15TH NATIONAL COMPUTER CONFERENCE

17-19 November, 1997 – 17-19 Rajab, 1418 H

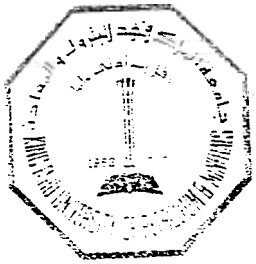
INFORMATION SUPER HIGHWAY

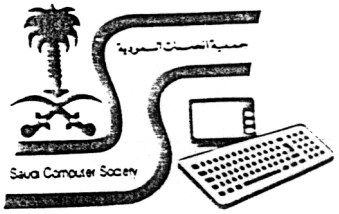
Trends and Impact

Proceedings

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

Saudi Computer Society





Component Selection and Pipelining using Stochastic Evolution Algorithm

Sadiq M. Sait and Talal Maghrabi
College of Computer Sciences & Engineering
King Fahd University of Petroleum and Minerals
KFUPM Box 673, Dhahran-31261, Saudi Arabia
e-mail: sadiq@ccse.kfupm.edu.sa, maghrabi@ccse.kfupm.edu.sa

Abstract

High-level synthesis is the process of translating a high-level program like specification of the behavior of a digital circuit into a structural design in terms of interconnected set of Register-Transfer level components. Component selection and pipelining (CS&P) is one of the important problems in HLS. The time complexity of the exhaustive search algorithm for this problem is exponential. In this paper, we investigate the application of Stochastic Evolution (SE) for solving component selection and pipelining and compare it with simulated annealing (SA). A realistic component library with multiple implementations of operators is used for component selection. Pipelining is done based on the constraints of latency and pipestage delay that are specified. Experimental results on different types of DFG's are reported.

Keywords: Stochastic evolution, Simulated annealing, Component Selection, Pipelining, High-Level Synthesis, Search Algorithms.

1 INTRODUCTION

High-level synthesis (HLS) [1] is the process of translating a high level program like specification of the behavior of a digital circuit into a structural design in terms of interconnected set of Register-Transfer (RT) level components via some intermediate representation. For most applications such as those in digital signal processing (DSP), etc., the intermediate form (IF), as indicated in Figure 1, is a data flow graph (DFG). Nodes of the DFG correspond to operations and arc determine the flow of data in the circuit. Operations scheduling and hardware allocation are two important phases in HLS of circuits from behavioral descriptions. Scheduling distributes the executions of operations throughout time steps. Allocation of hardware cells includes: functional unit, register and bus allocation, and their interconnections [2, 3, 4].

To improve the performance of the generated hardware, the DFG can also be pipelined. Pipelining involves placing registers at appropriate locations in the DFG (between operators) in order to improve the throughput of the computation. Hardware units corresponding to operations are taken from a component/cell library. Most component libraries contain several implementations of the same operator. Component selection involves determining the best selection of components from a realistic library containing many different implementations per operator. The different implementations differ in cost, area, speed, etc. When such a library is available, it is necessary to combine pipelining with the selection of appropriate components from the library to satisfy a given set of constraints. The goal of component selection and pipelining is to maximize the use of slower components and minimize the use of faster components while satisfying a given set of constraints.

In this paper we describe a heuristic based on stochastic evolution (SE) for component selection and pipelining (CS&P). We pose CS&P as a combinatorial optimization problem, and compare the performance of SE and simulated annealing (SA) on a set of randomly generated DFGs [5]. The paper is divided into 5 sections. In the following section (Section 2) we present the background of the work. Definitions, an example, previous work (literature review) and the state space searched are discussed in the section. In Section 3 we introduce the basic SE algorithm, and its application to the CS&P problem. Details of

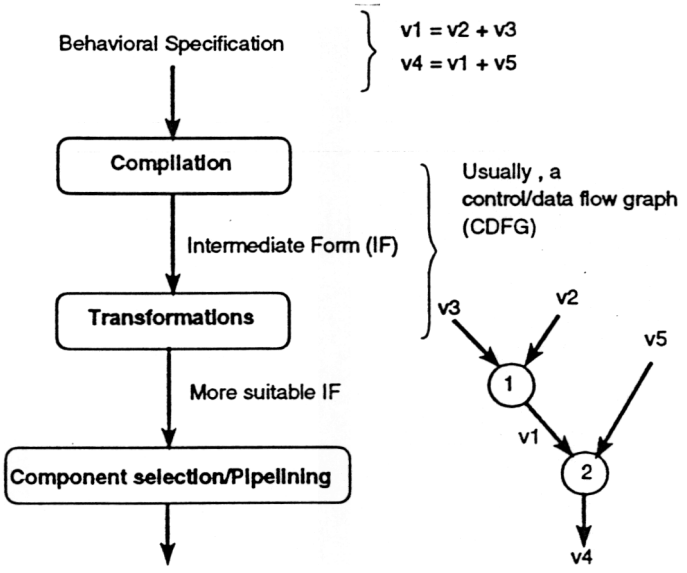


Figure 1: Steps involved in HLS.

the general iterative heuristic, and its adaptation to efficiently solve the CS&P problem are presented. Experimental results and comparison with the Simulated Annealing algorithm are detailed in Section 4. Finally, discussion and conclusions are presented in Section 5.

2 BACKGROUND

The CS&P problem can be formally stated as follows: given a $DFG(V, E)$ where V represents a set of vertices, a set of directed edges $E \subseteq V * V$, a component library CL consisting of a set of three tuples $(Component\_type, Area, Delay)$, and constraints on *pipestage delay* (PS_delay) and *latency*, it is required to find an *assignment* of vertices to components and a partition so as to minimize the cost, where:

- **PS_delay** : Is the sample inter-arrival delay, which is the clock cycle of the design. That is, it is the delay between the arrival of two consecutive input samples. It is also the inverse of throughput.
- **Assignment** : If we associate a type with every vertex v (called $Vertex\_type(v)$, such as *, ÷, +, -), then an assignment is a function from $V \rightarrow CL$ such that if $Assignment(v) = c$ then $Vertex\_type(v) = Component\_type(c)$.
- **Partition** : A *Partition* is a collection of subset of vertices that belong to the same pipeline stage.
- **Latency** : Is the product $n \times PS\_delay$, where n is the number of pipestages. This is also the total time taken before the first output appears.

Example

The DFG consists of vertices which correspond to different operators. The directed edges of the DFG determine the interconnections between vertices, and the flow of data from the input to the output stages. The DFG shown in Figure 2 is an example of the problem. It consists of three multipliers and two adders. The PS_delay constraint is 10 nano-seconds and latency is 20 nano-seconds. The objective is to map the components from the multiple component library shown in Table 1 onto the vertices to obtain a cost-optimal solution. A cost-optimal solution is obtained by an appropriate mapping of components from the component library onto the vertices of the DFG.

The multiple component library shows different types of operators along with the area and delay of each of the operators. The availability of a large number of hardware units corresponding to each function is assumed. Observe that for the same function, fast circuits have larger number of gates and hence are

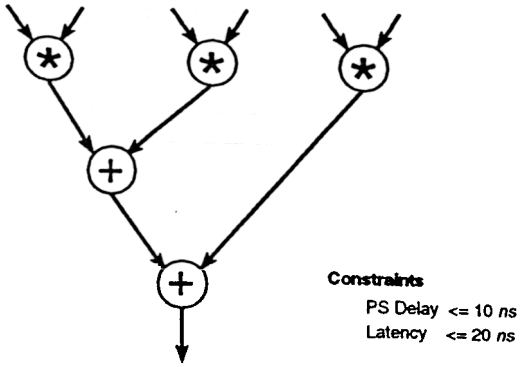


Figure 2: A sample data flow graph

Component Type	Component Name	Area Gates	Delay ns
*	Mpy1	100	30
*	Mpy2	200	20
*	Mpy3	250	10
*	Mpy4	300	5
+	Add1	50	20
+	Add2	70	8
+	Add3	80	5
+	Add4	100	2
Register	Reg	50	

Table 1: Example of a component library.

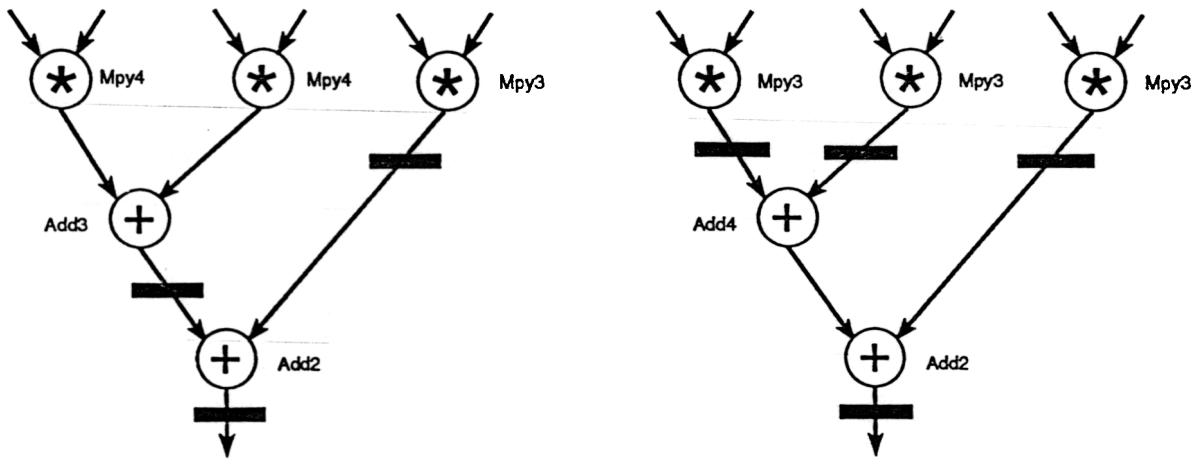


Figure 3: Two possible solutions to the example shown in Figure 1

more costly. Different iterative/constructive techniques may be used to obtain a cost-optimal solution [6]. For the DFG given in Figure 2, and the component library given in Table 1, Figure 3(a) shows one possible solution. This consists of two multipliers of type *Mpy*4, one of type *Mpy*3 and two adders of types *Add*3 and *Add*2 respectively. This solution requires three registers. The total cost is 1150 gates. Figure 3(b) shows another solution, where three multipliers of type *Mpy*3 and two adders of types *Add*2 and *Add*4 are used. This solution requires a total of four registers and its total cost is 1120 gates. Both the solutions satisfy the PS_delay and the latency constraints.

Search Space

The CS&P problem can be phrased in terms of a state-space as follows. A state, or configuration, consists of an assignment of vertices to components, and, selecting for an edge if it should have a register or not. Let n be the total number of nodes (vertices), and m the number of edges in the DFG. Also, let k be the number of types of components for each operator. Each node can be assigned from one of the k types available in the component library. This can be done in k^n ways. For each such assignment, the m edges can have registers assigned to them in 2^m ways. Therefore, the total number of possible configurations are $k^n \times 2^m$. Of course, all of these will not be legal configurations, since assignments of registers to edges is restricted by the rules of pipelining, and all assignments of components to vertices may not satisfy the given constraints. The size of the search space can be reduced to k^n by selecting an assignment of components to vertices from k^n different possibilities, and then using a constructive heuristic to pipeline such that latency and PS_delay constraints are satisfied.

Literature Review

There have been several optimization techniques suggested in the past for solving specific NP-hard problems. The existing optimization techniques [7] can be classified into two types: iterative and constructive.

Constructive heuristics to solve the CS&P problem have been proposed in literature. In [8, 9], Bakshi et al show, that by judiciously selecting vertices to be replaced by slower (less costly) ones, while satisfying constraints, cost can be reduced. Using their proposed heuristic, excellent results are obtained for reasonably sized graphs. Other techniques for synthesizing pipelined datapaths have also been proposed. Scheduling and hardware sharing (allocation) algorithm for synthesizing both pipelined and non-pipelined datapaths is presented in [10]. Shin and Hwang [11] describe the SODAS-DSP system, a pipelined datapath synthesis system, targeted for application specific DSP chip design. The design space of pipelined datapaths is explored to produce an optimal design through facilitated user interaction. Tools like Sehwa [12] and those from GS corporation R&D laboratories [10] have also been used to solve (component selection and pipelining) CS&P problems. These tools pipeline a given DFG so as to optimize area and performance for a given set of constraints, usually on the throughput or latency of the design. However they all assume a single implementation for functional units which forces them to use the same component on non-critical and critical paths, resulting in designs that are inefficient and costly. Implementations of SLIMOS [13] and MOSP [14] differ from the above approach. They start from a multiple implementation library and then select one single implementation per operator, resulting in designs that contain single implementations, thereby leading to the same design inefficiencies as previously discussed methods. A method for pipelining VLSI/ULSI systems for effective communication is proposed in [15]. This proposed method is simple and effective. It also increases the performance. Further details and more on related work can be found in [8, 10, 11, 12, 13, 14, 15, 16]. An excellent review of recent literature on constructive methods to solve the CS&P problem is found in [9].

However, when the size of the graphs grow, then, due to the large search space it becomes very difficult to reach a solution that satisfies constraints using only constructive techniques. It is then that we have to resort to iterative heuristics which have a hill climbing capability. General iterative techniques search the solution space by moving from one solution to another to obtain low-cost or optimal solutions. Examples of iterative techniques are Simulated Annealing (SA) [3], Simulated Evolution [17], Tabu Search (TS) [18, 19, 2], Genetic Algorithm (GA) [20] and *Stochastic Evolution* (SE) [21]. In the area of VLSI design automation (DA) and HLS, iterative techniques have been used to solve a large number of hard optimization problems. For example, in [2, 3], the problem of scheduling and allocation is solved using two iterative techniques, namely GA and TS. Details of other work on the use of general iterative heuristics in the area VLSI DA and HLS can be found in [22, 17, 23, 24, 25]. In the following section we present the SE algorithm and a heuristic based on it to solve the CS&P problem.

The stochastic evolution algorithm is a general iterative heuristic [21] used to solve combinatorial optimization problems. It is a special instance of a more general class of iterative heuristics discussed by Nahar et.al, in [26]. It has been applied to several NP-hard problems such as the traveling salesman, network bisection, and VLSI cell placement [17, 21, 22]. It is stochastic because the decision to accept a move is a stochastic decision. Moves which improve the cost function are accepted with probability one, and bad moves may also get accepted with a non-zero probability. This feature gives SE hill-climbing property.

SE is conceptually simple and elegant. The algorithm is general, in the sense that it can be tailored to solve most known combinatorial optimization problems. It has the capability of escaping local minima, and is blind, i.e., it does not know when the optimal solution has been reached, and has to be told when to stop. However, it has the following fundamental differences with the popular simulated annealing (SA) algorithm [26, 27, 28]: (1) in SA a perturbation of current state (solution) is a single (neighbor) move, while for SE it is a compound move; (2) the acceptance probability of an up-hill move in SA decreases with decreasing values of the temperature, whereas in SE such probability gets increased whenever the search is suspected to have reached some local optimum, and reset to its initial value otherwise; and (3) SE introduces the concept of a *reward* (R) whereby the search algorithm cleverly rewards itself whenever it makes a good move.

SE is based on the concept of state model [21]. A state model is described as a finite set M of movable elements, a finite set L of locations, and the state S defined as as function $S : M \rightarrow L$ satisfying certain constraints.

ALGORITHM *Stochastic_Evolution*(S_0, p_0, R);

```
S_Best = S = S_0;                          /* save initial state */
C_Best = C_Cur = Cost(S);
p = p_0;                                   /* initialize control parameter */
ρ = 0;                                     /* initialize counter */
Repeat
    C_pre = Cost(S)
    S = Perturb(S, p)
    C_cur = Cost(S)
    Update(p, C_pre, C_cur)
    If (C_Cur < C_Best) Then
        S_Best = S                         /* save best state */
        C_Best = C_Cur
        ρ = ρ − R                          /* decrement counter by R */
    Else
        ρ = ρ + 1                          /* increment counter */
    EndIf
Until ρ > R                                /* stopping criteria */
Return (S_Best)                            /* report best state */
```

Figure 4: Stochastic Evolution Algorithm [21].

The Algorithm

The general structure of the algorithm is given in Figure 4. The input to SE is an initial state S_0, an initial value p_0 of the control parameter p, and parameter R which is used in the stopping criterion. The initial state S_0 is a valid state satisfying all the constraints specified by the problem under consideration. The initial state is assumed to be the best state on invocation of the algorithm.

After initialization, the algorithm enters a **Repeat** loop which is executed **Until** the counter ρ exceeds R. Inside the **Repeat** body the cost of the current state is first calculated and stored in C_{pre}. Then, the *Perturb* function (see Figure 5) is invoked to make a compound move from the current state S. *Perturb* scans the set of movable elements M according to some apriori ordering and moves every $m \in M$

```
FUNCTION Perturb(S, p);
For Each (m ∈ M) according to some apriori ordering Do
   S' =Neighbor(S, m);
   Gain(m) = Cost(S) − Cost(S');
   If (Gain(m) > Randint(−p, 0)) Then
      S = S';
   EndIf;
Endfor;
S =Make_State(S); /* make sure S satisfies constraints */
Return(S);
```

Figure 5: *Perturb* Algorithm.

to a new location $l \in L$. Recall that the current state S is actually a function $S : M \rightarrow L$. When a move is performed, a new state S' is generated, which is *a unique* function $S' : M \rightarrow L$ such that $S'(m) \neq S(m)$ for some movable object $m \in M$. To evaluate the move, the gain function $Gain(m) = Cost(S) - Cost(S')$ is computed. If the calculated gain is greater than some integer r, the move is accepted and S' replaces S as the current state. The integer r is randomly generated in the interval $[-p, 0]$, i.e., $-p \leq r \leq 0$. Since $r \leq 0$, moves with positive gain are always accepted.

After scanning all the movable elements $m \in M$, the **Make_State** routine makes sure that the final state satisfies the state constraints. If the state constraints are not satisfied then **Make_State** *reverses* the fewest number of latest moves until the state constraints are satisfied. This procedure is required when perturbation moves that violate the state constraints are accepted.

```
PROCEDURE Update(p, Cpre, Ccur);
If (Cpre = Ccur) Then /* possibility of a local minimum */
   p = f(p); /* increment p to allow larger up-hill moves */
Else
   p = p0; /* re-initialize p */
EndIf;
```

Figure 6: The *Update* procedure.

The new state generated by *Perturb* is returned to the main procedure as the current state, and its cost is assigned to the variable C_{cur}. Then the routine **Update** (Figure 6) is invoked to compare the previous cost C_{pre} to the current cost C_{cur}. If $C_{pre} = C_{cur}$, there is a good chance that the algorithm has reached a local minimum and therefore, p is increased by replacing it by $f(p)$ to allow larger up-hill moves.

The SE algorithm retains the state of lowest cost among those produced by the function *Perturb*. Each time a state is found which has a lower cost than the best state so far, SE decrements the counter by R, thereby rewarding itself by increasing the number of iterations. This allows a more detailed investigation of the neighborhood of the newly found best solution. If S, however, has a higher cost, ρ is incremented, which is an indication of no improvements.

3.1 Algorithm *Perturb*_CS&P

The core of the SE algorithm is the procedure *Perturb*. In order to tailor SE for CS&P, we replace the *Perturb* function in Figure 4 by another procedure *Perturb*_CS&P(S, p, lt, pd); given in Figure 7.

Referring to Figure 7, the first time the *Perturb*_CS&P function is invoked, the initial solution is the current solution. The procedure begins with a For loop, that is executed n times, where n corresponds to the number of nodes in the DFG. The ordering of these nodes is fixed. The index i corresponds to label of nodes, which are assigned randomly or assigned depending on the level of nodes in the DFG.

```
Algorithm Perturb_CS&P(S, p, lt, pd);
   /* S is the current state */
   /* p is a non-negative control parameter */
   /* lt is the latency constraint */
   /* pd is the PS_delay  constraint */
For i = 1 to n Do
   S_new = Neighbor(S_cur, i)
   /* pipeline the DFG to determine number of registers */
      Pipeline(S_new)                          /* pipeline the DFG */
   Gain = Cost(S_cur) − Cost(S_new)
   If (Gain > Randint(-p,0)) Then
      S_cur = S_new
   EndIf
   If ((Latency_Test(S_new)) = TRUE)) &&
      (PS_Delay_Test(S_new)= TRUE))) Then
         S_t = S_new
   EndIf
Endfor
Intensify(S_t)
Return(S_t)
End
```

Figure 7: The *Perturb* function for CS&P.

In order to generate a neighbor state, the current state has to be disturbed. As discussed earlier, the search space can be traversed by either disturbing an edge (adding/deleting a register), or by disturbing the assignment of the node. Disturbing an edge may lead to several illegal solutions, and will result in wastage of time. In our heuristic, the set of movable elements are restricted to the nodes of the DFG, each of which can be replaced by another from the component library. A new (neighbor) solution is obtained by replacing one of the components of the DFG by another of the same function from the component library. This selection is done randomly. Of course, it is ensured that the delay of the replaced component is less than or equal to the specified PS_delay.

When a neighborhood move is made by disturbing the assignment of node from the component library then the assignment of registers to edges has to be updated. This is because the pipestage delay and latency constraints may no longer hold. The DFG with new values of component delays is pipelined, and this may cause a change in the number of registers. Once a component is disturbed the number of registers and the cost of the new state is calculated. The new cost resulting from the neighborhood move will be due to two terms: (1) the change in the cost of the replaced component and (2) the change in the number of registers. The cost function of the DFG is the cost (number of gates) of all the components of the DFG and the number of pipeline registers. That is, the total cost of the solution is:

$$\text{Total cost} = \text{Cost of Registers} + \sum_{i=1}^{n} \text{Cost}_i \qquad (1$$

where n is the number of vertices in the DFG. The cost of the DFG after disturbing the current state changes. Suppose that a component with ID i whose current cost is C_i is replaced by component with cost C_r. If T is the current cost then the new cost is given by:

$$\text{New Cost} = T - C_i + C_r + \text{New Cost of Registers} \qquad (2$$

where the new cost of the registers is obtained after pipelining the DFG. If the gain (due to change in the cost of the replaced component plus the change in the cost of registers) is less than the random number generated between $-p$ and 0, then the solution is rejected and the *Perturb* function is again executed with state S_{cur} as the current state. If the gain is greater than the generated random number, then the state S_{new} is accepted and the *Perturb* function is invoked with S_{new} as the current state S_{cur}.

Following the acceptance of the new state, a test is made to check if the state is valid. A state is valid if the PS_delay constraint and the latency constraints are satisfied. If the test passes, then the new state is saved in S_t.

To check for latency, the DFG is traversed from the inputs towards the output. If on any path between the input and the output the latency of the DFG exceeds that of the specified latency then the state is rejected. The PS_delay constraint is checked by scanning all the components of the DFG. If the delay of the components in a particular PS stage is greater than the specified PS_delay, then also the state is rejected. For a particular state to be accepted as S_t it is necessary that both the constraints should be satisfied. If the constraints are violated, then the state maybe accepted, (depending on the gain) but is not saved in S_t. The algorithm is repeated until all the vertices of the DFG are perturbed. By keeping track of the last valid state that has to be returned at the end of the execution of the *Perturb*_CS&P, it is not necessary to reverse any moves till a valid state is obtained.

After scanning and replacing all n nodes, the entire DFG is scanned starting from the inputs towards the output and the delays of the components between two consecutive registers is calculated. If the combined delay of the components between two consecutive registers is less than the specified PS_delay then there is a possibility of replacing one or more components by slower components of the same type. If this is possible then some fast components are replaced by slower (cheaper) ones. The replacement of the components should be done such that the the PS_delay and latency constraints are satisfied. By doing this the number of slower components that are used increases and the cost of the solution decreases. This step is similar to the greedy *intensification* step used in TS [18, 19].

The number of iterations for which the SE algorithm is made to run depends on the parameter R, which specifies the reward criteria as explained earlier. In our implementation, we used a value of R between 5 and 25.

Update Function Parameters for CS&P

This function is responsible for updating the value of the control parameter p. Initially, a value equal to the least difference between two components of the component library (CL) is assigned to p, and $f(p)$ was assigned a value equal to the maximum difference between two components in the CL. While experimenting with these values of p and $f(p)$ it was observed that the initial drop in the cost was less because only adders/subtractors could be disturbed. As the iterations increased the changes in the costs were high as the value of $f(p)$ was high.

On keeping the value of $f(p)$ large it was observed that the variations in costs were large. Therefore $f(p)$ had to be reduced. This was done by decreasing $f(p)$ in steps of some multiple of 100 and continuing until the range of variations were lessened. Further reduction of $f(p)$ produced poor results. As the value of p is equal to the least difference between any two components of the CL, only adders/subtractors could be disturbed. Therefore the value of p had to be increased so that multipliers could also be disturbed at the initial value of p. Hence p was increased keeping $f(p)$ constant. On increasing p a certain point was reached beyond which poor results were obtained. Using the above scheme, best values for p and $f(p)$ were determined.

4 EXPERIMENTAL RESULTS AND COMPARISON WITH SA

We used SE on problems of various sizes. However, we needed a measure of how good our solutions were in terms of cost and the required run time. We decided to compare our heuristic with simulated annealing (SA), another well known optimization technique. In this section we briefly describe SA and then discuss our experimental runs using SE and SA.

Simulated Annealing

Simulated annealing is a technique for solving combinatorial optimization problems [26, 27]. It belongs to a class of iterative improvement schemes. It has been applied to several combinatorial optimization problems from various fields like traveling salesman problem, graph partitioning, quadratic assignment, matching, linear arrangement, and scheduling. Resource constraint problems have also been solved using

SA. In the areas of engineering, simulated annealing has been applied to VLSI physical design, image processing, code design, facilities layout, network topology design etc., [29, 28].

```
Algorithm Simulated_Annealing(S0, T0, α, β, M, Maxtime)
/* S0 is the initial solution */
/* T0 is the initial temperature */
/* α is the cooling rate */
/* β is a constant */
/* M represents the time until the next parameter is updated */
/* Maxtime is the total allowed time for the annealing process */
begin
   T=T0
   Sc = S0
   Time = 0
   Repeat
         Call Metropolis(Sc, T, M, lt, pd)
         Time = Time + M
         T = α * T
         M = β * M
   until (Time≥ Maxtime)
   Output best solution
End
```

Figure 8: Simulated annealing algorithm

Simulated annealing procedure shown in Figure 8 starts with an initial solution S_0, an initial temperature T_0, cooling rate α, a constant β which controls the time spent in annealing at a particular temperature, $Maxtime$ that is the total time allowed for the annealing process, and M that represents the time until the next parameter is updated.

```
Procedure Metropolis(Scur, T, M, lt, pd)
   /* Scur is the current state */
   /* T is the initial temperature */
   /* lt is the latency constraint */
   /* pd is the PS_delay constraint */
For i = 1 to M Do
   Snew = Neighbor(Scur, i)
          Pipeline(Snew)          /* pipeline the DFG */
   Gain = Cost(Scur) − Cost(Snew)
      If (Gain > Randint() < e^(−Δc/T))Then
         If ((LATENCY_TEST(Snew) = TRUE) &&
            (PS_DELAY_TEST(Snew) = TRUE)) Then
            Scur = Snew
         EndIf
      EndIf
Endfor
Return Scur
End
```

Figure 9: The *Metropolis* procedure.

The core of the simulated annealing algorithm is the *Metropolis* procedure. This procedure (shown in Figure 9) simulates the annealing process at a given temperature T [30]. It is responsible for producing neighbor solutions and also defining the basis for accepting the new state. The acceptance of the new state

depends on the parameters α, β, T, $Maxtime$, and M.

In this procedure the current state is disturbed to obtain a neighboring state. After obtaining a new state the *cost* of the state is computed in the same way as was done for SE (Section 3). If the cost of the new solution S_n is better than the cost of the current solution S_c, and if latency and PS_delay constraints are met, then the new solution is accepted and S_c is set to S_n. If the new solution has a higher cost in comparison to the original solution S_c, *Metropolis* will accept the new solution on a probabilistic basis. This is to accept up-hill moves. At higher temperatures the probability of large up-hill moves is high and at lower temperatures the probability is small. To accept up-hill moves a random number is generated. The random number generation is assumed to follow a uniform distribution. If this random number is smaller than $e^{-\Delta c/T}$, where Δc is the difference in costs, $(\Delta c = c(S_n) - c(S_c))$, and T is the temperature, the up-hill solution is accepted. That is, the probability that an inferior solution is accepted is $P(random < e^{-\Delta c/T})$. At very high temperatures, (when $T \rightarrow \infty$), $e^{-\Delta c/T} = 1$, and hence the above probability approaches 1. When $T \rightarrow 0$, the probability $e^{-\Delta c/T}$ falls to zero.

Metropolis is also provided with the value M, which is the amount of time for which annealing must be applied at a temperature T. The procedure *Simulated_annealing* simply invokes *Metropolis* at decreasing temperatures. Temperature is initialized to a value T_0 at the beginning of the procedure, and is slowly reduced in a geometric progression; the parameter α is used to achieve cooling. The amount of time spent in annealing at a temperature maybe gradually increased as temperature is lowered [27]. This is done using the parameter $\beta \geq 1$. The variable *Time* keeps track of the time being expended in each call to the *Metropolis*. The annealing procedure halts when *Time* exceeds the allowed time.

In SA, in each iteration only valid solutions are accepted as neighboring solutions, and the cost is calculated only if the current state produced is a valid state. The set of parameters T_0, α, β and M specify the cooling schedule. The method proposed in [28] was used to determine the best values of these parameters. The initial temperature was chosen such that most of the initial transitions to new states are accepted independent of their goodness. That is, the initial acceptance ratio $\chi(T_0)$ is kept close to unity, where $\chi(T_0)$ is given by:

$$\chi(T_0) = \frac{\text{Number of moves accepted at } T_0}{\text{Total number of moves attempted at } T_0} \tag{3}$$

To determine a good value of T_0, initially a very low value is chosen. The acceptance ratio is then calculated by running the algorithm for a fixed number of iterations. At low values of T_0 it was found that $\chi(T_0)$ was less than 0.5. Then T_0 was increased in small multiples, and this procedure was repeated until the $\chi(T_0)$ was close to unity in the range between 0.9 and 1.0. The choice of α should be such that temperature T_0 should be reduced at a uniform rate. Furthermore T_0 should not approach zero quickly. Therefore the temperature is reduced in geometric progression as shown in Equation 4.

$$T_{k+1} = \alpha \times T_k, \; k = 0, 1, ..., \tag{4}$$

In our problem best results were obtained for $\alpha = 0.98$. M, the number of times the *Metropolis* loop is executed at a given temperature was chosen to be equal to the number of vertices in the DFG.

GRAPH	Depth	No. of Vertices
EWF	4	14
Graph 1	14	23
Graph 2	11	23
Graph 3	10	23
Graph 4	15	27
Graph 5	13	17

Table 2: Characteristic of input graphs.

Experimental Results

SE algorithm was tested on different DFGs with varying complexities. The DFGs differed in the number of nodes, types of operators and the complexity of the interconnections between the nodes. The depth

of the DFGs varied from 4 to 15. The characteristics of the DFGs are shown in Table 2. A total of si different graphs were used for experimentation. Due to the non-deterministic nature of the algorithm, th procedure was run on the DFGs a number of times, and the best and average results tabulated. Latenc and PS_delay for each of the DFGs were different as shown in the Table 3. The DFGs are labeled EW (Elliptical wave filter) and Graph 1 through 5.

DFG	R	Cost (gates) Avg.	Cost (gates) best	Iterations	Latency	PS_delay	Time in seconds
EWF	10	23175	21100	27	86	37	7
	15	22277	21100	55			19
	20	21938	21100	118			27
	25	21278	21100	129			29
Graph 1	10	36592	36013	94	247	63	223
	15	36544	35641	185			431
	20	35301	34749	328			753
	25	33505	34958	234			621
Graph 2	10	72667	69360	342	203	63	223
	15	68362	67896	346			815
	20	68063	67889	622			1462
	25	67883	67742	892			2060
Graph 3	10	39016	37847	119	187	63	161
	15	38360	37101	175			234
	20	37209	36689	398			533
	25	37753	37447	328			434
Graph 4	10	59589	58912	136	307	63	591
	15	57266	57158	308			1173
	20	55841	55247	447			1719
	25	55123	54259	831			3183
Graph 5	10	40235	39147	148	286	63	318
	15	40492	39901	287			627
	20	39647	38440	475			1071
	25	39228	38926	515			1112

Table 3: Stochastic Evolution results (with no intensification).

DFG	Latency	PS Delay	Stochastic Evolution Avg cost	Stochastic Evolution Best cost	Stochastic Evolution Time	Simulated Annealing Avg cost	Simulated Annealing Best cost	Simulated Annealing Time	% Red.
EWF	86	37	22167	21100	241	20674	20674	239	−2.60 %
Graph 1	247	63	34743	34517	2451	34596	34569	2050	+0.15 %
Graph 2	203	63	67009	65931	2332	66281	66209	2237	+0.42 %
Graph 3	187	63	37221	36689	1388	37289	37199	1375	+1.39 %
Graph 4	307	63	55909	54259	3939	52901	52696	3189	−2.88 %
Graph 5	286	63	39183	38440	2283	39083	39042	2141	+1.54 %

Table 4: Stochastic Evolution and Simulated Annealing results.

SA was also run on the same DFGs. Table 4 shows the results of SA and SE, both the average cc and best cost are tabulated, where the cost is indicated in terms of the number of gates. Improvement SA over SE is shown as % reduction. A bar chart depicting the improvement is given in Figure 10. It seen that in four of the six DFG's SE performs better than SA.

The variation in costs for SA is initially large and then it gradually decreases and remains consta after a certain period of time. This is because the cooling schedule in SA decreases gradually hence

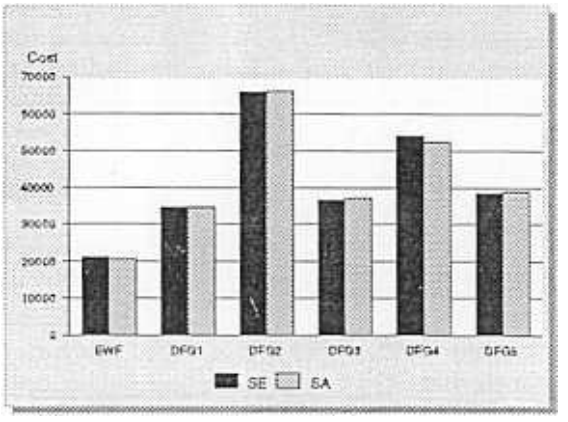


Figure 10: Barchart showing the results of SE and SA.

accepts solutions with higher costs initially and then progressively accepts solutions with less variations in costs. Then a point is reached when it accepts only good solutions, i.e., it accepts solutions whose cost is lesser than the current solution. In case of SE there is an initial drop in the cost in the first few iterations then the cost varies within a range. This is depicted in Figures 11 and 12.

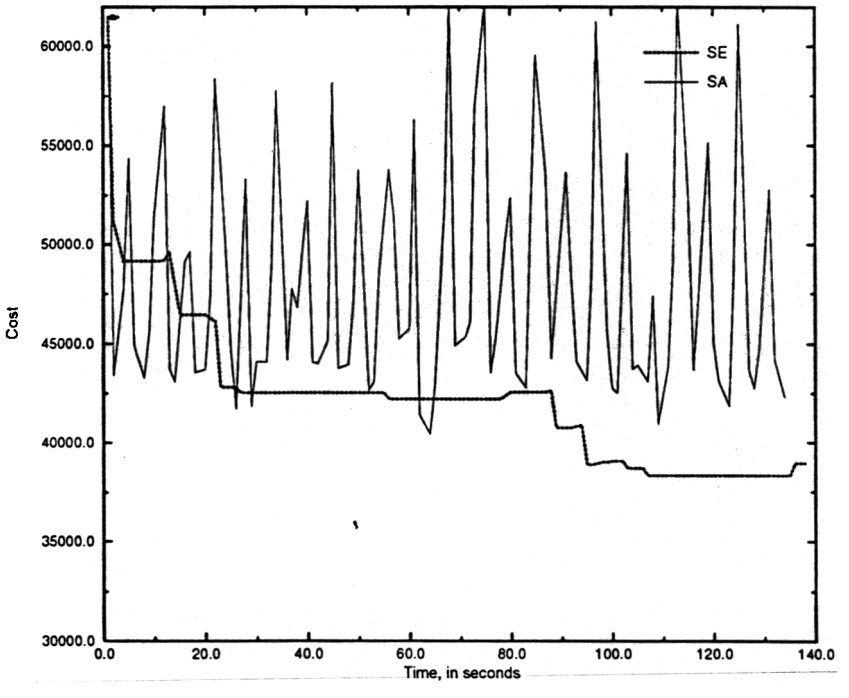


Figure 11: Plot showing the variation of cost for SA and SE during the initial stages.

5 DISCUSSION AND CONCLUSION

Different iterative/constructive techniques may be used to obtain a cost-optimal solution for the CS&P problem. A cost-optimal solution for this problem is obtained by minimizing the area occupied in terms of the number of gates without violating the constraints, and also pipelining the DFG.

Stochastic evolution is a promising optimization technique. In this paper we described a heuristic based on stochastic evolution (SE) for component selection and pipelining (CS&P). We posed CS&P as a

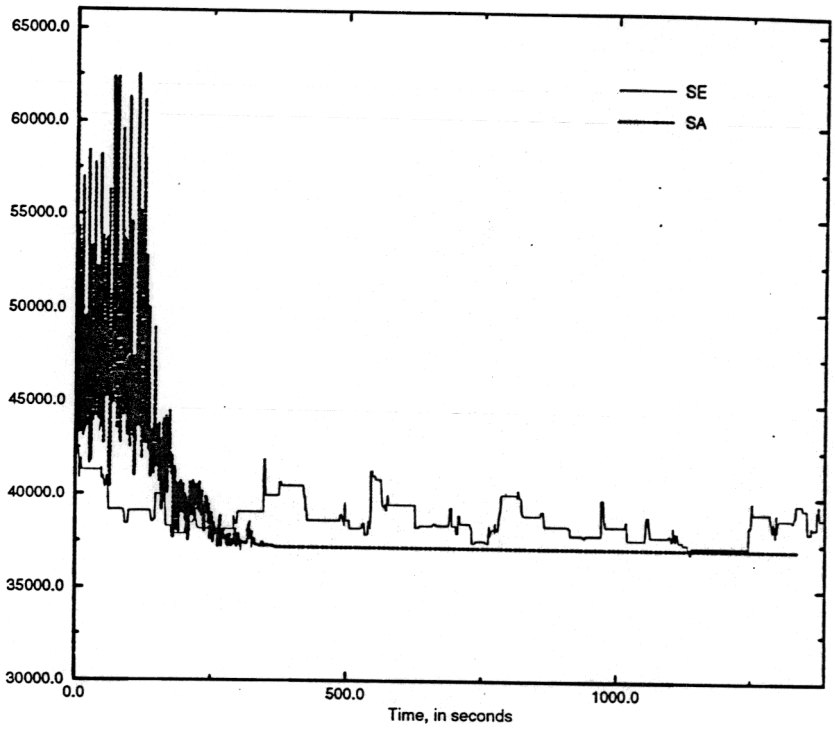


Figure 12: Plot of DFG Graph 4 showing the variation of cost versus time using SA and SE technique

combinatorial optimization problem, and compared the performance of SE and simulated annealing (S on a set of randomly generated DFGs. A realistic component library with multiple implementations operators was used for component selection. Pipelining was done based on the constraints of latency a pipestage delay that are specified. For pipelining, the DFG is scanned from the inputs towards the outp When the sum of all the delays in the path exceeds the specified PS_delay constraint then a registe placed. This is done for all the branches until the entire DFG satisfies the PS_delay constraint.

The *Perturb* function of SE is invoked to make a compound move from a current state S. *Pert* scans the set of movable elements M according to some apriori *ordering* and moves every $m \in M$ to a l location $l \in L$. For our CS&P problem, various ordering strategies have been experimented with. Orde of nodes based on their level in the DFG, random ordering, etc., were used to disturb the component the DFG in order to determine if it affected the solution. It was observed that the ordering (of node DFG) did not affect the quality of solution. This is because, irrespective of the ordering, the selectio the components from the component library is done randomly.

We believe that SE algorithm always run much faster than other stochastic iterative algorithms s as simulated annealing. The reason is that, for SE, the parameter p, which controls how steep of a the algorithm can climb, may be relatively large only when there is a strong evidence of the search get stuck at a local minimum. Otherwise p is such that only small up-hill moves are allowed. SE does have a hot regime like simulated annealing where the algorithm will be performing almost a random w thus wasting runtime resources. Although we ran both the algorithms for the same time, comparably g results were obtained using SE after only a few iterations.

In SE, after scanning all the movable elements $m \in M$, the **Make_State** routine makes sure t the final state satisfies the state constraints. If the state constraints are not satisfied then **Make_St** *reverses* the fewest number of latest moves until the state constraints are satisfied. This procedur required when perturbation moves that violate the state constraints are accepted. In our implementat we store the last valid state, thereby avoiding backtracking by undoing some last moves.

The iteration bound R (reward) acts as the expected number of iterations the SE algorithm ne until $C_{cur} < Cost(S_{best})$, i.e., an improvement in cost takes place. If such an improvement occur $q < R$ iterations, then the remaining $R - q$ iterations are added to the next R iterations to be perforn

Consequently the quality of the final state obtained increases with the running time of the SE algorithm. It was found that the algorithm ran for few hundred iterations and the results obtained were poor. On increasing the the value of R the results improved and the algorithm ran for larger number of iterations. It was found that best results in reasonable time were obtained when the value of R was between 20 and 25. In general if the value of R is increased then the chances of obtaining better results also improve. If R is set to be too large, then SE algorithm wastes time during the last set of iterations because it cannot find better states. However if R is chosen too small, the SE algorithm might not have enough time to improve the initial state. To obtain cost-effective results, methods to determine an optimal value of R must be investigated.

The update function in the SE algorithm determines the range of negative gains to be accepted. Different methods could be formulated to determine this range to improve the results. A scheme for selecting the parameters of the update function has been experimented with to obtain cost-effective results.

Simulated annealing is another promising optimization technique. For the purpose of comparison we applied it to the CS&P problem. Experimental results on different types of DFGs are reported and compared with SE. The representation of the problem and the perturb strategies used in SA were the same as those for SE.

Acknowledgments

Authors acknowledge King Fahd University of Petroleum and Minerals for all support.

References

[1] M. C. McFarland, A. C. Parker, and R. Camposano. "The high-level synthesis of digital systems". *Proceedings of the IEEE*, 78(2):301–318, 1990.

[2] S. Ali, Sadiq M. Sait, and M. S. T. Benten. "Application of tabu search in high-level synthesis of digital systems". *International Conference on Electronics, Circuits and Systems*, pages 423–428, 1994.

[3] S. Ali, Sadiq M. Sait, and M. S. T. Benten. "GSA: Scheduling and allocation using genetic algorithm". *European Design Automation Conference with Euro-VHDL*, pages 84–89, 1994.

[4] Sadiq M. Sait, S. Ali, and M. S. T. Benten. "Scheduling and allocation in high-level synthesis using stochastic techniques". *Microelectronics Journal* , 27(8):693–712, October 1996.

[5] M. Farook . "Component selection and pipelining using stochastic evolution. MS Thesis, KFUPM". June 1996.

[6] Sadiq M. Sait and Habib Youssef. *Iterative Computer Algorithms and their Applications in Engineering.* IEEE CS-Press (To Appear), 1997.

[7] L. Ramachandran and D. D. Gajski. "An algorithm for component selection in performance optimized scheduling". *In Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 92–95, 1991.

[8] S. Bakshi and D. D. Gajski. "A component selection algorithm for high-performance pipelines". *EuroDac '94 + Euro VHDL '94*, pages 400–405, 1994.

[9] S. Bakshi and D. D. Gajski. "Component Selection for High-Performance Pipelines". *IEEE Transactions on VLSI Systems*, 4(2):181–194, June 1996.

[10] K. S. Hwang, A. E. Casavant, C. T. Chang, and M. A. d'Abreu. "Scheduling and hardware sharing in pipelined data paths". *In Proceedings of the IEEE International Conference on Computer-Aided Design*, 1989.

[11] Hong-Shin and Sun-Young Hwang. "Design of a pipelined datapath synthesis system for digital signal processing". *IEEE Transactions on Very large Scale Integration (VLSI) Systems*, 2(3):292–303, 1994.

[12] N. Park and A. C. Parker. "Sehwa: A software package for synthesis of pipelines from behavioral specifications". *IEEE Transactions on Computer-Aided Design*, 7:356–370, 1988.

[13] R. Jain, A. Parker, and N. Park. "Module selection for pipelined synthesis". *In Proceedings of the 25th Design Automation Conference*, pages 542–547, 1988.

[14] R. Jain, A. Parker, and N. Park. "Module selection for pipelined synthesis with multi-cycle operations". *In Proceedings of the IEEE Conference on Computer-Aided Design*, pages 212–215, 1990.

[15] D. Audet, Y. Savaria, and N. Arel. "Pipelining communications in large VLSI/ULSI systems". *IEEE Transactions on Very large Scale Integration (VLSI) Systems*, 2(1):1–9, 1994.

[16] C. T. Hwang, Y. C. Hsu, and Y. L. Lin. "PLS: A scheduler for pipeline synthesis". *IEEE Transactions on Computer-Aided Design*, pages 24–27, 1989.

[17] R. M. King and P. Bannerjee. "ESP: Placement by simulated evolution". *IEEE Transactions on Computer-Aided Design*, 8(3), March 1989.

[18] F. Glover. "Tabu Search - Part I". *ORSA Journal of Computing*, 1:190–206, 1989.

[19] F. Glover. "Tabu Search - Part II". *ORSA Journal of Computing*, 2:4–32, 1990.

[20] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., 1989.

[21] Y. G. Saab and V. B. Rao. "Combinatorial optimization by stochastic evolution". *IEEE Transactions on Computer-Aided Design*, 10(4):525–535, April 1991.

[22] T. A. Ly and Jack T. Mowchenko. "Applying simulated evolution to high-level synthesis". *IEEE Transactions on Computer-Aided Design*, 12(3):389–409, March 1993.

[23] L. Song and A. Vannelli. "VLSI placement using tabu search". *Microelectronics Journal*, 17(5):437–445, 1992.

[24] A. Lim and Y.-M. Chee. "Graph Partitioning Using Tabu Search". In *1991 IEEE International Symposium on Circuits and Systems*, pages 1164–1167, 1991.

[25] J. P. Cohoon and W. D. Paris. "Genetic placement". *IEEE Transactions on Computer-Aided Design*, 6(6):956–964, November 1987.

[26] S. Nahar, S. Sahni, and E. Shragowitz. "Simulated annealing and combinatorial optimization". *In Proceedings of the 23rd Design Automation Conference*, pages 293–299, 1986.

[27] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by simulated annealing". *Science*, 220(4958):671–680, 1983.

[28] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing.* John-Wiley and Sons Ltd, 1989.

[29] Sadiq M. Sait and Habib Youssef. *VLSI Design Automation: Theory and Practice.* Mc-Graw Hill Book Co., Europe, 1995.

[30] N. Metropolis et al. "Equation of state calculations by fast computing machines". *Journal of Chem, Physics*, 21:1087–1092, 1953.