A Genetic Algorithm for the Synthesis of Structured Data Paths

C Mandal R M Zimmer
Department of Information Systems and Computing
Brunel University, England UB8 3PH, UK
crmandal@hotmail.com

Abstract

The technique presented here achieves simultaneous optimization of schedule time and data path component cost within a structured data path architecture, using a genetic algorithm. The data path architecture has been designed to overcome the problem of random interconnections between data path components by buses, which makes subsequent physical design more difficult. The data path is organized as architectural blocks (A-blocks), some or none global memory units, all interconnected by a few global buses. Each A-block has a local functional unit, local memory elements and local interconnections. The operations are scheduled such that the required data transfers are achieved using the few available global buses, and their interconnections to the A-blocks. The synthesis is guided by user specified architectural parameters, such as the number of A-blocks and global buses. The benchmark examples synthesized by this technique compare well with those synthesized by other commonly known synthesis techniques.

1. Introduction

A number of systems such as HAL [1], STAR [2], SAM [3], PARBUS [4], CASS [5], COBRA [5], and GABIND [6] support the high-level synthesis of digital systems. Most of the current synthesis systems generate data paths with random interconnections between data path elements, which may lead to use of greater routing area during physical design. The technique presented here supports the synthesis of structured data paths, specifically avoiding random global interconnects. The aim is to produce designs with a simple and predictable layout structure, conserving on-chip wiring resources. This synthesis algorithm is referred to as structured architectural synthesis technique or *SAST*.

SAST essentially takes as input, precedence constraints between operations represented as a partial order, and outputs a schedule of operations and transfers, and a data path to implement the schedule. The generated data path is organized as architectural blocks (A-block), and optional global memory blocks. Each A-block has a local functional unit (FU), local storage and internal interconnections. The A-blocks and the memory blocks, if any, are interconnected by a few global buses. The structure of the data path is characterized by a set of architectural parameters, such as, the number of A-blocks, the number of global memories, the number of global buses, the number of access links which connect an A-block to the global buses and the maximum number of writes per time step to storage locations in an A-block. The last parameter becomes relevant if a memory with a fixed number (e.g. one or two) of write ports is to be used to implement storage in an A-block. SAST delivers the following: *i*) a schedule of operations, *ii*) the A-block in which each operation is scheduled, *iii*) the schedule of all transfers over the global buses, satisfying the architectural constraints, and *iv*) the composition of the FU in each A-block, in terms of specific implementations of operators from a module database. The option to pick up modules from a data base permits the flexibility of using units which are pipelined or combinational and also units varying in speed and size. SAST can handle specifications with multiple basic blocks [7]. This requires certain variables carrying data across basic blocks to be located at predetermined locations. If the value destined for such a variable is defined or available only outside the A-block or memory where the variable is supposed to be located then, a transfer from a suitable A-block or memory to the appropriate destination for its assignment needs to be made.

The main feature of this work is that random long-distance interconnects between data path elements are avoided. This makes this technique attractive for synthesizing designs targeted towards programmable structures, where global wiring resources are limited. The experimental results indicate that this technique compares favorably, in terms of schedule time and component cost with other synthesis techniques that do not attempt to generate data paths free of random long distance interconnects. A brief review of related work is given in section 2. In section 3 the structured architecture synthesis problem is discussed. The GA based synthesis algorithm is presented in section 4. Some results for SAST are given in section 5 which is followed by the concluding remarks in section 6.

2. Related work

Initial work on data path synthesis led to the development of several scheduling and allocation techniques. Force directed scheduling in HAL [1] attempts to minimize the cost of hardware operators while trying to find a schedule within a specified number of time steps. The technique works by greedily minimizing a measure called *force*. STAR [2] is a program for data path allocation and binding, for scheduled designs. It tries to minimize the component cost and interconnection cost, as estimated through multiplexer usage. The algorithm used in SAM [3] is based on the scheduling ideas developed for force directed scheduling [1]. This algorithm uses the notion of force to measure the effect that a tentative scheduling of an operation would have on the resource requirements. Another method for integrated scheduling and binding has

been presented by Balakrishnan et al. [8]. The basic scheme is to schedule an operation and then bind it, along with its associated source and destination operands. The operands are bound to storage elements, while each operation is bound to a functional unit. This iteration is carried on till all the operations are scheduled. More recently a problem space genetic algorithm has been developed by Dhodhi et al. [9] which does concurrent scheduling and allocation.

The reported techniques address optimizations for the data path with respect to its performance or the cost of the components used, but not particularly the physical design cost. Estimation of the physical design cost, at this stage, is expensive. An approach to handle this problem is to impose restrictions on the structure of the data path so that it will have a predictable layout structure. This approach has been adopted in the current work, and also in varying degrees in GABIND [6], a related earlier work by one of the authors, and also in the works reported in STAR [2], PARBUS [4], CASS [5] and COBRA [5]. SAST, in particular, permits better control over the structure of the synthesized data path, by means of architectural parameters.

3. The Structured Architecture Synthesis Problem

It is necessary to find a schedule of operations such that each operation is scheduled in one of the A-blocks. The composition of an FU is determined by all the operations that it has to perform. It is also necessary to find a schedule of transfers of values between the A-blocks using the permitted buses as access links. It is assumed that sufficient storage is available in an A-block. There are a set of global buses interconnecting the A-blocks to permit the transfer of data between them. Each A-block is connected to the global buses by means of a specific number of *access links*. The number of access links limit the maximum transfer bandwidth between an A-block and the global buses.

A functional unit in an A-block is a set of one or more hardware operators such that in any time step only one operation can be initiated and in any time step only one result can be generated. Operations scheduled on an FU are not permitted to have input or output conflicts. Similarly, execution conflicts are not permitted in which operations try to execute simultaneously on the same hardware. It may be noted that multiple operations may execute on a pipelined unit without execution conflict.

If a variable is required by an operation scheduled in an A-block, it should either be available in that A-block or it should be transferred from another A-block or memory where it is already available. A variable becomes available in an A-block at a particular time step if it is either defined there or transferred therein, in that time step.

Certain variables, referred to here as *program variables* are meant to reside at specific storage locations in specific A-blocks, as explained later in section 4.1. These are initialized as being available for use in the appropriate A-block from the first time step. Variables in an A-block are stored in local storage elements. Any two variables which are live [7] at the same time need to be assigned to distinct locations. The present implementation also permits the use of multiple implementations of an operator, such as a slow adder or a fast adder. Use of pipelined operators, such a pipelined multipliers is also supported.

Thus several decisions need to be taken, which are as follows: *i*) The time step where an operation is to be scheduled. *ii*) The A-block in which the operation is to execute. *iii*) The particular module that will implement an operation in the FU in an A-block. *iv*) The time step when an input for an operation is to be transferred over a global bus, if it is not already available in the local A-block. *v*) If such a transfer is required, then the A-block from where the value should be obtained. It may be noted that a value may be present in more than one A-block. *vi*) Transfers between A-blocks that may be required for defining *program variables* (explained in section 4.1) – indicating the time step, source and destination.

4. GA Based Scheduling Algorithm

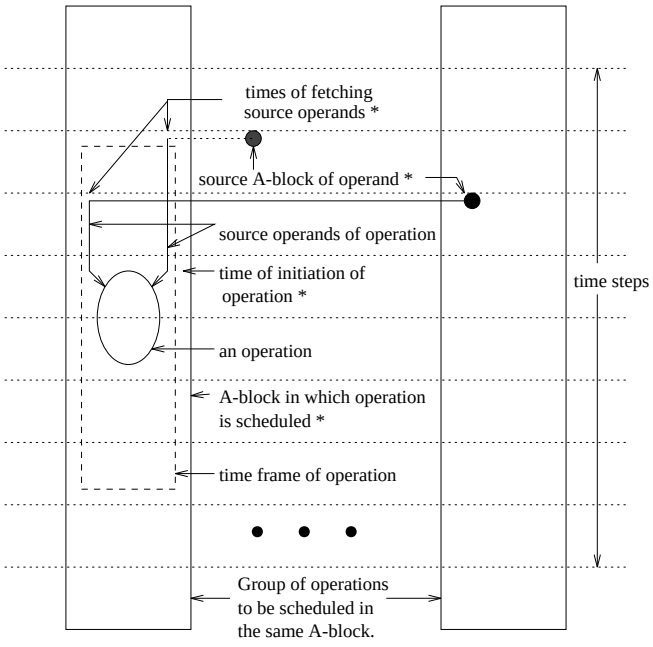
A genetic algorithm has been designed and implemented for solving the scheduling problem. A brief overview of the GA is given now. The detailed description follows in the sub-sections that follow. In view of the complex nature of the problem a structured solution representation has been used, as against a simple bit string. An initial population of solutions is generated at random. New solutions are obtained by inheriting values of decision variables from parent solutions, selected from the population. The decision values of the solution attributes are not independent and so the solution representation resulting from inheritance could correspond to an infeasible solution. To handle this situation a completion algorithm has been used to obtain a feasible solution from the solution representation resulting from a crossover. The completion algorithm is also used to obtain a feasible solution from a solution representation obtained by randomly assigning values to solution attributes, while generating the initial population of solutions. A scheduling heuristic has been used in the completion algorithm and this has been found to improve the performance of the genetic algorithm. A population control mechanism had to be employed to sustain diversity in the population, while at the same time retaining solutions with good overall and partial fitness. The genetic algorithm is run up to a fixed number of iterations and this serves as the stopping criterion. The last improvement in solution cost (i.e. when the best solution is obtained) usually occurs well before all the iterations are completed.

In the rest of this section the solution representation, the cost function, the parent selection scheme, the crossover scheme, the completion algorithm, the replacement scheme and the heuristic to enhance the performance of the genetic algorithm are explained.

4.1. Solution representation

Each solution comprises of several decisions which are required for the proper implementation of the design. Figure 1 indicates the decisions required for scheduling an operation. For each operation the time when it is to be scheduled and the A-block where it has to be scheduled are stored. For each input operand of an operation the A-block from where this value is to be obtained and the transfer time are given. If the operand is present in the same A-block then the time of transfer is redundant, as no transfer is necessary between A-blocks.

With loop based computations, which are very common, some of the variables defined in some basic block are required for subsequent iterations of a loop. Such variables are referred to as *program variables*. A program variable needs to reside at a fixed loca-



* marked entries correspond to design decisions related to the scheduling of the operation that need to be taken.

Figure 1: Decisions for scheduling an operation.

tion before the basic block in which it is used starts executing. For each program variable the time step of assignment and the A-block from where the value is to be obtained are indicated.

The period after which the result of an operation becomes available after it has been initiated on an FU depends on how long the particular module implementing the operation in the FU takes to deliver the result. For example, an addition could be implemented by a fast adder in a single time step or by a slow adder in a two time steps. Similarly, a multiplication could be implemented by a combinatorial multiplier or by a pipelined multiplier. The decisions involved in determining the composition of the FU need to be represented. It is necessary to indicate which operations an FU can implement and also the modules used for implementing these operations. The former need not be stored explicitly because it is fully implied by the union of all the types of operations that are scheduled on it. However, the module information needs to be stored explicitly.

Thus there are three types of information to be represented, which are as follows: *i*) Information directly related to the scheduling of operations, *ii*) information indicating the scheduling of variable transfers and *iii*) information regarding the composition of FUs. A structured representation is used for storing the above information. This is suitable for performing the algorithmic crossover (described in the section 4.5), which leads to a feasible solution representation.

It is often desirable to partially normalize a representation to reduce redundancies in the representation arising from permutation of attribute assignments. It may be noted that permutations of operation to A-block bindings alone do not correspond to equivalent solutions because the program variables are also bound to specific A-blocks. Such a permutation would, in general, lead to distinct transfer requirements.

4.2. Cost function

The scheduling algorithm tries to find a schedule of operations and transfers within a specified number of time steps. The solution cost is constructed to indicate the cost of the hardware and the extra time steps used in the schedule. It is of the form

$$C = (penalty)(extra\ \ time\ \ steps) + (cost\ \ of\ \ FUs).$$

The penalty is chosen to accord priority to finding a solution within the specified number of time steps. The penalty on the extra number of time steps is a constant chosen to be an order of magnitude higher that maximum possible cost of the FUs. In addition the cost of FUs is also separately accessible for performing population control, to be explained later in section 4.7.

4.3. Parent selection

The parents are selected on the basis of their costs using the roulette wheel technique [10]. This being a minimization problem, the selection probability of a parent is computed taking into account the maximum cost of solutions in the population as follows: $p_{s_i} = \frac{C_{max} + \delta - C_i}{N_{sols}(C_{max} + \delta) - \sum_i C_i}$, where p_{s_i} is selection the probability for solution i, $\delta \geq 0$, C_i is the cost of the solution, C_{max} is the maximum solution cost in the current population and N_{sols} is the number of solutions in the population. Solutions with higher cost are selected with lower probability. If $\delta = 0$ then the solutions with cost C_{max} will never be selected. Selection is done with replacement so that a member solution of the population may participate more than once in crossovers, in one generation.

```
procedure crossover()
1. chose two parents from the population
    of solutions.
2. mutate a each parent according to the mutation
    probability.
3. for each operation to schedule do
4.   inherit the various scheduling information
       of the operation  (such as, the A-block
       where it is to be scheduled, the time when
       the operation is to be initiated, for each
       input operand, the source A-block and the
       transfer time) from the two parents.
5. for each of the program variables do
6.   inherit the time of assignment and the
     source A-block from the the two parents.
7. for each of the A-blocks
8.   inherit library module to implement
     operations to be realized in the FU of
     this A-block from the two parents.
```

Figure 2: Generating initial attributes of offspring by crossover.

4.4. Crossover

New solutions are generated through crossover. An outline of the crossover mechanism used in SAST is given in figure 2. An example illustrating the formation of operation scheduling attributes through crossover and its subsequent completion is given in exam-

ple 1. First two parent solutions are selected. These go through a mutation and then the actual crossover takes place to generate a raw offspring. The crossover proceeds with inheritance of solution attributes values from each of the two parents. These attributes include schedule times and A-block bindings of operations, transfer times for operation inputs and the defined program variables. The FU configuration of the solution is also formed by inheritance from the parents. Inheritance of the attributes from either of the two parents proceeds in the (inverse) ratio of their solution costs. This may be considered to be a discrete multi-point crossover scheme. The solution representation available after inheritance, in general, not feasible. This is corrected by applying the completion algorithm.

4.5. Solution completion

It was noticed that optimization obtained only by applying the genetic operators of mutation and crossover, with small enough population sizes to be practical, do not perform very well. This is because of the vast numbers of solution representations generated that do not correspond to a feasible solution. A procedure for *solution completion* is applied to the raw solution resulting from attribute inheritance during crossover. Solution completion is also applied while generating new solutions because the randomly generated attributes used to construct the initial solutions may not correspond to feasible solutions either. The procedure is essentially a list scheduling algorithm with some programming intricacies to support the various features for structured architecture synthesis. A simplified version is shown in figure 3. The main data structures are a pair of lists, the ready list and the active list. A pair of these lists are used for scheduling operations and another pair for scheduling assignments. Operations or assignments in both types of lists are ready for scheduling in the current time step. However, it is only attempted to schedule operations or assignments from the corresponding active list. In each iteration the ready lists are processed to transfer some operations or transfers to the corresponding active lists. It is first attempted to schedule operations in the active list on the unit indicated in the solution representation for that operation. If this attempt to schedule the operation fails then it is attempted to schedule these operations on other available FUs. This is done to utilize FUs which may otherwise go unutilized in the current time step and is done only after it has been attempted to schedule all the operations on the active list on the designated FU. If any operation gets scheduled then the process of transferring operations to the active list from the ready list and then scheduling them is repeated. The intention of maintaining an active list of operations is to give priority to the operations in this list over the operations in the ready list for scheduling in the current time step. Assignments are normally handled after all the operations in the current time step have been scheduled. To avoid any excessive adverse effect of such a bias, assignments are sometimes attempted before trying the second round of scheduling operations, as indicated above, on other available FUs. When no more scheduling is possible, data structures are updated to close the current time step and scheduling proceeds from the next time step. Data structures have been chosen so that single step, multi-cycle and pipelined operators implementing operations are handled homogeneously as the scheduling is done.

A scheduling heuristic is also used intermittently with the intention of improving the quality of the solutions in the population. The heuristic may be used while transferring operations from the ready list to the active list (line 4, in figure 3). Normally operations are selected from the active list for scheduling at random (line 5, in figure 3). However, if the heuristic is being used then operations are chosen from the list on the basis of the scheduling heuristic. The application of the heuristic is explained in the section 4.6.

While trying to schedule an operation in an A-block at a specific time, first it is checked whether the FU can be used without input-conflict, output-conflict or execution-conflict. Next the availability of operands is checked. If an operand is not present in the current A-block then it needs to be transferred from another A-block, in the current or a preceding time step. For an operand or variable to be transferred at a particular time a free transfer path from the source to the destination needs to be identified. Thus a free bus and a free access link at the source and destination A-blocks have to be found. An operation can be scheduled in an A-block only if the FU can be used without conflict, and the operands are available or can be made available.

The inward transfer of a variable currently unavailable is made as follows. The variable can be transferred any time between the first time step and the current time step. It can be transferred from any A-block where the variable is available at the time the transfer is being attempted. The transfer is first attempted at the time and from the A-block indicated for that value in the solution. If the transfer cannot be satisfied this way then other times and A-blocks are considered in the following order: $t_s+1, t_s-1, t_s+2, \ldots$ and $(b_s+1) \bmod tot_b, (b_s+2) \bmod tot_b, \ldots$, respectively, where t_s is the desired time of transfer, b_s is the desired source A-block and tot_b is the total number of A-blocks. The order of scanning is block major (i.e. the block index changes slower).

Table 1: Crossover of scheduling attributes of a hypothetical operation.

Attribute	P1	P2	CS	SP
Initiation time	3	4	3	1
A-blk. where scheduled	1	2	1	1
Source A-blk. of left operand	1	2	2	2
Transfer time of left operand	3	4	4	2
Source A-blk. of right operand	2	1	2	1
Transfer time of right operand	3	3	3	1 or 2

Example 1 Consider an operation having inputs *v0* and *v1*. Table 1 shows hypothetical scheduling attributes values of the operation in the two parent solutions (column 'P1' and 'P2'), and those of the resulting offspring solution (column 'CS'). The parent from which the attribute is inherited is shown in column 'SP'. There are obvious inconsistencies in the inherited attribute values. These may be corrected by the completion algorithm as follows.

Assume that this operation occurs in the active list while scheduling for time step '3'. Let us assume that A-block '1' is available for this operation. The algorithm would find that it is not feasible to transfer the left operand into the A-block in time step 4, and would consider all feasible time steps for transferring in this

operand so as to monotonically recede from the time step indicated in the offspring. Thus if the feasible transfer times for the left attribute were time steps 2 and 3 then the algorithm would first consider time step 3 and then time step 2. Let us assume that it is feasible to transfer in the first operand in the third time step from A-block '3'. Now while considering the second operand suppose that it is not feasible to transfer it from A-block '2', as indicated in the offspring attribute. The algorithm would then try to source the operand from other A-blocks. Let us assume that it succeeds in sourcing the operand from A-block '1'. This operation is now scheduled in time step 3. ∎

```
procedure complete_solution()
1. prepare initial ready lists of operations and
      variable assignments.
2. while (operations and assignments remain
      to be scheduled)
3. { decide whether heuristic scheduling is to be used
        <sch_heur_flg> or priority will be given to
        transfers <priority_trn_flag>.
4.   transfer some operations to active list from
        ready list.
5.   try to schedule active operations on units
        indicated in the chromosome.
6.   if (priority_trn_flag)
7.     try to schedule active assignments.
8.   try of schedule remaining operations on
       other units.
9.   if (an operation has been scheduled)
10.    redo iteration.
11.  if (not priority_trn_flag)
12.    try to schedule active assignments.
13.  update ready list of operations.
14.  update status of FUs.
15.  bring in ready transfer candidates to active
       transfer list.
16.  move some transfers from ready list to active list.
17.  update data structures and flags.
18.  increment the time step.
19. }
```

Figure 3: Completion algorithm.

4.6. Application of Heuristic

The heuristic assists the completion algorithm. It is applied stochastically. The heuristic is based on a weight computed for each operation, which is defined as $w_i = \sum_{o_j \succ o_i}(d_j + W)$, where o_i and o_j are operations, o_j is a successor of o_i and W is a fixed positive value. While selecting an operation to schedule using the heuristic, it is chosen at random in proportion of its computed weight. A stochastic choice is made to avoid excessive bias to a particular decision.

The heuristic is applied at two places, while selecting operations from the active list and while transferring operations from the ready list to the active list. While completing a solution it is applied with a certain probability that is taken as a parameter. Even when it is being applied it is turned on and off at random as scheduling progress through the time steps to avoid excessive bias from the heuristic which might undo the evolutionary process.

4.7. Replacement

The replacement policy is designed to ensure that all solutions generated stay in the population for at least one iteration. This is done by introducing all the new solutions generated through crossover during one generation of the GA into the population, and replacing an equal number of existing solutions. The offsprings are stored in an adjoint pool, to be introduced into the main population once all the offsprings from the current generation are produced. The solutions to be replaced are mostly chosen at random. This could lead to removal of apparently good solutions, with low cost, from the population. To counter this a scheme has been used, at the same time, to retain the solutions with better costs, and also maintain a diversity of FU configurations in the population.

During implementation it had been observed that solutions with low cost FU configurations initially have schedules requiring more time steps than are desirable. These, therefore, have a higher cost and tend to get displaced. The population is then left mostly with solutions having expensive FU configurations. In order to retain low cost FU configurations a fixed number of *buckets* of a certain capacity are used to retain solutions having the same FU cost, although they may differ in their solution costs. Solutions which are in these buckets do not get replaced by a newly generated solution. These buckets are used to forcibly retain solutions with a range of low FU costs, even if their solution cost is high.

When a new solution is generated, first a check is made to see whether it can be placed in one of these buckets. If the cost of the solution matches FU cost of one of the buckets then it is introduced there if there is space in that bucket. Otherwise it replaces an inferior solution from that bucket, if any. In the absence of a matching bucket, the solution is placed in a free bucket, if one is available. Otherwise, solutions from the most expensive bucket are released. If the FU cost of these exceed the that of the new solution under consideration and this solution is put in.

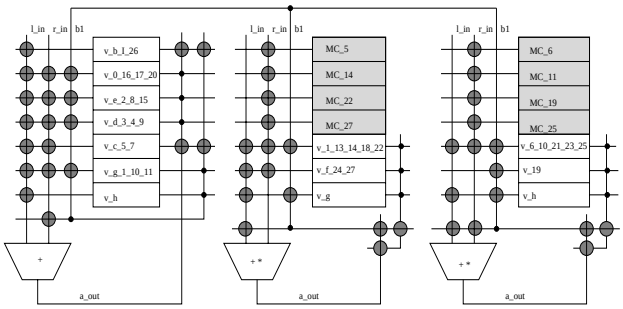


Figure 4: Structured architecture for Elliptic Wave Filter (EWF) in 18 time steps.

5. Experimentation

SAST has been implemented in 'C' in a SUN SPARC-5 under Solaris. It has been used successfully to synthesize designs satisfying given architectural requirements. In particular the differential equation solver [1], fifth order elliptic wave filter (EWF) [11] and discrete cosine transform (DCT) [12] examples were worked out and the results have been given in table 2. The experimentation has been done to investigate the effectiveness of the basic scheduling algorithm, the ability to use the appropriate implementation of an operator when many are possibly available and to find schedules under tight architectural constraints. All the designs require up to two concurrent writes per A-block. The run times for the tabulated examples vary between two to five minutes, depending

Table 2: Comparison of results with other synthesis techniques.

System	No. time steps	No. +	No. *	No. Bus, Blk., A. link	No. Reg.
Elliptic wave filter scheduled in 18 steps using multi-cycle multipliers					
SAST	18	3	2	1, 3, 1	13
COBRA	18	3	2	3, 3, -	12
CASS	18	3	2	5, 4, -	16
HAL	18	3	2	—	12
PSGA	18	3	2	—	10
Elliptic wave filter scheduled in 19 steps using multi-cycle multipliers					
SAST	19	2	2	2, 3, 2	12
COBRA	19	3	2	3, 3, -	13
CASS	19	2	2	4, 4, -	17
HAL	19	2	2	—	12
PSGA	19	2	2	—	9
Elliptic wave filter using pipelined multipliers					
SAST	18	2	1	2, 3, 2	12
COBRA	18	2	1	3, 3, -	13
HAL	18	3	1	—	12
PSGA	18	3	1	—	10
SAM	19	2	1	—	12
STAR	19	2	1	—	11
PARBUS	19	2	1	—	12

System	No. time steps	No. +	No. –	No. *	No. Bus, Blk., A. link	No. Reg.
Discrete Cosine Transform scheduled in 20 steps.						
SAST	20	2	3	1	2, 3, 2	18
COBRA	20	2	2	2	3, 3, -	12

on the difficulty of the problem. The run time is determined by the number of generations of the GA that need to be executed before a desirable solution is obtained. Each generation is completed quickly.

The differential equation example was synthesized with a choice of fast and slow adders. SAST synthesized a data path of three A-blocks and one global bus. The FUs configuration in the three A-blocks were: $\langle$slow +,<, –$\rangle$, $\langle$2-cycle *, +$\rangle$ and $\langle$2-cycle *$\rangle$. SAST uses a slow adder to make use of the available slack time and a fast adder otherwise. This is achieved by scheduling the operations such that such in one of the A-blocks *all* the scheduled additions have a slack time. Seven storage cells are used for the three program variables and other intermediate results.

The elliptic wave filter example has been scheduled in 18 and 19 time steps using two-cycle multipliers and single cycle adders. It has also been scheduled in 18 time steps using pipelined multipliers. The usage of adders, multipliers and storage for the various cases are indicated in table 2. The architectural characteristics of the solutions are indicated in the column labeled 'No. Bus, Blk., A. link,' to indicate the number of buses, A-blocks and access links per block. The structured architecture for EWF in 18 time steps is given in figure 4.

The architecture for DCT was chosen to have three A-blocks, two global buses and two access links per block. SAST was permitted to use both a pipelined and a multi-cycle multiplier and it finds a schedule using only one pipelined multiplier, two subtracters and three adders, which is a desirable solution.

6. Concluding Remarks

We present here SAST, a technique for synthesizing structured architectures with a simple and predictable layout structure. It relies on a GA for scheduling and allocation. The target architecture is characterized by the number of A-blocks, global memories, global buses, access links an A-block can have and the number of write ports used in the local storage for an A-block. SAST is able to handle multiple implementations of operations varying in speed, including multi-cycle and pipelined implementations. In all cases the FU cost of designs synthesized by SAST compare very favourably with those of other systems. An important feature of this work is that random long-distance interconnects between data path elements in the synthesized design are avoided. Designs produced by SAST compare favorably with other systems that do not attempt to generate structured data paths, and the run times for the tested examples are reasonable.

References

[1] P. G. Paulin and J. P. Knight, "Force-directed scheduling for ASICs," *IEEE Trans. on C. A. D.*, June 1989.

[2] F.-S. Tsai and Y.-C. Hsu, "STAR - An automatic data path allocator," *IEEE Trans. on C. A. D.*, pp. 1053–1064, Sep. 1992.

[3] R. J. Cloutier and D. E. Thomas, "Thc combination of scheduling, allocation and mapping in a single algorithm," in *Procs. of the 27th ACM/IEEE DAC*, pp. 71–76, June 1990.

[4] C. Ewering, "Automatic high-level synthesis of partitioned busses," in *Procs. of 1990 IEEE International Conference on Computer Aided Design*, pp. 304–307, 1990.

[5] A. A. Duncan and D. C. Hendry, "Area efficient dsp synthesis," in *Procs. of the 1995 European Design Automation Conference*, pp. 130–135, Sep. 1995.

[6] C. A. Mandal, P. P. Chakrabarti, and S. Ghose, "Allocation and binding for data path synthesis using a genetic approach," in *9th International Conference on VLSI Design*, pp. 122–125, 1996.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *COMPILERS Principles, Techniques and Tools*. Addison-Wesley Publishing Company, June 1987.

[8] M. Balakrishnan and P. Marwedel, "Integrated scheduling and binding: A synthesis approach for design space exploration," in *Procs. of the 26th ACM/IEEE DAC*, pp. 68–74, 1989.

[9] M. K. Dhodhi, F. H. Hielscher, R. H. Storer, and J. Bhaskar, "Datapath synthesis using a problem-space genetic algorithm," *IEEE Trans. on C. A. D.*, vol. 14, no. 8, pp. 934–944, 1995.

[10] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub. Co. Inc., 1989.

[11] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*. Prentice Hall, 1984.

[12] J. P. Neil and P. B. Denyer, "Simulated annealing based synthesis of fast discrete cosine transform blocks," in *Algorithmic and Knowledge Based CAD for VLSI* (G. Taylor and G. Russel, eds.), ch. 4, pp. 75–93, Peter Peregrinus, 1992.