LIST SCHEDULING AND SIMULATED ANNEALING

IN A HW/SW CO-DESIGN ENVIRONMENT

by

SEILA GONZALEZ-ESTRECHA

KENNETH G. RICKS, COMMITTEE CHAIR

JEFF JACKSON
PU WANG

A THESIS

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in the Department
of Electrical and Computer Engineering
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2012

ABSTRACT

For decades combinatorial problems have been studied in numerous disciplines. The scheduling problem, in which finding the optimal solution cannot be defined in a polynomial time, belongs to this type of problem. Several methodologies for the scheduling problem have been described in order to obtain efficient results. This thesis employs a combination of two methods to solve the task-scheduling problem. The first method, known as list-based heuristics, finds a feasible solution to the problem quickly, but with no guarantee of obtaining the optimal solution. The second method is a very well known search technique called simulated annealing. The simulated annealing technique explores all feasible solutions and obtains better solutions. Also, simulated annealing accepts worse solutions with a probability. This acceptance probability avoids local minimums in the algorithm. Furthermore, this thesis introduces the concept of hardware acceleration in order to improve the final algorithm's overall execution time. By transferring software functionality to dedicated hardware, the design achieves a significant reduction in execution time.

This thesis is dedicated to Micaela and Dave.

LIST OF ABBREVIATIONS AND SYMBOLS

$<$	Precedence constraint
Δf	Difference between the new solution and the current solution
API	Application program interface
ASIC	Application-specific integrated circuit
Avg	Average
c_k	k^{th} control parameter
CP	Constraint programming
CPLD	Complex programmable logic device
CPU	Central processing unit
DAG	Directed acyclic graph
E	Finite set of edges
ε	Constant close to 1
Exec Time	Execution time
FP	Floating point
FPGA	Field programmable gate array
FSM	Finite state machine
HDL	Hardware description language
HW/SW	Hardware/software
i	i^{th} state

I/O	Input output
K	Constant used in the probability formula
k_B	Boltzmann constant
LED	Light-emitting diode
LUT	Look up table
MILP	Mixed integer linear programming
NP	Non polynomial
NRE	Non-recurring engineering cost
PLD	Programmable logic device
PLL	Phase-locked loops
RAM	Random-access memory
RISC	Reduced instruction set computing
ROM	Read only memory
SA	Simulated annealing
T	Finite set of tasks
T	Temperature
T_i	i^{th} task
VLSI	Very large scale integration
VHDL	VHSIC Hardware Description Language

ACKNOWLEDGMENTS

I would like to thank my committee chairperson Dr. Kenneth Ricks for his support and help in this research project. I would also like to express my gratitude to my other committee members, Dr. Jeff Jackson and Dr. Pu Patrick Wang, for their interesting questions and support. This thesis would not have been possible without my parents Julio and Micaela, my sisters Inma and Almu, and my parents-in-law, Patricia and John. Thank you to all of them for their support and love. Finally I would like to thank my husband David Wheat for his constant encouragement, support, and love. This thesis is dedicated to him and to my beautiful new baby Micaela.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

For decades, combinatorial optimization has been studied by researchers in numerous fields. The goal of finding optimality in a combinatorial problem is a big challenge. The scheduling problem is one such combinatorial problem, in which finding the optimal solution within the search space becomes intractable for a polynomial time. There are many characteristics to take into consideration in order to properly define and solve the problem, including precedence constraints, execution times, preemption, and priorities.

Depending on its characteristics, the scheduling problem can be solved using different methods. A summary of the most important methods is described below. Unfortunately, many of these methods require a long computational time. In order to improve the final computational time, reconfigurable hardware and software co-design are implemented here [1].

1.1 Overview of the Classical Scheduling Problem

The scheduling problem becomes evident whenever a number of tasks to be executed can be assigned in a different order. The fact that software is seen as a set of tasks, to be executed serially or in parallel, has resulted in the task-scheduling problem becoming one of the most challenging problems.

The task scheduling problem's general model contains a set of resources, task systems, constraints, and performance measures [2]. The resources consist of a set of processors that can be identical or different, according to the problem specification. The task system can be defined as a graph with a tree structure. The tree graph is a directed acyclic graph (DAG) $TG = (T,E)$ in which $T=\{t_1,t_2,t_3 \ldots t_n\}$ is a set of tasks to be executed, and where E is a set of edges that defines precedence constraints on the graph. Thus, $T_i<T_j$ means T_i must be executed completely before T_j can start to execute, and this relationship is represented in the graph as a directed edge from node T_i to node T_j. Other constraints in the problem also can be defined such as priorities. List scheduling refers to a scheduling problem with assigned priorities as an additional constraint. In list scheduling, every task has an associated priority and the order of execution depends on both the precedence constraints and also the priority constraints. Later, an exhaustive explanation of list-based heuristics is provided. Also, the scheduling problem can involve either non-preemptive scheduling or preemptive scheduling. Non-preemptive scheduling does not permit a processor to be taken away from a task until the task has finished its execution. In contrast, preemptive scheduling allows a task to be interrupted during its execution, allowing another task to be assigned to the processor. Finally, the scheduling problem involves several metrics with the most important being the schedule-length time, called the makespan.

The general scheduling problem belongs in the category of NP-complete problems along with numerous other problems such as the traveling salesman problem, routing problems, and the knapsack problem. The term NP-complete refers to non-deterministic polynomial time, which means that the optimal solution in a search space cannot be found in a polynomial time.

1.2 Combinatorial Optimization and Scheduling Methods

Combinatorial problems have been studied for the past few decades, and many techniques have been applied to solve them. Most of these combinatorial problems are NP-complete problems. Two alternatives can be used to solve them. The first option, which is to obtain the optimal solution, has the drawback of requiring a very large computational time due to the deep search in the search space for the optimal solution. The algorithms used to solve combinatorial problems in this manner are called optimization algorithms. An alternative is to find a suboptimal solution within a shorter computational time, using algorithms known as approximation algorithms. This classification is often flexible, and occasionally some algorithms utilize both approaches. Examples of these two types of algorithms include branch and bound as an optimization algorithm, local search and randomized algorithms as approximation algorithms [3]. Approximation algorithms may also be classified as either general algorithms or tailored algorithms. A general algorithm can be applied to a wide range of combinatorial problems. In contrast, a tailored algorithm is meant to be used for a specific problem. The simulated annealing (SA) algorithm is a general approximation algorithm.

As a combinatorial problem, the scheduling problem has been addressed in several ways [4]. In order to clarify these different approaches, a reduced taxonomy of the different methods is described in Figure 1.

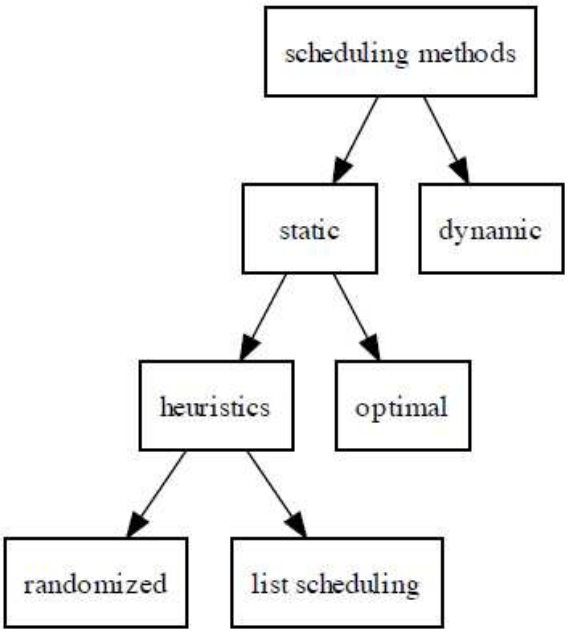


Figure 1.Taxonomy of Task Scheduling.

At the first division, scheduling methods are differentiated into static scheduling and dynamic scheduling. Static scheduling requires knowledge of all task system characteristics, such as execution times and constraints, prior to execution. Conversely, dynamic scheduling, also called online scheduling, does not have any prior information about the task system that is going to be executed [5]. This thesis is focused on static scheduling and therefore, two methods are listed under static scheduling.

Heuristics are one of the most common methods used due to their computational efficiency. The heuristic obtains a solution in a polynomial time, but optimality is not guaranteed; this type of approach can be also defined as an approximation algorithm. One class of heuristic used by many schedulers is list scheduling. The next sub-classification listed beneath static scheduling uses optimal methods. Optimal methods for solving static scheduling involve trying to find an optimal solution by completely searching the search space with a branch-and-

bound technique [6]. This search can be extremely long, and sometimes the optimal solution cannot be found. In the scheduling problem, mixed integer linear programming (MILP), constraint programming (CP), and a hybrid of both are optimal approaches that have been used to solve the problem [7]. The static scheduling problem can be defined as a set of mathematical constraints, and a constraint solver can be used to solve it. This approach can almost guarantee optimality, but with a large computational cost.

Finally, two types of heuristics are included in this taxonomy: list scheduling and randomized heuristics. In randomized methods, a global view of the solution space is given. One example of randomized methods is evolutionary algorithms, in which a random mutation is allowed to generate new solutions [8]. SA is another example of a randomized method, in which the algorithm changes into different states based upon a probability [9]. The drawback of these algorithms is their long computational time, but they avoid local minimums, which is the main disadvantage for other heuristics such as local search [10] and list-based heuristics. This thesis uses a list-based heuristic combined with a randomized SA heuristic. By using both methods, the number of feasible solutions in the search space is reduced, therefore reducing the computational time that the SA algorithm requires in order to find the optimal solution.

1.3 List-Based Heuristics

The use of heuristics yields a solution in a polynomial time, but without assurance of obtaining the optimal solution. One of the most common heuristics used for the scheduling problem is list scheduling. In the list-based heuristic, a priority is assigned to every task, adding a constraint to the initial problem. Therefore, feasible solutions of the scheduling problem decrease in number. Feasible schedules are all possible schedules that meet all of the problem constraints.

There are several methodologies in the manner in which priorities are assigned. One methodology, for instance, is priority assignment by level [11], where the level of a node is the longest path from the node to the exit node, and where the exit level is assigned priority one and does not have successors. Other priority assignments include the co-level and priority assigned randomly.

Figure 2 shows an example of the methodology to be used with list-based heuristics. Given the following DAG a feasible schedule can be found:

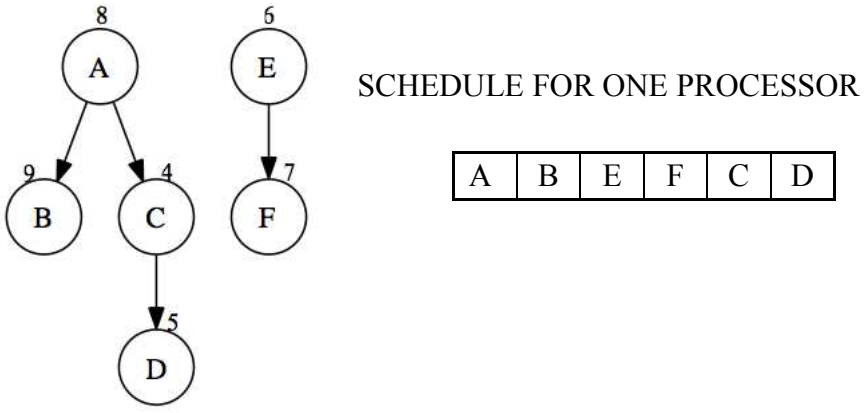


Figure 2. List Scheduling.

In this example, priorities are assigned randomly and tasks with higher priorities execute before tasks with lower priorities. If two tasks have the same priority, their execution order is random. At the beginning, the ready queue has *task A* and *task E*, so *task A* is executed first because it has greater priority. After *task A* is executed, the ready queue contains *task B*, *task C*, and *task E* still waiting to be executed. Observing the priorities, *task B* is executed, leaving the ready queue with *task C* and *task E*. After *task E* is executed, the ready queue contains *task F* and *task C*. Thus, *task F* is executed, and finally *task C* and *task D* are the last ones to be scheduled.

1.4 Thesis Outline

This thesis is divided into five chapters. In Chapter 2, the different aspects of the SA algorithm and its applications are discussed. Chapter 3 describes the software implementation of the list-based heuristic combined with SA. Many graph examples are defined in order to properly test the algorithm. Also, a study of the computational time of the algorithm for most task systems is plotted. In Chapter 4, a hardware-software co-design implementation is defined. By migrating some parts of the previous software implementation to a reconfigurable hardware implementation, it is possible to reduce the total computational cost of the algorithm. Next, Chapter 5 compares the different computational times of the pure software implementation and the hardware-software co-design implementation. Furthermore, Chapter 5 provides a study of how computational time is increased or decreased when some changes are made to the parameters of the SA algorithm. Finally, this thesis ends with Chapter 6 in which conclusions and some ideas for future research are described.

CHAPTER 2

SIMULATED ANNEALING

2.1 The SA Algorithm

SA is a probabilistic heuristic method used to obtain a suboptimal solution of a specific combinatorial problem. The algorithm approximates a global solution of a function within a large search space. Through a sequence of multiple solutions, the algorithm avoids local minimums by accepting new solutions with a probability. Since its introduction by Kirkpatrick, SA has been applied to numerous combinatorial optimization problems, such as the traveling salesman problem, the routing problem, scheduling problems, etc. [12]. The SA approach has also been applied to diverse areas of engineering, including image processing, the computer-aided design of integrated circuits, code design, and neural network theory.

An earlier concept of SA was defined in 1953 using a Monte Carlo technique [13]. The concept, known as the Metropolis algorithm, consists of simulating the evolution of a solid to reach a thermal equilibrium. The Monte Carlo technique generates a sequence of states for the solid in the following manner. Given a state i with an energy E_i and the next state j produced by a small distortion in the mechanism with energy E_j, if the difference between these states is less than or equal to zero, state j becomes the current state. On the other hand, if the difference between states is greater than zero, there is an acceptance probability defined as:

$$\exp\left(\frac{E_i - E_j}{k_B T}\right) \tag{2.1}$$

where T is the temperature of the thermal equilibrium and k_B is a constant known as the Boltzmann constant.

SA imitates the annealing process in order to solve combinatorial problems. The physical process of annealing consists of heating up a solid rapidly, and then cooling it very slowly until the solid reaches the thermal equilibrium. When the solid achieves the thermal equilibrium at a temperature T, the solid has a probability of being in a state with energy E, and this probability is defined by the Boltzmann distribution [10]:

$$P_T\{X = i\} = \frac{1}{Z(T)} exp\left(\frac{-E_i}{k_B T}\right) \tag{2.2}$$

where X denotes the current state of the solid, and $Z(T)$ is called the partition function and consists of the summation of all possible states.

It is possible to obtain a number of solutions for the SA algorithm by applying the Metropolis algorithm. The different solutions for the combinatorial problem are analogous to the different states of the solid. Also, the cost function is equivalent to the energy of the state. So it is possible to define the SA algorithm as a number of iterations in which the Metropolis algorithm is applied, and the algorithm is evaluated by decrementing a control parameter. This control parameter is the temperature. A typical characteristic of the SA algorithm is that it also accepts solutions with a probability, even when the energy of the state is greater than the current solution for a minimization problem. This acceptance probability helps avoid local minimums. This feature is the main difference between local search algorithms and the SA algorithm.

2.2 The Implementation of the Algorithm

In practice, the implementation of the algorithm needs to take four main points into consideration: a problem representation, the mechanism for transitions, acceptance criterion, and a cooling schedule. A precise description of the combinatorial problem is a representation of the solution space and the cost function. The solution space includes only feasible solutions. A mechanism to generate new solutions needs to be created. The new solution is created from the current solution within a neighborhood structure. The neighborhood structure includes a small change to the current solution. For instance, the neighborhood structure mechanisms of swapping, inversion, and permutation are the most common rearrangements. Most of the time the acceptance criterion follows the Metropolis criterion as follows:

$$P\{accept\} = \begin{cases} 1 & if\ \Delta f \leq 0 \\ exp\left(\frac{-\Delta f}{c}\right) & if\ \Delta f > 0 \end{cases} \tag{2.3}$$

where *c* is the control parameter. This control parameter refers to the temperature multiplied by the Boltzmann constant, as the algorithm is defined in the previous section. *Δf* is the difference between the new solution and the current solution for a minimization problem. Therefore, by definition, *Δf* refers to the difference between states of energy. The algorithm's schedule is determined by the specification of main parameters in the algorithm that are described in the following section.

In Figure 3, the SA algorithm is described in pseudocode [10]. As an example of a mechanism for generating a new solution, a swap mechanism is described.

```
procedure SIMULATED_ANNEALING
BEGIN
   set i_start to initial_solution
   set c_0 to initial_temperature
   set L_k to iteration_number
   set c_k to c_0
   set i to i_start
   REPEAT
      for l = 1 to L_k do
      begin
         set x to e(i)
         set e(j) to e(i_neighbor)
         set e(j_neighbor) to x
          if f(j) ≤ f(i) then
             set i to j
          else
              if exp( (f(i) - f(j)) / c_k ) > random (0,1] then
                    i set to j
             end
         end
      end

      C_k set to C_k * ε
   until stopcriterion
end
```

Figure 3.SA Algorithm.

2.3 Cooling Schedules

The term "cooling schedule" refers to the specification of the SA algorithm's parameters. The initial value of the control parameter, the stop criterion, the decrement of the control

parameter, and the length of the Markov chain are the main parameters that need to be tuned in order to obtain a good schedule.

The initial value of the control parameter needs to be large enough for possible acceptance solutions in subsequent transitions. The algorithm will stop when it reaches the thermal equilibrium. The *stop criterion* parameter will stop the algorithm if there are enough solutions in the Markov chain that continue unchanged. Also, a *decrement of the control parameter* needs to be done with small changes that usually decrease the control parameter as shown in Equation 2.4

$$c_{k+1} = \varepsilon * c_k\,,\ k = 1,2,\ldots, \tag{2.4}$$

where ε is a constant close to 1, but smaller than 1. The range of values usually lies between 0.8 and 0.99. This range of values permits a slow "cooling" in the algorithm. By eliminating drastic changes in the cooling process, an optimal solution may be ensured.

Finally, the *length of the Markov Chain* depends on the size of the problem [14]. For each value of c_k a number of transitions should be accepted. These theoretical definitions of cooling schedules are the simplest ones; other cooling schedules are more elaborate [10].

CHAPTER 3

THE SOFTWARE IMPLEMENTATION OF THE ALGORITHM

This chapter analyzes the software implementation of a list-based heuristic combined with SA. First, the initial solution is implemented using a list-based heuristic for a DAG. Second, the SA algorithm is applied to find the optimal solution from all the possible feasible solutions. Finally, some time measures are observed to decide partitions for the hardware/ software (HW/SW) co-design implementation.

3.1 Implementation Process

The different techniques for solving combinatorial problems have been widely used in order to find optimal solutions. The case of the scheduling problem consists of minimizing the total execution time of a task system. The task system can be defined by a DAG, in which vertices are tasks to be executed, and the edges represent the precedence between them [15]. Using a list-based heuristic, priorities are assigned to every task in the system; therefore an additional constraint is added to the general problem.

The approach of using a list-based heuristic for the task scheduling problem does not take a global view of the problem [16], hence optimal solutions are hard to obtain. The scheduling problem can be visualized as a global optimization problem; therefore global optimization techniques can be applied to it. Combining the list-based heuristic and a global optimization technique includes the advantages of both methods: how to obtain the solution and what makes

the solution good. In this thesis, a list-based heuristic is combined with the randomized heuristic SA. By combining these two methodologies, the search space is reduced [17].

The software implementation is written in the C programming language. In the C implementation, the DAG is represented as a precedence matrix. This matrix gives a global view of the task dependencies. A row and a column define each task. The matrix is asymmetric, and the part to the right of the main diagonal is meant to be zero because the graph is acyclic.

	A	B	C	D	E	F
task A	0	0	0	0	0	0
task B	**1**	0	0	0	0	0
task C	1	0	0	0	0	0
task D	0	0	1	0	0	0
task E	0	0	0	0	0	0
task F	0	0	0	0	1	0

Figure 4. Example of a precedence matrix.

When a "1" is placed in the matrix, a precedence relationship is represented; otherwise a zero is set if no precedence relationship is found. For instance, if a "1" is placed in column A and row B, task A needs to be executed before task B as indicated by an edge from task A to task B in the DAG. The definition of function cost in the SA algorithm is the makespan. Thus, if the algorithm is defined for only one processor, all feasible solutions are optimal solutions. Therefore, the algorithm is defined to assign tasks to two processors and communication between tasks is neglected.

In the algorithm, a ready queue is implemented. The ready queue contains all tasks that are ready to execute, i.e. all constraints have been satisfied. This queue will be updated when tasks are executed and new tasks become ready for execution.

The C implementation of the list-based algorithm is illustrated in Figure 5.

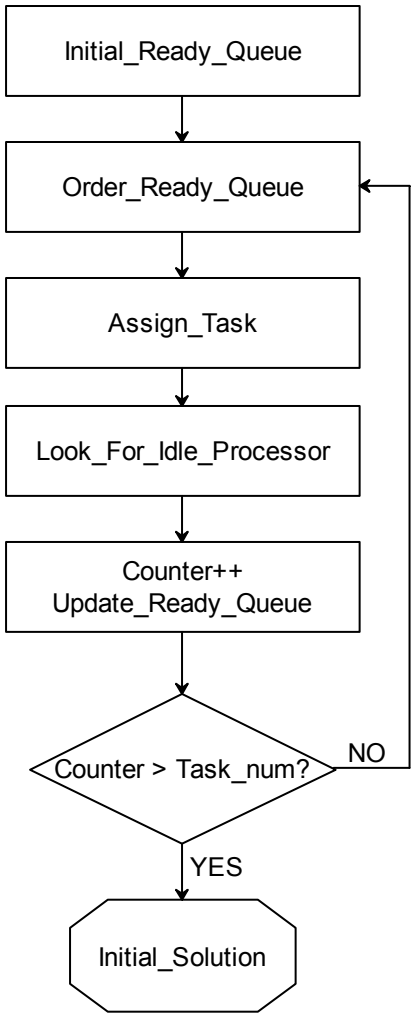


Figure5. List-Based Heuristic Flow Chart.

At first, the ready queue is initialized. Then, the ready queue is ordered from higher priorities to lower, and the algorithm assigns the task with highest priority to the first processor. In addition, the algorithm checks if the number of tasks in the ready queue is equal to one. If this is the case, the other processor needs to be idled at that time since there is no work for it to perform. Finally, a counter is incremented to observe the number of tasks that have been assigned to the processors. If this counter is less than the final task number in the DAG, the ready

queue is updated and the loop repeats beginning at the point where the ready queue is ordered. After applying the list-based heuristic, an initial solution is generated. The SA algorithm will swap tasks in this initial solution to generate new feasible solutions.

The parameters of the SA algorithm need to be chosen carefully in order to obtain a good solution and acceptable execution times. The main parameters that are included in the algorithm are: initial temperature, freezing temperature, iteration number, K, and ε. All of them have the following values respectively: 100, 1, 100, 0.01, and 0.95. The initial temperature, as its name indicates, is the highest temperature to be utilized in the algorithm. On the other hand, the freezing temperature is the lowest temperature to stop the annealing process. Furthermore, for each temperature, the algorithm is executed in a number of iterations, in this case 100. The constant K also guarantees that the acceptance probability is close to zero when temperatures reach the freezing point; therefore those solutions will not be accepted. Finally, the ε constant is employed to decrease the current temperature in the algorithm. Figure 6 illustrates the SA algorithm implemented in this thesis.

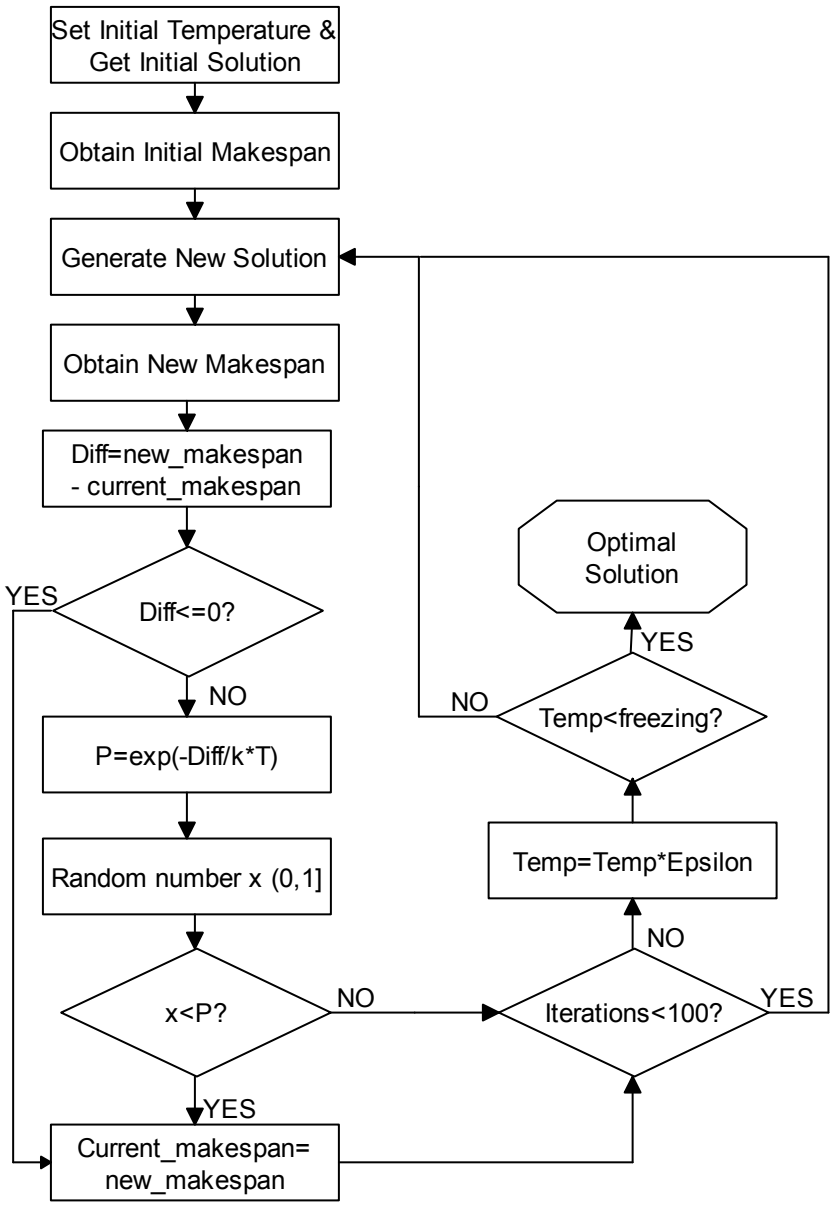


Figure 6. Simulated Annealing Flow Chart.

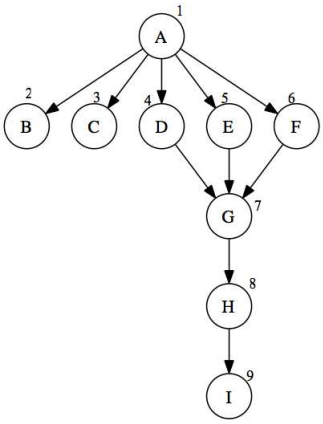
First of all, the temperature is set to the highest initial temperature. Then by applying the list-based heuristic, the initial solution is found. The current makespan of the schedule is set to that of the initial solution. At this point the different loops start executing. A new solution is generated by randomly selecting one task on one processor and swapping this task with its other pair-wise. When a task is selected randomly, the algorithm looks for tasks that meet all the

constraints and therefore that can be executed at the time of the selected task. For instance, task A is chosen to be swapped from processor 1. At that time, task B also meets all the constraints and can be executed. Therefore, task B will take the place of task A, and task A will take the place of task B. When there are no tasks to swap, one processor is idled. In this example, task A will swap to processor 2, and processor 1 will be idled. Then, the new schedule makespan is calculated from the new solution generated. The current makespan is subtracted from the new makespan, and that difference constitutes an important part of the second stage of the algorithm. If the difference is negative or zero, the new solution is accepted, so the new schedule becomes the current schedule. In contrast, if the difference is positive, the new solution will only be accepted with a probability. In this way, even though the makespan of the new solution is not as good as the current solution, the new solution can be added to the solution set to direct the search and help avoid local minimums. Equation 2.3 is employed to calculate the probability. The most important variables in this formula are the current temperature and the difference between schedule makespans. Then, a random number is generated between 0 and 1. If the probability calculated is greater than the random number, once again the new schedule is accepted as the current one. Otherwise, the current schedule remains the same and this new solution is rejected. This process is repeated 100 times for each temperature. The temperature decreases slowly by a factor, ε, and the process repeats.

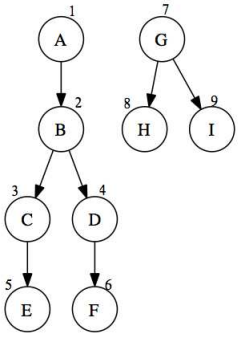
3.2 Task Systems

This combined algorithm is tested with thirteen different graphs shown in Figure 7. The numbers beside each task represent the priority of the task. Graphs 1 and 2 were manually generated specifically to test the algorithm. The rest of the graphs were selected from the

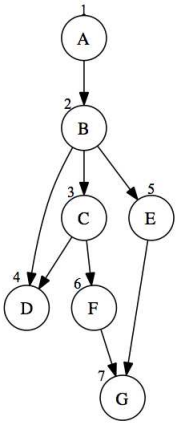
literature [18] and do not have a specific pattern. The set of graphs in Figure 7 represents a good test set, since, as a whole, they were not designed specifically for this algorithm. They have different numbers of nodes, and the tree structure for the graphs is ensured, so there are never cycles among tasks. Also, the different graphs support nodes having various numbers of predecessors and successors.



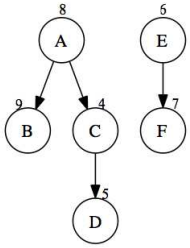
Graph 1



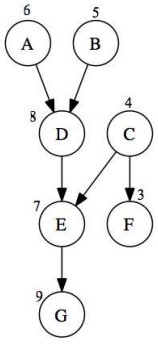
Graph 2



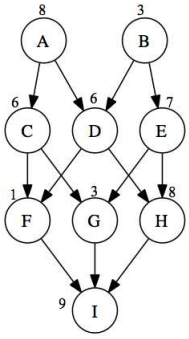
Graph 3



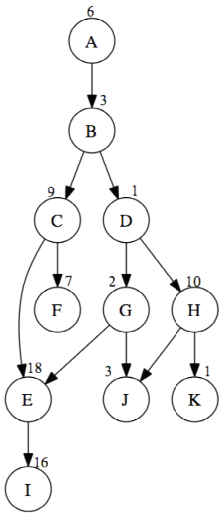
Graph 4



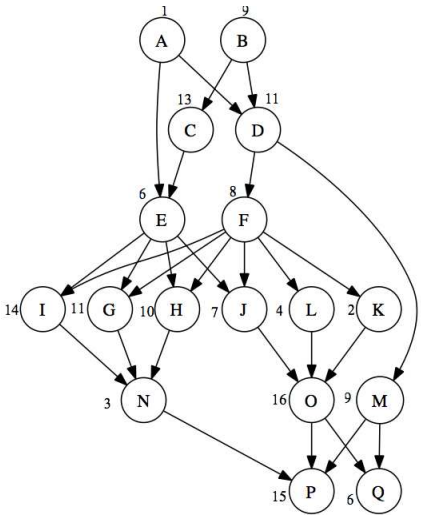
Graph 5



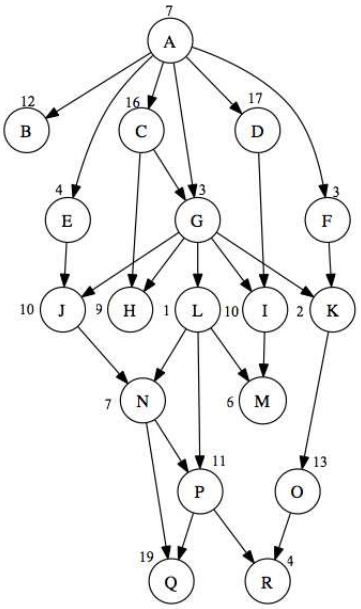
Graph 6



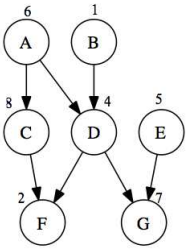
Graph 7



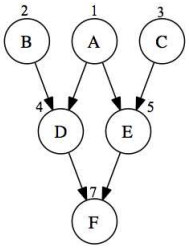
Graph 8



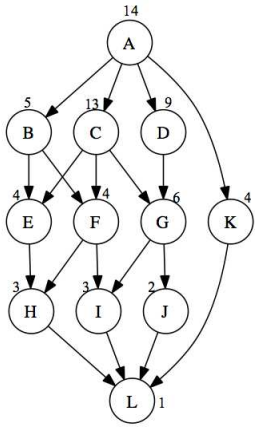
Graph 9



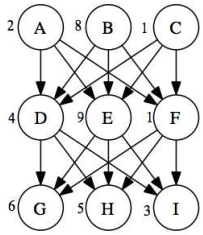
Graph 10



Graph 11



Graph 12



Graph 13

Figure 7. Task Graph Systems.

It is observed that the task structures are diverse in form. In these task systems an AND-join precedence relationship is assumed, which means that all predecessors of a task need to be executed before the task is ready for execution. The priorities are generated randomly, higher numbers represent higher priorities, and tasks with higher priorities take place earlier. All tasks are aperiodic and their execution times are defined before the software starts executing. Their execution times range from 1 unit time for task A to n unit time for task n, respectively, and the scheduler is a nonpreemptive scheduler. Thus if the graph contains seven tasks, task A will have an execution time of 1, task B will have an execution time of 2, and the task named G will have an execution time of 7 time units.

3.3 Results and Observations

In order to make a decision about the partition process in the HW/SW co-design implementation, the timing results are divided into two parts. First, the average execution time of the list-based heuristic algorithm is calculated for every DAG. Second, the average execution time of the SA algorithm is calculated using all of the task systems illustrated in Figure 7.

Candidates for hardware acceleration are those parts of the overall algorithm that require the longest time to execute.

The algorithm is implemented in C code and is executed on an Intel Core 2 Duo processor at a speed of 2.53 GHz. The execution time of the different parts of the algorithm is measured using the time library provided by the C compiler. The *gettimeofday* function is called in order to obtain the time in microseconds or miliseconds. The time is stored in a *timeval* structure. This function is called at the beginning and end of each code module that needs to be measured. For instance, the *gettimeofday* function is called at the beginning and the end of the code for obtaining the initial solution, so the list-scheduling algorithm can be measured. In the same way, the SA algorithm is measured using the same function.

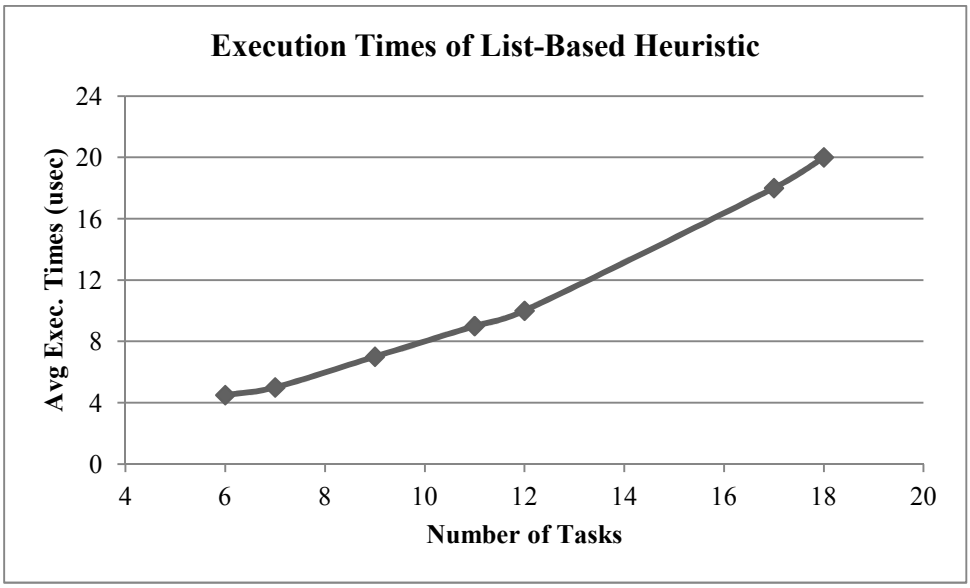


Figure 8. Execution Times of List Scheduling.

As shown in Figure 8 above, the execution time increases as the number of tasks increases. Since there are several graphs with the same number of tasks, the average of their execution times is calculated for those graphs. The measure of the list-based scheduling heuristic

is in microseconds. Thus, it is demostrated that the list-based scheduling heuristic is a fast method to obtain a feasible solution, but with no guarantee of having a good optimal one.

Next, an average execution time of the SA algorithm is shown in Figure 9.

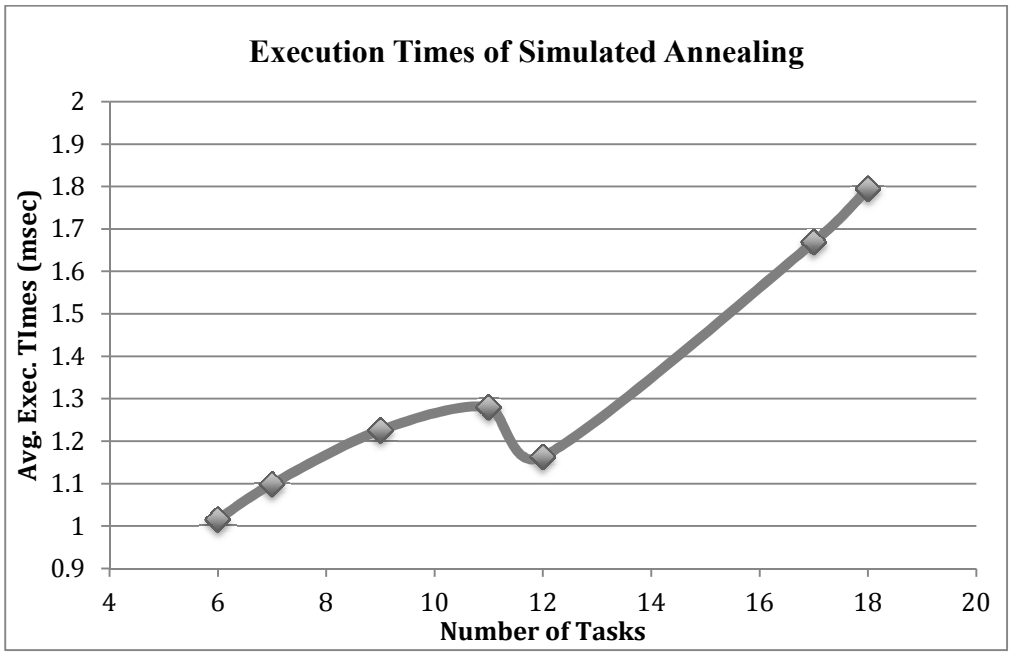


Figure 9. Execution Times of the SA Algorithm.

In Figure 9, it is observed that the average execution time for the SA algorithm does not increase directly proportional to the number of tasks. This effect is due to the random aspects of the algorithm. As new solutions are added to the solution set based upon a probability, the search is directed away from local minimums, thereby requiring more time to converge.

In Figure 9*,* the different execution times are measured in milliseconds. As expected, the most time-consuming part of the combined algorithm is the SA heuristic. Therefore, improvements in total execution time are more likely if hardware acceleration is applied to this portion of the algorithm. This approach is discussed in the next chapter.

3.4 Resulting Schedules

The task schedule applies the execution order of the DAG to each processor. The schedule length, or makespan, is the most important measure in terms of finding the optimal shortest length schedule. Also, task schedules are defined as feasible or nonfeasible. The schedule is considered to be feasible when it meets all the constraints in the problem, in this case the precedence and priority constraints. After applying the task graphs from Figure 7, the resulting schedules are shown in Figure 10.

A	F	D	G	H	
	E	C	B		I

Schedule Graph 1
Makespan = 28 time units

G	H	B	C	E
A	I		D	F

Schedule Graph 2
Makespan = 25 time units

A	B	C	D	G
		E	F	

Schedule Graph 3
Makespan = 17 time units

A	B	C	D
E	F		

Schedule Graph 4
Makespan = 14 time units

A	C	E	G
B	D	F	

Schedule Graph 5
Makespan = 16 time units

A	D	C	F	I
B	E	H	G	

Schedule Graph 6
Makespan = 23 time units

A	B	D	F	K	J	
		C	H	G	E	I

Schedule Graph 7
Makespan = 35 time units

B	C	F	L	I	J	K		P
A	D	M	E	G	H	N	O	Q

Schedule Graph 8
Makespan = 84 time units

A	D	E	F	J	H		O	M	R
	C	B	G	I	K	L	N	P	Q

Schedule Graph 9
Makespan = 92 time units

A	C	D	G
E	B		F

Schedule Graph 10
Makespan = 17 time units

C	A	E	
B		D	F

Schedule Graph 11
Makespan = 13 time units

A	D	B	E	F	I	L
	C	G	K	J	H	

Schedule Graph 12
Makespan = 40 time units

B		F		H	
A	C	E	D	G	I

Schedule Graph 13
Makespan = 29 time units

Figure 10. Final Schedules.

In the schedules shown above, the places filled with gray indicate that the processor has been idled. Having obtained the optimum results, and having measured the algorithm's execution time, the next step is to apply hardware acceleration to improve the algorithm's execution time.

CHAPTER 4

HARDWARE SOFTWARE CO-DESIGN OF A SA ALGORITHM

This chapter begins with motives for using a hardware implementation, and the different technologies necessary in order to implement the design. Then, it addresses the HW/SW co-design. This is followed by background information on FPGAs (field programmable gate array) and the NIOS II processor, and an explanation of the methodology used to change software partitions to hardware partitions. The chapter concludes with a description of the hardware implementation, and the use of resources obtained with this HW/SW co-design.

4.1 Motives for a Hardware Implementation

The large computational time required to solve a combinatorial problem with SA encourages finding ways to speed up the process. Some researchers have tried to optimize the algorithm [10]. Another approach has been the parallelization of the algorithm [19]. Parallel SA achieves a good speedup, but this speedup decreases when the task system is very complex. Compared with these software implementation methods, hardware implementations show a substantial improvement in the speed of algorithms. The past few years have seen the development of processors dedicated to implementing the SA algorithm. These processors are universal, so they can be used for any type of combinatorial problem [20]. Also, a Central Processing Unit (CPU) core, with special hardware modules in which the SA was implemented, has been designed in order to improve the execution time of this optimization algorithm [21]. Several hardware technologies could be used for dedicated hardware to perform the SA

algorithm. Full Custom very large scale integration (VLSI) designs are developed at the transistor level and custom manufacturing is needed; therefore their non-recurring engineering (NRE) costs and design times are elevated [22], but they provide the best performance possible. Application-Specific Integrated Circuits (ASICs) also require custom manufacturing that takes time to develop and involve high engineering costs. Again, the investment in this technology provides near optimal performance. Programmable logic devices (PLDs) have been well known for many years. They are a cheaper technology, and feature a short design time and added flexibility. Unfortunately, they do not achieve the best performance. FPGAs and complex programmable logic devices (CPLDs) are the highest density and most advanced programmable devices [23].

4.2 The HW/SW Co-design

The use of HW/SW co-design techniques began in the early 1990s, in attempts to solve the main problems that arose in embedded system design. By analyzing embedded system designs, the HW/SW co-design helped designers to decide if systems met characteristics such as performance, size, power, and synthesis methods [24]. Then, as now, HW/SW co-design techniques seek to migrate system functionality to dedicated hardware in order to improve the overall system performance. They also seek to place some of that system functionality in software to enhance flexibility and to reduce the total cost of the final product [25].

HW/SW co-design techniques vary according to the kind of application that is being developed. Many heuristics are being studied in a HW/SW co-design environment, including SA and list-based heuristics [25]. In order to obtain a good balance in the HW/SW co-design, reconfigurable hardware, such as FPGAs, is used. In this thesis, the use of HW/SW co-design

helps to improve the SA algorithm's performance by implementing the software components with long execution times using dedicated hardware. The resulting system has improved execution time due to the dedicated hardware while still maintaining many of the advantages provided by the software components.

4.3 Introduction to FPGAs and the Nios II Processor

For several decades, the use of FPGAs has been increasing, replacing ASIC-based designs due to their lower costs. FPGAs are devices capable of implementing large logic circuits, and have been extensively used for many different applications such as data processing and storage, digital signal processing, instrumentation, and telecommunication [26].

In the previous section, CPLDs and FPGAs are included in the same group, but there is an important difference between a CPLD and an FPGA. The CPLD contains AND and OR planes, in which the inputs of a CPLD can be "ANDed" together on the AND plane, and the outputs from the AND plane might be logically summed using the OR plane. An FPGA contains logic elements instead of AND and OR planes.

An FPGA can be divided into three main parts: logic elements, programmable interconnectors, and input/output (I/O) blocks. A logic element mainly consists of a look up table (LUT) or several logic gates feeding a flip-flop. The programmable interconnectors, also called routing channels, generate a network to perform more complex operations. These programmable switches connect logic elements with each other and also connect logic elements with I/O blocks. Finally, I/O blocks are connected to the I/O pins. The I/O blocks can be used as input, output, or bidirectional access points.

In this thesis, a low-cost FPGA is used to implement the algorithm due to its low cost, flexibility, and good performance. The manufacturer chosen is Altera, and the FPGA family is Cyclone II [27]. This type of FPGA belongs to the new generation of FPGAs. They have features such as phase-locked loops (PLLs) used for adjusting a high-speed clock signal, multiple accumulators, and multipliers utilized in this thesis for floating point (FP) operations. These FPGAs can be programmed to have a soft core processor such as Nios.

The Nios processor is a general-purpose Reduced Instruction Set Computer (RISC) processor [28]. It is available as a soft core processor, meaning that it is implemented in a Hardware Description Language (HDL), so it can be used on PLDs. This soft core processor is more flexible and its configurability is higher, but it has lower performance than hard core processors. The processor is implemented using logic elements from the FPGA, which gives the designer the flexibility of choosing many design components, such as memory sizes, number and type of peripherals, etc. In addition to the processor, a considerable number of FPGA HW/SW design tools are available to modify and test the processor.

This thesis makes use of a Nios II soft core processor to execute the list-based heuristic and SA algorithm. Most of the algorithm is implemented in C, but some parts of the SA algorithm are implemented in VHSIC Hardware Description Language (VHDL) as a custom hardware component, and interfaced to the Nios II processor.

4.4 HW/SW Co-design Implementation

Observing the results of the previous chapter, the SA algorithm takes 98.9% of the computational time of the entire software implementation. Therefore, a hardware implementation for most of the SA algorithm is designed. The list-based heuristic remains in software, and the

results of the initial schedule are sent to the hardware SA component. The hardware SA component behaves as a pseudo SA algorithm, except for some parts that are executed in software on the Nios II processor. Figure 11 shows the transition from software implementation to hardware implementation.

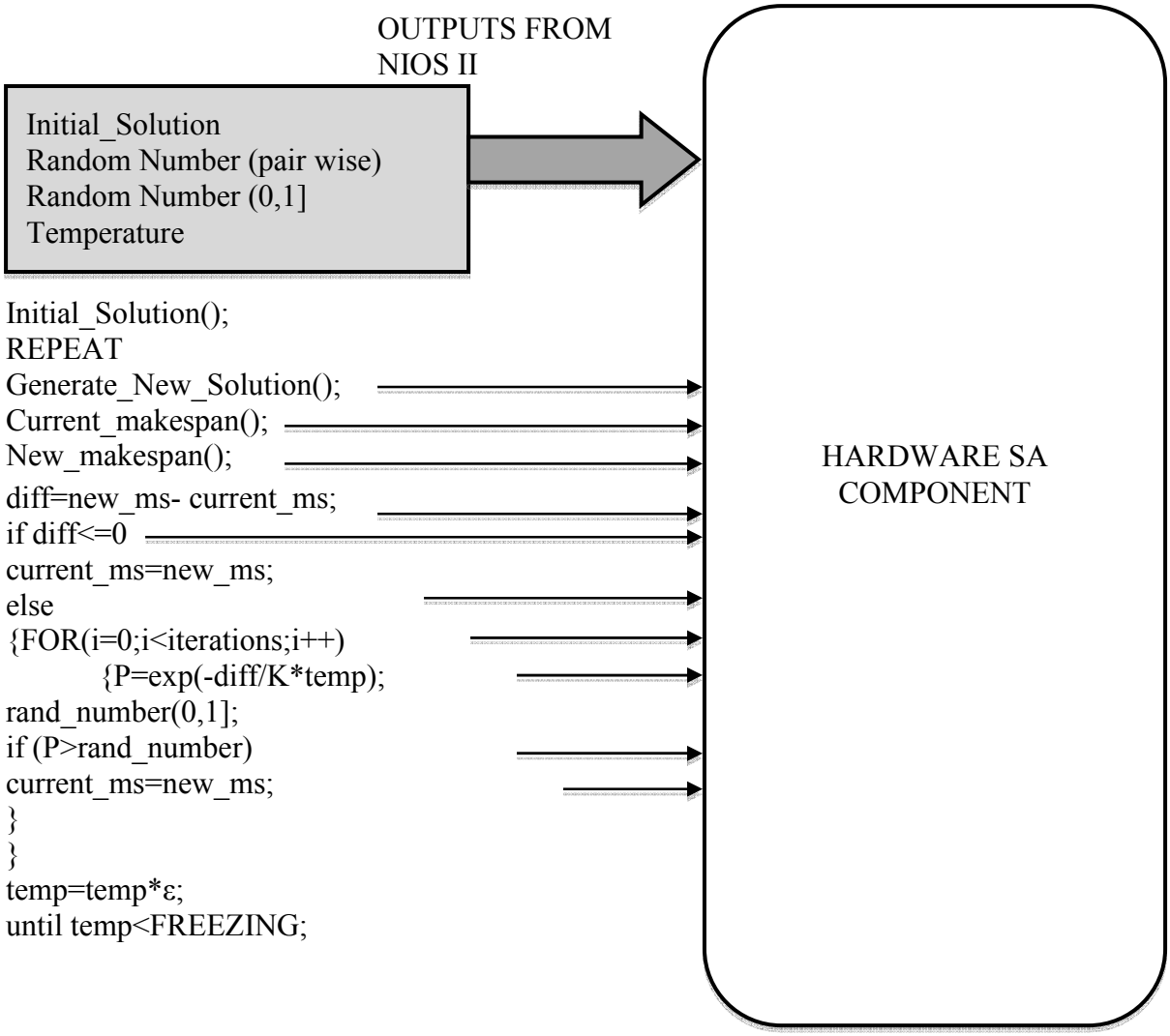


Figure 11. Software functionality migrated to dedicated hardware.

All arrows indicate a transition of functionality from a specific function in software to the hardware SA component. The elements enclosed in the gray box are the outputs from the

software executing on the Nios II processor into the hardware SA component. Thus, the initial solution, which is calculated in software, is sent to the hardware SA component through direct register access. Also, a random number integer is sent to the hardware SA component via direct register access to generate the new solution. Furthermore, two FP numbers are provided to the hardware SA component for different purposes. The first FP is compared with the probability generated in the hardware SA component. This FP value is a random number between 0 and 1. If the random number is smaller than the probability, the solution is accepted as a possible optimal solution. The second FP number is the current temperature in that moment in the algorithm. The temperature is not calculated in the hardware SA component due to a lack of resources, specifically an insufficient number of embedded multipliers.

4.5 Hardware Implementation Model

This section describes the hardware SA component. A modular view of the hardware SA component (most of the SA algorithm) is shown in Figure 12.

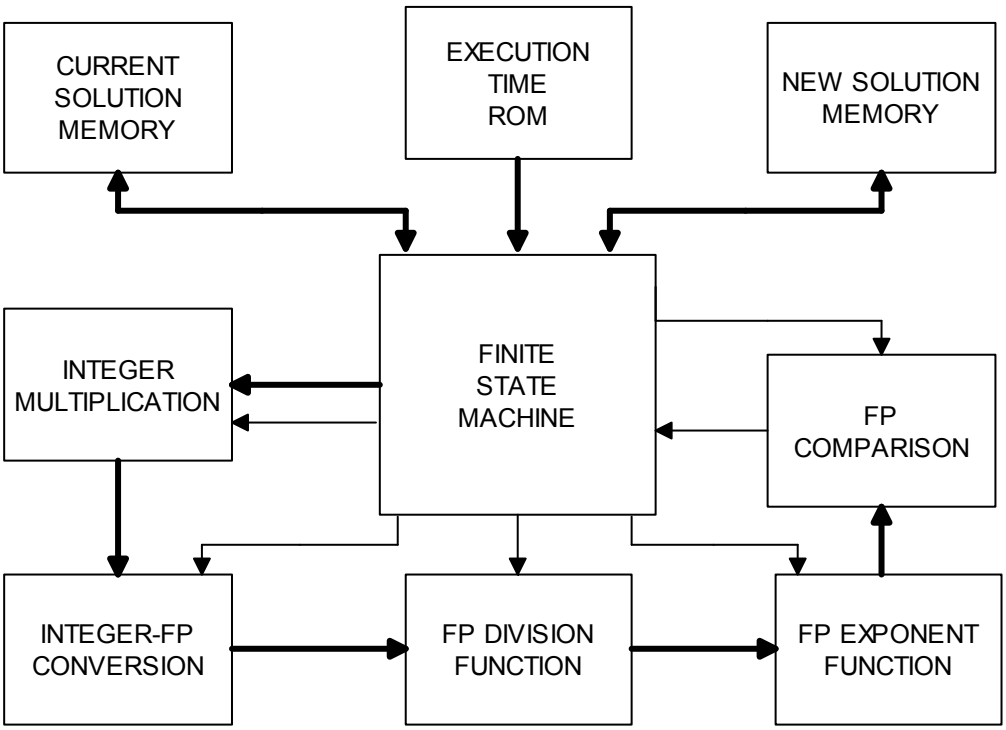


Figure 12. Modular Description of Hardware SA Component.

Current Solution Memory: This memory is a Random Access Memory (RAM) that contains the initial solution calculated by the list-based heuristic. This memory is also used to store all the possible solutions that are accepted in the algorithm.

New Solution Memory: This memory is another RAM memory. The memory contains all the possible random new solutions that are generated.

Execution Time ROM: This Read-Only Memory (ROM) is defined to store the execution time of every task.

These three memories are used to find the makespan of the current solution and the makespan of the new solution generated. With the schedules and the execution time of every task already established, the makespans of the current and new solutions are easy to calculate by the finite state machine (FSM).

Integer Multiplication: This component is utilized to multiply the difference between the new solution makespan and the current solution makespan by 100. In the software implementation, a constant of 0.01 is used as one of the divisors of the difference. The purpose of this constant is to approximate an acceptance criterion close to zero when temperatures in the algorithm are low. However, taking into consideration the amount of resources used by FP division, the design decision of multiplying by an integer has been made, in order to make better use of resources and eliminate one out of two FP divisions that are needed in this design. FP operations use embedded multipliers to perform operations. Although the integer multiplication takes four 9-bit embedded multipliers to obtain its results, by using this integer multiplier, the algorithm does not need to use FP division and FP multiplication.

Integer to FP Conversion: After multiplying the difference by 100 and subtracting this result from 0, a conversion from integer format to FP format is needed. In the hardware implementation, an IEEE 754 32-bit FP format is utilized, and the Altera Floating Point Megafunctions are used to calculate every operation that includes FP standard format [29].

FP Division: This megafunction divides the result from the integer to FP conversion by the current temperature that is received from the Nios II. All the FP operations in the hardware implementation contain a clock enable and a clear input. Thus, every operation is asserted with its clock enable only when necessary.

FP Exponent: After the division and the wait of the latency of the FP division component, an exponent of the result is calculated. Different operations need different latencies [29]. Table 2 in the next section presents the latency of each component.

FP Comparison: A comparison is made between the result from the FP exponent component, called probability P, and the random number in the range between 0 and 1 provided

by the software executing on the NIOS II. The FP comparison component asserts an output of 1 when the random number is smaller than the probability P; therefore the new solution generated previously is accepted as a current solution.

FSM: The FSM is in charge of most of the important operations in the algorithm. The FSM starts by enabling the write process in the RAM to store the initial solution. Then, the makespan of the initial solution is found. At the same time, the FSM asserts the write operation in the *New Solution Memory*. The write operation copies the initial solution into the new solution. Later, this schedule is modified, generating a new solution schedule. When the integer random number is received from the Nios II, a new solution is generated, and the FSM finds the new schedule makespan. Next, the difference between both schedule makespans is calculated. If the difference is less than or equal to zero, the solution is accepted, and the FSM enables the write operation in the *Current Solution Memory*. Thus, the new solution is copied into the *Current Solution Memory* and is declared as a current optimal solution. On the other hand, if the difference between schedule lengths is positive, a different set of states is asserted. These states are in charge of asserting the clock enable for every operation in order to calculate the probability and to compare with the random number. Also, when some latency is necessary, the FSM inserts the needed latency to keep the operations properly synchronized. If the solution is accepted, the FSM returns to the state in which the write operation starts to copy the new solution into the *Current Solution Memory*. Finally, a state to count the number of iterations in each temperature is implemented, followed by a last state in which the new temperature is received and the iteration counter is reset. The temperature received is used in subsequent iterations of the algorithm. In order to clarify the description of the hardware SA component, the FSM diagram is shown in Figure 13.

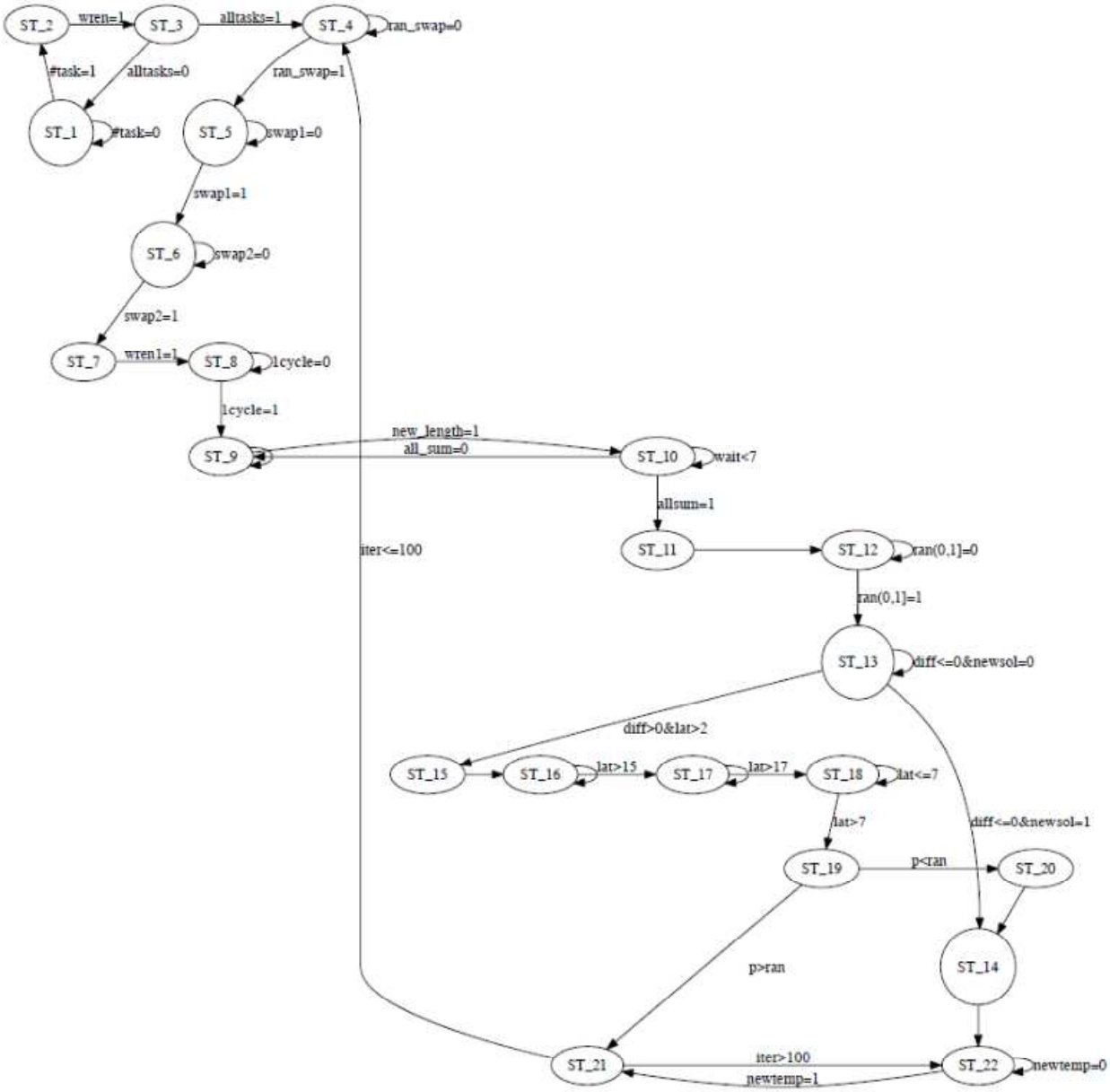


Figure 13. FSM of the Hardware SA Component.

4.6 Results

The co-design implementation is tested with all the graphs shown in Figure 7, obtaining the same makespan as the purely software implementation. In this part of the thesis, the list-

based heuristic still remains in software, but it is now executed using the Nios II processor. However, most of the SA algorithm is implemented in VHDL, with some parts in C, such as the random numbers to be used and the generation of a new temperature. The final makespan of each graph is shown on the LEDs interfaced to the FPGA. These results can be demonstrated by the use of a conduit interface that allows information to be shown outside of the logic component [30].

The FPGA used belongs to the Cyclone II family [27], and it is embedded in the DE1 board. Table 1 shows the resources used for the implementation using a Nios II processor.

Table 1.Resources Used for the Hardware SA Component.

Total Logic Elements	7,589/18,752 (40%)
Total Registers	4800
Total Pins	143/315 (45%)
Total Memory Bits	70,016/239,616 (29%)
Embedded Multiplier 9-bit Elements	51/52 (98%)
Total PLL	0/4 (0%)

In Table1 the noticeable use of almost all of the board's multipliers, almost 100%, is due to the use of FP operations. Also, since the design works at a frequency of 50MHz, it was not necessary to divide or multiply the clock rate.

The latencies for the different FP components are shown in Table 2. These latencies show the number of cycles necessary to perform a specific operation. These latencies are necessary to obtain good outputs for a function. It is observed that the FP exponent function has the longest latency. In contrast, the conversion of an integer number to a FP number only requires a clock cycle to be performed.

Table 2.Latency of Each Function.

Function	Latency (Clock Cycles)
Integer Multiplication	3
Conversion to FP	1
FP Division	15
FP Exponent	18
Comparison of Two FP Numbers	8

CHAPTER 5

COMPARISON BETWEEN DESIGNS

This chapter starts with an explanation of the tool, a counter core, used to measure the speed of the different designs. Next, a study of the modification of several parameters in the SA algorithm is explained. In closing, the chapter provides a comparison of both designs, the purely software design and the HW/SW co-design.

5.1 The Counter Core

In order to profile the code designed for the algorithm, a non-interfering tool is implemented in the Nios II processor [31]. The tool consists of a set of counters that precisely measures code execution times. By adding an instruction at the beginning and the end of the code that is going to be measured, the counter core determines the number of clock cycles required for that section of the code. Other alternatives for measuring execution time are available in the Nios II processor, including interval time peripherals and the GNU profiler [32].

The counter core is integrated in the SOPC Builder, and no RAM is needed for its implementation [33]. Furthermore, the counter core utilizes two counters for each section. It has two main sections: a time section and an event section. The time section employs a 64-bit clock cycle counter, while the event section operates with a 32-bit event counter. By using macros and functions defined in the Nios II application program interface (API), several operations can be performed, such as reset the counter, starting and stopping the counter, printing a report with the

main values, measuring the time of a section, and measuring the total time of the program. Lastly, the API can be used to obtain a CPU frequency and to observe the number of events.

5.2 Study of Several Parameters of the SA Algorithm

This section compares the pure software design with the HW/SW co-design. Obtaining similar results in the final solution, it is possible to observe the execution times required for each task system as the SA algorithm's temperature parameter and iteration parameter are modified. In Table 3, the algorithm is tested for different initial temperatures, while the number of iterations is set to 50 and the freezing temperature is set to 1.

Table 3.Execution Times for Several Temperatures.

# Tasks	Software Only Design			HW/SW Co-Design			Speedups		
	Temperatures			Temperatures			Temperatures		
	250	200	100	250	200	100	250	200	100
18	4.6732	4.5379	4.0867	0.2296	0.2215	0.1928	20.3536	20.4871	21.1965
17	4.9070	4.7743	4.1673	0.2288	0.2206	0.1917	21.4466	21.6423	21.7386
12	5.0163	4.8951	4.2796	0.2267	0.2184	0.1896	22.1275	22.4134	22.5717
11	4.6397	4.5017	3.9802	0.2263	0.2180	0.1892	20.5424	20.65	21.037
9	4.1821	4.1883	3.7402	0.2362	0.2179	0.1888	17.7057	19.2212	19.8104
7	4.2705	3.8273	3.3834	0.2261	0.2178	0.1888	18.8877	17.5725	17.9205

In Table 3, the different execution times for several temperatures may be observed. The execution times displayed are in seconds, and the algorithm is executing on the Nios II processor with a frequency of 50MHz. The difference among designs is obvious, with the HW/SW co-design having a significant reduction in execution times. The execution times for higher temperatures are greater than those for lower temperatures due to the number of iterations of the algorithm. However, this time difference is not very substantial compared with the variation of the number of iterations in the algorithm, as discussed below. Also, the number of tasks affects

the execution times of the algorithm, having more effect in the purely software design. Furthermore, Table 3 shows all the speedups between designs. All the speedups for different temperatures are compared with no major difference between them. All the speedups are approximately in the range of 17.5 – 23.5 for different task systems, being greater when task systems have 11 tasks or more.

Although for most task systems the execution time increases with the number of tasks, some task systems with lesser number of tasks are slower than others with greater number of tasks. For instance, the task system with 17 tasks is slower than the task system with 18 tasks when tested using the software only implementation. This behavior is due to the number of probabilities accepted in the algorithm every time that a different task system is executed. However, it is very difficult to predict how many solutions are going to be accepted, since the probabilities are compared to a random number. Thus, every time that a probability is accepted, the new solution needs to be stored as the current solution, which will increase the overall execution time of the algorithm.

In the next table, Table 4, the algorithm is executed with a different number of iterations on each temperature with the initial temperature set to 100 and the freezing temperature set to 1.

Table 4. Execution Times for Different Numbers of Iterations.

# Tasks	Software Only Design		HW/SW Co-Design		Speedup	
	Iterations		Iterations		Iterations	
	150	100	150	100	150	100
18	12.1359	8.1843	0.5506	0.3716	22.0412	22.0244
17	12.5355	8.4306	0.5501	0.3710	22.7876	22.7239
12	12.8683	8.5434	0.5492	0.3694	23.4310	23.1277
11	11.9275	7.9811	0.5488	0.3690	21.7338	21.5880
9	11.4577	7.8308	0.5494	0.3697	20.8549	21.1815
7	11.3245	7.5186	0.5505	0.3699	20.5712	20.3260

Again, the execution times in the software only design are quite large compared to those of the HW/SW co-design. Both designs are executed on the Nios II processor with a speed of 50Mhz, and the values are shown in seconds. The total execution time increases to 12 seconds in the software only design, when the number of iterations is set to 150 iterations for each temperature. This increase is due to a substantial increase in the number of total iterations in the algorithm. For instance, for every calculated temperature, the algorithm executes 100 or 150 times until the temperature decreases below the freezing temperature. Using HW/SW co-design, a very important reduction in the algorithm's total execution time is achieved. Also, Table 4 shows that some task systems with fewer tasks take more time to execute than others with a higher number of tasks. Again, this effect is due to the number of probabilities accepted. Furthermore, just as in Table 3, the difference in execution times for different task numbers is more noticeable in the software only design. Finally, the speedup results are shown in Table 4, which compares both designs. In this table, speedups are in the range of 20 to 23. Comparing the speedups from Table 4 with Table 3, speedups from Table 4 are greater than those of Table 3. This also is due to the overall number of iterations. Although the initial temperature is set to 100 in Table 4, the total number of iterations is significantly higher than the number of iterations from Table 3.

Figure 14 and Figure 15 show the speedup of the algorithm, comparing both designs. Figure 14 shows the speedup when the algorithm is tested with the minimum number of iterations in the overall SA algorithm. The data plotted is from that shown in Table 3 for a temperature of 100. In contrast, Figure 15 demonstrates the speedup between designs when the algorithm is tested with the maximum total number of iterations. The data plotted in Figure 15 is from Table 4. Comparing both figures, a slight increase of the speedup for some of the task

systems is observed when the total number of iterations increases. These results show that the use of a HW/SW co-design provides an important improvement in the execution time of the SA algorithm for the scheduling problem.

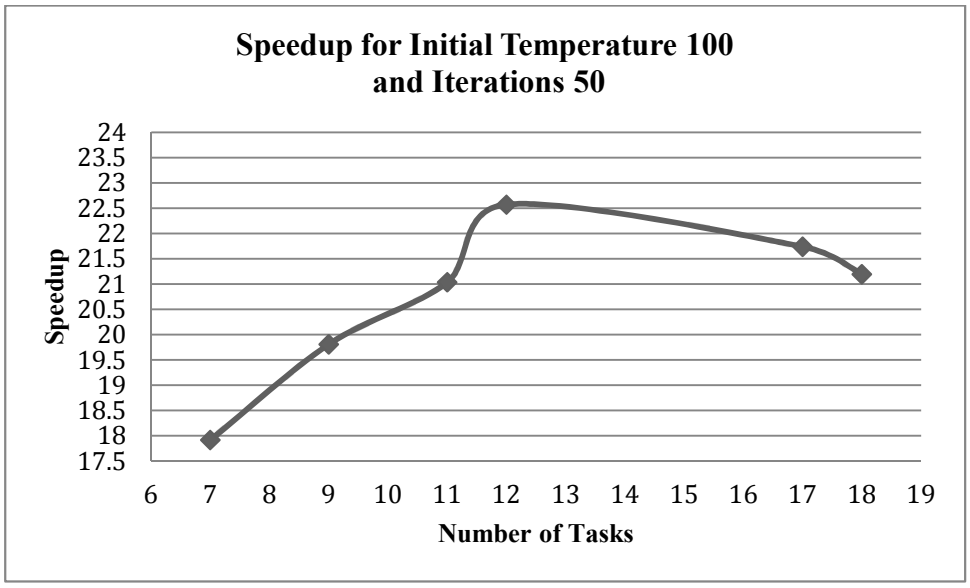


Figure 14. Speedup for Initial Temperature 100 and Iterations 50.

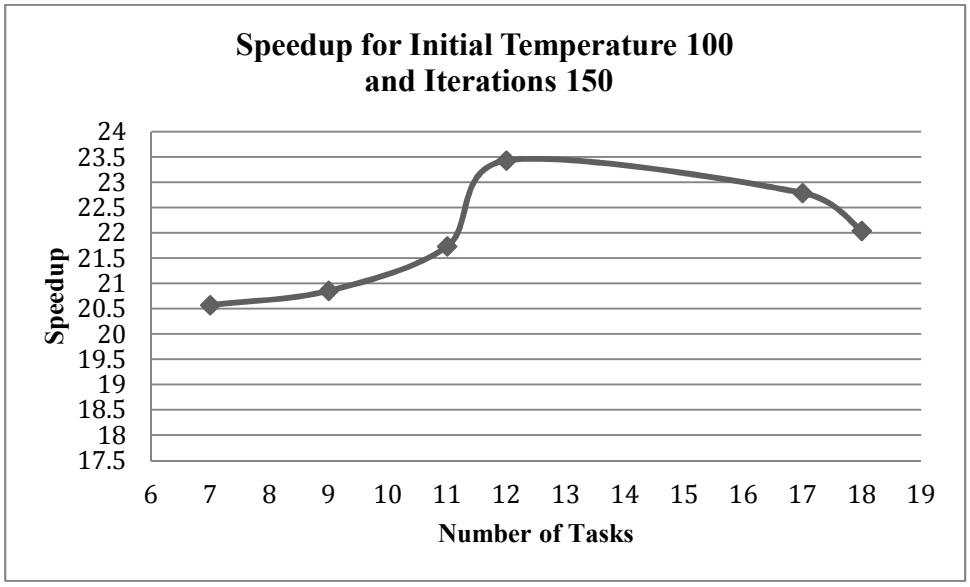


Figure 15. Speedup for Initial Temperature 100 and Iterations 150.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This thesis addresses the scheduling problem, an important combinatorial problem, by combining a heuristic and an optimization technique known as SA. The definition of the problem and the main techniques used to solve it are discussed. Among those techniques, the list scheduling heuristic can obtain sufficient results in terms of execution time, but it never assures a global optimal solution. At the same time, it is observed how SA may approximate an optimal solution, avoiding local minimums and in a relatively short amount of time. In contrast, exact methods might obtain optimal results but require very large execution times. Next, the different aspects of the SA technique and its main applications are addressed. Then, the two techniques of list-based heuristics and the SA are combined, and implemented as a software algorithm in which several different task graphs were tested. Furthermore, the results of the software implementation are studied. Finally, by using a reconfigurable environment, a HW/SW co-design technique to decrease the final execution time of the algorithm is implemented. The results comparing the software implementation and the HW/SW co-design demonstrate a significant improvement in the overall algorithm execution time.

6.2 Future Research

Use of a HW/SW co-design environment has shown an improvement in the final execution time. Future research might employ a parallel version of the SA algorithm [34]. Furthermore, the scheduling problem could be solved using different search techniques such as tabu search [21], and likewise applying this reconfigurable environment in order to improve the overall execution time. Another approach might involve combining two of these search techniques; a hybrid algorithm could be defined in order to obtain a faster polynomial execution time [35]. By adding a genetic algorithm to a SA algorithm, the advantages of the genetic algorithm would be included. Thus, on each temperature of the SA algorithm, a population would be generated and genetic operators would be applied to it. Introducing these changes, the thermal quasi-equilibrium could be easy to reach. Moreover, by using advanced FPGAs, the entire SA algorithm could be integrated as a hardware component. These FPGAs have more resources to integrate more FP data and thus would be able to perform more operations such as two FP divisions. For instance, the new advanced generation of FPGAs such as the Stratix family developed by Altera [36] contains a substantial amount of resources including 18 by 18-bit multipliers, more than 800,000 logic elements, and a generous RAM.

REFERENCES

[1] J. E. Beck and D. P. Siewiorek, "Automatic configuration of embedded multicomputer systems," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol 17, no 2, pp. 84-95, Feb 1998.

[2] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, New York: John Wiley & Sons, 1976.

[3] C. H. Papadimitriou and K.Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, New York: Prentice-Hall, 1982.

[4] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering,* vol 14, no 2, pp. 141–154, Feb 1988.

[5] J. Sgall, "On-line scheduling," *Online Algorithms: The state of the art. Springer Lecture Notes in Computer Science*, vol 1224, pp. 196-231, 1998.

[6] W. H. Kohler and K. Steiglitz, "Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems," *Journal of the ACM*, vol 21, no 1, pp. 140-156, 1974.

[7] C.T. Hwang, J.H. Lee, and Y.C. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol 10, no 4, pp. 464–475, April 1991.

[8] M. K. Dhodhi, Imtiaz Ahmad, A.Yatama, and Ishfaq Ahmad, "An Integrated Technique for Task Matching and Scheduling onto Distributed Heterogeneous Computing Systems," *Journal of Parallel and Distributed Computer,* vol 62, no 9, pp. 1338-1361, Sep 2002.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, no. 4598, pp. 671-680, May 1983.

[10] Emile Aarts and Jan Korst, *Simulated Annealing and Boltzmann Machines,* New York: John Wiley & Sons, 1989.

[11] H. El-Rewini, T.G. Lewis, and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems*, New Jersey: Prentice Hall, 1994.

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science,* Vol. 220, no. 4598, pp. 671-680, May 1983.

[13] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equation of State Calculations by Fast Computing," *Journal of Chemical Physics,* vol 21, no 6, pp. 1087-1092, 1953.

[14] P. J. M. van Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Applications,* Dordrecht, Holland: Reidel, 1987.

[15] Thomas L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communication of the ACM,* vol 17, no 12, pp. 685-690, December 1974.

[16] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy," *Real Time Systems*, vol 4, no 2, pp. 145-165, June 1992.

[17] B. Earl Wells and Chester C. Carroll, "An Augmented Approach to Task Allocation: Combining Simulated Annealing with List-Based Heuristics," *Proceedings - Euromicro Workshop on Parallel and Distributed Processing,* pp. 508–515, January 1993.

[18] K. G. Ricks (2006). Hardware Implementation of a Parallelized Genetic Algorithm for Task Scheduling [Online].Available: http://kricks.eng.ua.edu/research/thesis/tirumalai.pdf

[19] T. M. Nabhan and A. Y. Zomaya, "Parallel simulated annealing algorithm with low communication overhead," *IEEE Transactions on Parallel and Distributed Systems*, vol 6, no 12, pp. 1226-1233, December 1995.

[20] B. Eschermann, O. Haberl, O. Bringmann, and O. Seitzr, "COSIMA: A Self-Testable Simulated Annealing Processor for Universal Cost Functions," *Proceedings, Euro ASIC '92*, pp. 374-377, June 1992.

[21]. R. Schneider and R. Weiss, " Hardware Support for Simulated Annealing and Tabu Search," *Proceedings, Parallel and Distributed Processing* ,vol 1800, pp. 660-667, 2000.

[22] F. Vahid and T. Givargis, *Embedded System Design–A Unified Hardware/Software Introduction,* New York: John Wiley & Sons Inc, 2000.

[23] J. O. Hamblen, T. S. Hall, and M. D. Furman, *Rapid Prototyping of Digital Systems,* New york: Springer Science and Business Media LLC, 2008.

[24] W. Wolf, "A Decade of Hardware/Software Codesign," *IEEE Computer SOC,* vol 36, no 4, pp. 38-43, April 2003.

[25] S. M. Loo and B. E. Wells, "Task Scheduling in a Finite-Resource, Reconfigurable Hardware/Software Codesign Environment," *INFORMS Journal on Computing,* vol 18, no 2, pp. 151-172, Spring 2006.

[26] A. K. Maini, *Digital Electronics: Principles, Devices, and Applications,* England: John Wiley & Sons Inc, 2007.

[27] Altera Corp. (2007, Feb). Section I. Cyclone II Device Family Data Sheet. [Online]. Available: http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1_01.pdf

[28] Altera Corp. (2010, July). Nios II Processor Reference Handbook. [Online]. Available: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[29] Altera Corp. (2010, July). Floating-Point Megafunctions User Guide. [Online]. Available: http://www.altera.com/literature/ug/ug_altfp_mfug.pdf

[30] Altera Corp. (2010, July). SOPC Builder Components. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii54004.pdf

[31] Altera Corp. (2007, Oct). Performance Counter Core. [Online]. Available: http://www.ict.kth.se/courses/IL2207/0708/docs/qts_qii55001.pdf

[32] Altera Corp. (2010, May). Profiling Nios II Systems. [Online]. Available: http://www.altera.com/literature/an/an391.pdf?GSA_pos=1&WT.oss_r=1&WT.oss=Profiling Nios II Systems.

[33] Altera Corp. (2010, July). SOPC Builder Components. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii54004.pdf

[34] D. J. Ram, T. H. Sreenivas, and K. G. Subramaniam , "Parallel Simulated Annealing Algorithms," *Journal of Parallel and Distributed Computing,* vol 37, no 2, pp. 207-212, Sep 1996.

[35] Feng-Tse Lin, Cheng-Yan Kao, and Ching-Chi Hsu , "Applying the Genetic Approach to Simulated Annealing in Solving Some NP-Hard Problems," *IEEE Transaction On Systems Man and Cybernetics,* vol 23, no 6, pp. 1752-1767, Dec 1993.

[36] Altera Corp (2011, June). Overview for the Stratix IV Device Family. Available: http://www.altera.com/literature/hb/stratix-iv/stx4_siv51001.pdf

APPENDIX

```
--SIMULATED ANNEALING ALGORITHM IMPLEMENTED ON NIOS II PROCESSOR.
--DEVELOPED BY SEILA GONZALEZ ESTRECHA
--THE UNIVERSITY OF ALABAMA 2010

--LIBRARIES
LIBRARY ieee;
USE ieee.std_logic_1164.all;
useieee.std_logic_signed.all;
useieee.std_logic_arith.all;

--MAIN ENTITY
ENTITY TEST IS
PORT
(csi_clock_clk   : IN STD_LOGIC;
avs_switches0_write   : IN STD_LOGIC ;
avs_switches0_writedata: IN STD_LOGIC_VECTOR (31 DOWNTO 0);
coe_ledr_export   : OUT STD_LOGIC_VECTOR(9 DOWNTO 0)

);
END TEST;

--MAIN ARCHITECTURE
ARCHITECTURE BEHAVIOR OF TEST IS

--SIGNALS
SIGNAL wren,w_sol,key,valid,clear,en_conv,en_div,en_exp:STD_LOGIC;
SIGNAL en_mult,en_comp,alb,en_fpint,en_mult1,cl:STD_LOGIC;
SIGNAL count:STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL addr_rom,latency:STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL addr_sol,task_num:STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL addr:STD_LOGIC_VECTOR(6 DOWNTO 0);
SIGNAL length1,length0,len0,len1,iter,q_rom:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL makespan,ms_new,dif:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL SW,data,q_out,q_sol,data_sol,res_mult:STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL swap,swap1:STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL dif_32,res,res_d,res_e,ran,temp:STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL res_mul,res_mul2,res_int:STD_LOGIC_VECTOR(31 DOWNTO 0);

--CONSTANTS
CONSTANT initial_temp:STD_LOGIC_VECTOR(31 DOWNTO
0):="01000010110010000000000000000000";--100.00(fp)
CONSTANT epsilon:STD_LOGIC_VECTOR(31 DOWNTO 0):="00111111011100110011001100110011";
--0.95
```

```
--FINITE STATE MACHINE
TYPE states IS
(read_s,write_s,wait_s,makespan0,delay,new_solution,new_solution1,new_solution2,makespan1,compar
ator,comparator1,diff,copy_newsol,write_newsol,pause,conv,div,exp,compare,compare1,accept_sol,iterat
ion,freezing);
SIGNAL  state:states;

--DESIGN COMPONENTS
--MEMORY RAM TO STORE THE INITIAL SCHEDULE AND THE ACCEPTANCE SCHEDULES
COMPONENT altsyncram
GENERIC (
clock_enable_input_a: STRING;
clock_enable_output_a: STRING;
intended_device_family: STRING;
lpm_hint: STRING;
lpm_type: STRING;
numwords_a: NATURAL;
operation_mode: STRING;
outdata_aclr_a: STRING;
outdata_reg_a: STRING;
power_up_uninitialized: STRING;
ram_block_type: STRING;
widthad_a: NATURAL;
width_a: NATURAL;
width_byteena_a: NATURAL
);
PORT (
wren_a: IN STD_LOGIC ;
clock0: IN STD_LOGIC ;
address_a: IN STD_LOGIC_VECTOR (6 DOWNTO 0);
q_a: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
data_a: IN STD_LOGIC_VECTOR (15 DOWNTO 0)
);
END COMPONENT;

--MEMORY ROM THAT STORES THE EXECUTION TIMES OF EVERY TASKS

COMPONENT exec_times IS
PORT
(
address: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
clock: IN STD_LOGIC  := '1';
q: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
);
END COMPONENT;

--MEMORY RAM THAT STORES ALL THE NEW POSSIBLE SOLUTIONS
COMPONENT solutions IS
PORT
(
address: IN STD_LOGIC_VECTOR (5 DOWNTO 0);
clock: IN STD_LOGIC ;
data: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
wren: IN STD_LOGIC ;
q: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
```

```
);
END COMPONENT;

--COMPARATION FLOATING POINTS
COMPONENT  comparasion_altfp_compare_66c IS
 PORT
(
aclr:IN  STD_LOGIC := '0';
alb:OUT  STD_LOGIC;
clk_en:IN  STD_LOGIC := '1';
clock:IN  STD_LOGIC;
dataa:IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
datab:IN  STD_LOGIC_VECTOR (31 DOWNTO 0)
 );
 END COMPONENT;

 --MULTIPLICATION IN FLOATING POINT
COMPONENT  multiplication_altfp_mult_dgk IS
 PORT
(
aclr:IN  STD_LOGIC := '0';
clk_en:IN  STD_LOGIC := '1';
clock:IN  STD_LOGIC;
dataa:IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
datab:IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
result:OUT  STD_LOGIC_VECTOR (31 DOWNTO 0)
 );
 END COMPONENT;

 --CONVERSION FROM INTEGER TO FLOATING POINT STANDARD
COMPONENT conversion IS
PORT
(
aclr: IN STD_LOGIC ;
clk_en: IN STD_LOGIC ;
clock: IN STD_LOGIC ;
dataa: IN STD_LOGIC_VECTOR (31 DOWNTO 0);
result: OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
);
END COMPONENT;

--DIVISION OF FLOAT NUMBERS
COMPONENT  division_altfp_div_pst_b1f IS
 PORT
(
aclr:IN  STD_LOGIC := '0';
clk_en:IN  STD_LOGIC := '1';
clock:IN  STD_LOGIC;
dataa:IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
datab:IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
result:OUT  STD_LOGIC_VECTOR (31 DOWNTO 0)
 );
 END COMPONENT;

 --EXPONENT OF A FLOAT NUMBER
COMPONENT  exponent_altfp_exp_fkd IS
```

```
 PORT
(
aclr:IN  STD_LOGIC := '0';
clk_en:IN  STD_LOGIC := '1';
clock:IN  STD_LOGIC;
data:IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
result:OUT  STD_LOGIC_VECTOR (31 DOWNTO 0)
 );
 END COMPONENT;

 --MULTIPLICATION OF A INTEGER WITH A CONSTANT
COMPONENT  mem_mult_altmemmult_1jo IS
 PORT
(
clock:IN  STD_LOGIC;
data_in:IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
result:OUT  STD_LOGIC_VECTOR (15 DOWNTO 0);
result_valid:OUT  STD_LOGIC;
sclr:IN  STD_LOGIC := '0'
 );
 END COMPONENT;

 --CONVERTION FROM FP TO INTEGER
COMPONENT  fp_to_int_altfp_convert_hvn IS
 PORT
(
aclr:IN  STD_LOGIC := '0';
clk_en:IN  STD_LOGIC := '1';
clock:IN  STD_LOGIC;
dataa:IN  STD_LOGIC_VECTOR (31 DOWNTO 0);
result:OUT  STD_LOGIC_VECTOR (31 DOWNTO 0)
 );
 END COMPONENT;
BEGIN

--INSTANCES
altsyncram_component : altsyncram
GENERIC MAP (
clock_enable_input_a => "BYPASS",
clock_enable_output_a => "BYPASS",
intended_device_family => "Cyclone II",
lpm_hint => "ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=MEM1",
lpm_type => "altsyncram",
numwords_a => 128,
operation_mode => "SINGLE_PORT",
outdata_aclr_a => "NONE",
outdata_reg_a => "CLOCK0",
power_up_uninitialized => "FALSE",
ram_block_type => "M4K",
widthad_a => 7,
width_a => 16,
width_byteena_a => 1
)
PORT MAP (
wren_a => wren,
clock0 =>csi_clock_clk,
```

```
address_a =>addr,
data_a => data,
q_a =>q_out
);
ROM_INSTANCE: exec_times PORT MAP(address=>addr_rom, clock=>csi_clock_clk, q=>q_rom);
RAM_SOLUTIONS_INSTANCE:solutions PORT
MAP(address=>addr_sol,clock=>csi_clock_clk,data=>data_sol,
wren=>w_sol,q=>q_sol);

--MULTIPLIES THE DIFFERENCE BETWEEN SCHEDULE LENGHTS PER 100.
MEM_MULT_INSTANCE:mem_mult_altmemmult_1jo PORT MAP(clock=>csi_clock_clk,
data_in=>dif,result=>res_mult,
result_valid=>valid,sclr=>clear);

--CONVERTION OF THE NEGATIVE DIFFERENCE MULTIPLICATION TO A STANDARD FLOATING
POINT
CONVERSION_INSTANCE:conversion PORT
MAP(aclr=>clear,clk_en=>en_conv,clock=>csi_clock_clk,dataa=>dif_32,result=>res);

--DIVISION OF THE PREVIOUS RESULT BY THE TEMPERATURE
DIVISION_INSTANCE:division_altfp_div_pst_b1f PORT
MAP(aclr=>clear,clk_en=>en_div,clock=>csi_clock_clk,
dataa=>res,datab=>temp,result=>res_d);

--CALCULATION OF THE EXPONENT OF THE PREVIOUS RESULT
EXPONENT_INSTANCE:exponent_altfp_exp_fkd PORT
MAP(aclr=>clear,clk_en=>en_exp,clock=>csi_clock_clk,data=>res_d,result=>res_e);

--MUTL_INSTANCE:multiplication_altfp_mult_dgk PORT
MAP(aclr=>clear,clk_en=>en_mult,clock=>csi_clock_clk,dataa=>temp,datab=>epsilon,result=>res_mul);

--COMPARATION IF A RANDOM NUMBER BETWEEN 0 AND 1 IS LESS THAN THE PREVIOUS
RESULT
COMPARE_INSTANCE:comparasion_altfp_compare_66c PORT
MAP(aclr=>cl,alb=>alb,clk_en=>en_comp,clock=>csi_clock_clk,
dataa=>ran,datab=>res_e);

--CONDUIT: IT SHOWS THE MAKESPAN OF THE LAST ACCEPTED SOLUTION
coe_ledr_export<="00" & makespan;

--PROCESS TO OBTAIN DATA FROM NIOS II PROCESSOR.
--IT GETS THE FIRST INITIAL SOLUTION
PROCESS (avs_switches0_writedata)
BEGIN
IF avs_switches0_write='1' THEN
SW<=avs_switches0_writedata(15 downto 0);
END IF;
END PROCESS;

--PROCESS THAT DEVELOPS THE FINITE STATE MACHINE
PROCESS (csi_clock_clk)
BEGIN
IF RISING_EDGE(csi_clock_clk) THEN
CASE state IS
WHEN read_s=>--INITIALIZATION STATE
wren<='0';
```

```
key<='0';
en_div<='0';
en_exp<='0';
en_conv<='0';
en_mult<='0';
temp<=initial_temp;
IF avs_switches0_write='0' THEN
                                        state<=read_s;          --IT STAYS HERE UNTIL READ A
                                                                TASKS FROM NIOS II
ELSE
IF count="001" THEN
state<= write_s;
data<=sw;--SECOND, IT WRITES THE TASK NUMBER INTO INITIAL MEM
data_sol<=sw;--ALSO, IT WRITE SAME SOLUTION IN GENERATE SOLUTION MEM
IF sw/=31 THEN
addr_rom<=sw(4 DOWNTO 0); --ROM THAT STORES EXECUTION TIMES
END IF;
ELSE
task_num<=sw(5 DOWNTO 0); --IT FIRST TAKES NUMBER OF TASKS
count<=count+1;
state<=read_s;
END IF;
END IF;

WHEN write_s => --IT WRITES IN BOTH MEMORIES: INITIAL AND GENERATE SOLUTION
wren<='1';
w_sol<='1';
state<=wait_s;

WHEN wait_s =>
wren<='0';
w_sol<='0';
IF data=31 THEN --IF THE DATA IS 31 IT JUMPS TO OTHER ADDRESS
IF addr=task_num-1 THEN --THIS INCLUDES TASK SCHEDULE IN THE SECOND PROCESSOR
addr<="0001011";
addr_sol<="001011";
END IF;
ELSE
addr<=addr+1;
addr_sol<=addr_sol+1;
END IF;

IF addr=task_num+10 THEN --WHEN THE TASKS ARE IN THE MEMORIES
state<=makespan0;-- SET THE ADDRESSES LINE TO AN EMPTY CELL
addr<="1001101";
addr_sol<="101101";
ELSE
state<=read_s;--IF THE INITIAL SOLUTION IS NOT COMPLETELY STORED
END IF;

WHEN makespan0=>--DISABLE ALL THE CLOCK FOR COMPONENTS
en_conv<='0';
en_div<='0';
en_exp<='0';
en_comp<='0';
cl<='1';
```

```
IF avs_switches0_write='0' THEN --STAY HERE UNTIL RECEIVE THE RANDOM NUMBER
state<=makespan0;
ELSE
addr_sol<=sw(5 DOWNTO 0); --RECEIVE THE POSITION TO SWAP
state<=delay;
END IF;
IF key='0' THEN--IT CALCULATES THE NEW MAKESPAN
IF len1>len0 THEN
makespan<=len1;
ELSE
makespan<=len0;
END IF;
END IF;

WHEN delay=>--START SWAP INSTRUCTION
cl<='0';
IF count<3 THEN
state<=delay;
count<=count+1;
ELSE
count<="000";
swap<=q_sol;--TAKE DATA FROM MEMORY
addr_sol<=addr_sol+"001011";
state<=new_solution;
END IF;

WHEN new_solution=> -- SECOND PART OF SWAP INTRUCTION
IF count<3 THEN
state<=new_solution;
count<=count+1;
ELSE
data_sol<=q_sol; --FIRST SWAP DATA
addr_sol<=addr_sol-"001011";
key<='0';
state<=new_solution1;
END IF;

WHEN new_solution1=>
w_sol<='1';--WRITING THE SWAPING DATA
state<=new_solution2;

WHEN new_solution2=>
w_sol<='0';
IF key='0' THEN
data_sol<=swap; --SET SECOND DATA TO SWAP
addr_sol<=addr_sol+"001011";
key<='1';
state<=new_solution1;
ELSE
addr_sol<="000000"; --START EVERY COUNTER TO COMPUTE MAKESPAN
count<="000";
length0<="00000000";
length1<="00000000";
state<=makespan1;
END IF;
```

```
WHEN makespan1=>
IF count<"011" THEN
count<=count+1;
state<=makespan1;
ELSE
addr_rom<=q_sol(4 DOWNTO 0);
addr_sol<=addr_sol+1;
count<="000";
state<=comparator;
end if;

WHEN comparator=>
IF count<"011" THEN
count<=count+1;
state<=comparator; --CALCULATE THE MAKESPAN OF EVERY PROCESSOR
ELSE
IF addr_sol<task_num THEN
length0<=length0+q_rom;
END IF;
IF addr_sol>"001011" AND addr_sol<task_num+11 THEN
length1<=length1+q_rom;
END IF;
IF addr_sol>task_num+10 THEN
state<=comparator1;
ELSE
state<=makespan1;
END IF;
count<="000";
END IF;
WHEN comparator1=> --IT TAKES THE MAXIMUN LENGTH AS THE NEW MAKESPAN
IF length0>length1 then
ms_new<=length0;
ELSE
ms_new<=length1;
END IF;
state<=diff;

WHEN diff=> --DIFFERENCE BETWEEN THE NEW MAKESPAN AND THE OLD ONE.
dif<=ms_new - makespan;
addr<="0000000";
addr_sol<="000000";
count<="000";
IF avs_switches0_write='0' THEN --IT WAITS HERE UNTIL GETS A NUMBER BETWEEN 0 AND 1
state<=diff;
ELSE
ran<=avs_switches0_writedata;
state<=copy_newsol;
END IF;

WHEN copy_newsol=>
IF dif <=0  THEN --IF DIFFERENCE <0, IT ACCEPTS THE NEW SOLUTION
makespan<=ms_new;
wren<='0';
IF count<"011" THEN
count<=count+1;
state<=copy_newsol;
```

```
ELSE
data<=q_sol;
state<=write_newsol;
END IF;
ELSE--IF DIFERENCE IS GREATER THAN ZERO WAIT LATENCY OF MULTIPLICATION
IF latency<"00010" THEN
latency<=latency+1;
state<=copy_newsol;
ELSE
latency<="00000"; --IT MULTIPLIES DIFFERENCE BY 100 AND RESTS TO 0
dif_32<="00000000000000000000000000000000"-  res_mult;
state<=conv;
END IF;
END IF;

WHEN conv=> --ENABLE THE CONVERSION FROM INTEGER TO STANDARD FLOAT OF 32 BITS
en_conv<='1';
state<=div;

WHEN div=> --ENABLE THE DIVISION OF THE PREVIOUS MULTIPLICATION BY THE ACTUAL
TEMPERATURE
en_div<='1';
IF latency<"01110" THEN
latency<=latency+1;
state<=div;
ELSE
latency<="00000";
state<=exp;
END IF;

WHEN exp=> --ENABLE THE EXPONENT FUNCTION FOR THE PREVIOUS DIVISION
en_exp<='1';
IF latency<"10001" THEN
state<=exp;
latency<=latency+1;
ELSE
state<=compare;
latency<="00000";
END IF;

WHEN compare=> --COMPARE THE PREVIOUS RESULT WITH THE RANDOM NUMBER FROM 0-1
en_comp<='1';
IF latency<"00111" THEN
state<=compare;
latency<=latency+1;
ELSE
state<=compare1;
latency<="00000";
END IF;

WHEN compare1=> --IF P<RAN THEN ACCEPT THE NEW SOLUTION
IF alb='1'THEN
addr<="0000000";
addr_sol<="000000";
count<="000";
cl<='1';
```

```
state<=accept_sol;
ELSE
iter<=iter+1;
state<=iteration;
END IF;
WHEN accept_sol=> --ACCEPTING THE NEW SOLUTION WITH A PROBABILITY
makespan<=ms_new;
wren<='0';
key<='1';
cl<='0';
IF count<"011" THEN
count<=count+1;
state<=accept_sol;
ELSE
data<=q_sol;
state<=write_newsol;
END IF;

WHEN write_newsol=>
wren<='1';
state<=pause;

WHEN pause=>
wren<='0';
count<="000";
IF addr=task_num-1 THEN
addr<="0001011";
addr_sol<="001011";
ELSE
addr<=addr+1;
addr_sol<=addr_sol+1;
END IF;
IF addr>task_num+10 THEN
iter<=iter+1; --INCREASE THE ITERATION NUMBER
state<=iteration;
ELSE
IF key='1' THEN
state<=accept_sol;
ELSE
state<=copy_newsol;
END IF;
END IF;

WHEN iteration=> --REPEAT FOR 50 ITERATIONS
IF iter<"110010" THEN
key<='1';
state<=makespan0;
ELSE
state<=freezing; --WHEN 50 ITERATIONS TEMPERATURE DECREASES
END IF;

WHEN freezing=>
IF avs_switches0_write='0' THEN
state<=freezing;--STAY HERE UNTIL RECEIVE THE NEW TEMPERATURE
ELSE
iter<="00000000";
```

```
temp<=avs_switches0_writedata;
state<=iteration;
END IF;

WHEN OTHERS =>
state<=freezing;
wren<='0';
END CASE;
END IF;
END PROCESS;

--PROCESS THAT CALCULATES THE LENGTH SCHEDULE FOR EVERY PROCESSOR IN THE
INITIAL SOLUTION MEMORY
PROCESS (csi_clock_clk)
BEGIN
IF FALLING_EDGE(csi_clock_clk) THEN
IF addr<task_num AND state=wait_s THEN
len0<=len0+q_rom;
END IF;
IF addr>"001011" AND addr<task_num+11 AND state=wait_s THEN
len1<=len1+q_rom;
END IF;
END IF;
END PROCESS;

END Behavior;
```