A Genetic Algorithm based Approach for Multi-Objective Data-Flow Graph Optimization

Birger Landwehr

Dept. of Computer Science XII,
University of Dortmund, Germany
landwehr@ls12.cs.uni-dortmund.de

Abstract: This paper presents a genetic algorithm based approach for algebraic optimization of behavioral system specifications. We introduce a chromosomal representation of data-flow graphs (DFG) which ensures that the correctness of algebraic transformations realized by the underlying genetic operators *selection*, *recombination*, and *mutation* is always preserved. We present substantial fitness functions for both the minimization of overall resource costs and critical path length. We also demonstrate that, due to their flexibility, genetic algorithms can be simply adapted to different objective functions which is examplarily shown for power optimization. In order to avoid inferior results caused by the counteracting demands on resources of different basic blocks, all DFGs of the input description are optimized concurrently.

Experimental results for several standard benchmarks prove the efficiency of our approach.

1 Introduction

One of the first steps in the design-flow of digital systems is concerned with formulating the behavioral specification in an appropriate hardware description language (e.g. VHDL). This behavioral description serves as a basis for the subsequent design steps starting with high-level synthesis which is generally understood as a mapping of operations of the data-flow graph to control steps and to suitable components of a given library. Even though much effort has been spent during the last years in this research area (e.g. see [10] for an overview of recent publications) the actual question of how to suitably formulate the behavioral description such that synthesis can produce efficient results has been often underrated. However, it seems obvious that even optimal synthesis algorithms can only produce results as good as the given behavioral specification allows (the same applies to conventional software compilers for which the presented approach is applicable, too).

Apart from the classical domain of high-level language compilers, the main area of research and application of algebraic optimization methods has been extended to high-level synthesis during the recent years. Algebraic optimization techniques have been employed for improving resource utilization [11], tree height minimization [4, 5], maximization of data throughput [7][6] and minimization of power consumption [2] (see [11] for a comprehensive overview). Although all published approaches are very powerful in their own domain, most of them have certain restrictions concerning the set of supported transformation rules (commutativity, associativity, distributivity, and some special domain-specific rules are usually exploited) and the objective function to be optimized. I.e. these approaches are rather specialized as compared to general-purpose methods and hence difficult to adapt or to extend. Another drawback is their restriction to single basic blocks. Although all basic blocks of input description can be optimized sequentially, this procedure can potentially lead to inferior results due to conflicting demands on resources.

The remainder of this paper is organized as follows: In the next section, we introduce the genetic algorithm based on [8] including the chromosomal representation, the genetic operators and fitness functions. Section 3 describes the extension of the basic algorithm for the concurrent DFG optimization and section 4 concludes the paper with experimental results.

2 Data-flow graph optimization by a genetic algorithm

Genetic algorithms (GAs) have been proven as robust and powerful methods for searching vast solution spaces. Due to their capability of overcoming local optima, solutions found by genetic algorithms are usually close to the global optimum. Its general principle is to optimize a population's fitness in the course of generations driven by a randomized process of recombination, mutation, and selection[1].

The very first step in adapting genetic algorithms to a certain task is to find a suitable chromosomal representation for the given problem. A general problem in transforming algebraic expressions or DFGs is to represent their tree or graph structure by a linear data structure: the *chromosome*. This is particularly relevant since all transformations performed by the GA should be correctness preserving. In the next section we describe how to cope this problem with a novel chromosomal representation.

2.1 Chromosomal representation

Each chromosome of the population represents one semantically equivalent formulation of the original DFG. Each gene (or to be precise: its position on the chromosome) represents the functionality of one subexpression. We denote genes by Greek letters α, β, γ, etc.

[1] Due to the lack of space we assume that the reader is familiar with the principle of genetic algorithms. For a comprehensive introduction see e.g. [3].

Example: We use expression $((a * 2) + (b * 2)) + 1$ as a running example. Even though this example is quite simple, it is useful to clearly demonstrate the chromosomal representation and the particular genetic operators. Figure 2 depicts its chromosomal representation $\alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta$ with genes $\alpha : \beta + 1$, $\beta : \gamma + \delta$, $\gamma : a * 2$, and $\delta : b * 2$.

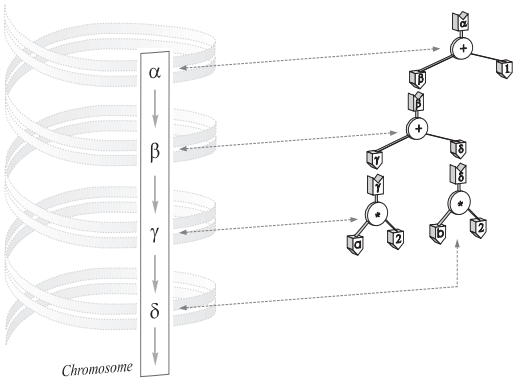

Figure 1: Chromosomal representation of expression $((a * 2) + (b * 2)) + 1$

In order to express that a certain subexpression can be represented and substituted in the end by an equivalent one (e.g. $a * 2 = a + a$) we introduce *alleles*: an allele is one (of several) phenotypes of a certain gene. We distinguish alleles of the same gene by roman numbers I, II, etc.[2] The set of alleles for a specific gene represents the set of functionally equivalent expressions which are mutually replaceable without changing the semantics of the entire DFG. Nevertheless, the resulting DFG can be distinguished by potentially different hardware realizations.

Figure 2 shows new alternative alleles for gene δ. Obviously, each allele of gene δ can be replaced by any other allele without changing the semantics of this expression. In the same way, algebraic transformations can be applied to other genes of the chromosome.

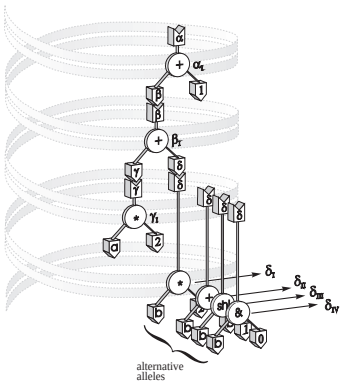

Figure 2: Alternative alleles

Figure 3 presents the gene pool for our running example after applying the associativity and distributivity laws and the simplification of multiplications[3]. Due to the fact that each chromosome implicitly represents the structure of a data-flow graph, the creation of any DFG can be performed very efficiently. For instance, chromosome $\alpha_{\mathrm{II}} \rightarrow \beta_{\mathrm{III}} \rightarrow \gamma_{\mathrm{III}} \rightarrow \delta_{\mathrm{IV}} \rightarrow \epsilon_{\mathrm{II}} \rightarrow \zeta_{\mathrm{I}}$ represents expression a shl $1+$ inc $(b \,\&\, 0)$[4].

	I	II	III	IV	V
α	$\beta + 1$	$\gamma + \epsilon$	inc(β)	cadd$(\gamma, \delta, 1)$	
β	$\gamma + \delta$	$\zeta * 2$	shl$(\zeta, 1)$	$\zeta + \zeta$	$\zeta \,\&\, 0$
γ	$a * 2$	$a + a$	shl $(a, 1)$	$a \,\&\, 0$	
δ	$b * 2$	$b + b$	shl $(b, 1)$	$b \,\&\, 0$	
ϵ	$\delta + 1$	inc(δ)			
ζ	$a + b$				

Table 1: Extended gene pool for the running example

At the beginning, this gene pool serves as a basis for the creation of the initial population. Later in the course of the genetic algorithm it will be extended during so-called "epochs" (see section 2.2) by applying transformation rules to a subsequent population in order to enable new variants of DFGs.

2.1.1 Genetic operators

The chromosomal representation introduced in the last section guarantees that alleles at the same gene position can be interchanged without changing the semantics of the entire DFG. This property is necessary for defining the main genetic operators, namely crossover and mutation.

Crossover: During crossover the genetic information of (usually) two parents is recombined and transmitted to the offspring. The desired effect of crossover is to combine all positive properties of the parents onto a new individual in order to increase its fitness. In the meaning of transforming algebraic expressions, crossover recombines the subexpressions of the parental data-flow graphs. In our approach we use the uniform crossover scheme which has lead to the best optimization results. Uniform crossover can be sketched as follows:

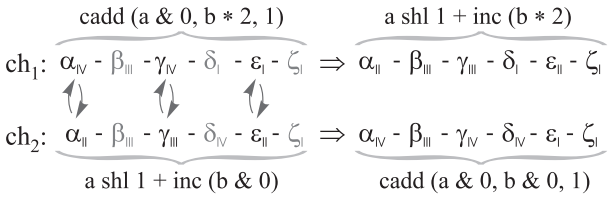

Figure 3: Crossover

At each gene position α, β, γ etc. two alleles are interchanged with a probability $p_c (0 \leq p_c \leq 0.5)$ among the parental chromosomes ch_1 and ch_2.

Mutation: Generally, the crossover operator is only able to recombine the parental properties whereas mutation is capable of providing the offspring with new genetic information (or to be precise: alleles). Its principle has been implicitly shown in figure 3: a data-flow graph can be transformed without changing the semantics by substituting an existing allele by another one at same gene position. For example, the substitution of gene δ_{I} by δ_{IV} leads to the transformation $(a * 2) + (b * 2) + 1 \Rightarrow (a * 2) + (b \,\&\, 0) + 1$ (cf. table 1).

[2] Operations of the original DFG are thus represented by the alleles $\alpha_{\mathrm{I}}, \beta_{\mathrm{I}}, \gamma_{\mathrm{I}}$ etc.

[3] *inc* means the increment operation, *cadd* stands for the ternary carry-add operation.

[4] Consider that chromosomes may also contain *redundant genes* which have no direct influence to the created DFG but can be reactivated instantly by small mutations or crossover.

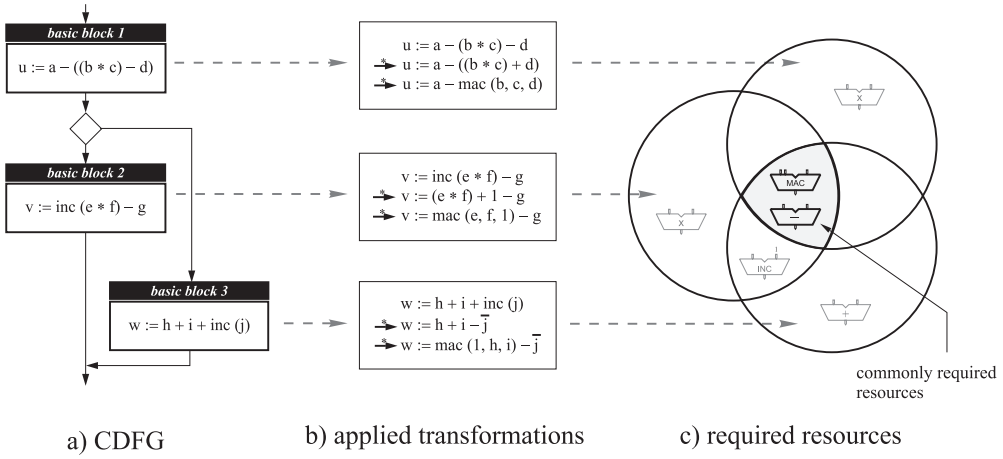

Figure 4: Concurrent basic block optimization

Selection: Selection favors individuals with a higher fitness compared to other individuals of the population to survive ("survival of the fittest") and thus become the cofounders the next generation. We presume the probability of an individual to be selected is proportional to its fitness. This enables to a certain extent even individuals with a lower fitness to survive and hence to transmit their gene information to the offspring. The method used in our approach is so-called *tournament selection* [1] which has been proven to be very efficient.

2.1.2 Fitness functions

The fitness of an individual is obviously crucial for the transmission of its gene information to the next generation. We now present suitable fitness functions for minimization of the critical path length and resource costs. We also show how the fitness function for resource optimization can be adapted for the minimization of power consumption. All presented fitness functions can be applied separately or (for multi-objective optimization) weighted to each individual of the population.

Minimization of critical path length
The fitness of a chromosome *ch* is equivalent to the critical path length *CP* of its data-flow graph *DFG* (*ch*). The critical path can be efficiently computed by a simple ASAP (or ALAP) scheduling.

$$f_{time}\,(ch) = CP\,(DFG\,(ch)) \qquad (1)$$

Minimization of resource costs
Since an exact computation of resources required for implementing a design is known to be NP-complete, we have to estimate the expected costs by heuristics. In order to keep computation times of the GA small, we employ an extended ASAP scheduling approach controlled by a priority list (operations on the critical path are scheduled first, then operations depend of their mobility).

$$f_{area}\,(ch) \quad = \quad \sum_{k \in K} b_k * c_k \qquad (2)$$

The fitness of an individual is defined as sum over all costs c_k for a number of b_k components of a type k. In our case, all necessary component information including functionality and costs of each type are defined in a given library and thus available to the underlying scheduling approach.

Minimization of power consumption
Equation 2 can be directly taken over for estimating the power consumption. Since the power consumption of an implemented DFG is depend of the number and the types of the operations, it is sufficient to perform the *direct compilation* method, i.e. each operation of the DFG is mapped to a separate component c_k which is associated with certain power consumption b_k.

2.2 Genetic algorithm

Figure 3 presents the genetic algorithm for algebraic optimization that takes pattern from the standard GA in [3].

```
1    initialize individuals of the population p
2    FOR EACH epoch e DO
2.1    apply transformation rules to the current population p
2.2    FOR EACH generation g DO
2.2.2    compute fitness of all individuals of p
2.2.2    select individuals according to their fitness
2.2.3    create offspring by crossover
2.2.4    mutate offspring
2.2.5    replace individuals of the current population by the offspring
2.2.6    exit loop, if criterion T_g is fulfilled
2.2'   END
2.3    terminate, if criterion T_e is fulfilled
2'   END
```

Figure 3 : Outline of the genetic algorithm

The algorithm consists of an outer and an inner loop: The inner loop repeats the tasks of fitness computation, selection, crossover, mutation and replacement of individuals of the current population by the offspring as long as the loop exit criterion is not fulfilled which is in out case controlled by the state of a generation counter. The outer loop is required in order to extend the gene pool by new genes along with their alleles. For our initial gene pool (s. table 1), we created the two new genes ϵ and ζ along with their alleles $\epsilon_{\mathrm{I}}, \epsilon_{\mathrm{II}}$ and ζ_{I} by applying the associativity law to our initial expression. The entire algorithm terminates if criterion T_e is fulfilled either controlled by reaching a certain number of epochs or a maximum size of the gene pool (T_e is also fulfilled if the gene pool cannot be extended any longer).

3 Extension to concurrent optimization

Up till now, we restricted our approach for the sake of a better understanding to single data-flow graphs. In this section we describe how the chromosomal representation and the genetic operators must be extended in order to optimize all basis blocks concurrently and thus to lay the foundations for solutions optimized for the entire input description. However, the example depicted in (figure 4) makes clear that algebraic transformations can potentially result in increasing resource requirements for a single basic block, but decrease the implementation costs for the entire design[5] at the same time: if we observe the required resources for the single basis blocks during optimization, the implementation costs increase temporarily. The final result of one multiplier-adder and one subtractor can obviously be reached only if the entire demand of resources is estimated concurrently for all basic blocks. The extension for a current DFG optimization only affects marginal modifications of the representation and the genetic operators:

Representation: We have to extend the notion *individual* that have been associated so far with one *chromosome* to a set of chromosomes. Since each of it represents one basic block of the input description, the entire individual contains all DFGs (without control-flow information).

Genetic operators: The particular genetic operators must be handled differently: mutation can be applied to each basic block (chromosome) of the individual separately without any risks of side effects. However, crossover must be applied to so-called *homologous* chromosomes which represent the same basic blocks of the individuals to be recombined. The selection operator must be also extended from single chromosomes to individuals.

4 Experimental results

We applied the presented algorithm to several standard benchmarks for optimizing the critical path length as well as resource costs. The running times of the optimization routine were for all examples approximately one minute (SparcStation 20)[6]. Table 2 shows the component requirement after synthesis for the original and the optimized data-flow graph. The underlying ILP-based synthesis system [9] guarantees that all computed results for both the original and transformed version are optimal with respect to the total costs of components. All synthesis results have been produced for the 1.0μ component library from VLSI [12] with the following associated component costs: add [vdp1add01] : 2405 $k\lambda^2$, sub [vdp1sub001] : 2433 $k\lambda^2$, mult [vdp3mlt004] : 14717 $k\lambda^2$.

For the examinated examples, we achieved a gain of up to 30 % concerning the critical path length (FIR_{cp}) and an area gain of up to 53 % ($Edge_{area}$)) compared to the original descriptions.

[5] We presume that each basic block can be executed exclusively thus resource sharing among basic blocks becomes possible.

[6] On the basis of empirical tests we determined the following genetic parameters: population size: 80 individuals; number of individuals in the population to be replaced by the offspring: 60; number of generations: 40, mutation rate: 0.1, crossover rate: 0.5.

benchmark[a]	#cs	resources	gain
EWF_{orig}	14	3 +, 2 *	
EWF_{cp}	12	3 +, 2 *	14 %
EWF_{area}	14	3 +, 1 *	40 %
FIR_{orig}	10	2 +, 1 *	
FIR_{cp}	7	3 +, 3 *	30 %
FIR_{area}	10	2 +, 1 *	0 %
BF_{orig}	14	3 +, 2 *	
BF_{cp}	12	3 +, 2 *	14 %
BF_{area}	14	3 +, 1 *	40 %
$Edge_{orig}$	10	2 +, 2 -, 4 *	
$Edge_{cp}$	9	2 +, 2 -, 3 *	10 %
$Edge_{area}$	10	1 +, 2 -, 2 *	53 %

Table 2: Optimization results

[a] EWF: elliptical wave filter, FIR: finite impulse response filter, BF: bandpass filter, Edge: edge detection

References

[1] T. Blickle and L. Thiele. A Comparision of Selection Schemes used in Genetic Algorithms. Forschungsbericht TIK Nr. 11, Version II, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology, Zürich, 1995.

[2] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen. Optimizing Power Using Transformations. *IEEE Transactions on CAD, Vol. 14, No. 1*, pages 12–31, 1995.

[3] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[4] R. Hartley and A. E. Casavant. Tree-Height Minimization in Pipelined Architectures. *Proceedings of the International Conference on Computer-Aided Design*, pages 112–115, 1989.

[5] R. Hartley and A. E. Casavant. Optimizing Pipelined Networks of Associative and Commutative Operators. *IEEE Transactions on CAD, Vol. 13, No. 11*, pages 1418–1425, 1994.

[6] S.-H. Huang and J. M. Rabaey. Maximizing the Throughput of High Performance Applications Using Behavioral Transformations. *Proceedings of the EDAC*, pages 25–30, 1994.

[7] Z. Iqbal, M. Potkonjak, S. Dey, and A. Parker. Critical Path Optimization Using Retiming and Algebraic Speed-Up. *Proceedings of the 30th Design Automation Conference*, pages 573–577, 1993.

[8] B. Landwehr and P. Marwedel. A New Optimization Technique for Improving Resource Exploitation and Critical Path Minimization. *10th International Symposium on System Synthesis*, pages 65–72, 1997.

[9] B. Landwehr, P. Marwedel, and R. Dömer. OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming. *Proceedings of the EURO-DAC*, pages 90–95, 1994.

[10] Y.-L. Lin. Recent Developments in High-Level Synthesis. *ACM Transactions on Design Automation of Electronic Systems, Vol. 2, No. 1*, pages 2–21, 1997.

[11] M. Potkonjak and J. Rabaey. Optimizing Resource Utilization by Transformations. *IEEE Transactions on CAD, Vol. 13, No. 3*, pages 277–292, 1994.

[12] VLSI Technology Inc. Library Manuals, 1993.