GABIND: A GA Approach to Allocation and Binding for the High-Level Synthesis of Data Paths

Chittaranjan Mandal, Partha Pratim Chakrabarti, and Sujoy Ghose

***Abstract*—We present here a technique for allocation and binding for data path synthesis (DPS) using a Genetic Algorithm (GA) approach. This GA uses an unconventional crossover mechanism relying on a force directed data path binding completion algorithm. The data path is synthesized using some supplied design parameters. A bus-based interconnection scheme, use of multi-port memories, and provision for multicycling and pipelining are the main features of this system. The method presented here has been applied to standard benchmark examples and the results obtained are promising.**

***Index Terms*—Allocation, binding, data path synthesis.**

I. Introduction

Data path synthesis (DPS) involves scheduling of operations followed by allocation and binding. The latter step consists of several sub-tasks which include determining the mix of functional units (FUs) grouping variables and assigning these variable clusters to storage units, memory port assignment when multi-port memories are used in the design, mapping operations to the FUs, and mapping transfers to buses, when buses are used. The present work is concerned with the allocation and binding aspects of DPS. Earlier techniques attempted to solve these sub-tasks independently. Contemporary techniques attempt to handle the sub-tasks and other tasks such as scheduling in larger groups to ensure better optimality of the final design. However, all these sub-tasks are known to be NP-Complete and amalgamating many of these as a single problem is computationally prohibitive. It is therefore desirable to be able to solve the sub-tasks in overlapping combinations and move from one set of sub-tasks to another with a set of solutions rather than just a single solution. Multiple heuristics and randomization schemes may be used to find multiple solutions that are of the same cost or are nondominating. These concerns have motivated us to develop a Genetic Algorithm (GA), called GABIND, for synthesizing optimized data paths from a given scheduled data flow graph. GABIND builds on previously developed successful heuristics, such as force [1], by incorporating them into the GA.

Several researchers have worked on the DPS problems and several systems such as HAL [1], SAST [2], Facet [3], STAR [4], SAM [5], and Vital-NS [6] have been developed to solve the problem. While all techniques attempt to optimize schedule time and cost of storage and FUs, current techniques place an emphasis on interconnect optimization. HAL, SAM, Vital-NS, and STAR are some of the systems that perform interconnect optimization, in addition to the other DPS related optimizations. HAL was the first to make use of the force-directed algorithm to perform scheduling and data path optimizations. SAM combines scheduling, allocation, and mapping in a single algorithm. The algorithm uses the notion of force [1] to measure the effect that a tentative scheduling of an operation would have on the resource requirements. VITAL-NS performs scheduling, allocation and binding sub-tasks of DPS. FU registers and buses are partially allocated during the scheduling stage and finally optimized data paths are produced using heuristic techniques. STAR treats three important aspects of the binding task:

1) data transfer binding;
2) operation assignment; and
3) variable binding.

It solves the problem in three phases. First the data transfer bindings are performed. Subsequently, register binding and operation binding may be performed independently. In each case a restricted branch and bound algorithm is used to obtain the assignments. A problem space genetic algorithm (PSGA_Syn) has been proposed in [7] which does concurrent scheduling and allocation. Their method is presently oriented toward a point to point architectural model and does not take into account architectural constraints such as the number of FUs and buses. SAST performs scheduling, allocation and binding, under strong architectural constraints at the cost of foregoing multiplexer optimization.

GABIND performs the following tasks: formation of FUs, binding operations to FUs, binding transfers to buses, allocating storage, binding variables to storage units and allocating switches to interconnect FUs and memory units to the interconnecting buses. An important aim of developing GABIND was to be able to satisfy all transfers using a given number of buses and not relying on an unpredictable number of point-to-point interconnections. The output is an optimized data path which correctly implements the computation given to GABIND in the form of scheduled data flow graphs. The optimization is performed to jointly minimize the cost of the FUs, the storage units and the switches for interconnection used in the data path. Specifications for subsequent synthesis of the controller are also generated.

The rest of the paper is organized as follows. The architectural considerations used for the synthesis scheme are described in Section II. The GA to solve the problem is described in Section III. GABIND employs an algorithmic crossover, which is described in Section IV. The experimental results and conclusions are given in Sections V and VI, respectively.

II. Underlying Architectural Considerations

The optimization performed by GABIND is based on the architectural considerations described in this section. GABIND takes as input a scheduled data flow graph (SDFG) of the operations. Such schedules may be obtained as output of a design-space exploration scheme that gives a set of schedules [8] or any scheduling. It accepts the *number of FUs* and *buses* as user specified design parameters. The former indicates the total number of sites where operations may be performed while the latter indicates the total number of paths for carrying data transfers. The minimum number of FU sites should equal the maximum number of operations that are scheduled to execute concurrently in the given SDFG. An additional FU site generally leads to an overhead in the interconnection and the control logic. The capability of an FU is determined by the set of operations of the SDFG that it needs to execute. Arithmetic pipelining, often used for multiplication, in an FU is supported and has been used for some of the examples.

Storage is implemented using *multi-port memories and register files* in addition to individual registers. By placing several variables in a single unit the number of independent sources and sinks of data is reduced. The cost of some memory units will be known in advance. Cost of other memory units, having p-ports and n cells, is computed by GABIND using the formula: $c_m(n,\ p) = n(\alpha p + \beta) + \gamma p$, where α is the cost of the access logic per port per cell, β is the cost of each cell, and γ is the cost of the driver and other logic per port of the memory. In order to achieve low access time for a memory, the maximum number

Manuscript received January 28, 1999; revised July 19, 1999 and March 23, 2000. The work of P. P. Chakrabarti was supported in part by the Department of Science and Technology, Government of India.

The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, WB 721302, India (e-mail: chitta@cse.iitkgp.ernet.in; ppchak@cse.iitkgp.ernet.in; sujoy@cse.iitkgp.ernet.in).

Publisher Item Identifier S 1063-8210(00)10100-3.

of cells that a memory can have is restricted to some predefined "small" number, input as a design parameter.

All components are connected to one or more buses. The connection may be switched or un-switched, ie. physical. Interconnecting buses are often major contributors to the routing area for data paths. The number of buses serves as an effective handle to control their proliferation. A sufficient number of buses need to be present to satisfy concurrent transfers between the FUs and the memories that are eventually formed. Multiple data transfers arising from a *common source* are identified for possible use in interconnect optimization. Such transfers may be routed through a common bus making better use of existing connections.

III. The GA-Based Solution

GABIND employs a GA to perform optimizations and solve the problem. Our technique makes use of the GA as an efficient randomized search scheme for finding good solutions. It also differs from the usual GAs on several aspects, explained below. The motivating factor for taking this approach was to have a GA to solve the problem in reasonable time with a population of solutions of practical size. We found it fit to incorporate known good heuristics such as force to speed up the search process. Qualitative justifications for the design decisions for several aspects of the GA are given at appropriate places. The main features of the GA are as follows.

1) Design Representation: A structured solution representation has been used. For each operation and each transfer there are fields indicating the FU or the bus to which it is bound, respectively. The binding of a variable indicates the memory and the number of ports that it has. Individual binding decisions of operations, transfers and storage are highly interdependent.

2) Crossover: This is the most important step in the GA. Application of a traditional recombinant crossover often results in offsprings that do not represent a feasible solution, thus wasting computation time. A randomized heuristic algorithmic crossover has been used to ensure that a crossover always results in a feasible solution. The role of the heuristic is to avoid generating extremely poor solutions. The randomization ensures that the application of the heuristic does not seriously arrest the search that takes place in course of the GA-based optimization. This technique has been successfully applied in [2].

Three steps are involved in the crossover. First, it is determined which of all the attributes of both the parent solutions will be considered for inheritance. Then a tentative partial data path (TPDP) is formed by inheriting some of these attributes. Finally, the complete offspring solution is formed by completing the partial solution. Details of the crossover are given in Section IV.

3) Population Control: It is important to ensure that diversity of the population is sustained throughout the run of the GA. This has been achieved as follows. First, a minimum number of solutions having the kth, $k > 1$, best overall solution cost are retained. This policy is implemented for up to a fixed value of k. Second, the minimum number of distinct memory configurations in the population is maintained above a certain minimum number. This condition may not be satisfied at the beginning but once sufficient memory configurations have been produced, a certain number of solution groups having the same memory configuration are maintained. These memory configurations are tracked according to the memory configuration cost only. Third, a few solution groups with the same memory configuration as lower cost solutions are also maintained. The conscious decision to ensure memory diversity has been taken in view of the vast number of memory formations possible, as compared to FU formations.

4) Parent Selection: Crossover is performed between two solutions taken from the population of solutions. A solution is selected only once during one generation to ensure maximum participation of solutions in the crossover. The selection policy gives preference to choosing parent solutions which are more fit. To choose better fit parents, a list of solutions whose cost is less than some threshold is maintained. The threshold is determined according of the distribution of the solution costs in the population. Solutions can be picked up from this list at random.

Due to the strong interdependence between binding decisions, it is likely that two good solutions will have highly incompatible solution attributes. A crossover between such a pair of solutions is very likely to produce an offspring of high cost or low fitness value. This was experimentally observed during development. It has been suggested in [9] that special precautions need to be taken to handle such a case, as an excessive amount of type II deceptability could undermine the GA for the particular problem. Therefore, a provision has been made, to choose parents that are genetically less incompatible. Parents may be chosen such that they have identical memory configurations. During crossover the use of core attributes helps by reducing the incidence of "noisy attributes."

5) Other Aspects: First an initial population of feasible solutions is created. Each solution is produced by randomly generating feasible binding and allocation decisions. The cost of each solution is computed and stored. The population control data structures are then created. Offspring solutions which are produced are integrated into the main population of solutions only after the current generation is completed. They replace an equal number of solutions from the current population. This is a flexible compromise between replacing the entire population, and replacing just one solution. The GA is started to run for a certain minimum number of generations. Every time there is an improvement it is run for at least another fixed number of iterations in the hope of another improvement. Finally, the data path corresponding to any one of the best solutions obtained is output.

IV. Details of Crossover

Crossover is performed in five phases, described below. Actual allocations and bindings are made in the final phase.

1) Determining Prominent Solution Attributes: The cost of a solution is sensitive to the bindings. An unfavorable binding could give rise to additional data path elements. For this reason a 0/1 gradation is performed for the operation and transfer bindings in the parent solutions. The aim of transfer binding gradation is to consider only the more frequently accessed component connections to each bus before proceeding with the inheritance. Similarly, the aim of operation binding gradation is to consider for inheritance only the more frequently used functionality of each FU. In the implementation the better bindings are marked *core* while the inferior ones are marked *noncore*.

Variable to memory bindings are graded in a continuous scale. For a particular memory the points accessing it are determined. The importance of each such point has been defined as the number of variables of the memory that are accessed by that point. The importance of a variable has been defined as the $\sum$ (*importance of points that access the variable*). The *spread* of a memory has been defined as the total number of points accessing the memory. The relative importance of a variable has been defined as

$$\frac{\begin{array}{c}(min.\ spread\ among\ all\ mems.)*\\(\ imp.\ of\ var.)\end{array}}{\alpha_v(spread)*(max.\ imp.\ of\ var.\ in\ mem.)},\ \alpha_v \geq 1.$$

The above scheme is intended to distill out only some of the binding decisions which are likely to work together as good building blocks, while filtering out the noisy building blocks. This GA will still benefit from implicit parallelism, but less than the usual analytical value. We feel that for the usual analytic results to apply, the required population size would be too large to be useful.

2) Correspondence Between Data Path Elements: A matching between the data path components of the two parent solutions is used while performing inheritances. Affinity measures between components are computed based on similarity of bindings of operations, transfers and variables, with FUs, buses and memories, respectively. A greedy algorithm driven by edge weights is then used to match these.

3) Operation and Transfer Binding Inheritance Plan: A tentative plan of operation and transfer bindings to be inherited, time step by time step, is constructed. In each time step, either *core* operation or core transfer bindings are inherited first. The choice is made probabilistically. Next, associated core transfer or operation bindings, respectively, are attempted to be inherited. This is done by inspecting the buses or FUs one by one, respectively. If operation bindings are inherited first in a time step then core transfers connected with these operations are inherited provided the target bus is available. Similarly, the case of first inheriting transfer bindings is handled. The tentative binding inheritance plan implies a tentative allocation scheme for the data path to be constructed. The actual allocation and binding is explained later in this section.

4) Memory Formation: First, a blank memory configuration is formed by inheritance. A variable inherits the memory binding with a probability which is either the *register inheritance probability* parameter, or the *importance* of the variable in the memory, as defined earlier in this section. After inheritance is completed, in general, there will still be variables to be mapped to memories. These remaining variables are packed into the memories already constructed during inheritance. Those variables which could not be packed into these memories are packed into new memories. The choice of memories to be packed is governed by a simple heuristic. The heuristic is to choose the variable for which the number of unmapped variables that can still be packed into this memory without increasing the number of ports is maximum.

5) Final Generation of Actual Allocations and Bindings: The actual operation and transfer bindings are now made, time step by time step, in three phases: completing implied bindings, performing bindings by inheritance and completion of pending bindings. This also completely determines allocation of all data path components.

The first phase is trivial involving only bookkeeping steps. For the second phase, first the operations are processed and then the transfers are handled. For each operation binding in the inheritance plan if the corresponding FU is available, then the actual binding is set. If the FU does not already implement that type of operation then possibility of doing so is decreased. Similarly, transfer bindings are inherited but with some additional processing. While making a transfer binding if the existing links between and FUs, system ports and the memories with the buses suffice to support the transfer then the binding is directly made. If new links need to be introduced at both the source and the destination of the transfer then the inheritance is not made. If only one new link is needed then the inheritance is done probabilistically. Whenever a new link is introduced the data path is updated.

After the first two phases, in general, some operations and transfers will still remain unmapped. The operation and then the transfer bindings are made using a force directed completion algorithm, time step by time step. The decisions are made in a best first approach selecting the binding that leads to the least force. The force is computed in a way to encourage utilization of existing data path components, and discourage introduction of new components.

V. Experimental Results

GABIND has been tested on a Silicon Graphics Indigo (IRIS) workstation [R4000SC RISC CPU, 100 MHz (int.), 50 MHz (ext.)] with the standard benchmark examples of Facet [3], differential equation solver (Diffeq.) [1] and elliptic wave filter (EWF) [10]. The results have been tabulated along with those of some other well-known systems in Table I. The columns of the table indicate the technique, the number of multiplexer channels (#M), the number of links (#L), the number of storage cells (#C), the memory configuration, the FU configuration and the run time. A memory configuration of the form $\langle x,\ y\rangle$, indicates y memories each having x ports. GABIND is able to synthesize the designs using only single or double port memories. A double port memory of one cell is equivalent to a register. The results indicate that the cost of FUs and total number of multiplexer channels are consistently kept low. Sometimes the storage requirements are marginally higher than competing systems. It may be noted that because of the high level of design, DPS techniques usually cannot be compared exactly. It was observed

TABLE I
RESULTS OF RUNNING GABIND OF FACET, DIFFEQ, AND EWF

System name	#M	#L	#C	memory config.	FU config.	CPU time
Facet in 4 time steps, 3FUs						
Facet	11	—	8	—	—	—
Splicer	8	—	7	—	—	3s
HAL	6	13	5	—	—	—
Vital-NS	6	12	5	—	—	1.5s
GABIND	5	11	6	$<2,3>$ $<1,2>$	$\langle+\rangle$, $\langle+\vert\star\rangle$, $\langle-\&/\rangle$	28s
Diffeq. in 4 time steps						
Using single cycle multipliers and 5 FUs						
Splicer	11	—	6	—	—	—
HAL	10	25	5	—	—	40s
Vital-NS	12	22	5	—	—	3s
GABIND	8	18	5	$<2,5>$ $<1,1>$	$2\star$, +, -, <	38s
Using single cycle multipliers and 3 FUs						
GABIND	12	16	6	$<2,4>$ $<1,2>$	$\langle+\star\rangle$, $\langle\star\rangle$, $\langle+,-,<\rangle$	32s
Diffeq. in 8 time steps, 2FUs, 1 pipelined multiplier						
HAL	13	19	5	—	—	120s
Vital-NS	13	17	5	—	—	2.5s
GABIND	7	13	5	$<2,2>$ $<1,2>$	$\langle\star\rangle$, $\langle+,-,<\rangle$	24s
EWF in 17 time steps, pipelined multiplier						
HAL	31	—	12	—	3+, $2\star$	120s
SAM	31	50	12	—	3+, $2\star$	—
PSGA_Syn	—	—	10	—	3+, $2\star$	10s
Vital-NS	32	50	11	—	3+, $2\star$	110s
STAR	26	—	11	—	2+, $1\star$	—
GABIND	29	29	13	$<2,5>$ $<1,1>$	2+, $1\star$	210s
EWF in 18 time steps, pipelined multiplier						
HAL	34	—	12	—	3+, $1\star$	240s
SAM	30	40	12	—	3+, $1\star$	—
PSGA_Syn	—	—	10	—	3+, $1\star$	10.2s
Vital-NS	33	40	10	—	3+, $1\star$	140s
GABIND	31	35	11	$<2,6>$ $<1,1>$	3+, $1\star$	251s
EWF in 19 time steps, pipelined multiplier						
HAL	26	—	12	—	2+, $1\star$	360s
SAM	21	40	12	—	2+, $1\star$	—
PSGA_Syn	—	—	9	—	2+, $1\star$	10.2s
Vital-NS	29	40	11	—	2+, $1\star$	200s
STAR	28	—	11	—	2+, $1\star$	—
GABIND	27	33	14	$<2,4>$ $<1,2>$	2+, $1\star$	255s

that the solution quality is not critically sensitive on the GA parameters. In general, the time taken by the algorithm depends on the total number of time steps used in the schedule and is proportional to it. A larger population size is required for designs involving higher number of FUs or time steps. The same GA parameters were used for all designs, although the optimal result is obtained for the smaller examples with a smaller population size.

VI. Conclusion

Given a schedule of operations, GABIND is able to synthesize globally optimized data paths in terms of the cost of the FUs, multiplexing switches and storage elements. The synthesized data paths compare well with those produced by other contemporary systems. Operation pipelining and multicycling are supported. Storage implementation can accommodate individual registers, single or multi-port memories. GABIND relies on the GA to perform optimization. For this GA we have developed a specific crossover based on a force directed completion algorithm. We have shown experimentally that the GA framework can be applied successfully for structured representations suitable for DPS.

Acknowledgment

The authors thank the anonymous referees for their comments, which helped to greatly improve this paper.

References

[1] P. G. Paulin and J. P. Knight, "Algorithms for high-level synthesis," *IEEE Des. Test. Comput.*, pp. 18–31, Dec. 1989.
[2] C. Mandal and R. M. Zimmer, "High-level synthesis of structured data paths," in *IFIP TC10 WG 10.5 Int. Conf. Computer Hardware Description Languages and Their Applications*, Apr. 20–25, 1997, pp. 92–94.
[3] C. J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital-systems," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 379–395, July 1986.
[4] F.-S. Tsai and Y.-C. Hsu, "Star—An automatic data path allocator," *IEEE Trans. Computer-Aided Design*, pp. 1053–1064, Sep. 1992.
[5] R. J. Cloutier and D. E. Thomas, "The combination of scheduling, allocation and mapping in a single algorithm," in *Proc. 27th ACM/IEEE DAC*, June 1990, pp. 71–76.
[6] A. Kumar, A. Kumar, and M. Balakrishnan, "Heuristic search based approach to scheduling, allocation and binding in data path synthesis," in *Proc. VLSI Design '95*, 1995, pp. 75–80.
[7] M. K. Dhodhi, F. H. Hielscher, R. H. Storer, and J. Bhaskar, "Datapath synthesis using a problem-space GA," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 934–944, 1995.
[8] C. A. Mandal, P. P. Chakrabarti, and S. Ghose, "A design-space exploration scheme for data-path synthesis," *IEEE Trans. VLSI Syst.*, vol. 7, pp. 3331–338, June 1999.
[9] M. D. Vose, "Generalizing the notion of schema in GAs (research note)," *Artif. Intell.*, vol. 50, pp. 385–396, 1991.
[10] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

Partitioning Algorithm to Enhance Pseudoexhaustive Testing of Digital VLSI Circuits

Bassam Shaer, David Landis, and Sami A. Al-Arian

***Abstract*—This brief introduces a partitioning algorithm, which facilitates pseudoexhaustive testing, to detect and locate faults in digital VLSI circuits. The algorithm is based on an analysis of circuit's primary input cones and fanout (PIFAN) values. An invasive approach is employed, which creates logical and physical partitions by automatically inserting reconfigurable test cells and multiplexers. The test cells are used to control and observe multiple partitioning points, while the multiplexers expand the controllability and observability provided by the test cells. The feasibility and efficiency of our algorithm are evaluated by partitioning numerous ISCAS 1985 and 1989 benchmark circuits containing up to 5597 gates. Our results show that the PIFAN algorithm offers significant reductions in overhead and test time when compared to previous partitioning algorithms.**

***Index Terms*—Digital VLSI circuits, partitioning, primary input cones and fanout (PIFAN), pseudoexhaustive, testing.**

I. Introduction

VLSI circuits contain large numbers of active devices and have limited input/output (I/O) access; these characteristics make them difficult to test. The complexity of test generation and fault simulation grows with the number of transistors, while the limited I/O access greatly decreases the controllability and observability of the internal circuitry. Both of these factors limit the achievable test coverage using conventional automatic test-pattern generation (ATPG). The fundamental advantage of exhaustive testing is that it can provide 100% coverage for irredundant combinational digital circuits against any faults that do not introduce memory. However, the length of an exhaustive test for a digital circuit with N inputs is 2^N input test patterns. This limits the applicability of exhaustive testing because, as the primary input value N increases, test time becomes long or unfeasible.

Since true exhaustive testing is impractical for large VLSI circuits, partitioning to allow exhaustive test of subcircuits offers an attractive alternative. Goel [1] observed that with each doubling of the number of gates in a circuit, the cost of testing increases as the square of the previous cost. McCluskey [2] proposed the method of pseudoexhaustive testing. He partitions a circuit into p subcircuits such that each partition has an upper bound s on the number of inputs, while the total number of edges between the partitions is minimized. For efficient pseudoexhaustive testing, a circuit must be partitioned into a number of subcircuits with the following constraints: 1) the gates in each subcircuit must not be functionally dependent on too many inputs of the subcircuit and 2) the amount of control logic used to separate the subcircuits must be as small as possible.

A wide variety of automated techniques for circuit partitioning and testability enhancement are available [3]–[7], and their effects on the circuit under test differ greatly. However, all available methods of general partitioning and test vector generation for digital circuits require that large amounts of design for testability hardware be inserted [8].

Manuscript received April 26, 1999; revised August 17, 1999.

B. Shaer is with the Department of Electrical and Computer Engineering, University of Minnesota, Duluth, MN 55812 USA (e-mail: bshaer@d.umn.edu).
D. Landis is with Center for Electronic Design, Communications, and Computing, Pennsylvania State University, University Park, PA 16802 USA (e-mail: dll2@psu.edu).
S. A. Al-Arian is with the Computer Science and Engineering Department, University of South Florida, Tampa, FL 33620 USA (e-mail: alarian@usf.edu).

Publisher Item Identifier S 1063-8210(00)09522-6.