to guide the generation of the offsprings, such as one-point crossover [7], two-point crossover [7], and uniform crossover [34], etc. Starkweather et al. [38] also suggest that it is better to have two parent solutions in the crossover process. In our implementation, we adopt single point crossover approach on two parent solutions. In particular, we randomly select a mating position for a pair of parent solutions, and exchange their right parts to produce two new offspring solutions.

Fig. 4 gives an example regarding how crossover operator works: we randomly select a mating position for parent solutions S_1 and S_2. Assume the random selection is 3, new solutions S_3 and S_4 are produced by exchanging the right part of the parent solutions S_1 and S_2. For child solution S_3, the left part comes from parent solution S_1, and the right part is from parent solution S_2. Child solution S_4 is obtained similarly.



(a) Before exchange



(b) After exchange

Fig. 4: Crossover operator

As our solution consists of two parts: the task selection and the task deployment, therefore, the mutation operator will be applied on both task's selection vector and task deployment vector of each chosen solution. In particular, to mutate the task selection vector, we first randomly select a position, i.e., i $(1 \leq i \leq n)$, then the operator flips the value of t_i in vector $\overrightarrow{T}_n$. When mutating the task deployment vector, we also randomly select a position j $(1 \leq j \leq n)$ and randomly choose a value between 1 to k, i.e., y, then the operator sets the value of d_j to y, i.e., the the jth task is deployed to processor π_y.

For each new offspring solution, we evaluate its fitness value and use it to replace a solution in the population which has the least fitness. However, if the new offspring solution has the least fitness value, it is removed. Hence, the solution population remains the same among generations.

F. Repair Operator

Based on the discussion in Section V-B, the initial randomly generated solutions may not be valid. Furthermore, even if the initial solutions are valid, after performing the crossover and mutation operations, the offsprings may not be valid either. To guarantee the validity of solution population, we define a heuristic repair operator.

The repair operator consists of steps: task transfer and task removal. In the task transfer step, we move tasks from the processor whose utilization demand exceeds its utilization bound to an available processor. By processor π_i being available to task τ_j, we mean that after accepting task τ_j, the total utilization demand on processor π_i is still below its bound.

Algorithm 3 shows the detailed procedure of task transfer.

Algorithm 3 TASK TRANSFER

```
Input: The number of processors $k$, invalid solution $S_i$, Task-Utilization matrix $\mathbf{U}_{n\times k}$.
Output: Solution $S_i'$ obtained after task transfer operation is finished.
1: for $i \leftarrow 1$ to $k$ do
2:   if the utilization demand on processor $\pi_i$ is above the bound then
3:     for each $\tau_j$ deployed on $\pi_i$ do
4:       if there exists an available processor $\pi_{min}$ on which the utilization demand of $\tau_j$ is minimized then
5:         move task $\tau_j$ from $\pi_i$ to $\pi_{min}$.
6:       end if
7:       if the utilization demand on $\pi_i$ is below the bound then
8:         break
9:       end if
10:    end for
11:  end if
12: end for
```

A brief of explanation of Algorithm 3 is as follows: we first check whether processor π_i $(1 \leq i \leq k)$ violates the utilization constraint (3) (Line 2). If yes, for each task τ_j that is assigned to π_i, we move it to an available processor π_{min} on which the utilization demand of τ_i is minimized. If there is no processor which can accept τ_j, we continue with the next task in processor π_i (from Line 4 to Line 6). If the utilization demand on π_i is below the bound, we stop the task transfer operation for π_i and check the next processor (from Line 7 and Line 9).

It is worth pointing out that after performing the task transfer operation, the accrued system values of the feasible solutions increase. In other words, the task transfer operation accelerates the convergence of the GA to the optimal solution.

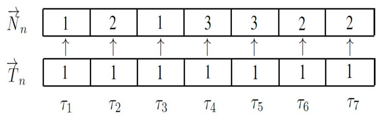
After task transfer step, if there still exist processors whose utilization exceed their utilization bounds, we perform the second step, i.e., remove tasks from processors to meet the constraints. The heuristic is to remove task that contributes the least value to system peformance, but consumes most computation resources, i.e., has the least value-resource ratio ρ_j

$$\rho_j = \frac{\sum_{i=0}^{m} b_{i,j} \times v_i}{avg(u_j)} \tag{9}$$

where $\sum_{i=0}^{m} b_{i,j} \times v_i$ is the total value of the applications that consist of task τ_j, and $ave(u_j) = (\sum_{e_{i,j} \neq +\infty} \frac{e_{i,j}}{p_j})/\eta$ is the average resource task τ_j needs on a processor, and η is the total number of processors on which the execution time of task τ_j is not equal to $+\infty$.

To illustrate how the repair operator works, we revisit Example 6 given below.

Example 7 (Example 6 Revisited): Consider a candidate solution given in Fig. 5. From Fig. 5, we know that task τ_1 and τ_3 are deployed to processor π_1, task τ_2, τ_6, and τ_7 to processor π_2, and task τ_4 and τ_5 to processor π_3. According to the Task-Utilization matrix $\mathbf{U}_{7\times 3}$ shown in Example 6, the utilization of each processor is 0.45, 1.0, and 0.7, and the utilization demand bound for processor π_1, π_2, and π_3 is 0.8284, 0.7798, and 0.8284, respectively. Clearly, processor π_2 violates the utilization constraint (3).



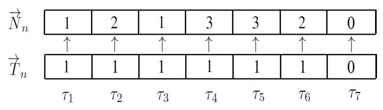
$\vec{N}_n$
1 2 1 3 3 2 2
$\vec{T}_n$
1 1 1 1 1 1 1
τ_1 τ_2 τ_3 $\tu_4$ τ_5 τ_6 τ_7

Fig. 5: A candidate solution before repair operation

In order to transform the invalid solution, we apply the first step of the repair operator. As processor π_1 is the only available processor to task τ_2, we move task τ_2 from processor π_2 to processor π_1. After moving task τ_2, the utilization of processor π_1, π_2, and π_3 becomes 0.6, 0.9, and 0.7, respectively. Right now, neither processor π_1 nor π_3 can accept tasks from processor π_2.

As processor π_2 still violates the utilization constraint (3), which is 0.8284, hence we take the second step of the repair process, i.e., remove tasks from processor to guarantee the satisfaction of the utilization bound constraints.

As both task τ_6 and τ_7 only belong to application $\mathcal{A}_3$. Hence, based on the Task-Utilization matrix $\mathbf{U}_{7\times 3}$, the value-resource ratio for task τ_6 and τ_7 are $\rho_6 = \frac{60}{(0.2+0.6)/2} = 150$ and $\rho_7 = \frac{60}{0.7} = 85.71$. As task τ_7 has the least value-resource ratio, it is removed from processor π_2, and the utilization demand on processor π_2 becomes 0.2, which is below the utilization bound. Fig. 6 shows the solution after the repair operation. □



$\vec{N}_n$
1 2 1 3 3 2 0
$\vec{T}_n$
1 1 1 1 1 1 0
τ_1 τ_2 τ_3 τ_4 τ_5 τ_6 τ_7

Fig. 6: A candidate solution after repair operation

G. Termination

In our implementation, the GA stops after performing a pre-defined $\mathcal{I}$ number of iterations. For each iteration, the crossover operator is performed $\mathcal{C}$ times, and on average the mutation operator on each offspring solution is performed $\mathcal{U}$ times. We will discuss the selection of these parameters in Section VI.

VI. Experimental Results

In this section, we describe two sets of experiments. The purpose of the first set of experiments is to investigate the performance of our GA-based approach by comparing it with the optimal value obtained through exhaustive searching of all possible solutions. Due to the exponentiality of the solution space, this set of experiments has to be on small scale, i.e., with few processors, tasks, and applications. The second set of experiments is to investigate how the GA-based approach performs when the scale of the problem becomes large. We compare it with two other commonly used heuristic approaches, i.e., UB [18] based approach, and Minimum Execution Time (MET) [19] based approach. The UB based approach selects the application with the largest value, and deploys tasks in such a way that the maximum utilization among all processors is minimized; while the MET based approach selects applications with the largest value and deploys tasks to the processor on which the task's execution time is minimized.

A. Experiment Settings

When the system size is decided, i.e., the number of processors, tasks, and applications is fixed, based on the utilization constraint (3), the feasibility that a set of tasks can be scheduled on the processors is affected by the total utilization demand of the task set. Therefore, in order to compare the performance of different approaches, we run the test cases with different utilization demand of the task set. The accrued value is normalized to the total application value.

In our implementation, we use UUnifast algorithm [39] to generate the utilization demand for each task. To be more specific, assume the total utilization demand of n tasks is U, the UUnifast algorithm uniformly distributes the U to task τ_i with $0 < ud_i < U$ $(1 \leq i \leq n)$. The algorithm guarantees $U = \sum_{i=1}^{n} ud_i$.

The step above only considers the case in which there is only one processor. When there are k processors, we first obtain the utilization demand ud_i for task $\tau_i (1 \leq i \leq n)$. We then apply the UUnifast algorithm again to choose the utilization demand of task τ_i on processor π_j, i.e., $u_{i,j}$. Because of processor heterogeneity, some tasks may not be able to be executed on certain processors. To reflect this, for each task τ_i, we first generate a random number k_0 in the range of $[0, \lceil k \times \varphi \rceil]$, where k is the number of processors in the system, and φ is a floating point value which is used to adjust the maximum number of unfeasible processors for the tasks, and $0 \le \varphi \le 1$. In our experiment, we set $\varphi = 0.3$. When k_0 is determined, we randomly select k_0 processors N_∞ and set their utilization for task τ_i to be $+\infty$. For the rest $(k - k_0)$ processors, we apply the UUnifast algorithm to decide the utilization demand of task τ_i on processor j where $N_j \notin N_\infty$ within the range of $(0, ud_i \times (k - k_0))$. In this way, the average utilization demand of task τ_i on a single processor remains the same, i.e., ud_i.

Algorithm 4 gives the detail for Task-Utilization matrix $\mathbf{U}_{n\times k}$ generation.

Algorithm 4 GENERATE TASK-UTILIZATION MATRIX

```
Input: Total utilization demand of task set $U$, number of tasks $n$, number of processor $k$, a floating point value $\varphi$.
Output: Task-Utilization matrix $\mathbf{U}_{n\times k} = (u_{i,j})_{n\times k}$.
1: Create vector $\overrightarrow{UD} = [ud_1, ud_2, \cdots, ud_n]$.
2: $\overrightarrow{UD} \leftarrow UUnifast(TU, n)$.
3: for $i \leftarrow 1$ to $n$ do
4:     randomly select a number $k_0 \in [0, \lceil k \times \varphi \rceil]$
5:     randomly select $k_0$ processors $j_1, \cdots, j_{k_0}$
6:     $u(i, j_l) \leftarrow \infty, l = 1, \cdots, k_0$
7:     $u(i, j_p) \leftarrow UUnifast(ud_i \times (k - k_0), k - k_0)$, $p = 1, \cdots, k$ and $j_p \notin \{j_1, \cdots, j_{k_0}\}$.
8: end for
```

For both sets of experiments, the value of applications is uniformly distributed from 1 to 100, the number of tasks that each application has is uniformly chosen from 1 to n (i.e., the number of tasks), and the tasks for each application are also randomly selected.

B. Deciding the Parameters for the GA-based Approach

In order to apply the GA-based approach, we have to first decide the following parameters: initial population size ($\mathcal{P}$), number of iterations ($\mathcal{I}$), number of crossover operations performed in each iteration ($\mathcal{C}$), and average number of mutations performed on each offspring solution ($\mathcal{U}$). As pointed out in [23], these values may be different under different scenarios.

For small scale of the ASTD problem, i.e., $k = 3$, $n = 12$, and $m = 20$, we set the parameters as $\mathcal{P} = 150$, $\mathcal{I} = 150$, $\mathcal{C} = 150$, and $\mathcal{U} = 1.5$; while for larger size of the problem, such as $k = 10$, $n = 80$, and $m = 120$, the parameter values are set as $\mathcal{P} = 600$, $\mathcal{I} = 550$, $\mathcal{C} = 400$, and $\mathcal{U} = 0.6$. The parameter values are obtained through experiments. The detailed procedure for obtaining the values is given in Appendix section.

C. Comparison

We first compare the performance of the GA-based approach with the exhaustive search, UB and MET based approaches when the system size is small, i.e., $k = 3$, $n = 12$, and $m = 20$. The performance of different approaches is evaluated based on the accrued value normalized to total application value and the results are shown in Fig. 7. As we can see from the figure that the system's accrued value obtained by the GA-based approach is close to the optimal result and is larger than the value obtained by the UB and MET based approaches. This is because in our GA-based approach, whether the task is kept or removed is decided by the value-resource ratio (9). In other words, the task which contributes the least value to the system (i.e., it is shared by few applications), but consumes most computation resources will be removed if the resource is not enough. While the traditional heuristics do not consider task sharing among the applications.

From Fig. 7, we can also see that when the average utilization demand of the applications' task set increases, the system accrued value decreases for all four approaches. For instance, the accrued value of the GA-based approach decreases from 100% to 17.03% when the average utilization demand increases from 0.5 to 2.5. The reason for this change is that when the average utilization demand increases, on average, each task requires more computation resources, therefore, fewer tasks can be executed due to the resource constraint, and hence fewer applications are supported, resulting lower accrued value.

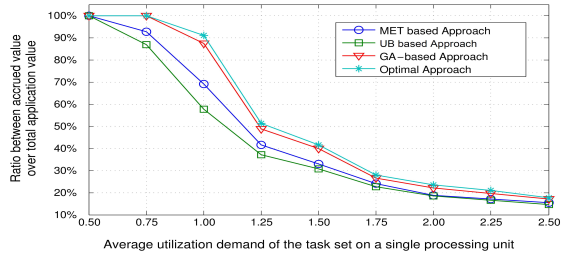


Fig. 7: Performance comparison between GA-based approach, UB based approach, MET based approach, and Exhaustive Search approach ($k = 3, n = 12, m = 20$)

In the second set of the experiments, we investigate the performance of the GA-based approach when the problem size increases. In particular, we set the number of processors to 10, the number of tasks to 80, and number of applications to 120. The results are depicted in Fig. 8 which clearly shows that the GA-based approach outperforms both UB and MET based approaches. Furthermore, we have similar observation that when the average utilization demand increases, the system's normalized accrued value decreases. However, the decreasing rate is faster than the case when the system size is smaller.

Another observation from Fig. 7 and Fig. 8 is that when the average utilization demand on a single processor is either under utilized (such as < 0.75) or overloaded (such as > 1.75), the performance difference among these approaches is small. This is because when the average utilization demand is high, only few applications can be supported, the accrued value difference between different selections is small. When

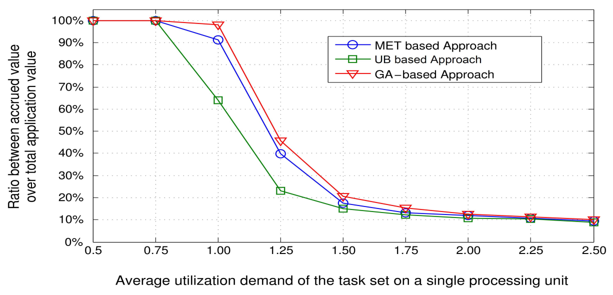


Fig. 8: Performance comparison between GA-based approach, UB based approach, and MET based approach ($k = 10, n = 80, m = 120$)

the average utilization demand is low, all the applications can be supported, therefore, the accrued values obtained by all converge to their maximal value of 1.

VII. Conclusion

In this paper, we have presented a GA-based approach to maximizing real-time system's value when applications are executed on heterogeneous processing units with resource constraints. The uniqueness of the problem this paper addresses is that different applications may share tasks; while most research in the literature assumes that applications are independent. Furthermore, the experimental results clearly show the superiority of GA-based approach over traditional heuristics.

It is worth pointing out that comparing to the simple heuristic UB based and MET based approaches, the time cost for executing our GA-based approach is significantly large. For instance, in the second set of experiment, it takes only 0.0243 seconds and 0.0227 seconds to obtain a solution by using UB based and MET based approach, respectively, while it takes 18.06 seconds with the proposed GA-based approach. Our next step is to investigate how to reduce the time cost without sacrificing the performance goal.

Acknowledgement

The work is supported in part by NSF CAREER CNS-0746643, NSF CNS-1018731, and NSF CNS-1035894.

VIII. Appendix

We first consider the GA parameter selection under the experimental setting with three processors, twelve tasks, and twenty applications, i.e., $k = 3$, $n = 12$, and $m = 20$, respectively, and investigate the GA performance under the following choices given by Table III.

TABLE III: Possible Parameter Values

Parameter	Values			
$\mathcal{P}$	100	150	200	250
$\mathcal{I}$	50	100	150	200
$\mathcal{C}$	50	100	150	200
$\mathcal{U}$	1	1.25	1.5	1.75

We follow the procedure proposed in [23] to decide the value of each parameter. The idea of the procedure is that we first calculate the value of the GA-based approach under all combinations of parameter values, then for each parameter, we choose the value under which the the value of the GA-based approach is the best.

To better explain the procedure, we use the selection of $\mathcal{U}$ value as an example. According to Table III, in addition to $\mathcal{U}$, there are three parameters and each parameter has four candidate values. Therefore, the total number of combinations for each candidate value of $\mathcal{U}$ is $4 \times 4 \times 4 = 64$. For each combination, we run the test for 10 times. Therefore, for each candidate value of $\mathcal{U}$, it has $64 \times 10 = 640$ test results. We compare the ratio between the accrued value obtained by the GA-based approach over total application value among these test cases and choose the value for $\mathcal{U}$ that has the largest accrued value ratio. Fig. 9a depicts relationship between $\mathcal{U}$ and the ratio. As we can see that the average accrued value ratios under four different $\mathcal{U}$ values are 52.65%, 52.69%, 52.81%, and 52.80%, respectively. Therefore, we choose $\mathcal{U} = 1.5$ because the average ratio of GA-based approach is the largest under $\mathcal{U} = 1.5$.

For GA-based approach, the more iterations we perform, the better solution we can obtain. This is also true for population size and the number of crossover. However, the running time will increase accordingly. Therefore, a tradeoff between performance and running time should be considered. In our implementation, we set the number of iteration to $\mathcal{I} = 150$ because when $\mathcal{I}$ becomes 200, the performance of the GA-based approach is not significantly increased, but the running time increases by 1/3 (see Fig. 9b). Similarly, we set both the population size and number of crossover to 150. In summary, the parameters for the GA-based approach are set as $\mathcal{P} = 150$, $\mathcal{I} = 150$, $\mathcal{C} = 150$, and $\mathcal{U} = 1.5$.

We use the same way to decide the parameter values for GA-based approach when the system size becomes larger, i.e., $k = 10$, $n = 80$, and $m = 120$. Under this scenario, the parameter values for GA-based approach are set as $\mathcal{P} = 600$, $\mathcal{I} = 550$, $\mathcal{C} = 400$, and $\mathcal{U} = 0.6$.

References

[1] J. Liu, E. Cheong, and F. Zhao, "Semantics-based optimization across uncoordinated tasks in networked embedded systems," in *Proceedings of the conference on Embedded software*, 2005, pp. 273–281.

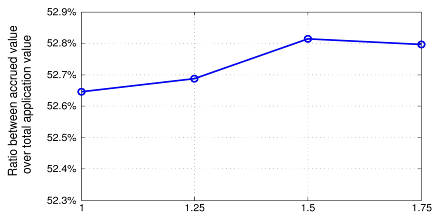
[2] C. Shelton, P. Koopman, and W. Nace, "A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems," in *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003, pp. 156–163.

[3] W. Nace and P. Koopman, "A graceful degradation framework for distributed embedded systems," in *Workshop on Reliability in Embedded Systems*, 2001.

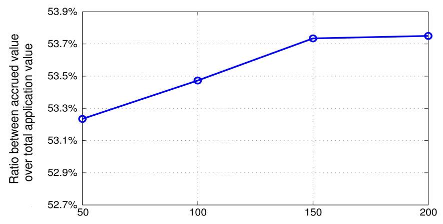
[4] ——, "A product family approach to graceful degradation," in *International Workshop on Distributed and Parallel Embedded Systems*, 2001, pp. 131–140.

[5] S. Baruah, "Task partitioning upon heterogeneous multiprocessor platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, may 2004, pp. 536 – 543.

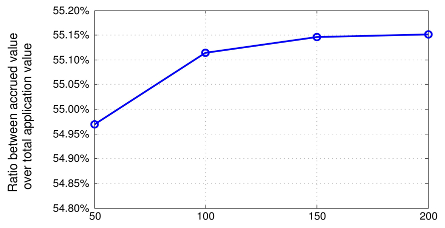
[6] J. H. Holland, *Adaptation in natural and artificial systems*. Cambridge, MA, USA: MIT Press, 1992.



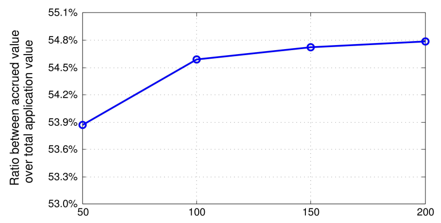
(a) Average number of mutation ($\mathcal{U}$)



(b) Number of iteration ($\mathcal{I}$)



(c) Population size ($\mathcal{P}$)



(d) Number of crossover ($\mathcal{C}$)

Fig. 9: Normalized accrued value for different parameter values

[7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[8] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *27th IEEE International Real-Time Systems Symposium*, Dec. 2006, pp. 101 –110.

[9] R. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," *Real-Time Systems*, vol. 47, pp. 1–40, 2011.

[10] S. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Systems*, vol. 32, pp. 9–20, 2006.

[11] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, dec. 2005, pp. 9 pp. –329.

[12] W. Liu, Z. Gu, J. Xu, X. Wu, and Y. Ye, "Satisfiability modulo graph theory for task mapping and scheduling on multiprocessor systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 8, pp. 1382 –1389, Aug. 2011.

[13] J. Lopez, M. Garcia, J. Diaz, and D. Garcia, "Worst-case utilization bound for edf scheduling on real-time multiprocessor systems," in *12th Euromicro Conference on Real-Time Systems*, 2000, pp. 25 –33.

[14] S. Baruah and J. Goossens, "The edf scheduling of sporadic task systems on uniform multiprocessors," in *Real-Time Systems Symposium, 2008*, Dec. 2008, pp. 367 –374.

[15] T. Gonzalez and S. Sahni, "Preemptive scheduling of uniform processor systems," *J. ACM*, vol. 25, pp. 92–101, January 1978.

[16] S. Funk, J. Goossens, and S. Baruah, "On-line scheduling on uniform multiprocessors," in *Real-Time Systems Symposium*, Dec. 2001, pp. 183 – 192.

[17] B. Andersson, G. Raravi, and K. Bletsas, "Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors," in *IEEE Real-Time Systems Symposium*, Dec. 2010, pp. 239 –248.

[18] S. Gopalakrishnan and M. Caccamo, "Task partitioning with replication upon heterogeneous multiprocessor systems," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, April 2006, pp. 199 – 207.

[19] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions," in *Heterogeneous Computing Workshop. Proceedings of Seventh*, March 1998, pp. 79 – 87.

[20] J. Puchinger, G. R. Raidl, and U. Pferschy, "The multidimensional knapsack problem: Structure and algorithms," *INFORMS J. on Computing*, vol. 22, no. 2, pp. 250–265, Apr. 2010.

[21] A. Freville, "The multidimensional 0-1 knapsack problem: An overview," *European Journal of Operational Research*, vol. 155, no. 1, pp. 1 – 21, 2004.

[22] H. Cho, B. Ravindran, and E. D. Jensen, "Utility accrual real-time scheduling for multiprocessor embedded systems," *J. Parallel Distrib. Comput.*, vol. 70, pp. 101–110, February 2010.

[23] G. Levitin, J. Rubinovitz, and B. Shnits, "A genetic algorithm for robotic assembly line balancing," *European Journal of Operational Research*, vol. 168, no. 3, pp. 811 – 825, 2006.

[24] J. Rubinovitz and G. Levitin, "Genetic algorithm for assembly line balancing," *International Journal of Production Economics*, vol. 41, no. 1-3, pp. 343–354, October 1995.

[25] S. Ponnambalam, P. Aravindan, and G. Naidu, "A multi-objective genetic algorithm for solving assembly line balancing problem," *Journal of Advanced Manufacturing Technology*, vol. 16, pp. 341– 352, 2000.

[26] Z. Michalewicz, *Genetic algorithms + data structures = evolution programs (3rd ed.)*. London, UK: Springer-Verlag, 1996.

[27] L. Wang, H. J. Siegel, V. R. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *J. Parallel Distrib. Comput.*, vol. 47, pp. 8–22, November 1997.

[28] A. J. Page, T. M. Keane, and T. J. Naughton, "Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system," *J. Parallel Distrib. Comput.*, vol. 70, pp. 758–766, July 2010.

[29] A. Y. Zomaya and Y.-H. Teh, "Observations on using genetic algorithms for dynamic load-balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 9, pp. 899–911, Sep. 2001.

[30] R. Hwang, M. Gen, and H. Katayama, "A comparison of multiprocessor task scheduling algorithms with communication costs," *Computers & Operations Research*, vol. 35, no. 3, pp. 976 – 993, 2008.

[31] J. Liu and F. Zhao, "Composing semantic services in open sensor-rich environments," *Network, IEEE*, vol. 22, no. 4, pp. 44 – 49, july 2008.

[32] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, January 1973.

[33] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[34] P. C. Chu and J. E. Beasley, "Constraint handling in genetic algorithms: The set partitioning problem," *J. Heuristics*, vol. 4, pp. 323–357, 1998.

[35] D. Powell and M. M. Skolnick, "Using genetic algorithms in engineering design optimization with non-linear constraints," in *Proceedings of the Conference on Genetic Algorithms*, 1993, pp. 424–431.

[36] P. C. Chu and J. E. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *J. Heuristics*, vol. 4, pp. 63–86, June 1998.

[37] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *Computer*, vol. 27, pp. 17–26, June 1994.

[38] T. Starkweather, S. Mcdaniel, D. Whitley, K. Mathias, and D. Whitley, "A comparison of genetic sequencing operators," in *Proceedings of the Conference on Genetic Algorithms*, 1991, pp. 69–76.

[39] E. Bini and G. C. Buttazzo, "Biasing effects in schedulability measures," in *Proceedings of the Conference on Real-Time Systems*, 2004, pp. 196–203.