Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2017) 25: 901 – 912
© TÜBİTAK
doi:10.3906/elk-1501-64

# A metaheuristic based on the tabu search for hardware-software partitioning

**Mehdi JEMAI**[*], **Sonia DIMASSI, Bouraoui OUNI, Abdellatif MTIBAA**
Laboratory of Electronics and Microelectronics, National Engineering School of Monastir, University of Monastir, Monastir, Tunisia

**Received:** 08.01.2015 • **Accepted/Published Online:** 07.03.2016 • **Final Version:** 10.04.2017

**Abstract:** Several metaheuristics have become increasingly interesting in solving combinatorial problems. In this paper, we present an algorithm involving a metaheuristic based on tabu search and binary search trees to address the problem of hardware-software partitioning. Metaheuristics do not guarantee an optimum solution, but they can produce acceptable solutions in a reasonable time. Our proposed algorithm seeks to find the efficient hardware-software partitioning that minimizes the logic area of a system on a programmable chip under the condition of time constraints. Our goal is to have a better trade-off between the logic area of the application and its execution time. Finally, we compare our algorithm to some metaheuristic-based algorithms.

**Key words:** Combinatorial optimization, heuristics, metaheuristics, neighborhood search, tabu search, hardware-software partitioning

## 1. Introduction

Embedded systems typically include two kinds of applications: specific application hardware parts, i.e. field programmable gate arrays and application-specific integrated circuits, and programmable parts, i.e. processors like digital signal processors or application-specific instruction processors. The software parts are much easier and faster to develop and modify compared to the hardware parts. Therefore, the development cost and time of the software is less expensive. However, the hardware provides a better performance. For this reason, designers try to minimize both hardware area and power consumption.

Partitioning is a crucial step in co-designing [1–4]; it plays an important role in allocating tasks correctly to the hardware part and the software part of the architecture according to the system constraints. We should decide which components of the system will be implemented in the hardware and which ones in the software. The large number and different characteristics of the components should be considered to find an optimal partition that meets all design specifications

Several studies have already been developed to solve the problem of software/hardware partitioning [5–9] and to automate the task of partitioning. Common approaches include exact algorithms such as branch-and-bound [10], integer linear programming [11], and dynamic programming [12]. These exact methods have allowed the finding of optimal solutions for problems having a small size. Although they are very slow and they have difficulties with applications that are large in size, this is because the progress, such as the computation time, is required to be solved, which could increase exponentially with the size of the problem. To overcome the drawback of these algorithms, significant progress has been witnessed with the emergence of a new generation

[*]Correspondence: jmehdie@gmail.com

of powerful and efficient approximate methods, often called metaheuristics [13,14]. Among these algorithms, we highlight simulated annealing algorithms [15], genetic algorithms [16], tabu search, and greedy algorithms [17].

The problem of hardware-software partitioning is essentially considered to be extremely difficult and it depends on technological parameters such as speed and consumption of the application, economic parameters such as the cost of design and manufacturing, and sociological parameters such as security and testability. These parameters are difficult to formulate and quantify and they are evolving, which explains the effectiveness of proposed approaches to solve the problem of hardware-software partitioning. Similarly, the diversity of technical choices, cost constraints, and time are more and more severe, requiring new methodologies and associated software tools to reduce design time and increase quality.

The objective is to participate in solving the hardware-software partitioning problem by proposing a new hardware-software partitioning technique based on metaheuristics such the tabu search, in order to provide the best solution in terms of execution time and quality.

This paper is structured in the following ten parts. The current section presents an introduction. From the second section to the sixth section, we respectively provide an overview of combinatorial optimization, heuristics, metaheuristics, tabu search, and binary search trees. In the seventh section, we give a formulation of the problem. The eighth section presents the proposed algorithm to solve the problem of the hardware-software partitioning as well as an illustrative example. Experimental results are shown in the ninth section. Finally, we end with a conclusion.

## 2. Combinatorial optimization

Combinatorial optimization encompasses many interesting areas of searches such as discrete mathematics and computer sciences. It has become important because many optimization problems are very difficult [18] and many practical applications should be formulated as a combinatorial optimization problem [19]. Although combinatorial optimization problems are often easy to identify, they are usually difficult to solve. Most of these problems are considered as NP-hard problems and therefore they do not have a valid and efficient algorithmic solution for all data [20].

A combinatorial optimization problem aims to find the best solution in a discrete set, the said set of feasible solutions. In general, this set is finite, but it is associated with a list of relatively fewer constraints that must satisfy the feasible solutions. Clearly, an optimization problem can have several optimal solutions for a set of solutions; the best solution (or optimal solution) is the one that minimizes or maximizes the objective function. Optimization assumes that the candidate solutions to a problem can be ordered according to one evaluation criterion or more, constructed on the basis of performance indicators. Thus, we will seek to minimize or maximize such criteria. Performance evaluation in this case is an essential element in the search for the best way to make decisions. Most specialists of combinatorial optimization have oriented their research towards the development of heuristics. A heuristic method is often defined as a procedure that better exploits the structure of the considered problem in order to find a solution of reasonable quality in a calculation time that is as low as possible [21].

## 3. Heuristics

A heuristic is a method for solving a particular type of problem in an approximate way. It can quickly provide a feasible solution, which is not necessarily optimal for a problem of NP-hard optimization.

Often the algorithms are too complex to get a result in a reasonable time for some problems, even if we could use phenomenal computing power. It therefore becomes necessary to seek the closest possible solution for an optimal solution by performing successive tests. Since we cannot try all combinations, strategic choices must be made. These choices usually depend on the problem treated, and they constitute what is called a heuristic. The purpose of a heuristic is therefore not to try all possible combinations to find one that addresses the problem to get a suitable approximate solution (which may be true in some cases) in a reasonable time.

To solve problems and make decisions, heuristics are considered in algorithms that require the exploration of a large number of cases because they can reduce their medium complexity by examining first the cases that have the potential to give the answer.

Though obtaining an optimal solution is not guaranteed, the use of heuristics offers many advantages over exact methods:

-Searching for an optimal solution can be totally inappropriate in some practical applications because of the size of the problem, the dynamic that characterizes the working environment, lack of accuracy in data collection, and difficulty of formulating the constraints in explicit terms or the presence of contradictory goals.

-When it is applicable, an exact method is often much slower than a heuristic method, which generates additional computing costs and difficulties in the response time.

-Research principles that are the basis of a heuristic method are generally more accessible to nonexpert users. The lack of transparency in some exact methods requires regular intervention by a specialist or even the designer of the method.

-A heuristic method can easily be adapted or combined with other types of methods. This flexibility greatly increases the possibilities of using heuristics.

## 4. Metaheuristics

A metaheuristic consists of a set of fundamental concepts that allow assistance in the design of heuristic methods for the optimization of a problem. The purpose of metaheuristics is similar to that of heuristics: to get good-quality solutions in a reasonable time. Metaheuristics are considered as a family of optimization algorithms to solve difficult optimization problems for which we do not know the most effective classic method; they are often used in combinatorial optimization.

Metaheuristics are usually stochastic algorithms, which are progressing towards an optimum sampling of an objective function. They do not require special knowledge about the problem optimized; indeed, the only required information is to be able to associate one value (or more) to a solution. In practice, they are used for problems that cannot be optimized by mathematical methods.

Thanks to metaheuristics, approximate solutions can be offered for larger classical optimization problems and for many applications that we were unable to treat previously [22,23]. In recent years, the interest in metaheuristics has increased continuously in operations research and artificial intelligence. The most classical metaheuristics are those based on the concept of the path. In this context, the algorithm generates one solution on the search space in each iteration. The concept of "neighborhood" is essential. The common approaches are simulated annealing and tabu search, while other approaches use the concept of population and they manipulate a set of solutions in parallel at each iteration, for example the genetic algorithms.

## 5. Tabu search

The tabu search is an optimization metaheuristic presented by Glover in 1986 [24]. This method is an iterative metaheuristic qualified in local searches at large.

### 5.1. Principle

The principle of a metaheuristic is to explore the search space composed of all feasible solutions in order to achieve the optimal solution that minimizes the objective function. Starting from an initial feasible solution, the process consists, in each iteration, of choosing the best solution in the neighborhood of the current solution, even if that does not lead to improvement. It is essential to note that this may lead to an increase in the value of the function: this is the case when all points of the neighborhood have a higher value. It is from this mechanism that we escape the local minima. To avoid the trap of local optima, in which this process can be easily grabbed, the tabu search uses a temporary storage structure to save the last visited solutions: the tabu list.

The mechanism is to prohibit (hence the "tabu" name) a return to the last explored positions. Already explored positions are stored in a FIFO (often called a tabu list) of a given size, which is an adjustable parameter of the heuristic. This stack must maintain complete positions, which, in some types of problems, may require the filing of a large amount of information. This difficulty can be circumvented by keeping in the memory only the previous movements associated with the value of the function to be minimized. Indeed, a solution remains prohibited for a number of iterations equal to the size of the list. Following this, the best nonprohibited neighbor will be selected for the next iteration.

### 5.2. Tabu dynamic list

One of the critical aspects of the use of a tabu search is the need to adjust the length of the tabu list to efficiently browse the solution space. If the list is too short, the search ends up exploring a local optimum, which has a slightly greater radius. Different explored solutions form a cycle that will repeat itself indefinitely. Conversely, if the list is too long, all movements can become tabu and the search stops if there is no new neighbors, which leads to a blockage. Adjusting the length of the tabu list mainly depends on the topology of the solution space, where the neighborhood and tabu are the only criteria for measurement elements. In general, the tabu list should be kept to a minimum length to avoid a cycle. If blocked, all movements are tabu. This situation is easy to detect. It is enough to reduce the length of the list to allow new movements.

### 5.3. Neighborhood function

The complexity of a solving approach based on the tabu search depends mainly on the size of the neighborhood of the current solution and the method of evaluation of each of these neighbors to determine the one that minimizes the function "cost".

### 5.4. Evaluation neighborhood

The best non-tabu neighbor among the neighborhood of the current solution will be selected for the next iteration. To do so we would be able to evaluate all the neighbors, however.

## 6. Binary search trees

A tree is a set of nodes, hierarchically organized from a distinguished node called the root. The tree is a more specific and important structure of computer science: for example, files in operating systems and programs processed by a compiler are organized in tree form. This structure is also used in other areas such as imaging and artificial intelligence.

A binary search tree consists of a set of nodes such that each node in the left subtree has a smaller value than that of the root, and any right subtree node has a greater value than that of the root. The left and right subtrees must also be binary search trees. This structure is often used to store and retrieve information and it optimizes the access time. Our goal of using binary search trees is to reduce the search space and get an optimized access time to data.

## 7. Problem formulation

The aim of this paper is to solve the following problem:

Given a control data flow graph G (V, E) and a system on a programmable chip (SOPC) circuit, the purpose is to find a possible hardware-software partitioning of the graph G (V, E) on the SOPC in order to minimize the hardware cost and to satisfy a temporal constraint. The parameters of node $v_i$ and graph G are as follows [25,26]:

- $\mathbf{H_L(v_i)}$ is the hardware latency of node $v_i$;
- $\mathbf{S_L\,(v_i)}$ is the software latency of node $v_i$;
- $\mathbf{A(v_i)}$ is the hardware cost in (a slice of) node $v_i$;
- **S** is an indicator vector defined as follows: S **(i)** where: i =1, 2..., $|V|$,
    - S (i) = 0 if the node (i) will be implemented in the software part;
    - S (i) = 1 if node (i) will be implemented in the hardware part;
- The latency of the graph related to one vector S is as follows:

$$L(G) = S_L\,(G) + H_L\,(G)\,, \tag{1}$$

where:

$$S_L\,(G) = \sum_{v_i \in G} (1 - S(i)) \times S_L(v_i)), \tag{2}$$

$$H_L\,(G) = max_{v_i \in G}[(S\,(i)\, \times H_L\,(v_i)\,) + \sum_{v_j \in G} ((S\,(j)\, \times \alpha_{ij}) \times H_L\,(v_j))\,], \tag{3}$$

$$\begin{cases} \alpha_{ij} = 1,\ if\, v_j \,\text{depends on}\, v_i \\ 0,\ \text{else.} \end{cases} \tag{4}$$

Hence, our partitioning problem can be modeled as an optimization problem as follows:

$$\begin{cases} \text{minimize} \sum\limits_{v_i \in G} S\,(i) \times A\,(v_i) \\ \text{subject to}\ L\,(G) \leq T_{con}, \end{cases} \tag{5}$$

where L(G) is the whole latency of the graph G and T$_{con}$ is the temporal constraint.

## 8. The algorithm

### 8.1. Principle

In this section, we present our hardware-software partitioning algorithm based on the tabu search and binary search trees. The binary search tree is built as follows: the root is a virtual node that is defined as the average of the largest and the smallest modules, where the modules having small size are assigned to the left subtree and those having large size are assigned to the right subtree. The tabu search will be applied to the left or right subtree according to the time constraint.

The proposed algorithm includes the following steps:

**- First:** Initialize the size of the generation and the temporal constraint $T_{con}$, and build the binary search tree by assigning the left subtree (LST) to the hardware part and the right subtree (RST) to the software part of the architecture. Finally, we calculate the execution time $T_{ex}$.

**- Second:** Compare the execution time $T_{ex}$ to the temporal constraint $T_{con}$.

```
If ($T_{ex} \leq T_{con}$) then
    Choose an initial solution $S_0$ among the individuals of the LST and the solution $S_0$* will present
      the individuals of the RST
Else
    Choose an initial solution $S_0$ among the individuals of the RST and the solution $S_0$* will present
      the individuals of the LST
End if;
```

- Third: Based on Eq. (6), calculate the fitness $f(S_0)$ of the solution $S_0$:

$$f(S0) = \sum_{v_i \in S0} S_0(i) \times A(v_i) \tag{6}$$

**- Fourth:** Find all neighboring solutions N (k) from the solution $S_0$ and select the better solution $S_1$. Where $S_1$ = Better (N (k)), k is the number of neighborhoods.

**- Fifth:** Compare the fitness $f(S_0)$ to the fitness $f(S_1)$.

```
If ($f(S_1) \leq f(S_0)$) then
    $S_0 <= S_1$
End if;
```

Update the FIFO list according to the following pseudocode:

```
1: Begin
2: FIFO list[n]; // n is the size of the FIFO list
3: If the FIFO list is empty then
4:   Filling the FIFO list with the solution f ($S_0$);
5: Else
6:   Sorting the FIFO list in a decreasing order;
7:     If (f ($S_0$) $\leq$ FIFO list [0]) then
8:       FIFO list [0] <= f ($S_0$);
9:     Else
10:      Reject the solution $S_0$;
11:    End if;
12: End if;
13: End.
```

If the solution f ($S_0$) exists in the FIFO list, then the algorithm will generate a new solution $S_0$ and in this case we have to start from step three.

```
- Sixth:
    If (finish condition) then
      Return the last element of the list FIFO and the solution $S_0$* as a final solution S
    Else
        If ($S_1$ is solution) then
          Go to step four
        Else
          Generate a new solution $S_0$ and go to step three
        End if;
    End if;
```

### 8.2. Illustrative example

To illustrate the proposed algorithm, we will apply it on the graph shown in Figure 1. The parameters of nodes are shown in Table 1. The temporal constraint $T_{con}$ is 70, the the size of the FIFO list is three, and the number of iterations is seven.

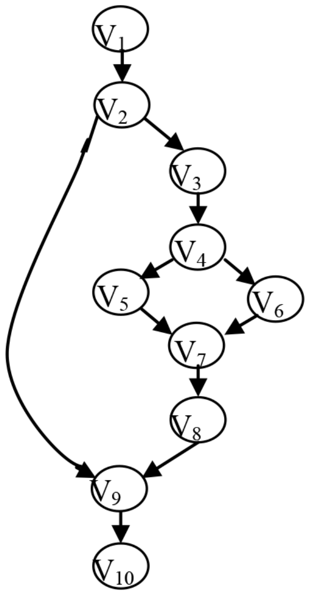

**Figure 1.** Graph G.

**Table 1.** Node parameters.

| Node | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | V10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $H_L(v_i)$ | 4 | 3 | 3 | 4 | 4 | 2 | 3 | 2 | 3 | 4 |
| $S_L(v_i)$ | 8 | 7 | 8 | 9 | 10 | 5 | 8 | 7 | 10 | 9 |
| $A(v_i)$ | 4 | 11 | 5 | 7 | 9 | 6 | 13 | 10 | 3 | 12 |

**First step:** The binary search tree is built as shown in Figure 2 by assigning the LST to the hardware part and the RST to the software part of the architecture. Therefore, the indicator vector is S = (1,0,1,1,0,1,0,0,1,0) and the execution time is $T_{ex} = 57$ ns.

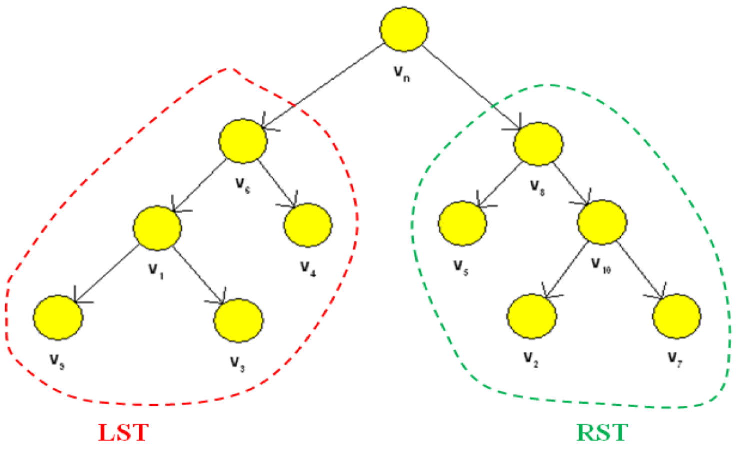

**Figure 2.** Building the binary search tree.

**Second step:** In this case $T_{ex} \leq T_{con}$, and then an initial solution $S_0$ will be chosen among the individuals of the LST. $S_0$ = (10111) and $S_0$* = (00000).

**Iteration 1**

**Third step:** The indicator vector S = (1,0,0,1,0,1,0,0,1,0), the fitness f($S_0$) = 20 slices, and L(G) = 62 ns.

**Fourth step:** The neighboring solutions of solution $S_0$ are N(0) = 00111, N(1) = 11111, N(2) = 10011, N(3) = 10101, and N(4) = 10110. The design results are shown in Table 2. The better solution is the one that has the smallest value of fitness while respecting the temporal constraint.

**Table 2.** Design results.

| Neighborhood | N(0) | N(1) | N(2) | N(3) | N(4) |
|---|---|---|---|---|---|
| The fitness f | 16 | 25 | 13 | 14 | 17 |
| Latency L(G) | 66 | 57 | 67 | 65 | 69 |

$$S_1 = \text{Better } (N(k)).$$
$$= N(2) = 10011.$$

**Fifth step:** The result in this step shows that f($S_1$) is less than f($S_0$), so the solution will be $S_1$ = 10011, and that will be assigned to the solution $S_0$. The FIFO list will be filled with the fitness of this solution. FIFO list = {13} .

**Sixth step:** If the stop condition is not reached and $S_1$ is the solution, we should go to the fourth step. Table 3 presents the results of other iterations.

**Table 3.** Results of iterations.

| | $S_0$ | f($S_0$) | L(G) (ns) | $S_1$ | f($S_1$) | L(G) (ns) | List FIFO |
|---|---|---|---|---|---|---|---|
| Iteration 2 | 10011 | 13 | 67 | 10001 | 7 | 70 | { 13,7 } |
| Iteration 3 | 10001 | 7 | 70 | 11001 | 12 | 65 | { 13,7,12 } |
| Iteration 4 | 10111 | 20 | 62 | 10011 | 13 | 67 | { 13,12,7 } |
| Iteration 5 | 11110 | 22 | 64 | 11010 | 15 | 69 | { 13,12,7 } |
| Iteration 6 | 11010 | 15 | 69 | 11011 | 18 | 62 | { 13,12,7 } |
| Iteration 7 | 11100 | 16 | 67 | 11101 | 19 | 60 | { 13,12,7 } |

When the stop condition is reached, the proposed algorithm returns the last element of the FIFO list and the solution $S_0*$, where $S_0$ = (10001) and $S_0*$ = (00000). Then the final solution is S = (1,0,0,0,0,0,0,0,1,0). Figure 3 shows the result of the proposed hardware-software partitioning algorithm.

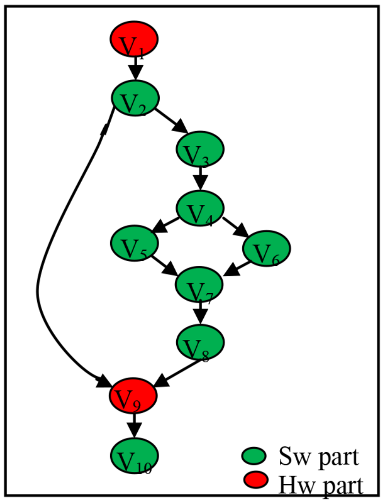

**Figure 3.** Final partitioned graph.

## 9. Experimental results

Our experiments were performed on the 16-DCT task graph composed of 56 nodes. The value of the latency is composed of the software execution time and the hardware execution time of the nodes. The hardware area occupied presents the number of slices used for each solution. These slices present the hardware part. However, the software part is presented by the PowerPC. Figures 4 and 5 show the simulated data: each algorithm was executed under Windows 7 on an Acer-PC (Intel Core 2 Duo T5500; 1.66 GHz; 1 GB of RAM) and was written in the JAVA language.

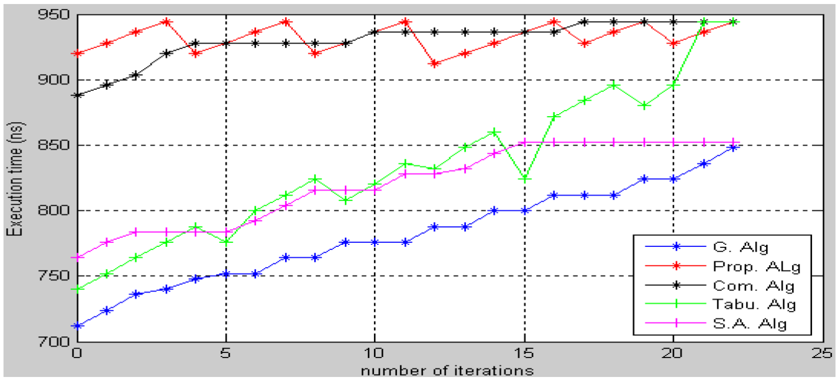

**Figure 4.** Evolution of the execution time during iterations.

In our experiments, we applied metaheuristic-based algorithms that involve a unique solution (tabu search, simulated annealing) and others that manipulate populations (genetic algorithm, combined algorithm based on binary search trees and genetic algorithm [27]).

To implement these algorithms we used the following parameters: in the case of simulated annealing, the initial temperature is 100, final temperature is 0, and $\alpha$ is 0.9. The parameters of the genetic algorithm were:

the initial population is composed of 10 individuals and each individual contains 56 nodes; the probability of a crossover is 0.6 and mutation probability is 0.1.

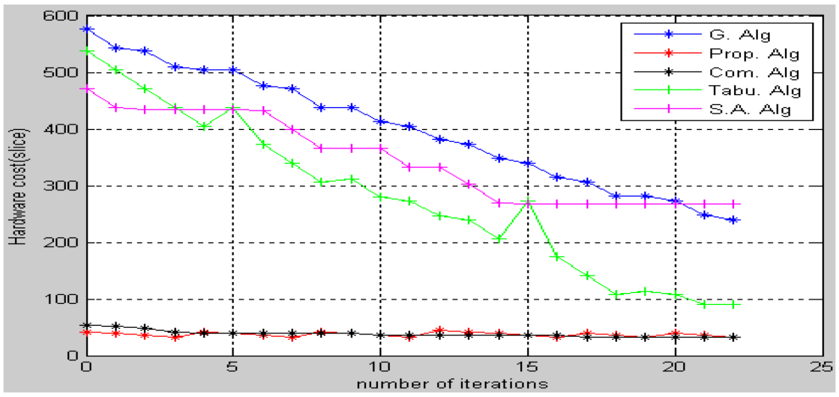

**Figure 5.** Evolution of the hardware cost during iterations.

According to Figures 4 and 5, we note that for the combined algorithm (genetic algorithm + binary search tree) and the proposed algorithm (tabu + binary search tree), the probability of convergence to the final solution is more important and that is through the use of binary search trees. For other algorithms (tabu, simulated annealing, and genetic algorithm), it requires several iterations to get the final solution. In addition, often they begin the search for the solution from a solution far from the final one, which requires more execution time.

Despite the results shown in Figures 4 and 5, we have concluded that there is no well-defined benchmark that allows having an effective comparison between the algorithms.

To overcome this difficulty, we have introduced a parameter $\beta$ defined as follows**:**

$$\beta = \frac{L}{A_{max} - A_L} \tag{7}$$

A$_{max}$: all nodes of the graph are implemented in the hardware part of the architecture.

A$_L$: the logic area consumed by the graph.

L: the whole latency of the graph.

Therefore, based on the above equation, a partitioning algorithm is classified to be good if it decreases the value of $\beta$. The use of this measure is intended to provide an idea of how the algorithm finds the best solution, as shown in Figure 6. The results show that our algorithm is the best one in terms of the $\beta$ value. It provides a gain reaching 5.17% compared to the tabu algorithm, 22.69% compared to the simulated annealing algorithm, and 21.27% compared to the genetic algorithm. When we compare our algorithm to the combined algorithm (genetic Algorithm + binary search tree), we note that we have the same value of $\beta$, but our algorithm is the fastest, as shown in Table 4. This is explained by the fact that the proposed algorithm in this paper uses a single solution, while the combined algorithm uses an entire population and sometimes it does not meet the right solution. Thus, our algorithm based on the tabu search has more of a chance to converge quickly towards the right solution.

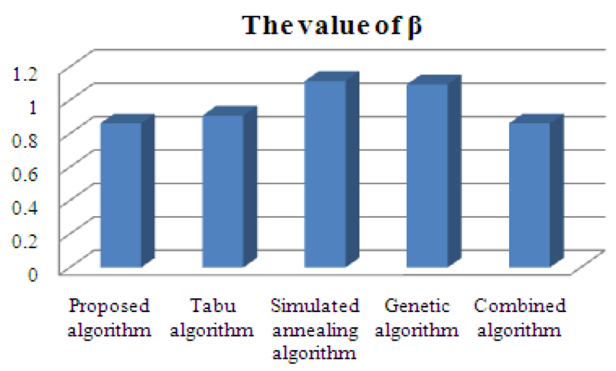

**Figure 6.** The value of $\beta$.

**Table 4.** Run time results.

| Algorithm | Proposed algorithm | Tabu algorithm | Simulated annealing algorithm | Genetic algorithm | Combined algorithm |
|---|---|---|---|---|---|
| Run time (ms) | 13,166 | 12,643 | 12,843 | 10,375 | 13,714 |

## 10. Conclusion

Despite the continuing evolution of computers and the dazzling progress of information technology, there will be certainly always a problem that is (NP) difficult, a critical size above which even a partial list of feasible solutions becomes prohibitive. Given these difficulties, most combinatorial optimization specialists have orientated their research towards developing metaheuristics. In effect, the metaheuristics revealed their high efficiency to provide approximate solutions of a good quality for a large number of conventional optimization problems and actual applications of large size. In this context, we have presented a new approach based on the tabu search to solve the problem of hardware-software partitioning to reduce the logic area. Our partitioning algorithm has been tested and compared to the simulated annealing algorithm, tabu algorithm, genetic algorithm, and a combined algorithm (binary search trees and genetic algorithm). The design results show that our approach provides better design results in terms of the hardware cost.

## References

[1] Hidalgo JI, Lanchares J. Functional partitioning for hardware-software codesign using genetic algorithms. In: IEEE 1997 New Frontiers of Information Technology, Proceedings of the 23rd EUROMICRO Conference; 14 September 1997; Budapest, Hungary. New York, NY, USA: IEEE. pp. 631-638.

[2] Ernst R, Henkel J, Benner T. Hardware-software co-synthesis for micro-controllers. IEEE Des Test Comp 1993; 10: 64-75.

[3] Harkin J, McGinnity TM, Maguire LP. Partitioning methodology for dynamically reconfigurable embedded systems. IEE Proc-E 2000; 147: 391-396.

[4] Bianco L, Auguin M, Gogninal G. Path analysis based partitioning for time constrained embedded systems. In: CODES/CASHE Proceedings of the Sixth Hardware/Software Codesign Conference; 1998. pp. 85-89.

[5] Bouraoui O, Ramzi A, Abdellatif M. Combining temporal partitioning and temporal placement techniques for communication cost improvement. Adv Eng Software 2011; 42: 444-451.

[6] Bouraoui O, Ramzi A, Abdellatif M. Temporal partitioning of data flow graph for dynamically reconfigurable architecture. J Syst Archit 2011; 57: 790-798.

[7] Bouraoui O, Abdellatif M. Optimal placement of modules on partially reconfigurable device for reconfiguration time improvement. Microelectron Int 2012; 29: 101-107.

[8] Ramzi A, Bouraoui O, Abdellatif M. A partitioning methodology that optimizes the communication cost for reconfigurable computing systems. Int J Autom Comput 2012; 9: 280-287.

[9] Mehdi J, Sonia D, Bouraoui O, Abdellatif M. Optimization of logic area for System on Programmable Chip based on hardware-software partitioning. In: ACEEE 2014 International Conference on Embedded Systems and Applications; 2224 March 2014; Hammamet, Tunisia. pp 236-238.

[10] Chatha K, Vemuri R. Hardware-software partitioning and pipelined scheduling of transformative applications. IEEE T VLSI Syst 2002; 10: 193-208.

[11] Banerjee S, Bozorgzadeh E, Dutt ND. Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration. IEEE T VLSI Syst 2006; 14: 1189-1202.

[12] Wu J, Srikanthan T. Low-complex dynamic programming algorithm for hardware/software partitioning. Inform Process Lett 2006; 98: 41-46.

[13] Reeves CR. Modern Heuristic Techniques for Combinatorial Problems. Oxford, UK: Blackwell Scientific Publications, 1993.

[14] Aarts E, Lenstra JK. Local Search in Combinatorial Optimization. New York, NY, USA: Wiley, 1997.

[15] Henkel J, Ernst R. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. IEEE T VLSI Syst 2001; 9: 273-289.

[16] Dick R, Jha N. MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. IEEE T Comput Aid D 1998; 17: 920-935.

[17] Eles P, Peng Z, Kuchcinski K, Doboli A. System level hardware/software partitioning based on simulated annealing and Tabu search. Des Autom Embed Syst 1997; 2: 5-32.

[18] Papadimitriou C, Steiglitz K. Combinatorial Optimization: Algorithms and Complexity. New York, NY, USA: Prentice Hall, 1982.

[19] Ribeiro C, Maculan N. Applications of Combinatorial Optimization. Basel, Switzerland: J.C. Baltzer, 1994.

[20] Garey M, Johnson D. Computers and Intractability: A Guide to the Theory of NP-Completeness. New York, NY, USA: W.H. Freeman and Company, 1979.

[21] Nicholson T. Optimization in Industry: Optimization Techniques. London, UK: Longman Press, 1971.

[22] Laporte G, Osman IH. Metaheuristics in Combinatorial Optimization. Basel, Switzerland: J.C. Baltzer, 1996.

[23] Osman IH, Kelly JP. Meta-Heuristics: Theory and Applications. Boston, MA, USA: Kluwer Academic Publishers, 1996.

[24] Glover F. Future paths for integer programming and links to artificial intelligence. Comput Oper Res 1986; 13: 533-549.

[25] Mehdi J, Bouraoui O. Hardware software partitioning of control data flow graph on system on programmable chip. Microprocess Microsy 2015; 39: 259-270.

[26] Pankaj KN, Dilip D. Multi-objective hardwaresoftware partitioning of embedded systems: a case study of JPEG encoder. Appl Soft Comput 2014; 15: 30-41.

[27] Sonia D, Mehdi J, Bouraoui O, Abdellatif M. Hardware-software partitioning algorithm based on binary search trees and genetic algorithm to optimize logic area for SOPC. Journal of Theoretical and Applied Information Technology 2014; 66: 788-794.