Journal of Circuits, Systems, and Computers
Vol. 32, No. 6 (2023) 2350105 (26 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218126623501050

Application of Uncertain Programming in Hardware/ Software Partitioning: Model and Algorithm[*]

Si Chen[†,‡,¶], Lida Huang[†,||], Guoqi Xie[†,‡,**], Renfa Li[†,††] and Keqin Li[§,‡‡]

[†]*College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, P. R. China*

[‡]*Center for Convergence of Automobile and Cyberspace, Research Institute of Hunan University in Chongqing, Chongqing 401120, P. R. China*

[§]*Department of Computer Science, State University of New York, New Paltz, New York 12561, USA*
[¶]*chensi_2018@hnu.edu.cn*
[||]*hld_jt@hnu.edu.cn*
[**]*xgqman@hnu.edu.cn*
[††]*lirenfa@hnu.edu.cn*
[‡‡]*lik@newpaltz.edu*

Received 16 June 2022
Accepted 22 September 2022
Published 9 November 2022

Hardware/software partitioning is a typical multi-stage decision optimization problem; most existing hardware/software partitioning methods ignore a fact that real-life decisions are usually made in an uncertain state. We should model the hardware/software partitioning problem in uncertain environments and deal with uncertainty. The state-of-the-art work proposed an uncertainty conversion method for hardware/software partitioning, but this method does not include the equivalent deterministic model and is not suitable for dealing with different types of uncertainties. In order to cope with different situations with various uncertainties, we should apply uncertain programming to build a model in uncertain environments and give different equivalent deterministic models to convert different uncertainties theoretically. In this paper, we present the process of applying uncertain programming to solve the hardware/software partitioning problem, including the model and algorithm. We convert the uncertain programming model into its equivalent deterministic models, including the expected value model and the chance-constrained programming model; we give details for the conversion methods of these two models. We present the custom genetic algorithm to solve the converted model, by incorporating a greedy idea in two steps of the genetic algorithm. Experimental results show that the

[*]This paper was recommended by Regional Editor Tongquan Wei.
[||,**]Corresponding authors.

custom genetic algorithm can find a high-quality approximate solution while running much faster for large input scales, compared with the exact algorithm.

Keywords: Uncertain programming; hardware/software partitioning; the custom genetic algorithm.

1. Introduction

1.1. *Background and motivation*

Hardware/software partitioning is a typical multi-stage decision optimization problem in embedded system design, where hardware and software are used for realizing different functional modules to achieve the optimal objective, under the condition of satisfying system performance constraints. There has been a lot of research on hardware/software partitioning, and various models and algorithms have been proposed, including exact algorithms[1–6] and heuristic algorithms.[7–12] However, most existing hardware/software partitioning methods ignore a fact that is also ignored by most optimization problems, that is, real-life decisions are usually made in an uncertain state.

Typically, when conducting hardware/software partitioning, we always assume that the system's relevant parameters are determinate values and pre-estimate these parameter values based on experience; uncertainty is introduced by this way because these pre-estimations are not completely accurate. We should model the hardware/software partitioning problem in uncertain environments and deal with uncertainty.

In order to model the optimization problems in uncertain environments, Liu provides uncertain programming theory,[13] which means the optimization theory in generally uncertain environments. Uncertain programming gives the concept of equivalent deterministic model to deal with uncertainty, and it has been applied to many problems, such as transportation,[14] assignment[15] and shortest path.[16] Although some works have considered the uncertainty of the hardware/software partitioning problem,[17–20] none of them have given the equivalent deterministic model, for converting the uncertainty theoretically.

The state-of-the-art work[20] proposed an uncertainty conversion method for hardware/software partitioning, but this method does not include the equivalent deterministic model and is not suitable for dealing with different types of uncertainties. In order to cope with different situations with various uncertainties, we should apply uncertain programming to build a model in uncertain environments and give different equivalent deterministic models to convert different uncertainties theoretically.

1.2. *Contributions*

In this paper, we present the process of applying uncertain programming to solve the hardware/software partitioning problem, including the model and algorithm. We pay attention to the theoretical approach, rather than the system design.

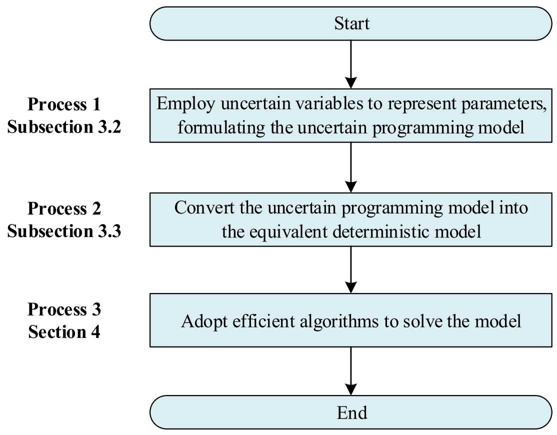
Start
Process 1
Subsection 3.2
Employ uncertain variables to represent parameters, formulating the uncertain programming model
Process 2
Subsection 3.3
Convert the uncertain programming model into the equivalent deterministic model
Process 3
Section 4
Adopt efficient algorithms to solve the model
End

Fig. 1. The detailed process of applying uncertain programming.

The detailed process has been displayed in Fig. 1. Specifically, we express the hardware/software problem as an optimization problem and solve it, including that: (1) we employ uncertain variables to represent system parameters that need to be pre-estimated, formulating the uncertain programming model; (2) we convert the uncertain programming model into the equivalent deterministic model, because the deterministic model can be solved mathematically; and (3) we adopt efficient algorithms to solve the model. Our main contributions are described below.

(i) We convert the uncertain programming model into its equivalent deterministic models, including the expected value model and the chance-constrained programming model. We give details for the conversion methods of these two models, including the objective conversion and the constraint conversion, and we finally present a unified form of the converted model.

(ii) We present the custom genetic algorithm to solve the converted model, by incorporating a greedy idea in two steps of the genetic algorithm. Experimental results show that the custom genetic algorithm can find a high-quality approximate solution while running much faster for large input scales, compared with the exact algorithm.

The rest of this paper is structured as follows. Section 2 shows related work. Section 3 presents the uncertain programming model. Section 4 proposes the custom genetic algorithm. Section 5 gives experimental results and Sec. 6 concludes the paper.

2. Related Work

Embedded system design is very complex, and many works have investigated it.[21–25] We conduct research on the hardware/software partitioning in design. The hardware/software partitioning problem aims at conducting a reasonable partitioning for hardware implementation and software implementation of system modules, for meeting system design requirements, which is an NP-hard optimization problem. Some methods have done many efforts to solve the hardware/software partitioning problem. Exact algorithms (such as integer linear programming,[1,2] dynamic programming[3,4] and branch-and-bound[5,6]) are used for the hardware/software partitioning problem with small inputs, while heuristic algorithms (such as genetic algorithm,[7,8] greedy algorithm[9,10] and tabu search[11,12]) are implemented to settle large-scale problems, for the reason that exact algorithms run slowly for large inputs.

All these existing methods run in deterministic environments, in which performance parameters of the system components are determinate, but these performance parameters cannot be accurately determined in fact. That is, the above existing methods do not consider making real-life decisions in an uncertain state. The hardware/software partitioning problem should be modeled in uncertain environments, and the uncertainty should be dealt with.

As a theory of modeling optimization problems in uncertain environments, uncertain programming presents the equivalent deterministic model to deal with uncertainty, and it has been applied to many problems.[14–16] For the hardware/software partitioning problem, some works have considered its uncertainty. In 2012, Jiang *et al.*[17] for the first time proposed an uncertain model for hardware/software partitioning; the uncertain model was then expanded to the cyber-physical system in 2019.[18] In that work, the authors elaborate on the case where system parameters are linear uncertain variables. Wang *et al.*[19] paid attention to the case where system parameters are normal uncertain variables in 2016. In 2021, Chen *et al.*[20] presented an uncertainty conversion method that can be applied to various forms of uncertain variables.

The above works have conducted a preliminary exploration of dealing with uncertainty, but none of them have given the equivalent deterministic model, for converting the uncertainty theoretically. In order to cope with different situations with various uncertainties, we should apply uncertain programming to build a model in uncertain environments and give different equivalent deterministic models to convert different uncertainties theoretically. In this paper, we present the process of applying uncertain programming to solve the hardware/software partitioning problem, including the model and algorithm.

3. Uncertain Programming Model

In this section, we elaborate on the modeling process of the uncertain programming model. First, we briefly introduce some basic definitions of uncertain programming theory. Second, we present the definition and formulation of the hardware/software partitioning problem. Third, we propose the equivalent deterministic models of the uncertain programming model, including the expected value model and the chance-constrained programming model. Fourth, we try to explain the relationship between the deterministic model and the uncertain programming model. At last, we instantiate the uncertain programming model through an example.

3.1. *Basic definitions*

For ease of understanding, in this subsection, we introduce some basic definitions that will be used later. These definitions come from uncertainty theory,[26] which was founded by Liu in 2007, and are applied in uncertain programming.

Following notions are utilized in definitions: (1) Γ represents a nonempty set; (2) a collection $\mathcal{L}$ of Γ's subsets is called a σ-algebra; (3) each element Λ in the σ-algebra $\mathcal{L}$ is called an event and (4) a number $\mathcal{M}\{\Lambda\}$ is assigned to each event Λ, which indicates the belief degree (i.e., probability) that the event Λ will occur.

Definition 1.[13,26] The *uncertain measure* is a function that maps the collection $\mathcal{L}$ to the range [0, 1]. If the set function $\mathcal{M}$ with events Λ can satisfy the following four axioms, it is called an uncertain measure.

- $\mathcal{M}\{\Gamma\} = 1$ for the universal set Γ (*normality*).
- $\mathcal{M}\{\Lambda_1\} \leq \mathcal{M}\{\Lambda_2\}$ whenever $\Lambda_1 \subset \Lambda_2$ (*monotonicity*).
- $\mathcal{M}\{\Lambda\} + \mathcal{M}\{\Lambda^c\} = 1$ for any event Λ (*self-duality*).
- $\mathcal{M}\{\cup_{i=1}^{\infty}\Lambda_i\} \leq \sum_{i=1}^{\infty} \mathcal{M}\{\Lambda_i\}$ for every countable sequence of events $\{\Lambda_i\}$ (*countable subadditivity*).

Definition 2.[13,26] If the set function $\mathcal{M}$ is an uncertain measure, the triplet $(\Gamma, \mathcal{L}, \mathcal{M})$ is called an uncertainty space. An *uncertain variable* is a measurable function γ that maps the uncertainty space $(\Gamma, \mathcal{L}, \mathcal{M})$ to the set of real numbers.

Definition 3.[13,26] The *uncertainty distribution* Φ is a description to an uncertain variable γ. For any real number x, Φ: $\mathcal{R} \to [0, 1]$ is defined by $\Phi(x) = \mathcal{M}\{\gamma \leq x\}$.

Definition 4.[13,26] For an uncertain variable γ, $\gamma_{\sup}(\beta) = \sup\{q|\mathcal{M}\{\gamma \geq q\} \geq \beta\}$ is called the ***β**-optimistic value* to γ, where $\beta \in (0, 1]$.

3.2. *Problem definition and formulation*

Considering that what we focus on is the modeling method of the uncertain programming for hardware/software partitioning problem, not the model itself, we use the simple series system model similar to Ref. 27. Suppose that the system is composed of a series of n basic scheduling modules, denoted as $M = \{M_1, M_2, \ldots, M_n\}$, where M_{i+1} follows M_i for $i = 1, 2, \ldots, n-1$. These modules can be functions or procedures, where the communication time between each other is not included in our model for simplicity. Each module M_i can be implemented in hardware or software while associating with hardware area a_i, execution time t_i and power consumption p_i. Figure 2 shows an example with $n = 6$.

We *employ uncertain variables to represent system parameters, formulating the uncertain programming model of the hardware/software partitioning problem.*

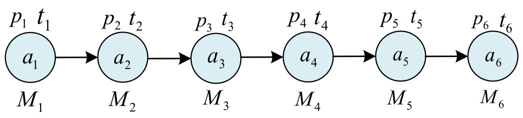

Fig. 2. An example system model with 6 modules.

Table 1. Notations and definitions.

Notation	Definition
a_i	Hardware area of implementing M_i in hardware
Φ_{ai}	Uncertainty distribution of a_i
Φ_{ai}^{-1}	Inverse uncertainty distribution of a_i
t_i^{s}	Execution time of implementing M_i in software
Φ_{ti}^{s}	Uncertainty distribution of t_i^{s}
$\Phi_{ti}^{\mathrm{s}^{-1}}$	Inverse uncertainty distribution of t_i^{s}
t_i^{h}	Execution time of implementing M_i in hardware
Φ_{ti}^{h}	Uncertainty distribution of t_i^{h}
$\Phi_{ti}^{\mathrm{h}^{-1}}$	Inverse uncertainty distribution of t_i^{h}
p_i^{s}	Power consumption of implementing M_i in software
Φ_{pi}^{s}	Uncertainty distribution of p_i^{s}
$\Phi_{pi}^{\mathrm{s}^{-1}}$	Inverse uncertainty distribution of p_i^{s}
p_i^{h}	Power consumption of implementing M_i in hardware
Φ_{pi}^{h}	Uncertainty distribution of p_i^{h}
$\Phi_{pi}^{\mathrm{h}^{-1}}$	Inverse uncertainty distribution of p_i^{h}
T_c	Total execution time limitation
P_c	Total power consumption limitation

The notations illustrated in Table 1 are defined on M, and we will briefly explain what they mean.

- If the module M_i is implemented in hardware, it will occupy the hardware area. Uncertain variable a_i is used to denote the hardware area of M_i, and Φ_{ai} is the uncertainty distribution of a_i.
- The execution of each module M_i takes time. If M_i is implemented in software, uncertain variable t_i^{s} is used to denote its execution time; if M_i is implemented in hardware, uncertain variable t_i^{h} is used to denote its execution time. Φ_{ti}^{s} and Φ_{ti}^{h} are uncertainty distributions of t_i^{s} and t_i^{h}, respectively.
- Each module M_i consumes power when running. If M_i is implemented in software, uncertain variable p_i^{s} is used to denote its power consumption; if M_i is implemented in hardware, uncertain variable p_i^{h} is used to denote its power consumption. Φ_{pi}^{s} and Φ_{pi}^{h} are uncertainty distributions of p_i^{s} and p_i^{h}, respectively.

The hardware/software partitioning problem we define here is to make a decision that which modules should be implemented in software and which in hardware, where each module can only be implemented in one way. That is, the problem is to find a bipartition B of M: $B = (M_{\mathrm{h}}, M_{\mathrm{s}})$, where $M_{\mathrm{h}} \cup M_{\mathrm{s}} = M$ and $M_{\mathrm{h}} \cap M_{\mathrm{s}} = \emptyset$. We describe our modeling method in detail, by taking the example of minimizing the hardware area while satisfying the time and power consumption constraints. That is, our objective is to get the minimal total hardware area, while the total execution time does not exceed T_c and the total power consumption does not exceed P_c. By the way, our method is a general method and also works when the objective is to optimize time or power consumption.

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ which represents the implementation way of n modules be a feasible solution to this partitioning problem, in which x_i is chosen from $\{0, 1\}$. If $x_i = 0$, it means that the module M_i will be implemented in hardware, otherwise in software. Thus, the total hardware area is $\sum_{i=1}^{n} a_i(1 - x_i)$, and the corresponding total execution time is $\sum_{i=1}^{n}[t_i^{\mathrm{s}} x_i + t_i^{\mathrm{h}}(1 - x_i)]$, and the corresponding total power consumption is $\sum_{i=1}^{n}[p_i^{\mathrm{s}} x_i + p_i^{\mathrm{h}}(1 - x_i)]$.

Based on the above definitions, we model the hardware/software partitioning problem B as an optimization problem and express it as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} a_i(1 - x_i)\,, \\ \text{subject to} \quad & \sum_{i=1}^{n} [t_i^{\mathrm{s}} x_i + t_i^{\mathrm{h}}(1 - x_i)] \le T_c\,, \\ & \sum_{i=1}^{n} [p_i^{\mathrm{s}} x_i + p_i^{\mathrm{h}}(1 - x_i)] \le P_c\,, \\ & x_i \in \{0, 1\}, \quad i = 1, 2, \ldots, n\,. \end{aligned} \tag{1}$$

3.3. *Equivalent deterministic models*

In this model, a_i, t_i^{s}, t_i^{h}, p_i^{s} and p_i^{h} are uncertain variables, so it is hard to solve the model directly. *We should convert the model into the equivalent deterministic model to solve.* Uncertain programming was founded by Liu[13] in 2009 as a type of mathematical programming involving uncertain variables, and it gives four ranking criteria for uncertain variables to establish a mathematical model, including expected value criterion, optimistic value criterion, pessimistic value criterion and chance criterion. In this paper, based on uncertain programming[13] and uncertainty theory,[26] we, respectively, take the expected value criterion and the optimistic value criterion as examples to describe the model conversion process in detail.

3.3.1. *Expected value model*

According to the expected value criterion,[13] for two uncertain variables γ and ρ, $\gamma > \rho$ if and only if $E(\gamma) > E(\rho)$. In this case, we can get the following template for converting the model with uncertain variables.

$$\begin{aligned}&\text{minimize} \quad y(\mathbf{x},\boldsymbol{\gamma}),\\ &\text{subject to} \quad c_j(\mathbf{x},\boldsymbol{\gamma}) \le 0, \quad j=1,2,\ldots,p.\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}&\text{minimize} \quad E[y(\mathbf{x},\boldsymbol{\gamma})],\\ &\text{subject to} \quad E[c_j(\mathbf{x},\boldsymbol{\gamma})] \le 0, \quad j=1,2,\ldots,p.\end{aligned}$$

In order to complete the model conversion, we need to calculate the expected values of the objective and constraint equations, respectively. For ease of understanding, we list a theorem used in calculations; we omit the proof of this theorem to save space, and the detailed proof can be found in Refs. 13 and 26.

Theorem 1. *Assume that* $\gamma_1,\gamma_2,\ldots,\gamma_n$ *are independent uncertain variables, and* $f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_n(\mathbf{x}), f_0(\mathbf{x})$ *are real-valued functions, then the expected value* $E[y(\mathbf{x},\boldsymbol{\gamma})]$ *of the function* $y(\mathbf{x},\boldsymbol{\gamma}) = f_1(\mathbf{x})\gamma_1 + \cdots + f_n(\mathbf{x})\gamma_n + f_0(\mathbf{x})$ *can be calculated as follows*:

$$E[y(\mathbf{x},\boldsymbol{\gamma})] = f_1(\mathbf{x})E[\gamma_1] + \cdots + f_n(\mathbf{x})E[\gamma_n] + f_0(\mathbf{x})\,.$$

(1) For the objective equation $\sum_{i=1}^{n} a_i(1-x_i)$, note that each $(1-x_i)$ is a real-valued function for $i=1,2,\ldots,n$. According to Theorem 1, its expected value can be computed as $E[\sum_{i=1}^{n} a_i(1-x_i)] = \sum_{i=1}^{n} E[a_i](1-x_i)$.

(2) For the constraint equation $\sum_{i=1}^{n}[t_i^{\mathrm{s}}x_i + t_i^{\mathrm{h}}(1-x_i)] \le T_c$, we need to calculate the expected value of $\sum_{i=1}^{n}[t_i^{\mathrm{s}}x_i + t_i^{\mathrm{h}}(1-x_i)] - T_c$. Note that each x_i and each $(1-x_i)$ are real-valued functions for $i=1,2,\ldots,n$, and $(-T_c)$ is also a real-valued function, so its expected value can be calculated as follows:

$$E\left[\sum_{i=1}^{n}[t_i^{\mathrm{s}}x_i + t_i^{\mathrm{h}}(1-x_i)] - T_c\right] = \sum_{i=1}^{n}\left[E[t_i^{\mathrm{s}}]x_i + E[t_i^{\mathrm{h}}](1-x_i)\right] - T_c\,.$$

Similarly, for the constraint equation $\sum_{i=1}^{n}[p_i^{\mathrm{s}}x_i + p_i^{\mathrm{h}}(1-x_i)] \le P_c$, we have

$$E\left[\sum_{i=1}^{n}[p_i^{\mathrm{s}}x_i + p_i^{\mathrm{h}}(1-x_i)] - P_c\right] = \sum_{i=1}^{n}[E[p_i^{\mathrm{s}}]x_i + E[p_i^{\mathrm{h}}](1-x_i)] - P_c\,.$$

Uncertainty theory gives the expected values of different types of uncertain variables, and we can directly use these results in calculations. For example, for linear uncertain variable $\gamma \sim \mathcal{L}(c,d)$, $E[\gamma] = (c+d)/2$, where c and d are real numbers with $c < d$.

3.3.2. *Chance-constrained programming model*

In practice, the expected value model is not suitable for all situations; when the stability of the uncertain variable is poor, the expected value cannot reflect the characteristics of the variable. Furthermore, we are not always concerned about maximizing or minimizing the expected value, but consider the belief degree of the events' occurrence. Sometimes, we may allow the decision to be made without satisfying the constraints, but the decision's measure of satisfying the constraints cannot be less than a certain confidence level. In this case, the chance-constrained programming model is a good choice.

According to the optimistic value criterion,[13] for two uncertain variables γ and ρ, $\gamma > \rho$ if and only if $\gamma_{\sup}(\beta) > \rho_{\sup}(\beta)$ for some predetermined confidence level $\beta \in (0, 1]$, where $\gamma_{\sup}(\beta)$ and $\rho_{\sup}(\beta)$ are the β-optimistic values of γ and ρ, respectively. For the uncertain constraints $c_j(\mathbf{x}, \boldsymbol{\gamma}) \leq 0$, the establishment of the uncertain constraints is related to a confidence level α: $\mathcal{M}\{c_j(\boldsymbol{x}, \boldsymbol{\gamma}) \leq 0\} \geq \alpha$, where $\mathcal{M}\{c_j(\boldsymbol{x}, \boldsymbol{\gamma}) \leq 0\}$ is the belief degree that event $c_j(\mathbf{x}, \boldsymbol{\gamma}) \leq 0$ occurs. In this case, we can get the following template for converting the model with uncertain variables.

$$\begin{aligned} &\text{minimize} \quad && y(\mathbf{x}, \boldsymbol{\gamma})\,, \\ &\text{subject to} \quad && c_j(\mathbf{x}, \boldsymbol{\gamma}) \leq 0, \quad j = 1, 2, \ldots, p\,. \end{aligned}$$

$$\Downarrow$$

$$\begin{aligned} &\text{minimize} \quad && \max \bar{y}\,, \\ &\text{subject to} \quad && \mathcal{M}\{y(\mathbf{x}, \boldsymbol{\gamma}) \geq \bar{y}\} \geq \beta\,, \\ & && \mathcal{M}\{c_j(\mathbf{x}, \boldsymbol{\gamma}) \leq 0\} \geq \alpha_j, \quad j = 1, 2, \ldots, p\,, \end{aligned}$$

where $\max \bar{y}$ means the β-optimistic value to the objective function $y(\mathbf{x}, \boldsymbol{\gamma})$.

In order to complete the model conversion, we list three theorems used in the conversion; we omit the proofs of these three theorems to save space, and detailed proofs can be found in Refs. 13 and 26.

Theorem 2. *Assume that $\gamma_1, \gamma_2, \ldots, \gamma_n$ are independent uncertain variables with uncertainty distributions. If f is a strictly increasing function, then $\gamma = f(\gamma_1, \gamma_2, \ldots, \gamma_n)$ is an uncertain variable, and*

$$\gamma_{\sup}(\beta) = f(\gamma_{1\sup}(\beta), \gamma_{2\sup}(\beta), \ldots, \gamma_{n\sup}(\beta))\,.$$

Theorem 3. *Assume that γ is an uncertain variable with uncertainty distribution Φ. Then its β-optimistic value is*

$$\gamma_{\sup}(\beta) = \Phi^{-1}(1 - \beta)\,.$$

Theorem 4. *Assume that $\gamma_1, \gamma_2, \ldots, \gamma_n$ are independent uncertain variables, $\Phi_1, \Phi_2, \ldots, \Phi_n$ are their uncertainty distributions and $f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_n(\mathbf{x}), f_0(\mathbf{x})$ are all nonnegative real-valued functions, then*

$$\mathcal{M}\left\{\sum_{i=1}^{n} f_i(\boldsymbol{x})\gamma_i \leq f_0(\boldsymbol{x})\right\} \geq \alpha$$

holds if and only if

$$\sum_{i=1}^{n} f_i(\boldsymbol{x})\Phi_i^{-1}(\alpha) \le f_0(\boldsymbol{x})\,,$$

where α is a confidence level.

(1) For the objective, we need to compute the β-optimistic value to $\sum_{i=1}^{n} a_i(1-x_i)$. We notice that $\sum_{i=1}^{n} a_i(1-x_i)$ is an increasing function to $a_1, a_2, \ldots, a_n$, and we denote $\sum_{i=1}^{n} a_i(1-x_i)$ as a. According to Theorem 2, a is an uncertain variable and $a_{\sup}(\beta) = \sum_{i=1}^{n} a_{i\sup}(\beta)(1-x_i)$. Based on Theorem 3, the β-optimistic value to $\sum_{i=1}^{n} a_i(1-x_i)$ is $\sum_{i=1}^{n} \Phi_{ai}^{-1}(1-\beta)(1-x_i)$, where Φ_{ai}^{-1} is the inverse uncertainty distribution of a_i.

(2) For the constraint, we give a lemma that will be used in the conversion based on the model characteristics and make a simple proof.

Lemma 1. *Assume that $\gamma_1, \gamma_2, \ldots, \gamma_n, \rho_1, \rho_2, \ldots, \rho_n$ are independent uncertain variables with uncertainty distributions $\Phi_1^{\gamma}, \Phi_2^{\gamma}, \ldots, \Phi_n^{\gamma}, \Phi_1^{\rho}, \Phi_2^{\rho}, \ldots, \Phi_n^{\rho}$, respectively, x_i takes value from $\{0,1\}$ for $i = 1, 2, \ldots, n$, and $f_0(\boldsymbol{x})$ is a nonnegative real-valued function, then the inequation*

$$\mathcal{M}\left\{\left(\sum_{i=1}^{n} x_i\gamma_i + \sum_{i=1}^{n}(1-x_i)\rho_i\right) \le f_0(\boldsymbol{x})\right\} \ge \alpha$$

holds if and only if

$$\sum_{i=1}^{n} x_i\Phi_i^{\gamma^{-1}}(\alpha) + \sum_{i=1}^{n}(1-x_i)\Phi_i^{\rho^{-1}}(\alpha) \le f_0(\boldsymbol{x})\,.$$

Proof. Note that each x_i and each $(1-x_i)$ are nonnegative real-valued functions, for $i = 1, 2, \ldots, n$; that is, these $2n$ terms are all nonnegative real-valued functions. According to Theorem 4, we consider the case of $2n$ independent uncertain variables. Considering that $\gamma_1, \gamma_2, \ldots, \gamma_{2n}$ are independent, we denote $\gamma_{n+1}, \gamma_{n+2}, \ldots, \gamma_{2n}$ as $\rho_1, \rho_2, \ldots, \rho_n$. Lemma 1 is then proven. □

Based on the above description, the two uncertain constraints hold with confidence levels as follows:

$$\mathcal{M}\left\{\sum_{i=1}^{n} x_i t_i^{\mathrm{s}} + \sum_{i=1}^{n}(1-x_i)t_i^{\mathrm{h}} \le T_c\right\} \ge \alpha_t\,, \tag{2}$$

$$\mathcal{M}\left\{\sum_{i=1}^{n} x_i p_i^{\mathrm{s}} + \sum_{i=1}^{n}(1-x_i)p_i^{\mathrm{h}} \le P_c\right\} \ge \alpha_p\,, \tag{3}$$

where α_t and α_p are the confidence levels of the time constraint and the power consumption constraint, respectively. It means that the belief degree of the time constraint occurrence cannot be less than α_t and the belief degree of the power consumption constraint occurrence cannot be less than α_p. According to the definitions in Sec. 3.2, $t_1^{\mathrm{s}}, t_2^{\mathrm{s}}, \dots, t_n^{\mathrm{s}}$, $t_1^{\mathrm{h}}, t_2^{\mathrm{h}}, \dots, t_n^{\mathrm{h}}$ are independent of each other. Based on Lemma 1, the uncertain time constraint (i.e., Eq. (2)) is converted into

$$\sum_{i=1}^{n}[\Phi_{ti}^{\mathrm{s}^{-1}}(\alpha_t)x_i] + \sum_{i=1}^{n}[\Phi_{ti}^{\mathrm{h}^{-1}}(\alpha_t)(1-x_i)] \le T_c\,, \tag{4}$$

where $\Phi_{ti}^{\mathrm{s}^{-1}}$ and $\Phi_{ti}^{\mathrm{h}^{-1}}$ are inverse uncertainty distributions of t_i^{s} and t_i^{h}, respectively. Similarly, the uncertain power consumption constraint (i.e., Eq. (3)) is converted into

$$\sum_{i=1}^{n}[\Phi_{pi}^{\mathrm{s}^{-1}}(\alpha_p)x_i] + \sum_{i=1}^{n}[\Phi_{pi}^{\mathrm{h}^{-1}}(\alpha_p)(1-x_i)] \le P_c\,, \tag{5}$$

where $\Phi_{pi}^{\mathrm{s}^{-1}}$ and $\Phi_{pi}^{\mathrm{h}^{-1}}$ are inverse uncertainty distributions of p_i^{s} and p_i^{h}, respectively.

Note that confidence levels β, α_t and α_p are independent and can be different values. These confidence levels generally depend on expert experience and are problem dependent, which need to be analyzed for specific problems.

Uncertainty theory gives the inverse uncertainty distributions of different types of uncertain variables, and we can directly use these results in conversion. For example, the inverse uncertainty distribution of linear uncertain variable $\mathcal{L}(c,d)$ is $\Phi^{-1}(\alpha) = (1-\alpha)c + \alpha d$.

3.3.3. *Converted model*

Finally, we use some notations to simplify the converted object and constraints, and we present a unified form of the converted model.

- For the expected value model, we make the following denotations:

$$\begin{aligned}
\phi_{ai} &= E[a_i]\,,\\
\phi_{ti} &= E[t_i^{\mathrm{s}}] - E[t_i^{\mathrm{h}}], \quad \phi_{pi} = E[p_i^{\mathrm{s}}] - E[p_i^{\mathrm{h}}]\,,\\
T_c' &= T_c - \sum_{i=1}^{n} E[t_i^{\mathrm{h}}], \quad P_c' = P_c - \sum_{i=1}^{n} E[p_i^{\mathrm{h}}]\,.
\end{aligned} \tag{6}$$

- For the chance-constrained programming model, we make the following denotations:

$$\begin{aligned}
\phi_{ai} &= \Phi_{ai}^{-1}(1-\beta)\,,\\
\phi_{ti} &= \Phi_{ti}^{\mathrm{s}^{-1}}(\alpha_t) - \Phi_{ti}^{\mathrm{h}^{-1}}(\alpha_t), \quad \phi_{pi} = \Phi_{pi}^{\mathrm{s}^{-1}}(\alpha_p) - \Phi_{pi}^{\mathrm{h}^{-1}}(\alpha_p)\,,\\
T_c' &= T_c - \sum_{i=1}^{n} \Phi_{ti}^{\mathrm{h}^{-1}}(\alpha_t), \quad P_c' = P_c - \sum_{i=1}^{n} \Phi_{pi}^{\mathrm{h}^{-1}}(\alpha_p)\,.
\end{aligned} \tag{7}$$

There is no doubt that $T'_c \geq 0$ and $P'_c \geq 0$, otherwise the constraints are unreasonable and the problem does not hold.[27] Then, the final equivalent deterministic model of problem B is as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} \phi_{ai}(1 - x_i)\,, \\ \text{subject to} \quad & \sum_{i=1}^{n} \phi_{ti} x_i \leq T'_c\,, \\ & \sum_{i=1}^{n} \phi_{pi} x_i \leq P'_c\,, \\ & x_i \in \{0,1\}, \quad i = 1,2,\ldots,n\,. \end{aligned} \tag{8}$$

We notice that minimizing the value of $\sum_{i=1}^{n} \phi_{ai}(1 - x_i)$ is equal to maximizing the value of $\sum_{i=1}^{n} \phi_{ai} x_i$, so B can be formulated as the following B_1:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} \phi_{ai} x_i\,, \\ \text{subject to} \quad & \sum_{i=1}^{n} \phi_{ti} x_i \leq T'_c\,, \\ & \sum_{i=1}^{n} \phi_{pi} x_i \leq P'_c\,, \\ & x_i \in \{0,1\}, \quad i = 1,2,\ldots,n\,. \end{aligned} \tag{9}$$

3.4. *Relationship between the deterministic model and the uncertain programming model*

In this subsection, we transform our partitioning uncertain programming model into the deterministic model by specifically setting the parameters of the uncertain programming model. In this way, we try to explain the relationship between the deterministic model and the uncertain programming model.

For the model of problem B in Sec. 3.2 (i.e., Eq. (1)), we consider two scenarios:

(i) If all parameters a_i, t_i^{s}, t_i^{h}, p_i^{s} and p_i^{h} are all known certain variables, Eq. (1) is a deterministic model that can be solved directly. Most of the current models are in this form, and we call it the original deterministic model here.

(ii) If all parameters a_i, t_i^{s}, t_i^{h}, p_i^{s} and p_i^{h} are all unknown uncertain variables, Eq. (1) is an uncertain programming model that should be converted into its equivalent deterministic model (i.e., Eq. (8)).

In the second scenario, for the equivalent deterministic model, we take the expected value model as an example and consider the linear case where the uncertain variable presents linear uncertainty distribution. Then, all parameters are linear uncertain variables, denoted by $\mathcal{L}(c_{ai}, d_{ai})$, $\mathcal{L}(c^{\mathrm{s}}_{ti}, d^{\mathrm{s}}_{ti})$, $\mathcal{L}(c^{\mathrm{h}}_{ti}, d^{\mathrm{h}}_{ti})$, $\mathcal{L}(c^{\mathrm{s}}_{pi}, d^{\mathrm{s}}_{pi})$ and $\mathcal{L}(c^{\mathrm{h}}_{pi}, d^{\mathrm{h}}_{pi})$, respectively, where c_{ai}, d_{ai}, c^{s}_{ti}, d^{s}_{ti}, c^{h}_{ti}, d^{h}_{ti}, c^{s}_{pi}, d^{s}_{pi}, c^{h}_{pi}, d^{h}_{pi} are all non-negative real numbers. According to uncertainty theory,[26] for linear uncertain variable $\gamma \sim \mathcal{L}(c, d)$, its expected value is $E[\gamma] = (c + d)/2$.

$\mathcal{L}(c, d)$ means that any value in the range (c, d) has the same probability to become the value of the current uncertain variable. In particular, we let all $c = d$, then the uncertain variable's value becomes a certain value. In this way, we have

$$a_i = c_{ai} = d_{ai}, \quad t^{\mathrm{s}}_i = c^{\mathrm{s}}_{ti} = d^{\mathrm{s}}_{ti}, \quad t^{\mathrm{h}}_i = c^{\mathrm{h}}_{ti} = d^{\mathrm{h}}_{ti}, \quad p^{\mathrm{s}}_i = c^{\mathrm{s}}_{pi} = d^{\mathrm{s}}_{pi}, \quad p^{\mathrm{h}}_i = c^{\mathrm{h}}_{pi} = d^{\mathrm{h}}_{pi}\,.$$

Based on the above settings and calculation formulas, the objective of Eq. (8) is calculated as follows:

$$\text{minimize} \quad \sum_{i=1}^{n} \left[\left(\frac{c_{ai} + d_{ai}}{2} \right) (1 - x_i) \right] = \sum_{i=1}^{n} a_i (1 - x_i)\,,$$

and the constraints of Eq. (8) are calculated as

$$\sum_{i=1}^{n} \left[\left(\frac{c^{\mathrm{s}}_{ti} + d^{\mathrm{s}}_{ti}}{2} \right) x_i + \left(\frac{c^{\mathrm{h}}_{ti} + d^{\mathrm{h}}_{ti}}{2} \right) (1 - x_i) \right] = \sum_{i=1}^{n} [t^{\mathrm{s}}_i x_i + t^{\mathrm{h}}_i (1 - x_i)] \le T_c\,,$$

$$\sum_{i=1}^{n} \left[\left(\frac{c^{\mathrm{s}}_{pi} + d^{\mathrm{s}}_{pi}}{2} \right) x_i + \left(\frac{c^{\mathrm{h}}_{pi} + d^{\mathrm{h}}_{pi}}{2} \right) (1 - x_i) \right] = \sum_{i=1}^{n} [p^{\mathrm{s}}_i x_i + p^{\mathrm{h}}_i (1 - x_i)] \le P_c\,.$$

In this way, we complete the conversion from the uncertain programming model's equivalent deterministic model to the original deterministic model. That is to say, the uncertain programming model is the same as the original deterministic model after making the specific conversion, which indicates that the original deterministic model is a special case of the uncertain programming model.

3.5. *Problem example*

In this subsection, we take Fig. 2 as an example and give specific values to each module's parameters to instantiate the model. We take the expected value model (i.e., Eq. (6)) as an example to give the specific calculation process. We assume that (1) the hardware area of each module is a normal uncertain variable, denoted as $\mathcal{N}(\mu, \sigma)$; (2) the execution time of each module is a linear uncertain variable, denoted as $\mathcal{L}(c, d)$ and (3) the power consumption of each module is a zigzag uncertain variable, denoted as $\mathcal{Z}(c, d, e)$. According to uncertainty theory,[26] the expected values of $\mathcal{N}(\mu, \sigma)$, $\mathcal{L}(c, d)$ and $\mathcal{Z}(c, d, e)$ are μ, $(c + d)/2$ and $(c + 2d + e)/4$, respectively.

We make the hardware area of each module meet the following assumptions:

$$\begin{aligned} a_1 &= \mathcal{N}(5, 0.2), \quad a_2 = \mathcal{N}(6, 0.3), \quad a_3 = \mathcal{N}(7, 0.1)\,, \\ a_4 &= \mathcal{N}(9, 0.1), \quad a_5 = \mathcal{N}(8, 0.2), \quad a_6 = \mathcal{N}(10, 0.1)\,. \end{aligned}$$

We make the software execution time and hardware execution time of each module meet the following assumptions:

$$\begin{aligned} t_1^{\mathrm{s}} &= \mathcal{L}(20,28), \quad t_2^{\mathrm{s}} = \mathcal{L}(30,36), \quad t_3^{\mathrm{s}} = \mathcal{L}(50,52), \quad t_4^{\mathrm{s}} = \mathcal{L}(40,45)\,, \\ t_5^{\mathrm{s}} &= \mathcal{L}(50,53), \quad t_6^{\mathrm{s}} = \mathcal{L}(35,38), \quad t_1^{\mathrm{h}} = \mathcal{L}(12,18), \quad t_2^{\mathrm{h}} = \mathcal{L}(8,12)\,, \\ t_3^{\mathrm{h}} &= \mathcal{L}(12,15), \quad t_4^{\mathrm{h}} = \mathcal{L}(10,20), \quad t_5^{\mathrm{h}} = \mathcal{L}(5,8), \quad t_6^{\mathrm{h}} = \mathcal{L}(7,8)\,. \end{aligned}$$

We make the software power consumption and hardware power consumption of each module meet the following assumptions:

$$\begin{aligned} p_1^{\mathrm{s}} &= \mathcal{Z}(12,15,18), \quad p_2^{\mathrm{s}} = \mathcal{Z}(15,19,20), \quad p_3^{\mathrm{s}} = \mathcal{Z}(14,18,20), \quad p_4^{\mathrm{s}} = \mathcal{Z}(16,22,15)\,, \\ p_5^{\mathrm{s}} &= \mathcal{Z}(20,28,30), \quad p_6^{\mathrm{s}} = \mathcal{Z}(18,20,21), \quad p_1^{\mathrm{h}} = \mathcal{Z}(4,7,8), \quad p_2^{\mathrm{h}} = \mathcal{Z}(5,8,10)\,, \\ p_3^{\mathrm{h}} &= \mathcal{Z}(5,7,10), \quad p_4^{\mathrm{h}} = \mathcal{Z}(6,10,12), \quad p_5^{\mathrm{h}} = \mathcal{Z}(8,12,15), \quad p_6^{\mathrm{h}} = \mathcal{Z}(7,10,15)\,. \end{aligned}$$

We suppose that the total execution time limitation T_c is 120 and the total power consumption limitation P_c is 80. After that, the objective of the model is as follows:

$$\text{maximize } 5x_1 + 6x_2 + 7x_3 + 9x_4 + 8x_5 + 10x_6\,,$$

and the constraints of the model are as follows:

$$\begin{aligned} &(24-15)x_1 + (33-10)x_2 + (51-13.5)x_3 + (42.5-15)x_4 + (51.5-6.5)x_5 \\ &\quad + (36.5-7.5)x_6 + (15+10+13.5+15+6.5+7.5) \le 120\,, \end{aligned}$$

$$\begin{aligned} &(15-6.5)x_1 + (18.25-7.75)x_2 + (17.5-7.25)x_3 + (18.75-9.5)x_4 \\ &\quad + (26.5-11.75)x_5 + (19.75-10.5)x_6 + (6.5+7.75+7.25 \\ &\quad + 9.5+11.75+10.5) \le 80\,. \end{aligned}$$

Next, we can *adopt efficient algorithms to solve the model*. We use Algorithm 2 described in Sec. 4 to solve the model and get the solution: $\mathbf{x} = \{0,1,0,0,0,1\}$, which means that modules M_2, M_6 are implemented in software and modules M_1, M_3, M_4, M_5 are implemented in hardware.

4. Algorithm

In this section, we propose the algorithm based on the characteristics of our model. Through the conversion in Sec. 3.3.3, the hardware/software partitioning problem has been transformed into a two-dimensional 0-1 knapsack problem, which has one more constraint than the traditional 0-1 knapsack. We come up with the custom genetic algorithm to settle this problem, by incorporating a greedy idea in the genetic algorithm.

4.1. *Greedy idea*

The greedy idea always makes the current best choice when solving problems.[28] For the traditional 0-1 knapsack problem, each item's profit-to-weight ratio is used as the basis for the greedy algorithm to fill the knapsack. In each iteration, the item with the largest profit-to-weight ratio is filled into the knapsack until the knapsack is full or there is no item that is suitable for the knapsack's remaining capacity. Considering that our hardware/software partitioning problem is an extension of the traditional 0-1 knapsack problem, the greedy idea for the 0-1 knapsack problem can be extended to solve it.[27]

In order to simplify the constraints' number and normalize the constraints, the vector representation method is used, which defines $\vec{v} = \langle 1, 1\rangle$ and $\vec{\phi}_i = \langle \phi_{ti}/T'_c, \phi_{pi}/P'_c\rangle$ for $i = 1, 2, \ldots, n$. This method can be used for multiple constraints, not just two. Then, the final equivalent deterministic model B_1 (i.e., Eq. (9)) can be reformulated as follows:

$$\begin{aligned}
&\text{maximize} \quad && \sum_{i=1}^{n} \phi_{ai} x_i \,,\\
&\text{subject to} \quad && \sum_{i=1}^{n} \vec{\phi_i} x_i \le \vec{v}\,,\\
& && x_i \in \{0,1\}, \quad i = 1, 2, \ldots, n\,.
\end{aligned}$$

The profit-to-weight ratio of each item i here can be defined as $\phi_{ai}/|\vec{\phi}_i|$, where $|\vec{\phi}_i|$ is denoted as follows:

$$\sqrt{\left(\frac{\phi_{ti}}{T'_c}\right)^2 + \left(\frac{\phi_{pi}}{P'_c}\right)^2}\,.$$

Then, based on the greedy idea, the item with the largest $\phi_{ai}/|\vec{\phi}_i|$ can be packed into the knapsack one by one if constraint $\sum_{i=1}^{n} \vec{\phi}_i x_i \le \vec{v}$ is satisfied.

4.2. *The custom genetic algorithm*

We use the genetic algorithm to solve the model for getting the approximate solution. In the process of implementing the genetic algorithm, we add the above greedy idea in two steps to optimize it.

(i) We use the greedy idea to generate an initial solution and add this solution to the initial population of the genetic algorithm to speed up the convergence of the genetic algorithm.

(ii) For the individual who does not meet the constraints (i.e., invalid solution) in the population of each generation, we use the greedy idea to modify it. There are many strategies when combining the genetic algorithm with the greedy idea. Reference 29 has studied different strategies in detail, and we choose the best strategy it proposes. Specifically, for this kind of individual, we take out the items that have been packed into the knapsack in ascending order of the profit-to-weight ratio, until the constraints are met, as shown in Algorithm 1.

Algorithm 1. The Solution Modification Algorithm

```
Input: time limitation $T_c'$, power consumption limitation $P_c'$, all relevant parameters of each module, original solution $\mathbf{x_o}(x_1, x_2, \ldots, x_n)$
Output: modified solution $\mathbf{x_m}(x_1, x_2, \ldots, x_n)$
1: $\mathbf{x_m} \leftarrow \mathbf{x_o}$;
2: $OverpackedFlag \leftarrow 0$;
3: Calculate $Time$ and $Power$ of $\mathbf{x_m}$;
4: if ($Time > T_c'$ or $Power > P_c'$) then
5:     $OverpackedFlag \leftarrow 1$;
6: end if
7: while ($OverpackedFlag$) do
8:     Remove the item $i$ with the smallest $\frac{\phi_{ai}}{|\vec{\phi}_i|}$ from the knapsack;
9:     $x_i \leftarrow 0$;
10:    Calculate $Time$ and $Power$ of $\mathbf{x_m}$;
11:    if ($Time \leq T_c'$ and $Power \leq P_c'$) then
12:        $OverpackedFlag \leftarrow 0$;
13:    end if
14: end while
15: return $\mathbf{x_m}$
```

More details about the custom genetic algorithm are presented in Algorithm 2.

Lines 1−2 initialize the parameters of the hardware/software partitioning problem and set the used time *Time* and used power *Power* of the greedy idea to 0.

Line 3 sorts each item i in descending order of the profit-to-weight ratio $\phi_{ai}/|\vec{\phi}_i|$ for satisfying the greedy strategy.

Lines 4−13 generate the greedy solution $\mathbf{x}'$ according to the greedy idea described above.

Line 14 mixes the greedy solution individual $\mathbf{x}'$ and random 0-1 sequences individuals to generate the first population P_o. Then, the genetic algorithm is applied to generate the approximate optimal solution $\mathbf{x}$.

Lines 15−16 generate an initial solution to the problem.

Line 17 determines whether the termination conditions of the propagation are met, such as generation numbers and convergence.

Lines 18−24 generate a new population P_o' through selection, crossover and mutation operations on the original population P_o, where the strategies for selection, crossover and mutation are roulette wheel selection, two-points crossover and one-point mutation, respectively.

Lines 25−31 update the current solution and ensure that the current optimal solution exists in the next population. In the end, the algorithm will return an approximate optimal solution $\mathbf{x}$.

Algorithm 2. The Custom Genetic Algorithm

```
Input: time limitation $T'_c$, power consumption limitation $P'_c$, all relevant parameters of each module
Output: solution $\mathbf{x}(x_1, x_2, \ldots, x_n)$
1: Encode the hardware/software partitioning problem's parameters;
2: Initialize $Time$ and $Power$ as 0;
3: Sort all items to make $\frac{\phi_{a1}}{|\vec{\phi}_1|} \geq \frac{\phi_{a2}}{|\vec{\phi}_2|} \geq \cdots \geq \frac{\phi_{an}}{|\vec{\phi}_n|}$;
4: for ($i \leftarrow 1; i \leq n; i{+}{+}$) do
5:     if ($Time + \frac{\phi_{ti}}{T'_c} \leq 1$ and $Power + \frac{\phi_{pi}}{P'_c} \leq 1$) then
6:         $x_i \leftarrow 1$;
7:         $Time \leftarrow Time + \frac{\phi_{ti}}{T'_c}$;
8:         $Power \leftarrow Power + \frac{\phi_{pi}}{P'_c}$;
9:     else
10:        $x_i \leftarrow 0$;
11:    end if
12: end for
13: Generate the greedy solution $\mathbf{x}'$;
14: Generate the first population $P_o$ with the greedy solution individual and random 0-1 sequences individuals;
15: Compute the fitness of each individual in $P_o$, where every individual has been modified by Algorithm 1;
16: Record the individual with the highest fitness as the solution $\mathbf{x}$ of the problem;
17: while (termination conditions) do
18:     Clear new population $P'_o$;
19:     while (size of $P'_o$ < size of $P_o$) do
20:         Perform selection in $P_o$ to generate two parents $p_1$ and $p_2$;
21:         Perform crossover on $p_1$ and $p_2$ by probability $crossover\_rate$ to generate two offsprings $o_1$ and $o_2$;
22:         Perform mutation on $o_1$ and $o_2$ by probability $mutation\_rate$ to generate individuals $o'_1$ and $o'_2$;
23:         Put individuals $o'_1$ and $o'_2$ into new population $P'_o$;
24:     end while
25:     Find the individual $o'_f$ with the highest fitness in $P'_o$, where every individual has been modified by Algorithm 1;
26:     if (fitness($o'_f$) > fitness($\mathbf{x}$)) then
27:         $\mathbf{x} \leftarrow o'_f$;
28:     else
29:         $o'_f \leftarrow \mathbf{x}$;
30:     end if
31:     $P_o \leftarrow P'_o$;
32: end while
33: return $\mathbf{x}$
```

5. Experimental Results

Considering that we have given a detailed example of the expected value model in Sec. 3.5, in this section, we take the chance-constrained programming model (i.e., Eq. (7)) as an example to conduct experiments. First, considering that we propose the custom genetic algorithm in Sec. 4, to verify the performance of this algorithm, we compare it with the exact algorithm. Second, considering that the chance-constrained programming model contains a lot of parameters, we study the influence of model parameters on the partitioning results. All codes of the experiments are written in Python and run on an Intel i5 computer of 2.9 GHz.

5.1. *Module parameter settings*

Without loss of generality, we use randomly generated instances in our experiments. We give the settings of each module's relevant parameters and system performance limitations in this subsection. In order to present different types of uncertainty distributions, we make different performances of each module follow different uncertainty distributions. Similar to Sec. 3.5, we assume that the hardware area, execution time and power consumption of each module are normal uncertain variable, linear uncertain variable and zigzag uncertain variable, respectively, denoted as $\mathcal{N}(\mu,\sigma)$, $\mathcal{L}(c,d)$ and $\mathcal{Z}(c,d,e)$, respectively. According to uncertainty theory,[26] the inverse uncertainty distribution of normal uncertain variable $\mathcal{N}(\mu,\sigma)$ is

$$\Phi^{-1}(\alpha) = \mu + \frac{\sigma\sqrt{3}}{\pi}\ln\frac{\alpha}{1-\alpha}\,,$$

and the inverse uncertainty distribution of linear uncertain variable $\mathcal{L}(c,d)$ is

$$\Phi^{-1}(\alpha) = (1-\alpha)c + \alpha b\,,$$

and the inverse uncertainty distribution of zigzag uncertain variable $\mathcal{Z}(c,d,e)$ is

$$\Phi^{-1}(\alpha) = \begin{cases} (1-2\alpha)c + 2\alpha d & \text{if } \alpha < 0.5\,,\\ (2-2\alpha)d + (2\alpha-1)e & \text{if } \alpha \geq 0.5\,. \end{cases}$$

For each module, the parameters need to meet the real situation; that is, the execution time and power consumption of hardware implementation are less than that of software implementation for the same module. We use the following rules to generate each module's parameters; these rules are consistent with Ref. 27 and refer to the corresponding settings in Refs. 18, 20, while meeting that $T'_c \geq 0$ and $P'_c \geq 0$.

- μ_{ai} is randomly generated from [0, 100] and σ_{ai} is randomly generated from [0, 5].
- c^{s}_{ti} is randomly generated from [0, 30] and d^{s}_{ti} is randomly generated from $[c^{\mathrm{s}}_{ti}$, 30]; c^{h}_{ti} is randomly generated from $[0, \lambda \cdot c^{\mathrm{s}}_{ti}]$ and d^{h}_{ti} is randomly generated from $[c^{\mathrm{h}}_{ti}$, $\lambda \cdot d^{\mathrm{s}}_{ti}]$, where $0 < \lambda < 1$ (taking 0.5 in the experiment).

- c^{s}_{pi} is randomly generated from $[0, 20]$, d^{s}_{pi} is randomly generated from $[c^{\mathrm{s}}_{pi}, 20]$, e^{s}_{pi} is randomly generated from $[d^{\mathrm{s}}_{pi}, 20]$; c^{h}_{pi} is randomly generated from $[0, \lambda \cdot c^{\mathrm{s}}_{pi}]$, d^{h}_{pi} is randomly generated from $[c^{\mathrm{h}}_{pi}, \lambda \cdot d^{\mathrm{s}}_{pi}]$ and e^{h}_{pi} is randomly generated from $[d^{\mathrm{h}}_{pi}, \lambda \cdot e^{\mathrm{s}}_{pi}]$, where $0 < \lambda < 1$ (taking 0.5 in the experiment).
- T_c is generated from $[\sum_{i=1}^{n} \Phi^{\mathrm{h}^{-1}}_{ti}(\alpha_t), \sum_{i=1}^{n} \Phi^{\mathrm{s}^{-1}}_{ti}(\alpha_t)]$ and P_c is generated from $[\sum_{i=1}^{n} \Phi^{\mathrm{h}^{-1}}_{pi}(\alpha_p), \sum_{i=1}^{n} \Phi^{\mathrm{s}^{-1}}_{pi}(\alpha_p)]$, which can guarantee that $T'_c \geq 0$ and $P'_c \geq 0$.

5.2. *Algorithm comparison*

In this subsection, we investigate the performance of the custom genetic algorithm, by comparing the results of our approximate algorithm with the exact algorithm. The exact algorithm we use is dynamic programming, which is a classical algorithm for finding the exact solution to the 0-1 knapsack problem and can be extended to solve the two-dimensional 0-1 knapsack problem.

Considering that our model introduces confidence level to both the objective and the constraints, the parameters of the model are not integers. For ease of calculation, we round down all parameters of Eq. (9) in this experiment. Then, we solve the model by using the custom genetic algorithm and dynamic programming, respectively. For the dynamic programming, we define $A(k, \phi_t, \phi_p)$ as the maximum value of $\sum_{i=1}^{k} \phi_{ai} x_i$ when only using the first k items, while satisfying that $\sum_{i=1}^{k} \phi_{ti} x_i \leq T'_c$ and $\sum_{i=1}^{k} \phi_{pi} x_i \leq P'_c$. Then, we have the following recursive relation for $A(k, \phi_t, \phi_p)$: if $\phi_{tk} > T_c$ or $\phi_{pk} > P_c$,

$$A(k, \phi_t, \phi_p) = A(k-1, \phi_t, \phi_p)\,,$$

else,

$$A(k, \phi_t, \phi_p) = \max \left\{ \begin{array}{c} A(k-1, \phi_t, \phi_p)\,, \\ A(k-1, \phi_t - \phi_{tk}, \phi_p - \phi_{pk}) + \phi_{ak} \end{array} \right\}.$$

First, we verify the quality of the approximate solution obtained by our algorithm by comparing it with the dynamic programming solution on small inputs. We have conducted experiments on cases where the number of modules is 25, 50, 100, 150 and 200, respectively. For each case, we generate 50 instances. For each instance, we randomly generate relevant parameters of each module and the performance limitations of the system according to the rules described in Sec. 5.1. In this experiment, we set $\beta = 0.8$, $\alpha_t = 0.9$ and $\alpha_p = 0.9$. Then, we, respectively, use dynamic programming (DP) and the custom genetic algorithm (CGA) to solve the model, and we record their results. For each instance, we denote A_h as the hardware area corresponding to the case that all modules are implemented in hardware, and denote A_s as the hardware area corresponding to the solution obtained by the algorithm. We record $\frac{A_s}{A_h}$ results of the two algorithms in Table 2, where δ is the solution error

Table 2. $\frac{A_s}{A_h}$ results and solution errors δ.

n	DP (%)	CGA (%)	δ (%)
25	47.59	48.01	0.42
50	46.46	46.98	0.52
100	46.33	46.84	0.51
150	44.26	44.80	0.54
200	40.91	41.43	0.52

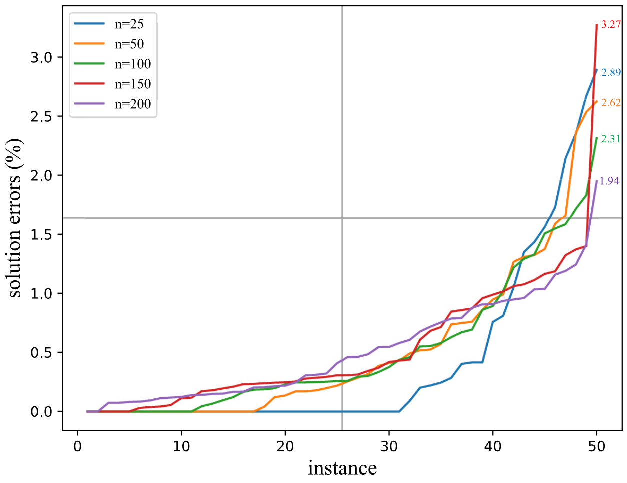

Fig. 3. Solution errors corresponding to different numbers of modules.

between the approximate solution and the exact solution. The values in Table 2 are the average results of 50 instances.

It can be seen from the table that for different module numbers, the errors between our algorithm and dynamic programming are very small, indicating that our algorithm can obtain a high-quality approximate solution. In addition, in order to further observe the quality of the approximate solution, we, respectively, show the solution errors of 50 instances corresponding to different numbers of modules. For easy observation, we sort the results and display them in Fig. 3.

It can be seen from Fig. 3 that the solution error of each instance is very small. When the number of modules is 25, 50, 100, 150 and 200, the corresponding maximum solution errors are 2.89%, 2.62%, 2.31%, 3.27% and 1.94%, respectively. Meanwhile, we can see that the custom genetic algorithm has the ability to obtain the optimal solution (i.e., the solution error is 0). For example, when the number of modules is 25, the algorithm obtains the optimal solution in more than half of the instances.

Second, we compare the running time of the two algorithms on different input scales. We use the above parameter settings to conduct experiments on cases where

Table 3. The average running time on different input scales.

n	DP (s)	CGA (s)
50	6.64	60.79
100	59.28	62.50
200	618.09	66.46
500	5822.68	80.22
1000	—	103.67
2000	—	194.93
5000	—	533.68

the number of modules is 50, 100, 200, 500, 1000, 2000 and 5000, respectively. We record the running time of the two algorithms, and the values in Table 3 are the average results of 50 instances. As the input scale increases, the running time of dynamic programming increases extremely, while the running time of our algorithm grows slowly. For large input scales (in this experiment it refers to the input scale exceeding 1000), dynamic programming cannot solve the problem due to lack of memory, denoted as "—" in the table, while our algorithm can solve the problem in 500 s.

As mentioned above, we compare our algorithm with the exact algorithm dynamic programming in terms of solution error and running time. We can see that (1) our approximate solution has little error from the exact solution; (2) with the increase of the input scale, the running time of the dynamic programming method increases dramatically, while our algorithm can solve the problem in a relatively short time.

5.3. *Influence of parameters*

In this subsection, we study the influence of model parameters on the partitioning results. It can be seen from the model (i.e., Eq. (7)) in Sec. 3.3 that for a given system, under the condition that each module is determined (i.e., all modules' relevant parameters are known) and the system performance limitations are determined (i.e., T_c and P_c are known), the partitioning result is affected by confidence levels β, α_t and α_p.

We consider a system with 30 modules, and we randomly generate all modules' relevant parameters according to the rules described in Sec. 5.1. In this case, system's each module is determined. We study: (1) the influence of the objective's confidence level β on the partitioning result; (2) the influence of the constraints' confidence levels α_t and α_p on the partitioning result. For the system performance limitations T_c and P_c, we randomly generate them according to the characteristics of each problem, which ensures that $T_c' \geq 0$ and $P_c' \geq 0$ can be satisfied in all conditions.

First, we study the impact of objective's confidence level β on the partitioning results. We let α_t and α_p both be 0.9, and we take β as 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,

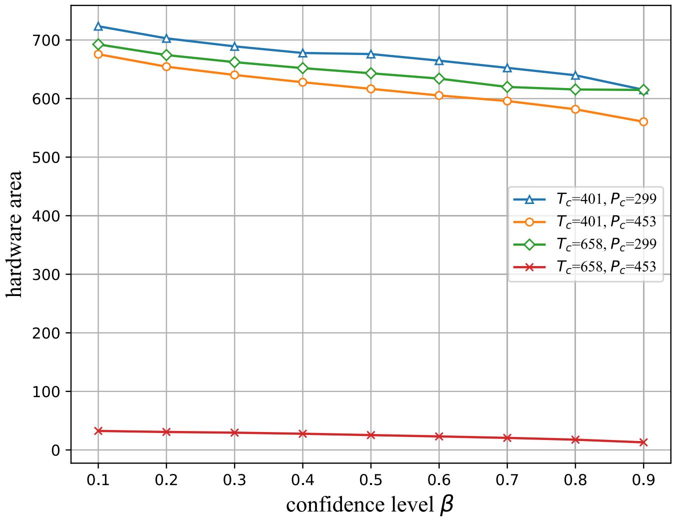

Fig. 4. Results for different β, corresponding to different T_c and P_c.

0.8 and 0.9, respectively, to calculate the hardware area. In this problem, T_c is generated from $[\sum_{i=1}^{n} \Phi_{ti}^{\mathrm{h}^{-1}}(0.9), \sum_{i=1}^{n} \Phi_{ti}^{\mathrm{s}^{-1}}(0.9)]$ and P_c is generated from $[\sum_{i=1}^{n} \Phi_{pi}^{\mathrm{h}^{-1}}(0.9), \sum_{i=1}^{n} \Phi_{pi}^{\mathrm{s}^{-1}}(0.9)]$. For each case, we use Algorithm 2 to solve the model and get the results; we select four sets of representative data and show them in Fig. 4. For different system limitations T_c and P_c, it can be seen that under the same constraints, the hardware area gradually decreases as β increases (i.e., we get more optimized results). There has a theorem in uncertain programming,[13] that is, for an uncertain variable γ, its β-optimistic value is a decreasing and left-continuous function of β. Therefore, $\gamma_{\sup}(\beta)$ (i.e., $\Phi_{ai}^{-1}(1-\beta)$) is a decreasing function of β, which is consistent with our experimental results. At the same time, we find that (1) for the same T_c, the hardware area decreases as P_c increases; (2) for the same P_c, the hardware area decreases as T_c increases. This is reasonable, because increasing the system performance limitations can give more space for minimization, so we can get more optimized results. For the red line in Fig. 4, it can be seen that its corresponding T_c and P_c are both the largest, which gives enough space for optimization, so it gets the best results compared to other three lines.

Secondly, we study the impact of constraints' confidence levels α_t and α_p on the partitioning results. For α_t and α_p, we make the following settings, respectively.

- $\boldsymbol{\alpha_t}$**:** We let β be 0.8, let α_p be 0.9 and take α_t as 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 and 0.9, respectively, to calculate the hardware area.
- $\boldsymbol{\alpha_p}$**:** We let β be 0.8, let α_t be 0.9 and take α_p as 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 and 0.9, respectively, to calculate the hardware area.

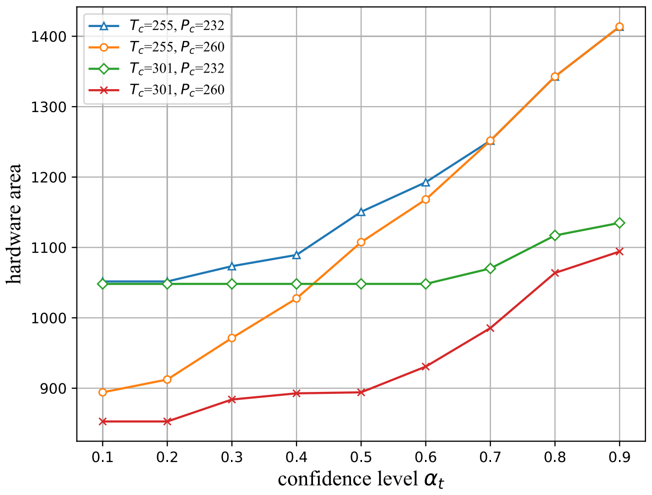

Fig. 5. Results for α_t, corresponding to different T_c and P_c.

In this problem, T_c is generated from $[\sum_{i=1}^{n} \Phi_{ti}^{\mathrm{h}^{-1}}(0.9), \sum_{i=1}^{n} \Phi_{ti}^{\mathrm{s}^{-1}}(0.1)]$ and P_c is generated from $[\sum_{i=1}^{n} \Phi_{pi}^{\mathrm{h}^{-1}}(0.9), \sum_{i=1}^{n} \Phi_{pi}^{\mathrm{s}^{-1}}(0.1)]$. For each case, Algorithm 2 is used to solve the model and get the results; for α_t and α_p, we select four sets of representative data, respectively, and present them in Figs. 5 and 6.

From Fig. 5, we can see that under the same constraints, the hardware area gradually increases as α_t increases. This is reasonable because a higher confidence level α_t means that the time constraint should be satisfied to a greater extent; that is,

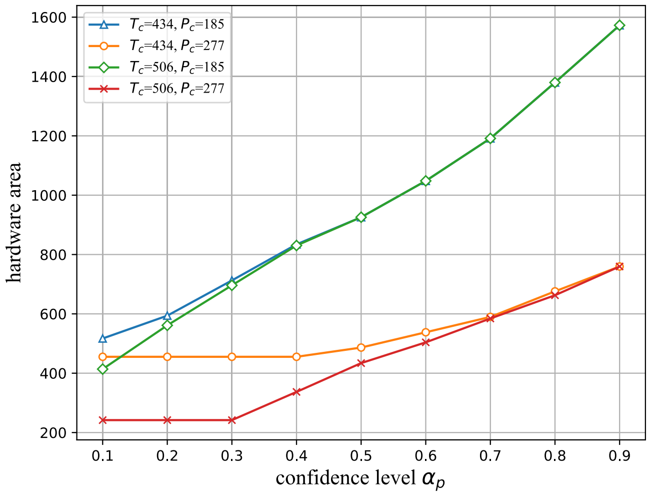

Fig. 6. Results for α_p, corresponding to different T_c and P_c.

the time constraint is stricter, which gives less space for minimization. In the same way, the hardware area decreases with the increases of T_c and P_c. We find that the blue line and the orange line overlap when α_t is 0.7, 0.8 and 0.9, indicating that P_c is already large enough in this case; at this time, it is the value of T_c that affects the result, and increasing the value of P_c will not change the result. We also find that the result corresponding to the green line remains the same when α_t goes from 0.1 to 0.6. Because T_c and P_c are large enough at this time, providing enough space for optimization, the stricter constraints brought by increasing α_t will not affect the current optimal solution. From Fig. 6, we can observe similar results with Fig. 5: (1) under the same constraints, the hardware area gradually increases as α_p increases; (2) the hardware area decreases with the increase of T_c and P_c and (3) the blue line and the green line overlap when α_p is larger than 0.4, which indicates that T_c is already large enough in this case.

6. Conclusion

In this paper, we present the process of applying uncertain programming to solve the hardware/software partitioning problem, including the model and algorithm. We elaborate on the detailed process, including that: (1) employing uncertain variables to represent parameters; (2) converting the uncertain programming model into the equivalent deterministic model and (3) solving the model. Our conversion methods cover different forms of uncertain variables (such as linear, zigzag and normal) and equivalent deterministic models (such as the expected value model and the chance-constrained programming model). Our custom genetic algorithm can find a high-quality approximate solution while running much faster for large input scales, compared with the exact algorithm. Our partitioning results become better as the objective confidence level increases and become worse as the constraint confidence level increases.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant Nos. 61972139, 61932010 and 62133014.

References

1. A. Ouyang, X. Peng, J. Liu and A. Sallam, Hardware/software partitioning for heterogenous mpsoc considering communication overhead, *Int. J. Parallel Program.* **45** (2017) 899–922.
2. S. Banerjee, E. Bozorgzadeh and N. D. Dutt, Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **14** (2006) 1189–1202.
3. W. Shi, J. Wu, G. Jiang and S. Lam, Multiple-choice hardware/software partitioning for tree task-graph on mpsoc, *Comput. J.* **63** (2019) 688–700.

4. J. Wu and T. Srikanthan, Low-complex dynamic programming algorithm for hardware/software partitioning, *Inf. Process. Lett.* **98** (2006) 41–46.
5. W. Jigang, B. Chang and T. Srikanthan, A hybrid branch-and-bound strategy for hardware/software partitioning, *Proc. 2009 Eighth IEEE/ACIS Int. Conf. Computer and Information Science* (2009), pp. 641–644.
6. K. S. Chatha and R. Vemuri, Hardware-software partitioning and pipelined scheduling of transformative applications, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **10** (2002) 193–208.
7. Z. A. Halim, B. S. Babu and M. Mustaffa, Hardware software partitioning using four levels hybrid algorithm technique, *Proc. 2020 IEEE 10th Symp. Computer Applications Industrial Electronics (ISCAIE)* (2020), pp. 42–47.
8. A. Iguider, K. Bousselam, O. Elissati, M. Chami and A. En-Nouaary, Heuristic algorithms for multi-criteria hardware/software partitioning in embedded systems codesign, *Comput. Electr. Eng.* **84** (2020) 106610.
9. N. Govil, R. Shrestha and S. R. Chowdhury, Pgma: An algorithmic approach for multi-objective hardware software partitioning, *Microprocess. Microsyst.* **54** (2017) 83–96.
10. J. W. Tang, Y. W. Hau and M. Marsono, Hardware/software partitioning of embedded system-on-chip applications, *Proc. 2015 IFIP/IEEE Int. Conf. Very Large Scale Integration (VLSI-SoC)* (2015), pp. 331–336.
11. N. Hou, F. He, Y. Zhou and Y. Chen, An efficient gpu-based parallel tabu search algorithm for hardware/software co-design, *Front. Comput. Sci.* **14** (2020) 1–18.
12. W. Shi, J. Wu, S. Kei Lam and T. Srikanthan, Algorithms for bi-objective multiple-choice hardware/software partitioning, *Comput. Electr. Eng.* **50** (2016) 127–142.
13. B. Liu, *Theory and Practice of Uncertain Programming* (Springer-Verlag, Berlin, Heidelberg, 2009).
14. H. Dalman, Uncertain programming model for multi-item solid transportation problem, *Int. J. Mach. Learn. Cybern.* **9** (2018) 559–567.
15. B. Zhang and J. Peng, Uncertain programming model for uncertain optimal assignment problem, *Appl. Math. Model.* **37** (2013) 6458–6468.
16. S. Majumder, M. B. Kar, S. Kar and T. Pal, Uncertain programming models for multi-objective shortest path problem with uncertain parameters, *Soft Comput.* **24** (2020) 8975–8996.
17. Y. Jiang, H. Zhang, X. Jiao, X. Song, W. N. N. Hung, M. Gu and J. Sun, Uncertain model and algorithm for hardware/software partitioning, *Proc. 2012 IEEE Computer Society Annual Symp. VLSI* (2012), pp. 243–248.
18. Y. Jiang, M. Wang, X. Jiao, H. Song, H. Kong, R. Wang, Y. Liu, J. Wang and J. Sun, Uncertainty theory based reliability-centric cyber-physical system design, *Proc. 2019 Int. Conf. Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (2019), pp. 208–215.
19. R. Wang, W. N. N. Hung, G. Yang and X. Song, Uncertainty model for configurable hardware/software and resource partitioning, *IEEE Trans. Comput.* **65** (2016) 3217–3223.
20. S. Chen, G. Xie, R. Li and K. Li, Uncertainty theory based partitioning for cyber-physical systems with uncertain reliability analysis, *ACM Trans. Design Autom. Electron. Syst.* **27** (2021) 1–19.
21. G. Xie, G. Zeng, J. Jiang, C. Fan, R. Li and K. Li, Energy management for multiple real-time workflows on cyber–physical cloud systems, *Future Generat. Comput. Syst.* **105** (2020) 916–931.

22. G. Xie, G. Zeng and R. Li, Safety enhancement for real-time parallel applications in distributed automotive embedded systems: A stable stopping approach, *IEEE Trans. Parallel Distrib. Syst.* **31** (2020) 2067–2080.
23. G. Xie, G. Zeng, X. Xiao, R. Li and K. Li, Energy-efficient scheduling algorithms for real-time parallel applications on heterogeneous distributed embedded systems, *IEEE Trans. Parallel Distrib. Syst.* **28** (2017) 3426–3442.
24. Y. Liu, G. Xie, X. Chen, L. Jin, Y. Tang and R. Li, An active scheduling policy for automotive cyber-physical systems, *J. Syst. Architect.* **97** (2019) 208–218.
25. G. Xie, G. Zeng, L. Liu, R. Li and K. Li, High performance real-time scheduling of multiple mixed-criticality functions in heterogeneous distributed embedded systems, *Syst. Architect.* **70** (2016) 3–14.
26. B. Liu, *Uncertainty Theory* (Springer-Verlag, Berlin, Heidelberg, 2010).
27. W. Jigang and T. Srikanthan, Efficient algorithms for hardware/software partitioning to minimize hardware area, *Proc. APCCAS 2006 - 2006 IEEE Asia Pacific Conf. Circuits and Systems* (2006), pp. 1875–1878.
28. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction To Algorithms*, 2nd edn. (MIT Press, Cambridge, MA, 2001).
29. Z. Michalewicz and J. Arabas, Genetic algorithms for the 0/1 knapsack problem, *Proc. 8th Int. Symp. Methodologies for Intelligent Systems* (1994), pp. 134–143.