Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2014, Article ID 103059, 15 pages
http://dx.doi.org/10.1155/2014/103059

Research Article

A Tabu Search-Based Memetic Algorithm for Hardware/Software Partitioning

Geng Lin,[1] Wenxing Zhu,[2] and M. Montaz Ali[3,4]

[1] *Department of Mathematics, Minjiang University, Fuzhou 350108, China*
[2] *Center for Discrete Mathematics and Theoretical Computer Science, Fuzhou University, Fuzhou 350108, China*
[3] *School of Computational and Applied Mathematics, Faculty of Science, University of the Witwatersrand, (Wits), Johannesburg 2050, South Africa*
[4] *TCSE, Faculty of Engineering and Build Environment, University of the Witwatersrand, (Wits), Johannesburg 2050, South Africa*

Correspondence should be addressed to Geng Lin; lingeng413@163.com

Received 24 March 2014; Revised 16 June 2014; Accepted 16 June 2014; Published 13 July 2014

Academic Editor: Jiaji Wu

Hardware/software (HW/SW) partitioning is to determine which components of a system are implemented on hardware and which ones on software. It is one of the most important steps in the design of embedded systems. The HW/SW partitioning problem is an NP-hard constrained binary optimization problem. In this paper, we propose a tabu search-based memetic algorithm to solve the HW/SW partitioning problem. First, we convert the constrained binary HW/SW problem into an unconstrained binary problem using an adaptive penalty function that has no parameters in it. A memetic algorithm is then suggested for solving this unconstrained problem. The algorithm uses a tabu search as its local search procedure. This tabu search has a special feature with respect to solution generation, and it uses a feedback mechanism for updating the tabu tenure. In addition, the algorithm integrates a path relinking procedure for exploitation of newly found solutions. Computational results are presented using a number of test instances from the literature. The algorithm proves its robustness when its results are compared with those of two other algorithms. The effectiveness of the proposed parameter-free adaptive penalty function is also shown.

1. Introduction

The embedded systems have become omnipresent in a wide variety of applications and typically consist of application specific hardware components and programmable components. With the growing complexity of embedded systems, hardware/software codesign has become an effective way of improving design quality, in which the HW/SW partitioning is the most critical step.

HW/SW partitioning decides which tasks of an embedded system should be implemented in hardware and which ones should be in software. A task implemented with a hardware module is faster but more expensive, while a task implemented with a software module is slower but cheaper. For this reason, the main target of HW/SW partitioning is to balance all the tasks to optimize some objectives of the system under some constraints [1].

In recent years, there has been an increasing interest in the study of the HW/SW partitioning problem. Several exact approaches [2–5] to the problem have been developed. However, since most of the exact formulations of the HW/SW partitioning problem are NP-hard [6], the practical usefulness of these approaches is limited to fairly small problem instances only. For larger instances, a number of heuristic algorithms, both traditional and general purpose, have been proposed.

There are two traditional families of heuristics: the software-oriented heuristic and the hardware-oriented heuristic. The software-oriented heuristic starts with a complete software solution and parts of the system are migrated to hardware until all constraints are satisfied. On the other hand, the hardware-oriented heuristic starts with a complete hardware solution and moves parts of the system to software until a constraint is violated.

Up to now, many general-purpose heuristics or metaheuristics have also been applied to solve the HW/SW partitioning problem. These include simulated annealing [7–9], genetic algorithm [10–12], tabu search [8, 13–15], artificial immune [16], and the Kernighan-Lin heuristic [17, 18]. There are also some other methods for solving the problem; see [19] for details.

It must be mentioned that the HW/SW partitioning problem is to minimize a cost while satisfying some design constraints. In fact, the problem is a discrete constrained optimization problem, and the optimal solutions usually lie on the boundary of the feasible region. Hence, it is necessary to develop effective techniques in handling the constraints.

This paper presents a tabu search-based memetic algorithm (TSMA) for solving the HW/SW partitioning problem. The TSMA uses an adaptive penalty function to convert the original problem into an equivalent unconstrained integer programming problem; both problems have the same discrete global minimizers. The TSMA is then applied to unconstrained integer programming problem in order to solve the original problem. The TSMA integrates a tabu search procedure, a path relinking procedure, and a population updating strategy. These strategies achieve a balance between intensification and diversification of the search process. The algorithm has been tested on four benchmark test instances in the literature. Experimental results and comparisons show that the proposed algorithm is effective and robust.

The remaining part of this paper is organized as follows. Section 2 gives the system model and problem formulation. Section 3 presents the parameter-free adaptive penalty function used in converting the original problem into the unconstrained problem. Section 3 also presents some theoretical properties of the resulting unconstrained problem. In Section 4, a full description of TSMA is presented. Experimental results on four benchmark test instances are presented in Section 5. Final remarks are given in Section 6.

2. Hardware/Software Partitioning Model and Formulation

The HW/SW partitioning model discussed in this paper is similar to that presented in [16]. In order to make the paper self-contained, this section briefly reviews the description of the HW/SW partitioning model considered in [16].

2.1. Function Description. The main function of the system is usually described by a high-level programming language, and then the function will be mapped into a control-data flow graph (CDFG). The CDFG is composed of nodes and arcs [20, 21]. Generally, the CDFG is a directed acyclic graph (DAG). In the DAG, the nodes are determined by the model granularity, that is, the semantic of a node. A node can represent a short sequence of instructions, a basic block, a function, or a procedure [12]. The arcs between nodes denote their relationship. Each node in DAG can receive data from its previous nodes and can send data to its next nodes.

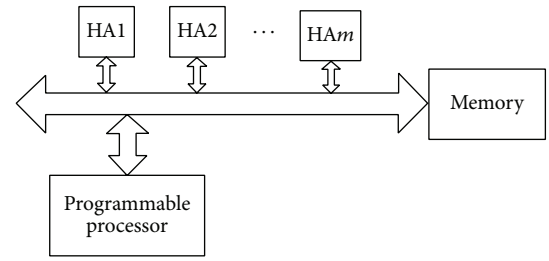

Figure 1: The partition model discussed in this paper [16].

2.2. Binary Integer Programming Formulation. In a directed acyclic graph (DAG) $G = (V, E)$ with node set $V = \{1, 2, \ldots, n\}$ and arc set E, every node is labeled with several attributes. The attributes for a node i are defined as follows.

(i) as_i denotes the cost of i in software implementation.

(ii) ts_i denotes the executing time of i in software implementation.

(iii) ah_i denotes the cost of i in hardware implementation.

(iv) th_i denotes the executing time of i in hardware implementation.

(v) r_i is an array, which denotes the set of incoming nodes of i and stores the set of corresponding communication times.

(vi) w_i is an array, which denotes the set of outgoing nodes of i and stores the set of corresponding communication times.

(vii) pc_i denotes the number of times that node i is executed.

Figure 1 [16] shows the partition model used in this paper, where "HA1", "HA2", ..., and "HAm" denote the m hardware nodes implemented by application specific integrated circuit (ASIC) or field programmable gate array (FPGA). The software nodes are executed in a programmable processor (CPU). All the nodes exchange data through a shared bus, and they share the common memory to store the interim data. All the hardware nodes without dependency can be executed concurrently [16].

After all nodes are determined to be implemented by hardware or software, all the nodes are scheduled. The total executing time T can be evaluated by the list schedule algorithm [11, 22], and the cost of the design can be evaluated. The detailed steps of the list schedule algorithm are shown in Algorithm 1.

In this paper, the objective is to minimize the total cost of the system under the time constraint. Since the cost of software is usually negligible, the HW/SW partitioning problem can be described as follows [11]:

$$\begin{aligned} \min \quad & \sum_{i \in H} ah_i \\ \text{s.t.} \quad & T \leq \text{TimeReq}, \end{aligned} \tag{P}$$

(1) Initialization. Set two attributes for every node, *start* and *finish*, which are used for recording the node's starting time and finishing time. Choose all nodes without data dependency from the node set and put them into a ready queue. Choose one software node from the queue to be executed by CPU, and all hardware nodes to be executed at the same time. Set 0 as the values of the *start* attributes for the running nodes. Set the sum of the executing time and the communication time as the values of the *finish* attributes. If there is a software node being executed by *CPU*, set *CPU* = *BUSY*.
(2) **repeat**
(3) Select the node with the minimal *finish* time from the ready queue, mark it as i.
(4) Remove node i from the queue. If i is a software node, set *CPU* = *FREE*.
(5) If *CPU* = *FREE*, choose one software node j from the queue to execute and set *CPU* = *BUSY*. Set $start[j] = finish[i]$, $finish[j] = start[j] + ts_j + comm[j]$, where $comm[j]$ denotes the communication time of node j with other nodes.
(6) Choose all hardware nodes to execute concurrently. Set their *start* and *finish* attributes as in Line (5).
(7) **until** The ready queue is empty
(8) **return** Choose the maximal *finish* time of all the nodes as the system executing time, denoted as T.

Algorithm 1: List schedule algorithm [16, 22].

where $i \in H$ means that node i is implemented by hardware, T denotes the total executing time, and TimeReq denotes the required time which is given in advance by the designer.

Let $x = (x_1, \ldots, x_n)^T \in \{0,1\}^n$ denote a solution of the problem (P). $x_i = 1$ ($x_i = 0$) indicates that node i is implemented by hardware (software). Then problem (P) can be formulated as the following constrained binary programming problem:

$$\begin{aligned} \min \quad & f(x) = \sum_{i=1}^{n} x_i ah_i \\ \text{s.t.} \quad & t(x) \le \text{TimeReq} \\ & x_i \in \{0,1\}, \quad i = 1, \ldots, n, \end{aligned} \tag{IP}$$

where $t(x)$ denotes the total executing time of x which can be evaluated by the list schedule algorithm, that is, Algorithm 1. We suppose without loss of generality that as_i, ts_i, ah_i, th_i, r_i, w_i, pc_i, and TimeReq are nonnegative integers.

3. The Adaptive Penalty Function

In this section, we introduce the basic form of the penalty function for constrained optimization problem and present an adaptive penalty function to convert the problem (IP) into an unconstrained binary optimization problem.

3.1. The Basic Penalty Function. A variety of penalty functions have been proposed to deal with constraints. Of them, the exact penalty function and the quadratic penalty function are most basic and widely used [23–25]. Using these penalty functions problem (IP) can be converted into the following unconstrained problem:

$$\chi(x, R, q) = f(x) + R \times \psi(x), \tag{1}$$

where $\psi(x) = \max\{0, t(x) - \text{TimeReq}\}^q$, R is the penalty parameter, and $q \ge 1$. For $q = 1$ (resp., $q = 2$), the optimization method that uses (1) is known as the exact (the quadratic) penalty function method.

Let $S = \{0,1\}^n$, and denote by $\overline{S}$ the feasible solution space of (IP); that is, $\overline{S} = \{x \mid x \in S, \text{ and } t(x) \le \text{TimeReq}\}$. We construct the following auxiliary problem:

$$\begin{aligned} \min \quad & \chi(x, R, q) = f(x) + R \times \psi(x), \\ \text{s.t.} \quad & x \in S, \end{aligned} \tag{UIP1}$$

where $(x) = \max\{0, t(x) - \text{TimeReq}\}^q$, $R > 0$ is the penalty parameter, and $q \ge 1$.

3.2. The Proposed Adaptive Penalty Function. It is a well-known fact that the penalty parameter R in (1) is sensitive and problem-dependent. Hence, it is difficult to choose a value for R [26] that can be used for a wide range of optimization problems. Therefore, a number of authors [27–31] have suggested parameter-free penalty functions to do away with the sensitive parameter R. Following the same reasonings, we propose the following param0065ter-free adaptive penalty function:

$$L(x) = \begin{cases} f(x) & \text{if } \varphi(x) = 0; \\ \varphi(x) + U & \text{if } \varphi(x) > 0, \ f(x) \le U; \\ \varphi(x) + f(x) & \text{if } \varphi(x) > 0, \ f(x) > U, \end{cases} \tag{2}$$

where

$$\varphi(x) = \max\{0, t(x) - \text{TimeReq}\} \tag{3}$$

and U is an upper bound on the global minimum value of problem (IP). Since the cost of the system is smaller than $\sum_{i=1}^{n} ah_i$, we can initialize $U = \sum_{i=1}^{n} ah_i$. The value of U needs to be updated by the current best known objective function value at a feasible solution.

Based on this adaptive penalty function, we construct the following problem:

$$\begin{aligned} \min \quad & L(x), \\ \text{s.t.} \quad & x \in S. \end{aligned} \tag{UIP2}$$

Definition 1. A solution $y \in \overline{S}$ is called a discrete local minimizer of problem (IP), if $f(y) \leq f(x)$, for all $x \in N(y) \cap \overline{S}$, where $N(y)$ is a neighborhood of y. Furthermore, if $f(y) \leq f(x)$, for all $x \in \overline{S}$, then y is called a discrete global minimizer of problem (IP).

Theorem 2. *Let U be an upper bound on the discrete global minimum value of problem (IP); then problems (IP) and (UIP2) have the same discrete global minimizers and global minimal values.*

Proof. Let x^* be a discrete global minimizer of problem (IP); then $\varphi(x^*) = 0$, and $f(x) \leq f(x^*)$, for all $x \in \overline{S}$. Consider two sets $\overline{S}$ and $S \setminus \overline{S}$.

For all $x \in \overline{S}$, by (3), we have $\varphi(x) = 0$, then

$$L(x) = f(x) \geq f(x^*) = L(x^*). \tag{4}$$

For all $x \in S \setminus \overline{S}$, we have two cases: (i) $\varphi(x) > 0$ and $f(x) \leq U$, and (ii) $\varphi(x) > 0$ and $f(x) > U$.

Case (i). By (2), we have

$$L(x) = \varphi(x) + U \geq \varphi(x) + f(x^*) > f(x^*) = L(x^*). \tag{5}$$

Case (ii). By (2), we have $L(x) = \varphi(x) + f(x)$. Since $f(x) > U > f(x^*) = L(x^*)$, it holds

$$L(x) = \varphi(x) + f(x) > U > L(x^*). \tag{6}$$

From (4), (5), and (6), it is obvious that x^* is a discrete global minimizer of problem (UIP2).

Conversely, let x^* be a discrete global minimizer of problem (UIP2), then

$$L(x) \geq L(x^*), \quad \forall x \in S. \tag{7}$$

Suppose that $x^* \notin \overline{S}$; that is, $\varphi(x^*) > 0$. If $f(x^*) \leq U$, then

$$L(x^*) = \varphi(x^*) + U > U \geq f(y), \tag{8}$$

where y is a discrete global minimizer of problem (IP); that is, $\varphi(y) = 0$. If $f(x^*) > U$, then

$$L(x^*) = \varphi(x^*) + f(x^*) > U \geq f(y). \tag{9}$$

Hence,

$$L(x^*) > f(y). \tag{10}$$

Since $L(y) = f(y)$, inequality (10) implies that $L(x^*) > L(y)$ which means that x^* is not a discrete global minimizer of problem (UIP2). So $x^* \in \overline{S}$, and problems (IP) and (UIP2) have the same discrete global minimizers and global minimal values. □

Theorem 3. *Suppose that y is a solution of problem (UIP2), and $L(y) < U$. Then y is a feasible solution of problem (IP).*

Proof. If y is an infeasible solution of problem (IP), then $\varphi(y) > 0$. By (2), if $f(y) \leq U$, we have $L(y) = \varphi(x) + U > U$. If $f(y) > U$, then $L(y) = \varphi(y) + f(y) > U$. So, it holds that $L(y) > U$, which contradicts the condition $L(y) < U$. Hence, the theorem holds. □

4. The Proposed Memetic Algorithm for HW/SW Partitioning

Memetic algorithms (MAs) are a kind of global search technique derived from Darwinian principles of natural evolution and Dawkins' notion of memes [32]. They are genetic algorithms that use local search procedures to intensify the search [33]. MAs have been applied to solve a variety of optimization problems [34–40].

In this section, a tabu search-based memetic algorithm, TSMA, is presented to solve the parameter-free unconstrained problem (UIP2). First, a general framework of TSMA is presented, followed by detailed descriptions of various components of the algorithm.

4.1. General Framework of the Hybrid Memetic Algorithm. The steps of the general framework of TSMA for solving (UIP2) is shown in Algorithm 2. Throughout its execution, the TSMA updates the upper bound U with the function value at the current best solution x^*.

Before the TSMA begins, its iterative process performs three main steps; see Algorithm 2. In step 1, it initializes the first upper bound U of (UIP2) with the known trivial solution where $x_i = 1, i = 1, 2, \ldots, n$. In step 2, it generates the initial random population set of size p; that is, $P = \{x^1, \ldots, x^p\}$, $x^k \in S$, $k = 1, 2, \ldots, p$. In step 4, the TSMA refines each member of the initial population P by the local search, the tabu search, and treats P as the starting population set.

As the iteration process begins, the TSMA repeatedly performs three main steps: crossover followed by tabu search, path relinking, and the updating of P. In step 10, a new solution x^0 is identified using the crossover operation followed by tabu search. In step 17, the path relinking procedure may be applied to x^0 in an attempt to generate an improved solution y from x^0. Finally, in step 22, the population updating process is performed to decide whether $y \in P$ and which solution should be replaced, provided that such an improved y has been generated. The remaining steps of Algorithm 2 are used to update U and x^*. This iterative process repeats itself until some stopping conditions are met.

The main iterative components of TSMA, the crossover followed by the tabu search procedure, the path relinking procedure, and the population updating strategy are explained in the subsequent subsections.

4.2. Crossover Operator. We adopt the fixed crossover operator [41] to generate a new offspring. First, a pair of individuals are selected randomly from the population. Next, if two selected individuals have the same bit value, their offspring inherits it. Otherwise, it takes value 0 or 1 randomly. Suppose that x and y are two selected individuals from the population. The pseudocode of the crossover operator is given in Algorithm 3.

4.3. The Tabu Search Procedure. Tabu search (TS) is a metaheuristic optimization that has been applied successfully to solve a number of combinatorial optimization problems [42–45]. It combines search strategies that are designed to avoid

Require: A directed acyclic graph;
Ensure: Approximate global maximal solution x^*;
(1) Initial $U = \sum_{i=1}^{n} ah_i$, $x^* = (1, \ldots, 1)$.
(2) Generate randomly a population $P = \{x^1, \ldots, x^p\}$.
(3) **for** $k = \{1, \ldots, p\}$ **do**
(4) Use the local search procedure (see Section 4.3) to minimize problem ($UIP2$) from every individual x^k, also denote the obtained solution by x^k.
(5) **if** $L(x^k) < U$ **then**
(6) Let $U = L(x^k)$, and $x^* = x^k$.
(7) **end if**
(8) **end for**
(9) **while** termination criteria (see Section 4.6) do not meet **do**
(10) Use the crossover operator (see Section 4.2) to generate a new offspring x^0, and apply the local search procedure to refine x^0, also denote it by x^0.
(11) **if** $L(x^0) < U$ **then**
(12) Let $U = L(x^0)$, and $x^* = x^0$.
(13) **end if**
(14) **if** $x^0 = x^*$ **then**
(15) $y = x^0$.
(16) **else**
(17) Use the path relinking procedure PR (see Section 4.4) to the solution x^0 and the current best solution x^* to obtain a solution y.
(18) **end if**
(19) **if** $L(y) < U$ **then**
(20) Let $U = L(y)$, and $x^* = y$.
(21) **end if**
(22) Use the population updating method (see Section 4.5) to update the population that is to decide if $y \in P$.
(23) **end while**
(24) **return** x^* and $f(x^*)$ as an approximate discrete global minimizer and global minimal value of problem (IP), respectively.

Algorithm 2: General structure of TSMA.

Require: Two selected individuals x and y;
Ensure: A new offspring z;
(1) **for** $i = \{1, \ldots, n\}$ **do**
(2) **if** $x_i = y_i$ **then**
(3) Let $z_i = x_i$.
(4) **else**
(5) z_i takes value 0 or 1 randomly.
(6) **end if**
(7) **end for**
(8) **return** z as a new offspring.

Algorithm 3: Crossover operator.

being trapped in a local minimizer and to present revisiting recently generated solutions [46]. The neighborhood structure and the tabu tenure with which TS is implemented are fundamental to achieving the above objectives. A new solution x^0 is generated from the existing solution x, within a defined neighborhood of x, by the operation called a "move." The solution x^0 is not generated again for a number of iterations (tabu tenure).

The tabu search procedure presented here differs mainly in solution generation. While we preserve the tabu tenure feature of TS, we do away with the neighborhood structure in generating x^0 from x. The solution generation procedure has been suggested here to conform to the problem at hand and to make the search greedier. In particular, a sequence of flips (each flip returns a solution) on the (allowed) components of x are carried out. The best solution from the sequence is considered as x^0. More specific to the problem considered, a flip gain $gain(i, x)$ of a node i (node i corresponds to the variable x_i) is defined as follows:

$$gain(i, x) = L(x) - L(x_1, \ldots, x_{i-1}, 1 - x_i, x_{i+1}, \ldots, x_n). \tag{11}$$

A positive gain is a gain when the objective value of problem (UIP2) decreases if the component (variable) x_i is flipped from the current value to its complement; that is, if $x_i = 1$, then its value is changed to $x_i = 0$. Therefore, for each variable x_i we keep an associated tabu tenure T_i which prevents x_i from being flipped again until T_i diminishes ($T_i \leq 0$). Hence, the ith gain ($i = 1, 2, \ldots, n$) is calculated using (11) when it is permitted by the tabu tenure T_i or some aspiration criterion is satisfied.

Require: An initial solution x^0.
Ensure: The best solution x^{best}.
(1) Set the parameters *Maxcount* and T_j, $j = 1, \ldots, n$. Initiate $count = 0$, $t_i = 0, i = 1, \ldots, n$, and $x^{\text{best}} = x^0$.
(2) **while** $count < Maxcount$ **do**
(3) Calculate the flip gain $gain(i, x^0), i = 1, \ldots, n$, using (11).
(4) $j = \text{argmax}\{gain(i, x^0) | t_i = 0 \text{ or } L(x^0) - gain(i, x^0) < L(x^{\text{best}}), i = 1, \ldots, n\}$.
(5) Let $x_j^0 = 1 - x_j^0$. Rename the solution corresponding to this change as x^0 again. Calculate T_j by (12), let $t_j = T_j$.
(6) **if** $L(x^0) < L(x^{\text{best}})$ **then**
(7) Let $x^{\text{best}} = x^0$.
(8) **end if**
(9) $count = count + 1$.
(10) **for** $i = \{1, \ldots, n\}$ **do**
(11) **if** $t_i > 0$ **then**
(12) Let $t_i = t_i - 1$.
(13) **end if**
(14) **end for**
(15) **if** $count > Maxcount$ **then**
(16) Break.
(17) **end if**
(18) **end while**
(19) **return** x^{best}.

Algorithm 4: The tabu search procedure.

At the beginning of the tabu search procedure, T_i ($i = 1, 2, \ldots, n$) are initialized by the user and if x_i is flipped the corresponding T_i is updated, as suggested by Cai et al. [42], as follows:

$$T_i = \begin{cases} T_i + T_i \times \dfrac{\sum_{k=1}^{p} x_i^k}{p} & \text{if } x_i = 1, \\ T_i + T_i \times \dfrac{p - \sum_{k=1}^{p} x_i^k}{p} & \text{otherwise.} \end{cases} \tag{12}$$

The updating rule (12) is called the feedback mechanism in Cai et al. [42], as this updating prevents T_i from being too large or too small.

We have proposed a simple aspiration criterion that permits a variable to be flipped in spite of being tabu if it leads to a solution better than the best solution x^{best} produced by the tabu search procedure thus far. There are a number of parameters which are needed to be initialized. These are the maximum numbers of iterations, *Maxcount*, and the T_i values ($i = 1, 2 \ldots, n$). The tabu search procedure we have suggested here is now summarized in Algorithm 4. Notice that the index j in step 4 denotes that the node j is not in the tabu list ($t_j = 0$) or when the aspiration criterion is met (which is a positive gain). The tabu search procedure process stops when *Maxcount* of iterations is reached and returns the best solution found in the process.

4.4. Path Relinking (PR) Procedure. The path relinking technique was originally proposed by Glover et al. [47] to explore possible trajectories connecting high quality solutions obtained by heuristics. It has been applied successfully to solve a number of optimization problems such as data mining [48], unconstrained binary quadratic programming [49], capacitated clustering [50], and multiobjective knapsack problem [51].

The PR procedure presented here explores solution trajectories using two good solutions. Within the iteration process of TSMA, there are always two such solutions: the current best solution x^* and the x^0 produced by the tabu search procedure. When $x^* \neq x^0$, PR is executed in an attempt to improve the current best solution x^*. At the beginning, PR identifies the number of variables whose values differ in x^0 and x^*. The indices of these variables are then kept in a set D. PR then starts its iteration process. For each index in D an iteration is executed. PR stops if an improved x^* has been found during an iteration.

An iteration starts with identifying the index j corresponding to the highest gain; that is, $j = \text{argmax}\{gain(i, x^0) \mid \text{for all } i\}$. Let the solution corresponding to the flipping of x_j^0 be $\underline{x}^0$. The following tasks are then performed to see (a) whether PR has obtained a new best solution or (b) whether PR is approaching towards the current x^*.

(a) If $L(\underline{x}^0) < L(x^*)$, then PR stops as it has found a new best solution.
(b) Otherwise, x^0 is modified by $x_j^0 = 1 - x_j^0$ where $j = \text{argmax}\{gain(i, x^0) \mid i \in D\}$. The index j is then removed from D.

PR then proceeds with next iteration for the next index in D. The pseudocode of the PR is illustrated in Algorithm 5.

4.5. Population Updating Strategy. A combination of intensification and diversification is essential for designing high quality hybrid heuristic algorithms [52, 53]. We have presented how the TSMA implements its search intensification

Require: An initiating solution x^0, the current best solution x^*.
Ensure: A solution y.
(1) Initiate $y = x^*$, $D = \emptyset$.
(2) **for** $i = \{1, \ldots, n\}$ **do**
(3) **if** $x_i^0 \neq x_i^*$ **then**
(4) $D = D \cup \{i\}$.
(5) **end if**
(6) **end for**
(7) **while** $D \neq \emptyset$ **do**
(8) Calculate the flip gains $gain(i, x^0), i = 1, \ldots, n$.
(9) $j = \text{argmax}\{gain(i, x^0) | i = 1, \ldots, n\}$.
(10) **if** $L(x^0) - gain(i, x^0) < L(x^*)$ **then**
(11) Let $x_j^0 = 1 - x_j^0$, and $x^* = x^0$, $y = x^*$.
(12) Break.
(13) **else**
(14) $j = \text{argmax}\{gain(i, x^0) | i \in D\}$.
(15) Let $x_j^0 = 1 - x_j^0$, $D = D \setminus \{j\}$.
(16) **end if**
(17) **end while**
(18) **return** y.

Algorithm 5: Path relinking (PR) procedure.

strategies via the tabu search procedure and PR; we now present its search diversification strategies. This is achieved by updating the population set P with the solution y obtained by the tabu search procedure or PR. In this subsection, y will be referred to as the trial solution.

Like any other algorithm [35, 38], the TSMA uses its population updating to cater for both quality and diversity of the member of P. A measure is needed to decide whether the trial solution y should be inserted into P and if so which solution should it replace. To this end, the following function is suggested when $x \notin P$:

$$g(x, P) = \begin{cases} L(x) & \text{if } L(x) < L(x^*), \\ L(x) - L(x^*) + \dfrac{d(x, P)}{n\lambda} & \text{otherwise,} \end{cases} \tag{13}$$

where $x^* \in P$ is the current best solution and $\lambda > 0$ is a parameter, which balances the relative importance of the solution quality and the diversity of the population. The function $d(x, P)$, $x \notin P$, is defined as

$$d(x, P) = \min_{x^k \in P} d\left(x, x^k\right), \tag{14}$$

where

$$d\left(x, x^k\right) = \sum_{i=1}^{n} \left| x_i^k - x_i \right|. \tag{15}$$

In the updating process, we want to achieve two objectives: solution quality and population diversity. At early stages of TSMA, points in P are scattered and diversely distributed. Therefore, a further diversification is not needed. Hence, the decision whether $y \in P$ and which solution deletes should be based on the contribution of the function value $L(y)$ in $g(y, P)$ and not purely be based on the location of y in the solution space with respect to the points in P (i.e., density of the points $\{y\} \cup P$). Notice that this always holds in (13) when $L(y) < L(x^*)$. For $L(y) \geq L(x^*)$ (y is likely to be far from x^*), $L(y) - L(x^*)$ in (13) is likely to dominate $d(y, P)/n\lambda$ in $g(y, P)$.

At later stages of TSMA, points in P are likely to be clustered together. Hence, diversification of the points in P is needed. Clearly, it is likely that $L(y) - L(x^*)$ (for y being more likely to be close to x^*) is small ensuring less contribution from it and more from $d(y, P)/n\lambda$ in $g(y, P)$.

With the functional form (13) of $g(x, P)$, we now device a strategy to achieve both objectives. Central to this strategy is the comparison of $p + 1$ measure values, namely, $\{g(x^k, \widehat{P}), g(y, P)\}$, $\widehat{P} = P \setminus \{x^k\} \cup \{y\}$, $k = 1, 2, \ldots, p$. Here $g(x^k, \widehat{P})$ (resp., $g(y, P)$) represents a measure with respect to the function value at x^k (resp., the function value at y) and its distance (sparsity) with respect to $\widehat{P}$ (sparsity of y with respect to P). The point with the minimum measure is then identified; denote it by x^l. In particular, we find

$$x^l = \text{argmin} \left\{ L\left(x^k\right) - L\left(x^*\right) + \frac{d\left(x^k, P\right)}{n\lambda}, x^k \in \widehat{P} \right\} \cup \left\{ L(y) - L\left(x^*\right) + \frac{d(y, P)}{n\lambda} \right\}, \quad \forall x^k \in P \tag{16}$$

(or $x^l = \text{argmin}\{L(x^k) - L(x^*) + d(x^k, P)/n\lambda, x^k \in \widehat{P}\} \cup \{L(y)\}$, for all $x^k \in P$, when $L(y) < L(x^*)$). When $x^l \neq y$, we update P as $P = P \setminus \{x^l\} \cup \{y\}$.

On the other hand, if $x^l = y$, then P is updated with y with a small probability. This is because inclusion of y in P with probability one will reduce the diversity of P. In particular,

Require: Population $P = \{x^1, \ldots, x^p\}$, the current best solution x^*, and a solution x.
Ensure: Updated population $P = \{x^1, \ldots, x^p\}$.
(1) Calculate $g(x, P)$ according to (13);
(2) **for** $k = 1, \ldots, p$ **do**
(3) calculate $g(x^k, (P \setminus \{x^k\}) \cup \{x\})$ according to (13).
(4) **end for**
(5) Let $x^l = \text{argmin}\{g(x, P), g(x^k, (P \setminus \{x^k\}) \cup \{x\}), k = 1, \ldots, p\}$.
(6) **if** $x^l \neq x$ **then**
(7) x is inserted into P, and x^l is deleted from P.
(8) **else**
(9) **if** rand$(0, 1) < p_r$ **then**
(10) Let $x^s = \text{argmin}\{g(x^k, (P \setminus \{x^k\}) \cup \{x\}) | k = 1, \ldots, p\}$.
(11) x is inserted into P, and x^s is deleted from P.
(12) **end if**
(13) **end if**
(14) **return** P.

Algorithm 6: Population updating procedure.

Table 1: The characteristics of four test instances.

No.	DAG	TimeSw	TimeHw	TimeReq	CostHw
1	30 nodes	9736	1782	4963	3039
2	60 nodes	19591	2279	9203	6085
3	90 nodes	28718	3060	13323	8399
4	120 nodes	37354	3056	16775	11187
5	300 nodes	92156	2940	39940	27819
6	500 nodes	155151	3094	66463	45171

update $P = P \setminus \{x^s\} \cup \{y\}$ is carried out with small probability, where $x^s = \text{argmin}\{L(x^k) - L(x^*) + d(x^k, P)/n\lambda, x^k \in \widehat{P}\}$, for all $x^k \in P$. Here x^s is replaced to increase the diversity in P. The pseudocode for the population updating procedure is presented with Algorithm 6.

Finally, we present the time required to update the set P. It needs $O(n)$ to calculate the distance between y and $x^k \in P$. The distance $d(y, P)$ can be calculated in time $O(pn)$, and it needs $O(p^2 n)$ to calculate $g(x^k, \widehat{P})$. Moreover, it takes $O(p)$ time to find the solution with the smallest and the second smallest values in $\{g(y, P)\} \cup \{g(x^k, \widehat{P}\}, k = 1, \ldots, p$. Therefore, the total running time of the population updating procedure is bounded by $O(p^2 n)$.

4.6. Termination Criteria. In this paper, we use two criteria to stop TSMA. If the maximum number of generations $G_{\max}$ is reached, then we stop the algorithm. On the other hand, when the quality of the best solution from one generation to the next is not improved after G_{no} generations, then we stop the algorithm. Therefore, the TSMA stops when any of these two criteria is met.

5. Computational Experiments

In this section, we present experiments performed in order to evaluate the performance of TSMA. The benchmark test instances used are introduced followed by the parameter settings for TSMA. Finally, computational results, comparisons, and analyses are reported. The algorithm is coded in C language and implemented on a PC with 2.11 GHz AMD processor and 1 G of RAM.

5.1. Test Instances. Task graphs for free (TGFF) [54] can create directed acyclic graphs (DAGs), that is, task graphs. Given identical parameters and input seeds, it can generate identical task graphs. We have used TGFF (Window Version 3.1) to generate four DAGs as hardware/software partitioning test instances. These DAGs were also used to test the performance of the artificial immune algorithm (ENSA-HSP) [16]. The TGFF (Window Version 3.1) can be obtained at the website (http://ziyang.eecs.umich.edu/~dickrp/tgff/), and the parameters are given in the Appendix. Since these DAGs do not contain cycles, all nodes in each DAG are executed only once; that is, $pc_i = 1$.

The characteristics of these test instances are given in Table 1. These are the times when all nodes are implemented by software (TimeSw), the time when all nodes are implemented by hardware (TimeHw), the required time (TimeReq), and the cost when all nodes are implemented by hardware (CostHw).

5.2. The Parameter Settings. The tabu search procedure requires parameters T_j, $j = 1, \ldots, n$ and *Maxcount*. We have used $T_j = 10$, $j = 1, \ldots, n$, and $Maxcount = 1.5 \times n$. These

Table 2: Experimental results of our algorithm with tuning of values p and λ on the test instance with 90 nodes.

λ, p	10		20		30		40	
	Cost	Time (s)	Cost	Time (s)	Cost	Time (s)	Cost	Time (s)
0.001	3570	15.092	3568	19.531	3561	23.687	3560	23.735
0.003	3571	15.651	3562	18.265	3564	18.625	3559	20.127
0.005	3567	16.539	3561	20.283	3560	15.982	3561	21.077
0.008	3560	16.585	3561	15.617	3566	19.234	3559	20.524
0.01	3562	16.516	3568	15.681	3569	16.484	3558	20.437
0.012	3568	16.597	3556	18.421	3568	16.102	3560	19.511

values are different from those used by Cai et al. [42] for unconstrained binary quadratic programming. Justifications for these choices are that the tabu search procedure proposed here is different from [42], and the scale of test problems used in this paper is smaller than that of the test problems used in [42]. Moreover, we have chosen these values based on our numerical experiments.

An important parameter of TSMA is the size p of the population set P. The bigger the value of p is, the higher the CPU time is. This is because $O(p^2n)$ operations are needed to update P. The parameter λ in (13) is used by TSMA to diversify P, and as such it has an important significance in the performance of TSMA. We have chosen the values of these two parameters after some numerical experiments.

We first conduct a number of preliminary experiments using the test instance number 3, that is, the instance with 90 nodes. We have fixed $T_j = 10$, $j = 1, \ldots, n$, $Maxcount = 1.5 \times n$, $G_{\max} = 200$, and $G_{\text{no}} = 20$ and run TSMA with various values of p and λ. This experiment was conducted to get suitable ranges for both parameters.

With the values of T_j, $Maxcount$, $G_{\max}$, and G_{no} chosen as above, we have then conducted another series of runs of TSMA using the test instance with 90 nodes. We have used $p \in \{10, 20, 30, 40\}$. For each value of p, six values of λ, that is, $\lambda \in \{0.001, 0.003, 0.005, 0.008, 0.01, 0.012\}$, are used to constitute (p, λ) pairs. Hence, there are 24 (p, λ) pairs. For each pair, TSMA was run 5 times, making a total of 120 runs. The average results of 5 runs on each (p, λ) pair are reported in Table 2.

The results in Table 2 show that $p = 20$ and $\lambda = 0.008$ are suitable values to choose. Hence, we have used these values throughout the numerical experiments. We present all parameter values used in TSMA in Table 3, where the column 2 presents the subsections where the corresponding parameter has been discussed.

5.3. Numerical Comparisons. In order to show the effectiveness of TSMA, we compare its results with those of the evolutionary negative selection algorithm (ENSA-HSP) [16] and traditional evolutionary algorithm (EA) [16]. TSMA was run 20 times on each test instance in Table 1; hence, average results are used in the comparison. Results of ENSA-HSP and EA have been taken from [16]. Comparisons of results are presented in Table 4, where columns under the captions "best," "mean," and "worst" report the best cost, the average cost, and the worst cost, respectively. The column under the caption "time" reports the average CPU times for TSMA. We are unable to present the CPU times for the other two algorithms as this would require running these algorithms ourselves, since the CPU times of EA and ENSA-HSP were not reported in [16]. The last column of Table 4 reports the percentage of improvement on the best results by TSMA. The results obtained by all three algorithms on the first 4 test instances are further summarized in Figures 2, 3, 4, and 5.

Table 3: Settings of parameters.

Parameters	Section	Value
p	4.1	20
T_j	4.3	10
$Maxcount$	4.3	$1.5 \times n$
λ	4.4	0.008
p_r	4.4	0.2
$G_{\max}$	4.6	500
G_{no}	4.6	50

Figures 3–5 clearly show the superiority of TSMA over the other two algorithms. Wide differences in costs are noticeable in these figures. Indeed, it can be seen from Table 4 that the TSMA has achieved 84, 65, and 176 less costs than the corresponding previous best obtained by ENSA-HSP for the instances numbers 2, 3, and 4, respectively. These improvements are quite significant.

5.4. Effectiveness of the Adaptive Penalty Function. In order to show the effectiveness of our proposed parameter-free adaptive penalty function, we have replaced the adaptive penalty function $L(x)$ in (2) with the exact penalty function $\chi(x, R, q)$ $(q = 1)$ in (1) and kept other ingredients of TSMA including the parameter values in Table 3 intact. We denote this implementation of TSMA by TSMA2. Since the parameter R in $\chi(x, R, q)$ is problem-dependent, we took $R = 50, 100,$ and 200 and run TSMA2 20 times on each of the test instances, for each R value. Average results including average CPU times (in seconds) are summarized in Table 5. A number of observations and comparisons using the results in Table 4 are now presented below.

(1) Dependency on R: it can be clearly seen in Table 5 that results are very much dependent on R values. While results are comparable on instance number 1 for all R values, results for the other instance are not the same.

Table 4: Obtained results for EA, ENSA-HSP, and TSMA.

DAG	EA [16]			ENSA-HSP [16]			TSMA				Improve
	Best	Mean	Worst	Best	Mean	Worst	Best	Mean	Worst	Time	
30 nodes	1350	1409	1473	1335	1339	1350	1335	1335.0	1335	1.078	0%
60 nodes	2923	2980	3072	2884	2912	2931	2800	2806.1	2814	12.625	2.91%
90 nodes	3804	3915	4100	3617	3726	3761	3552	3557.1	3564	41.545	1.80%
120 nodes	5363	5506	5619	5177	5308	5346	5001	5005.2	5021	145.259	3.40%
300 nodes							12556	12581.6	12639	1456.672	
500 nodes							20497	20578.8	20667	3204.913	

Table 5: Obtained results for TSMA2.

DAG	$R = 50$				$R = 100$				$R = 200$			
	Best	Mean	Worst	Time	Best	Mean	Worst	Time	Best	Mean	Worst	Time
30 nodes	1335	1335.0	1335	0.986	1335	1335.0	1335	1.015	1335	1335.0	1335	0.906
60 nodes	2800	2811.7	2820	15.424	2805	2811.5	2818	14.804	2809	2811.0	2818	12.047
90 nodes	3552	3557.2	3570	41.592	3556	3559.4	3565	36.209	3556	3559.6	3565	37.491
120 nodes	5005	5028.1	5069	180.434	5018	5035.1	5068	161.848	5018	5029.2	5055	179.020

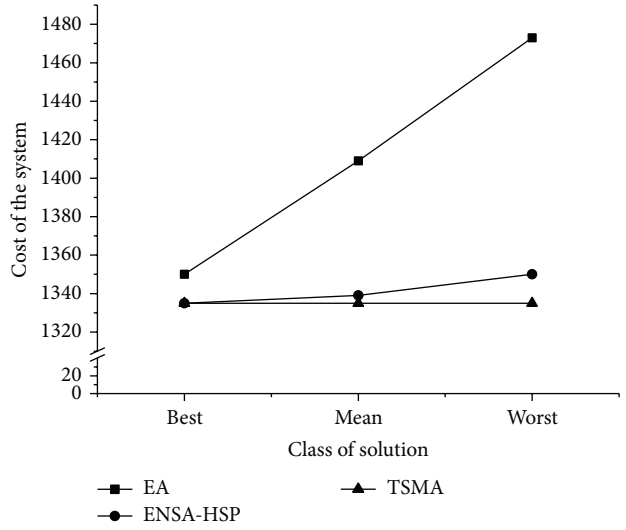

Figure 2: Experimental results on test instance with 30 nodes.

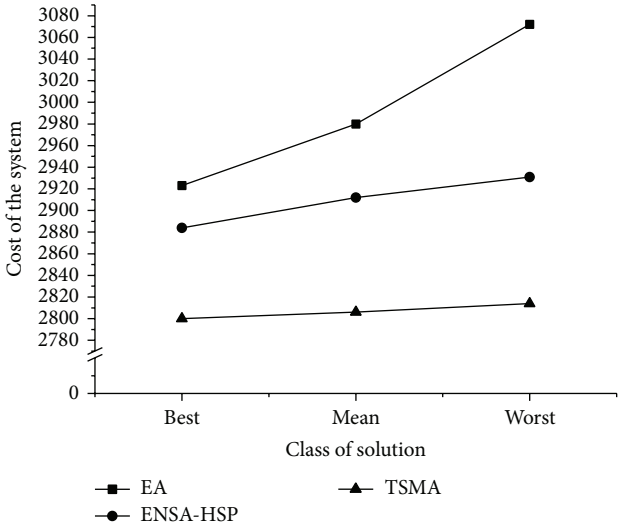

Figure 3: Experimental results on test instance with 60 nodes.

To give an example, in terms of the "mean" value, the best result was obtained for $R = 200$ on instance number 2, while, for the instance numbers 3 and 4, the best "mean" values were obtained using $R = 50$.

(2) Comparison using "mean": The TSMA performs better than TSMA2 for all instances, except for instance number 1 for which both perform the same.

(3) Comparison using "best": The TSMA performs better than TSMA2 on the last instance while both are equally matched on the remaining instances. TSMA2 obtained these results for instances 1, 2, and 3 for $R = 50$.

(4) Comparison using "time": The best CPU times are different for different R values in TSMA2. However, even if we take the best CPU times obtained by TSMA2 disregarding the R values then the results are still comparable between TSMA2 and TSMA.

A statistical view in terms of boxplot is now shown in Figures 6, 7, and 8 using optimal values obtained by TSMA and TSMA2 over 20 runs. Three figures are presented since the standard deviation is zero for instance number 1. These boxplots show that ranges over which the best results obtained over 20 runs vary. All three figures clearly show that the boxplots of TSMA has less height than those of the corresponding boxplots obtained by TSMA2 for three R values. This clearly establishes the superiority of the adaptive penalty function $L(x)$ presented in (2) over the exact penalty function (1).

6. Conclusion

In this paper, we have presented a memetic algorithm for the hardware/software partitioning problem. The algorithm has three main components: a local tabu search, a path

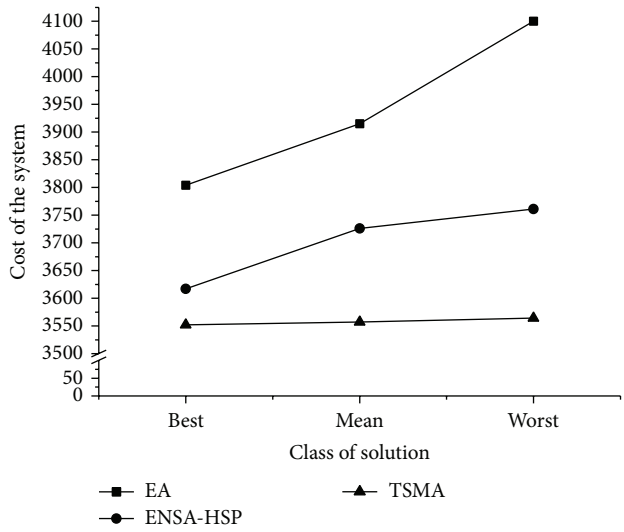

Figure 4: Experimental results on test instance with 90 nodes.

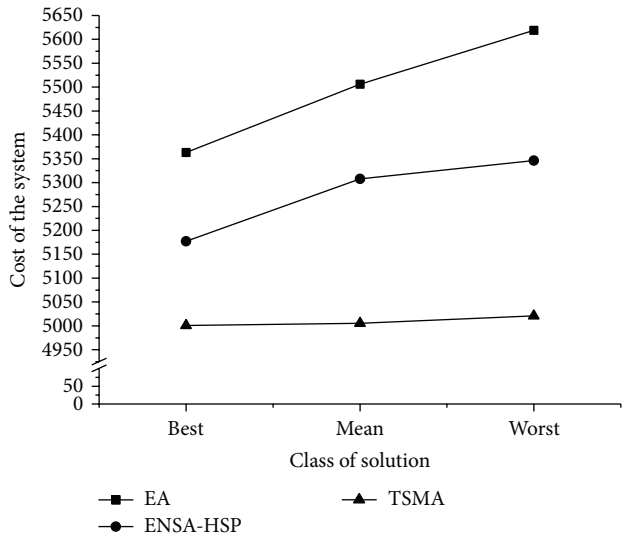

Figure 5: Experimental results on test instance with 120 nodes.

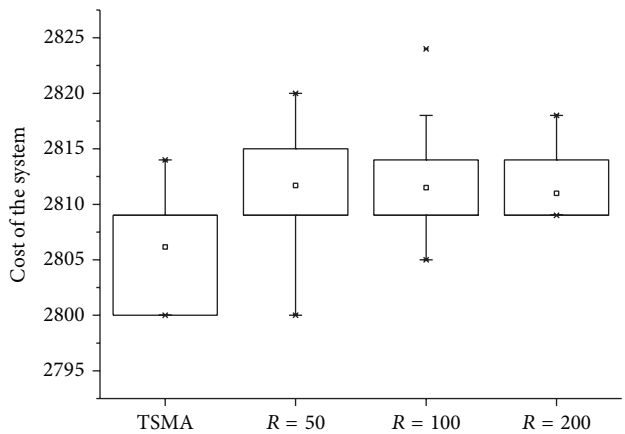

Figure 6: Boxplots of the test instance with 60 nodes.

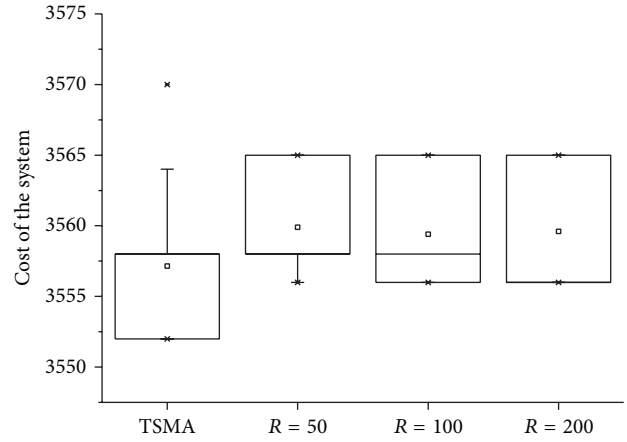

Figure 7: Boxplots of the test instance with 90 nodes.

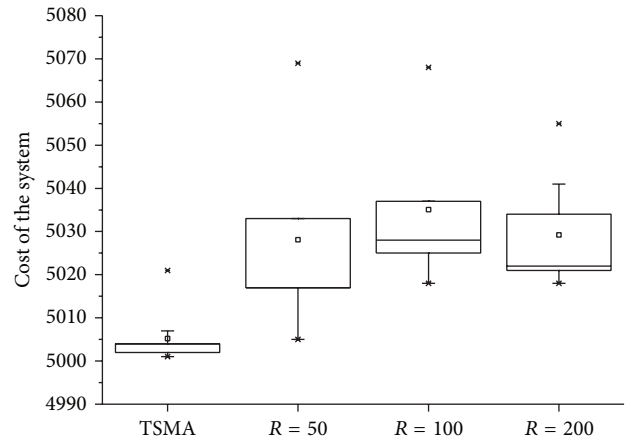

Figure 8: Boxplots of the test instance with 120 nodes.

relinking procedure, and population updating. The local tabu search is a "controlled local search" procedure used for search intensification. The path relinking procedure uses further exploitation of solutions and population updating is used for diversification of solutions. Motivations for each of these components within the proposed memetic algorithm are provided.

The hardware/software partitioning problem is converted into an equivalent unconstrained binary optimization problem by employing a parameter-free adaptive penalty function. We have motivated this penalty function first by theoretically and then by numerically comparing its performance with the exact penalty function. The memetic algorithm is then applied to this unconstrained problem, and robustness with respect to the quality of optimal solution is shown by comparing it with another two algorithms from the literature.

Appendix

In this paper, TGFF (Windows Version 3.1) used the following parameter settings [16] to generate test instances, where 30.tgffopt is for the DAG with 30 nodes, 60.tgffopt is for the

```
# 30.tgffopt
period_mul 1
task_degree 3 3
tg_cnt 1
task_cnt 31 1
task_type_cnt 30
trans_type_cnt 30
table_cnt 1
table_label TRANS_TIME
type_attrib time 10 10 1 1
trans_write table_cnt 1
table_label HARDWARE
type_attrib cost 100 50 0.5 1, time 150 75 0.5 1
pe_write table_cnt 1
table_label SOFTWARE type_attrib cost 50 30 0.5 1, time 300 100 0.5 1
pe_write tg_write #.tgff
vcg_write #.vcg
# 60.tgffopt
period_mul 1
task_degree 4 4
tg_cnt 1
task_cnt 61 1
task_type_cnt 60
trans_type_cnt 60
table_cnt 1
table_label TRANS_TIME
type_attrib time 10 10 1 1
trans_write table_cnt 1
table_label HARDWARE
type_attrib cost 100 50 0.5 1, time 150 75 0.5 1
pe_write table_cnt 1
table_label SOFTWARE type_attrib cost 50 30 0.5 1, time 300 100 0.5 1
pe_write tg_write #.tgff
vcg_write #.vcg
# 90.tgffopt
period_mul 1
task_degree 5 5
tg_cnt 1
task_cnt 91 1
task_type_cnt 90
trans_type_cnt 90
table_cnt 1
table_label TRANS_TIME
type_attrib time 10 10 1 1
trans_write table_cnt 1
table_label HARDWARE
type_attrib cost 100 50 0.5 1, time 150 75 0.5 1
pe_write table_cnt 1
table_label SOFTWARE type_attrib cost 50 30 0.5 1, time 300 100 0.5 1
pe_write tg_write #.tgff
vcg_write #.vcg
# 120.tgffopt
period_mul 1
task_degree 5 5
tg_cnt 1
task_cnt 121 1
task_type_cnt 120
trans_type_cnt 120
table_cnt 1
table_label TRANS_TIME
```

Algorithm 7: Continued.

```
type_attrib time 10 10 1 1
trans_write table_cnt 1
table_label HARDWARE
type_attrib cost 100 50 0.5 1, time 150 75 0.5 1
pe_write table_cnt 1
table_label SOFTWARE type_attrib cost 50 30 0.5 1, time 300 100 0.5 1
pe_write tg_write #.tgff
vcg_write #.vcg
# 300.tgffopt
period_mul 1
task_degree 6 6
tg_cnt 1
task_cnt 301 1
task_type_cnt 300
trans_type_cnt 300
table_cnt 1
table_label TRANS_TIME
type_attrib time 10 10 1 1
trans_write table_cnt 1
table_label HARDWARE
type_attrib cost 100 50 0.5 1, time 150 75 0.5 1
pe_write table_cnt 1
table_label SOFTWARE type_attrib cost 50 30 0.5 1, time 300 100 0.5 1
pe_write tg_write #.tgff
vcg_write #.vcg
# 500.tgffopt
period_mul 1
task_degree 6 6
tg_cnt 1
task_cnt 501 1
task_type_cnt 500
trans_type_cnt 500
table_cnt 1
table_label TRANS_TIME
type_attrib time 10 10 1 1
trans_write table_cnt 1
table_label HARDWARE
type_attrib cost 100 50 0.5 1, time 150 75 0.5 1
pe_write table_cnt 1
table_label SOFTWARE type_attrib cost 50 30 0.5 1, time 300 100 0.5 1
pe_write tg_write #.tgff
vcg_write #.vcg
```

Algorithm 7

DAG with 60 nodes, 90.tgffopt is for the DAG with 90 nodes, 120.tgffopt is for the DAG with 120 nodes, 300.tgffopt is for the DAG with 300 nodes, and 500.tgffopt is for the DAG with 500 nodes. See Algorithm 7.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This research was supported partially by the National Natural Science Foundation of China under Grants 11226236, 11301255, and 61170308, the Natural Science Foundation of Fujian Province of China under Grant 2012J05007, and the Science and Technology Project of the Education Bureau of Fujian, China, under Grants JA13246 and Jk2012037.

References

[1] J. I. Hidalgo and J. Lanchares, "Functional partitioning for hardware-software codesign using genetic algorithms," in *Proceedings of the 23rd EUROMICRO Conference*, pp. 631–638, September 1997.

[2] M. O'Nils, A. Jantsch, A. Hemani, and H. Tenhunen, "Interactive hardware-software partitioning and memory allocation

based on data transfer profiling," in *Proceedings of the International Conference on Recent Advances in Mechantronics (ICRAM '95)*, pp. 447–452, Istanbul, Turkey, August 1995.

[3] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen, "Lycos: the lyngby cosynthesis system," *Design Automation for Embedded Systems*, vol. 2, no. 2, pp. 195–235, 1997.

[4] R. Niemann and P. Marwedel, "An algorithm for hardware/software partitioning using mixed integer linear programming," *Automation for Embedded Systems*, vol. 2, no. 2, pp. 165–193, 1997.

[5] K. S. Chatha and R. Vemuri, "Hardware-software partitioning and pipelined scheduling of transformative applications," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 10, no. 3, pp. 193–208, 2002.

[6] A. Kalavade, *System-level codesign of mixed hardware-software systems, [Ph.D. dissertation]*, EECS Department, University of California, Berkeley, Calif, USA, 1995.

[7] J. Henkel and R. Ernst, "An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 9, no. 2, pp. 273–289, 2001.

[8] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "Systems level hardware/software partitioning based on simulated annealing and tabu search," *Design Automation for Embedded Systems*, vol. 2, no. 1, pp. 5–32, 1997.

[9] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design and Test of Computers*, vol. 10, no. 4, pp. 425–449, 2002.

[10] R. P. Dick and N. K. Jha, "MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 10, pp. 920–935, 1998.

[11] M. Purnaprajna, M. Reformat, and W. Pedrycz, "Genetic algorithms for hardware-software partitioning and optimal resource allocation," *Journal of Systems Architecture*, vol. 53, no. 7, pp. 339–354, 2007.

[12] P. K. Nath and D. Datta, "Multi-objective hardware-software partitioning of embedded systems: a case study of JPEG encoder," *Applied Soft Computing*, vol. 15, pp. 30–41, 2014.

[13] T. Wiangtong, P. K. Cheung, and W. Luk, "Comparing three heuristic search methods for functional partitioning in hardware-software codesign," *Design Automation for Embedded Systems*, vol. 6, no. 4, pp. 425–449, 2002.

[14] J. C. Wu, Q. Q. Sun, and T. Srikanthan, "Algorithmic aspects for multiple-choice hardware/software partitioning," *Computers and Operations Research*, vol. 39, no. 12, pp. 3281–3292, 2012.

[15] P. Liu, J. Wu, and Y. Wang, "Hybrid algorithms for hardware/software partitioning and scheduling on reconfigurable devices," *Mathematical and Computer Modelling*, vol. 58, no. 1-2, pp. 409–420, 2013.

[16] Y. G. Zhang, W. J. Luo, Z. M. Zhang, B. Li, and X. F. Wang, "A hardware/software partitioning algorithm based on artificial immune principles," *Applied Soft Computing Journal*, vol. 8, no. 1, pp. 383–391, 2008.

[17] M. López-Vallejo and J. C. López, "On the hardware-software partitioning problem: System modeling and partitioning techniques," *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 3, pp. 269–297, 2003.

[18] P. Arató, Z. Á. Mann, and A. Orbán, "Algorithmic aspects of hardware/software partitioning," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 1, pp. 136–156, 2005.

[19] W. Jigang, T. Srikanthan, and G. Chen, "Algorithmic aspects of hardware/software partitioning: 1D search algorithms," *IEEE Transactions on Computers*, vol. 59, no. 4, pp. 532–544, 2010.

[20] A. Kalavade and P. A. Subrahmanyam, "Hardware/software partitioning for multifunction systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 9, pp. 819–837, 1998.

[21] Y. Peng, M. Lin, and J. Yang, "Hardware-software partitioning research based on resource constraint," *Journal of Circuits and Systems*, vol. 10, no. 3, pp. 80–84, 2005 (Chinese).

[22] D. Gajski, N. Dutt, A. We, and S. Lin, *High-Level Synthesis Introduction to Chip and System Design*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1994.

[23] M. Sinclair, "An exact penalty function approach for nonlinear integer programming problems," *European Journal of Operational Research*, vol. 27, no. 1, pp. 50–56, 1986.

[24] A. Homaifar, S. H. Y. Lai, and C. X. Qi, "Constrained optimization via genetic algorithms," *Simulation*, vol. 62, no. 4, pp. 242–254, 1994.

[25] R. Sarker and C. Newton, "A genetic algorithm for solving economic lot size scheduling problem," *Computers & Industrial Engineering*, vol. 42, no. 2–4, pp. 189–198, 2002.

[26] T. P. Runarsson and X. Yao, "Stochastic ranking for constrained evolutionary optimization," *IEEE Transactions on Evolutionary Computation*, vol. 4, no. 3, pp. 284–294, 2000.

[27] D. W. Coit, A. E. Smith, and D. M. Tate, "Adaptive penalty methods for genetic optimization of constrained combinatorial problems," *INFORMS Journal on Computing*, vol. 8, no. 2, pp. 173–182, 1996.

[28] R. Farmani and J. A. Wright, "Self-adaptive fitness formulation for constrained optimization," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 5, pp. 445–455, 2003.

[29] B. Tessema and G. G. Yen, "An adaptive penalty formulation for constrained evolutionary optimization," *IEEE Transactions on Systems, Man, and Cybernetics A*, vol. 39, no. 3, pp. 565–578, 2009.

[30] M. M. Ali and W. X. Zhu, "A penalty function-based differential evolution algorithm for constrained global optimization," *Computational Optimization and Applications*, vol. 54, no. 3, pp. 707–739, 2013.

[31] H. J. C. Barbosa and A. C. C. Lemonge, "A new adaptive penalty scheme for genetic algorithms," *Information Sciences*, vol. 156, no. 3-4, pp. 215–251, 2003.

[32] R. Dawkins, *The Selfish Gene*, Oxford University, Oxford, UK, 1989.

[33] P. Moscato, "Memetic algorithms: a short introduction," in *New Ideas in Optimization*, D. Corne, M. Dorigo, and F. Glover, Eds., pp. 219–234, McGraw-Hill, New York, NY, USA, 1999.

[34] A. R. Buck, J. M. Keller, and M. Skubic, "A memetic algorithm for matching spatial configurations with the histograms of forces," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 4, pp. 588–604, 2013.

[35] G. Lin and W. X. Zhu, "An efficient mememtic algorithm for the max-bisection problem," *IEEE Transactions on Computers*, vol. 63, no. 6, pp. 1365–1376, 2014.

[36] M. Urselmann, S. Barkmann, G. Sand, and S. Engell, "A memetic algorithm for global optimization in chemical process synthesis

problems," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 5, pp. 659–683, 2011.

[37] G. Iacca, F. Caraffini, and F. Neri, "Memory-saving memetic computing for path-following mobile robots," *Applied Soft Computing*, vol. 13, no. 4, pp. 2003–2016, 2013.

[38] Z. Lü and J.-K. Hao, "A memetic algorithm for graph coloring," *European Journal of Operational Research*, vol. 203, no. 1, pp. 241–250, 2010.

[39] Z. Z. Zhang, O. Che, B. Cheang, A. Lim, and H. Qin, "A memetic algorithm for the multiperiod vehicle routing problem with profit," *European Journal of Operational Research*, vol. 229, no. 3, pp. 573–584, 2013.

[40] T. C. E. Cheng, B. Peng, and Z. Lü, "A hybrid evolutionary algorithm to solve the job shop scheduling problem," *Annals of Operations Research*, 2013.

[41] P. Chardaire, M. Barake, and G. P. McKeown, "A PROBE-based heuristic for graph partitioning," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1707–1720, 2007.

[42] Y. Q. Cai, J. H. Wang, J. Yin, and Y. L. Zhou, "Memetic clonal selection algorithm with EDA vaccination for unconstrained binary quadratic programming problems," *Expert Systems with Applications*, vol. 38, no. 6, pp. 7817–7827, 2011.

[43] N. F. Unver and M. M. Kokar, "Self controlling tabu search algorithm for the quadratic assignment problem," *Computers & Industrial Engineering*, vol. 60, no. 2, pp. 310–319, 2011.

[44] T. James, C. Rego, and F. Glover, "Multistart tabu search and diversification strategies for the quadratic assignment problem," *IEEE Transactions on Systems, Man, and Cybernetics A*, vol. 39, no. 3, pp. 579–596, 2009.

[45] S. Kulturel-Konak, "A linear programming embedded probabilistic tabu search for the unequal-area facility layout problem with flexible bays," *European Journal of Operational Research*, vol. 223, no. 3, pp. 614–625, 2012.

[46] F. Glover and M. Laguna, *Tabu Search*, Springer, New York, NY, USA, 1997.

[47] F. Glover, M. Laguna, and R. Martí, "Fundamentals of scatter search and path relinking," *Control and Cybernetics*, vol. 29, no. 3, pp. 653–684, 2000.

[48] H. Barbalho, I. Rosseti, S. L. Martins, and A. Plastino, "A hybrid data mining GRASP with path-relinking," *Computers & Operations Research*, vol. 40, no. 12, pp. 3159–3173, 2013.

[49] Y. Wang, Z. Lü, F. Glover, and J. Hao, "Path relinking for unconstrained binary quadratic programming," *European Journal of Operational Research*, vol. 223, no. 3, pp. 595–604, 2012.

[50] Y. M. Deng and J. F. Bard, "A reactive GRASP with path relinking for capacitated clustering," *Journal of Heuristics*, vol. 17, no. 2, pp. 119–152, 2011.

[51] R. P. Beausoleil, G. Baldoquin, and R. A. Montejo, "Multi-start and path relinking methods to deal with multiobjective knapsack problems," *Annals of Operations Research*, vol. 157, no. 1, pp. 105–133, 2008.

[52] L. Y. Tseng and S. C. Chen, "Two-phase genetic local search algorithm for the multimode resource-constrained project scheduling problem," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 4, pp. 848–857, 2009.

[53] K. Sörensen and M. Sevaux, "MA—PM: memetic algorithms with population management," *Computers & Operations Research*, vol. 33, no. 5, pp. 1214–1225, 2006.

[54] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," in *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, pp. 97–101, March 1998.