A New Optimization Technique for Improving Resource Exploitation and Critical Path Minimization

Birger Landwehr, Peter Marwedel

Dept. of Computer Science XII, University of Dortmund
D-44221 Dortmund, Germany

Abstract

This paper presents a novel approach to algebraic optimization of data-flow graphs in the domain of computationally intensive applications. The presented approach is based upon the paradigm of simulated evolution which has been proven to be a powerful method for solving large non-linear optimization problems. We introduce a genetic algorithm with a new chromosomal representation of data-flow graphs that serves as a basis for preserving the correctness of algebraic transformations and allows an efficient implementation of the genetic operators. Furthermore, we introduce a new class of hardware-related transformation rules which for the first time allow to take existing component libraries into account. The efficiency of our method is demonstrated by encouraging experimental results for several standard benchmarks.

1 Introduction

The very first step in the design-flow of digital systems is concerned with formulating the behavioral specification in a hardware description language such as VHDL [9]. This behavioral description forms the basis for all subsequent design steps starting with behavioral (or high-level) synthesis at the algorithmic level. High-level synthesis deals with the transformation of the behavioral description into a netlist of RT-level components and is generally understood as a mapping of operations of the data-flow graph (DFG) to control steps and to suitable components of an existing library [16]. Since high-level synthesis systems directly operate on the internal representation of the behavioral description it is quite obvious that the chosen formulation style has a lasting effect to later design steps and therefore to the final result.

Although the use of high-level synthesis systems has gained acceptance during the recent years, the actual question of how to suitably formulate a behavioral description has often been underrated. This question particularly arises in the domain of digital signal applications which are characterized by complex arithmetical computations resulting in complex data-flow graphs.

Figure 1 demonstrates the effect of different transformations for expression $((a*c)+(b*c))+d$ with respect to time and resource equirements.

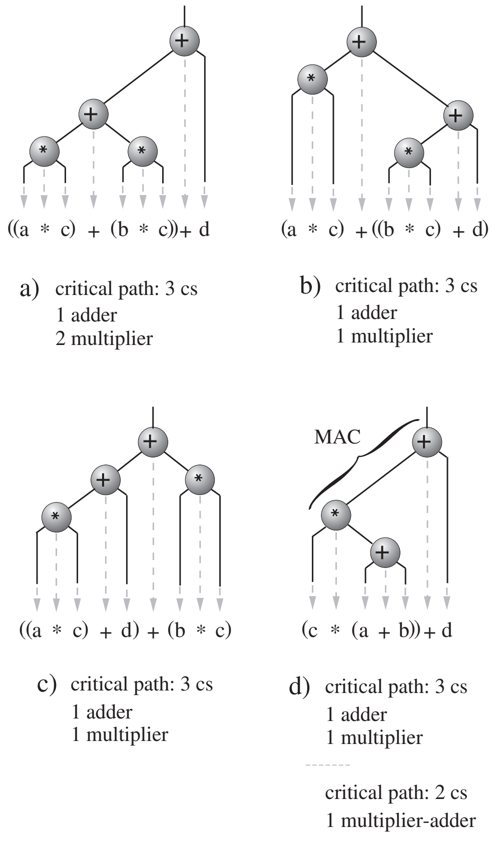

Figure 1: Time and resource requirements for equivalent expressions

Figure 1a presents the original expression requiring one adder and two multipliers with a critical path of 3 control steps[†]. After applying the associativity law (figure 1b), the order of operations has been changed such that component sharing could be improved. Therefore, only one multiplier is required after the transformation. The additional exploitation of commutativity shown in figure 1c neither leads to a further improvement of resource sharing nor to a shorter critical path, but may increase the applicability of other transformation rules. Figure 1d shows the expression after exploiting distributivity. Although implementation costs and critical path have not further changed, the application of a special component-driven transformation $(x * y) + z \Leftrightarrow \text{MAC}(x, y, z)$ is possible. This rule implies a mapping of the two operations to a single multiplier-adder-accumulator (MAC) in contrast to a separate implementation by one multiplier and one adder. As we can recognize, the use of this component-directed transformation has a major impact on the synthesis result: in the latter case, only one MAC is required because this component can also be employed as a simple multiplier or adder by applying the particular identity element to the corresponding input port. Since components like MACs are capable of performing two ore more operations in one execution step we call such functional units *complex components* or *BIC* (*built-in-chaining*)*-components* [15].

Even this simple example has shown that synthesis results are strongly dependent on the choice of a certain formulation style of the behavioral description. Obviously, it might be difficult to recognize in advance how to formulate a behavioral description that leads to the best synthesis result. This becomes almost impossible for complex data-flow dominated circuits. The goal of applying algebraic transformations can thus be stated as *making synthesis results mostly independent of the formulation style* or *to transform the input description such that synthesis yields better results than for the original description*.

The remainder of this paper is organized as follows: Section 2 gives an overview of the research on algebraic optimization in different areas. After introducing some hardware-related transformations in section 3 we describe the genetic algorithm including the chromosomal representation and the genetic operators in section 4. Section 5 presents experimental results for several standard benchmarks, and section 6 concludes the paper.

2 Related work

The use of algebraic transformations has been established in different domains: In classical computer-algebra systems such as MAPLE [5] or MATHEMATICA [21] they are indispensable for the transformation and simplification of algebraic expressions. In the domain of high-level-language compilers (see [2] for an overview) algebraic transformations are mainly used for tree height reduction, common subexpression elimination, constant folding, constant propagation and rather simple optimizations based on strength reduction.

In the area of high-level synthesis algebraic transformations have been particularly used for improving resource utilization [18] [17], tree-height minimization [6] [7], the maximization of data throughput [8] [10] and minimization of power consumption [3]. The use of algebraic transformations in combination with complex components (e.g. MACs) was first proposed in [14]. However, to the author's knowledge there is no optimization technique that exploits component-directed transformations in the same extent as the approach presented in this paper.

An approach based upon evolutionary programming for an area efficient design of Application Specific Programmable Processors (ASPP) has been published in [22]. ASPPs are programmable architectures which are designed for a set of different algorithms. The underlying approach is based on a genetic algorithm for transforming the particular data-flow graphs such that a given *behavioral kernel* (defined by a set of RT-level components) is optimally exploited by the algorithms.

Concerning the chromosomal representation of data-flow graphs and genetic operators (mutation and crossover), our method appears similar to [22]. However, it combines the concept of evolutionary programming with the algebraic optimization techniques for critical path minimization and improved resource exploitation on a finer level of granularity.

3 Overview of algebraic optimizations

The introducing example has already shown that apart from the exploitation of algebraic laws e.g. commutativity, associativity, distributivity etc., even hardware-related transformations have the potential of a considerable reduction of hardware costs and the critical path. In this section we demonstrate how hardware-related transformations can be specifically employed in order to reduce resource requirements.

3.1 General hardware-related transformations

First let us consider the expression $x * c$. Let c be a constant with $c = 2^n$. Obviously, the expression can be implemented in several ways: a) using a multiplier, b)

[†]In this example all operations are assumed to be single-cycled.

using a shifter, or c) by an offset in the wiring pattern. The transformation of the multiplication to a possibly hardwired shift-operation is well known as *strength reduction* [2], i.e. an expression with a cost intensive realization is always replaced by a "cheaper" expression. This optimization technique is usually exploited in almost all modern software compilers, however rarely in synthesis systems.

The exploitation of powers of two is not restricted to multiplications: Consider the expression $x_{[\mathrm{msb}:0]} + y$. [msb:0] represents the bit-slice† of x, and $y = 2^n$ is a constant again. Since an addition with 2^n increments only the upper bits [msb:n] whereas the lower bits [n-1:0] are not affected, we can state the following transformations: $x + 2^n \Leftrightarrow (x_{[\mathrm{msb}:n]} + 1) \,\&\, x_{[n-1:0]} \Leftrightarrow \mathrm{inc}(x_{[\mathrm{msb}:n]}) \,\&\, x_{[n-1:0]}$. Hence, additions with a power of two can be replaced by increment operations and thus lead to cheaper hardware realization. The same transformations also hold for $x - 2^n$, however applying a decrement operation. The reduction of an addition to an increment operation can also be exploited in order to find a cost efficient implementation for $1 - z$ with only one incrementer and one inverter: $1-z \Leftrightarrow -z+1 \Leftrightarrow \overline{z}+2 \Leftrightarrow \mathrm{inc}(\overline{z}_{[\mathrm{msb}:1]}) \,\&\, z_{[0:0]}$.

All transformations presented above are accompanied by an immediate reduction of the realization costs. However, we still have to take transformations into account which temporarily have the opposite effect:

Constant Unfolding is a technique that promises a further improvement in terms of cost and speed by splitting constants into a power of two and a remainder. Consider the rule $c = [c - r]^{\ddagger} + r$, where $[c - r] = 2^n$ is a constant. We can split expression $9 * x$ into $(2^3 + 1) * x$, which is equivalent to $2^3 * x + x$. Since $2^3 * x = x \,\&000$ can be implemented by an offset in the wiring pattern, only one adder instead of a multiplier is required to realize the expression. Another transformation rule is concerned with introducing identity elements which may be necessary to increase the applicability of further transformations: $a + a \Leftrightarrow (a * 1) + (a * 1) \Leftrightarrow a * (1 + 1) \Leftrightarrow a * 2$. Obviously, the use of identity elements is necessary for the formal proof of equivalence without knowledge of the rule $a + a \Leftrightarrow a * 2$. Although the introduction of identity elements helps to find new formulations of an initial expression, it also temporarily increases the implementation costs. Concerning the length of the critical path it was shown in [13] that the creation of additional operations in the DFG may have a positive effect on synthesis results. Therefore it is essential that the underlying optimization method does not reject transformations which temporarily lead to suboptimal results.

†Corresponding to the VHDL [9] notation `x : Bit_vector (msb DOWNTO 0)` where `msb` denotes the most significant bit, i.e. `msb := x'LENGTH - 1`.

‡The brackets mean an instant evaluation of the subexpression, e.g. constant $c = 9$ is splitted into $[9 - 1] + 1 = 8 + 1$.

3.2 Component-driven transformations

As we saw in the introduction, the use of a special component-driven transformation for employing multiplier-adders has the potential of a further reduction of both hardware costs and the critical path. In the following example we demonstrate how such transformations can be applied in order to exploit existing library components more efficiently: Consider the transformation rule $x + y + z \Leftrightarrow \mathrm{CADD}(x, y, z)$, with $z \in \{0, 1\}$. This rule implies that expression $x + y + z$ is mapped to one carry-adder. In combination with the transformations presented above the following rule allows to implement $x + y - z$ with $y = 2^n$ by only one carry-adder and one inverter: $(x + \overline{z}_{[\mathrm{msb}:n]} + 1) \,\&\, \overline{z}_{[n-1:0]} \Leftrightarrow \mathrm{CADD}(x, \overline{z}_{[\mathrm{msb}:n]}, 1) \,\&\, \overline{z}_{[n-1:0]}$.

This section has shown the applicability and the positive impact of hardware-related transformation rules concerning hardware costs and speed. We also have recognized that some transformations temporarily may increase the costs but allow the application of further rules which globally may decrease the costs. Since cost-driven heuristics do not work appropriately in this case, we formulated the problem using a probabilistic approach based on a genetic algorithm which will be presented in the following section.

4 Algebraic optimization by genetic algorithm

The general principle in natural evolution as well as evolutionary algorithms is the optimization of a population's fitness in the course of generations driven by the randomized processes of *selection*, *recombination* and *mutation*. Genetic algorithms as one representative of evolutionary algorithms (see [1] for an overview) have been proven to be very powerful for searching vast solution spaces. Solutions found by genetic algorithms are generally close to the global optimum.

4.1 Chromosomal representation

Each chromosome of the population represents one semantically equivalent formulation of an initial data-flow graph. The genes which are located on the chromosome—or the gene positions, to be precise—represent the operations of the DFG together with references to the predecessor operations.

Example: We use expression $((a * 2) + (b * 2)) + 1$ as a running example in order to demonstrate the chromosomal representation and the particular genetic operators. Figure 2 depicts the chromosome representation $\alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta$ with its genes $\alpha : \beta + 1$, $\beta : \gamma + \delta$, $\gamma : a * 2$, and $\delta : b * 2$.

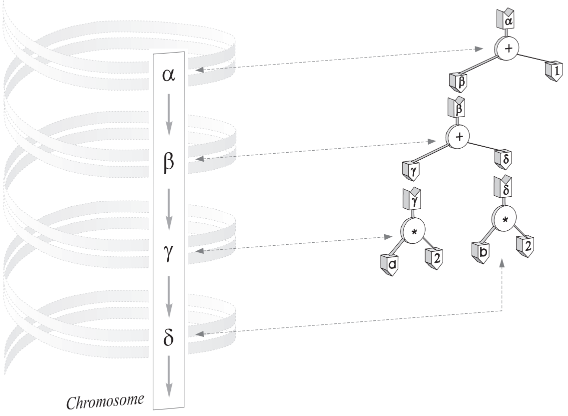

Figure 2: Representation of expression $((a * 2) + (b * 2)) + 1$ by a sequence of genes

Each gene has a specific phenotype, called its allele. We distinguish different alleles of the same gene by roman numbers I, II, etc. Operations of the original data-flow graph are thus represented by the alleles $\alpha_{\mathrm{I}}, \beta_{\mathrm{I}}$ etc. The set of alleles for one specific gene represents the set of functionally equivalent expressions. Since all alleles of the same gene are semantically equivalent they are mutually replacable without changing the functionality of the entire DFG. Nevertheless, the resulting DFG can be distinguished by potentially different hardware costs and critical path lengths.

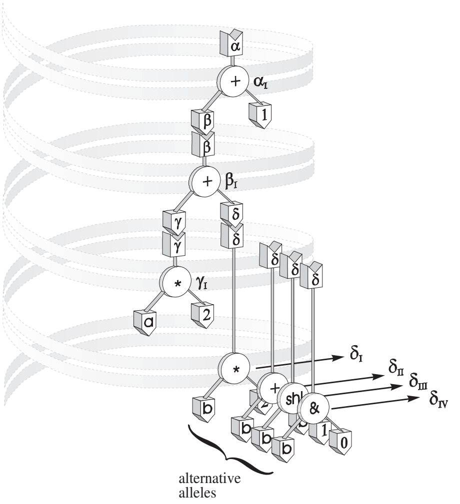

Figure 3: Alternative alleles

Figure 3 shows new alternative alleles for the gene δ after applying the transformations $b * 2 \Rightarrow b + b \Rightarrow b \text{ shl } 1 \Rightarrow b \;\&\; 0$. Obviously, each allele of gene δ can be replaced by any other allele without changing the semantics of the expression. In the same way, algebraic transformations can be applied to the other genes of the chromosome.

Figure 4 presents the gene pool for our example after applying associativity, distributivity and the demonstrated simplification of multiplications. The gene pool serves as a basis for the creation of the initial population: for each chromosome that represents one individual of the population we can arbitrarily select one allele at each gene position (figure 4). Due to the fact that each chromosome implicitly represents the structure of a data-flow graph, the creation of any DFG can be performed in linear time O(n). In this example, the chromosome $\alpha_{\mathrm{III}} \rightarrow \beta_{\mathrm{III}} \rightarrow \gamma_{\mathrm{IV}} \rightarrow \delta_{\mathrm{II}} \rightarrow \epsilon_{\mathrm{II}} \rightarrow \zeta_{\mathrm{I}}$ represents the expression $\mathrm{inc}((a + b) \text{ shl } 1)$. It should be mentioned that chromosomes may also contain some *redundant* genes (in this example: $\gamma \rightarrow \delta \rightarrow \epsilon$) which have no direct influence on the created DFG. However, redundant genes can be reactivated instantly by small mutations of the chromosome.

As we have seen, the chromosomal representation introduced above guarantees that all subexpressions referenced by different alleles at the same gene position are semantically equivalent. This means that every possible allele substitution at any gene position will subsequently lead to a new semantically equivalent data-flow graph. This property is crucial for preserving the correctness of the used genetic operators, namely *mutation* and *crossover*.

4.2 Genetic operators

4.2.1 Mutation

The principle of mutation was implicitly shown in figure 3: Without changing the entire semantics we can transform a data-flow graph by a simple gene mutation that substitutes the selected allele by another one at the same gene position. For example, the substitution of allele δ_{I} by δ_{IV} represents the transformation $(a*2)+(b*2)+1 \Leftrightarrow (a*2)+(b \;\&\; 0)+1$.

Obviously, the mutation operator can be implemented in time O(1).

4.2.2 Crossover

The goal of crossover is to recombine the parental properties and its transmission to the new offspring. In the meaning of transforming algebraic expressions, crossover recombines subexpressions of the parental data-flow graphs and can be sketched as follows:

1. Create two new chromosomes representing the children.
2. Select an arbitrary gene position.

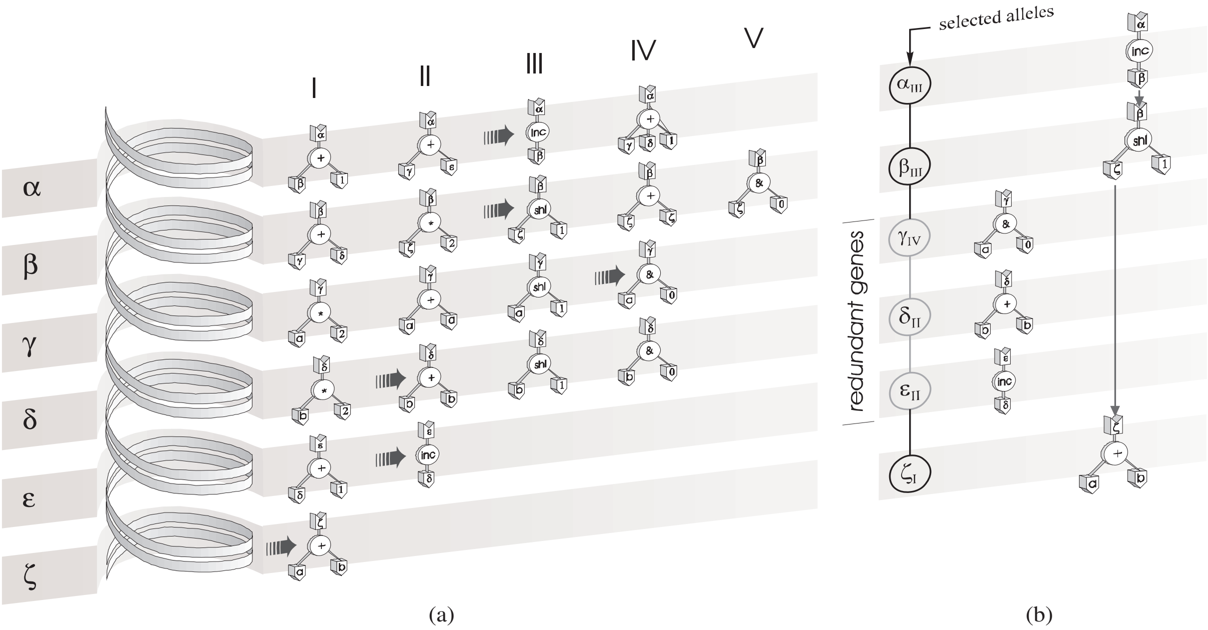

Figure 4: (a) Extended gene pool for the running example and
(b) exemplary creation of a DFG by selecting alleles at each gene position

3. Copy all alleles from first (second) chromosome up to the selected gene position to the first (second) child.
4. Copy the remainder of the first (second) chromosome the second (first) child.

Also crossover benefits from the underlying representation and always creates only those DFGs which are semantically equivalent to the initial specification. Obviously, crossover can be implemented in linear time O(n) where n represents the chromosome length.

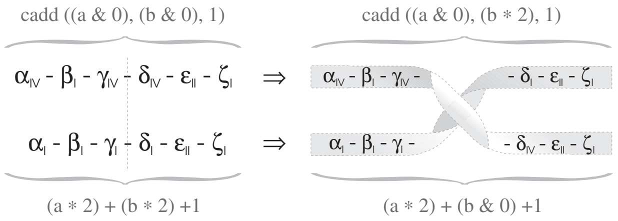

Figure 4: Crossover

Figure 4 demonstrates crossover for our running example. As initial chromosomes we use $\alpha_{\text{IV}} \rightarrow \beta_{\text{I}} \rightarrow \gamma_{\text{IV}} \rightarrow \delta_{\text{IV}} \rightarrow \epsilon_{\text{II}} \rightarrow \zeta_{\text{I}}$ corresponding to cadd$((a \ \& \ 0), (b \ \& \ 0), 1)$ and $\alpha_{\text{I}} \rightarrow \beta_{\text{I}} \rightarrow \gamma_{\text{I}} \rightarrow \delta_{\text{I}} \rightarrow \epsilon_{\text{II}} \rightarrow \zeta_{\text{I}}$ which corresponds to $((a * 2) + (b * 2)) + 1$. The crossover position has been chosen at gene position δ. In the resulting expressions, $(b \ \& \ 0)$ and $(b * 2)$ have been interchanged.

4.2.3 Selection

Selection is a crucial process in (simulated) evolution that favors individuals of higher fitness to survive ("survival of the fittest") and thus become the co-founders of the next population. Generally, we presume the probability of an individual to be selected is proportional to its fitness. This enables even individuals with a lower fitness to be selected and thus to transmit their gene information to the offspring.

In the meaning of the final hardware realization we define the fitness as weighted sum of the required functional units and the length of the critical path. In contrast to the critical path computation that can be done in linear time by an ASAP (or ALAP) scheduling, the exact computation of resource costs is NP-complete. Therefore, we have to employ resource estimation techniques (e.g. [19]) in order to value the effect of performed transformations. Surprisingly, experimental results have shown that even simple fitness functions are sufficient for producing excellent optimization results (see figure 5–6 and table 2). We used a combination of the critical-path length and hardware costs computed by *direct compilation*, i.e. each operation of the data-flow is associated with certain hardware costs. An advantage of using direct compilation is its efficient implementation in linear time and is thus crucial for the fast execution time of the optimization routine.

4.3 Skeleton of the genetic algorithm

Figure 5 presents the genetic algorithm that serves as a basis for algebraic optimization and takes pattern from the standard algorithm in [4].

```
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
2.2.1       compute fitness of all individuals
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 5: Skeleton 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 as long as the loop-exit criterion T_G is not fulfilled. The loop exit is usually controlled by the state of the generation counter or by the yielded gain of the population's fitness.

The outer loop is required in order to extend the current gene pool in two directions: On the one hand we introduced the new alleles δ_{II}, δ_{III} and δ_{IV} in our example to represent the transformations $a * 2 \Rightarrow a + a \Rightarrow a$ shl $1 \Rightarrow a \,\&\, 0$. On the other hand the chromosome is extended by new genes along with their alleles. In our example, we introduced gene ϵ with its alleles ϵ_{I} and ϵ_{II} and the new allele α_{II} in order to exploit the associativity.

In contrast to the actual composition of new DFGs including their recombination, mutation and selection in progress with the *generations*, we call the continuous extension of the current gene pool by transformations *epochs*. The termination of the entire algorithm is controlled by fulfilling T_E that can be realized in the same way as the loop-exit criterion T_G.

5 Experimental results

We applied the presented algorithm to several standard benchmarks. For each example we achieved a gain of up to 30 % concerning the critical path and an area gain of up to 26 %. 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.

All presented results have been computed on a SparcStation 20. The execution time of the optimization routine was for all examples approximately one minute for the chosen parameters.

Figure 6 and 7 present the initial and the optimized dataflow graph of the 5th-order elliptical wave filter[†] [11], respectively. All operations are assumed to be single-cycled. The *macro-nodes* in fig. 7 represent multiply-add subexpressions which have been bound to MACs. These complex components can also be used for computing additions by simply applying the corresponding identity elements to the appropriate inports (control steps 3, 8, 9, 11 in fig. 8).

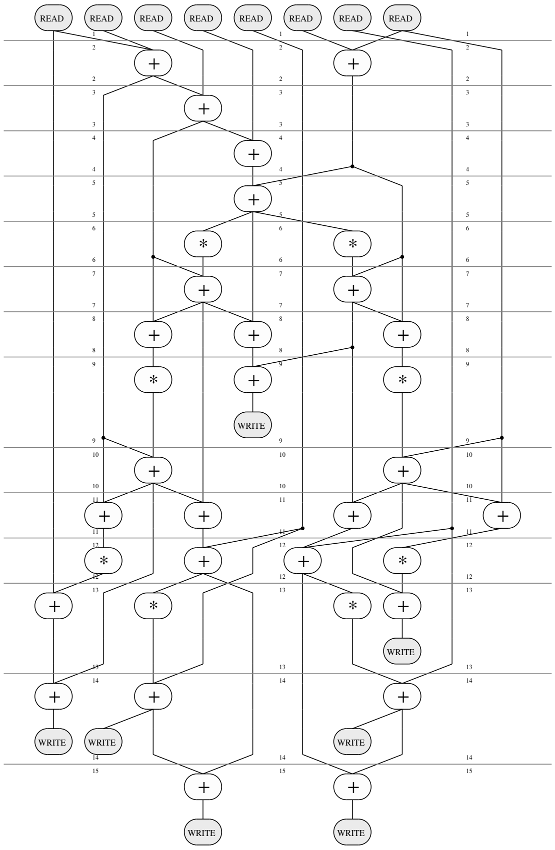

Figure 6: Initial DFG of the EWF-benchmark

[†] in contrast to some existing formulations of this benchmark containing only multiplications by 2, we dispense with the exploitation of the special transformation rule $x * 2 \rightarrow x \,\&\, 0$. I.e. we assumed multiplications with different coefficients in order to produce results for rather realistic applications

benchmark[a]	#cs	add $2405k\lambda^2$	sub $2433k\lambda^2$	mult $14717k\lambda^2$	mac $15435k\lambda^2$	gain (area[b]) $\frac{area_{orig}-area_{opt}}{area_{orig}}$	gain (#cs) $\frac{\#cs_{orig}-\#cs_{opt}}{\#cs_{orig}}$
EWF_{orig}	14	3	0	2	0		
EWF_{opt}	10	2	0	0	2	**2.6 %**	**28.6 %**
FFT_{orig}	8	3	4	12	0		
FFT_{opt}	8	0	4	5	4	**25.1 %**	0 %
FIR_{orig}	10	2	0	2	0		
FIR_{opt}	7	2	0	1	1	-2 %	**30.0 %**
$FDCT_{orig}$	6	4	2	8	0		
$FDCT_{opt}$	6	1	2	3	3	**26.1 %**	0 %
BF_{orig}	15	2	1	2	0		
BF_{opt}	14	0	1	0	2	**9.1 %**	**6.7 %**
$Edge_{orig}$	10	2	2	4	0		
$Edge_{opt}$	9	0	2	1	3	**3.9 %**	**10.0 %**

Table 2: Optimization results

[a]EWF: elliptical wave filter, FFT: fast fourier transformation, FIR: finite impulse response filter, FDCT: fast discrete cosine transformation, BF: bandpass filter, Edge: edge detection

[b]area of allocated functional units

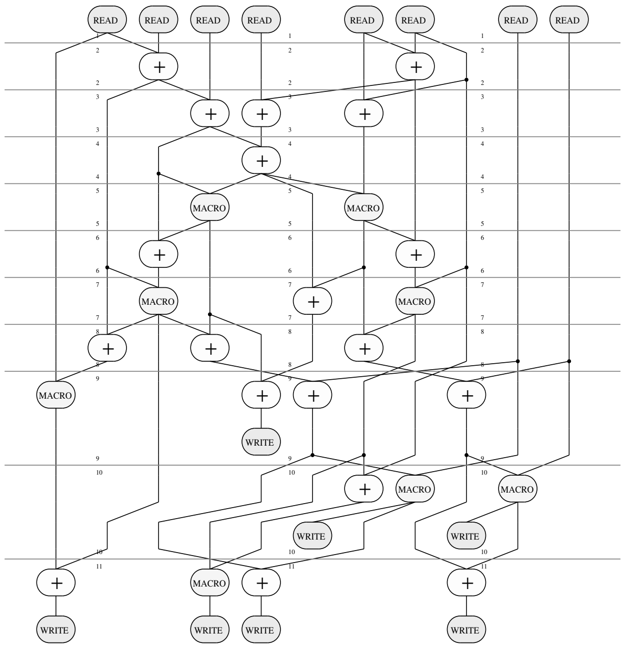

Figure 7: Optimized DFG of the EWF-benchmark

It seems to be quite obvious, that manually formulating a behavioral description with the same quality of the presented optimized DFG is virtually impossible. In this example, the critical path has been shortened from 14 to 10 control-steps requiring only two MACs and two adders instead of three adders and two multipliers.

Table 2 shows the synthesis results for several benchmarks before and after algebraic optimization. We used the high-level synthesis system OSCAR [12] for synthesizing the original and the optimized design. Due to its underlying integer programming formulation all presented results are optimal concerning the overall costs of functional units. Areas and delays[‡] of functional units have been adopted from the underlying 1.0μ VLSI component library [20]. The execution times of high-level synthesis were always less than one second.

6 Conclusion

We presented a genetic algorithm based approach for algebraic optimization of data-flow graphs. Due to the underlying chromosomal representation all genetic operators are correctness preserving and can be implemented very efficiently. Apart from standard transformation rules such as commutativity, associativity and distributivity, we also support hardware-related transformations. It has been shown that even these transformations have a positive effect to the quality of the achieved synthesis results. Since all rules are stored in an external library, they can be modified or extended by the designer.

The system has been implemented as a front end to an ILP-based synthesis system [12] and benefits from its capability of supporting complex component libraries. However, the approach can also be easily realized as a source-to-source (e.g. VHDL-to-VHDL) optimizer that

[‡]add [`vdp1add001`]: 16.1 ns , sub [`vdp1sub001`]: 16.7 ns, mult [`vdp3mlt004`]: 112.1 ns, mac [`vdp3mlt006`]: 112.1 ns

enables the employed synthesis system to support complex functional units (e.g. MACs).

The experimental results have shown the efficiency of our method. For all examined benchmarks (EWF, FFT, FDCT, FIR-Filter, Bandpass Filter, Edge Detection) we could achieve a considerable reduction of the critical path and/or the area.

References

[1] T. Bäck and F. Hoffmeister. Global Optimization by Means of Evolutionary Algorithms. *in: A. N. Antamoshkin (Ed): Random Search as a Method for Adaption and Optimization of Complex Systems, pages 17-21, Divnogorsk, UdSSR, March 1991, Krasnojarsk Space Technology University*, 1991.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys, Vol. 26, No. 4*, pages 345–420, 1994.

[3] 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.

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

[5] K. O: Geddes, G. H. Gonnet, and B. W. Char. MAPLE User's Manual (2nd ed.). Technical Report CS-82-40, University of Waterloo, 1982.

[6] 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.

[7] 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.

[8] 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.

[9] Design Automation Standards Subcommittee of the IEEE. IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-87). *IEEE Inc., New York*, 1988.

[10] 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.

[11] S. Y. Kung, H. J. Whitehouse, and T. Khailath. *VLSI and Modern Signal Processing*. Prentice Hall, 1985.

[12] 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.

[13] D. A. Lobo and B. M. Pangrle. Redundant Operator Creation: A Scheduling Optimization Technique. *Proceedings of the 28th Design Automation Conference*, pages 775–778, 1991.

[14] P. Marwedel. Matching System and Component Behaviour in the MIMOLA Synthesis Tools. *Proceedings of the EDAC*, pages 146–156, 1990.

[15] P. Marwedel, B. Landwehr, and R. Dömer. Built-in Chaining: Introducing Complex Components into Architectural Synthesis. *Proceedings of the ASP-DAC*, 1997.

[16] M. C. McFarland, A. C. Parker, and R. Camposano. The High-Level Synthesis of Digital Systems. *Proceedings of the IEEE, Vol. 78, No. 2*, pages 301–318, 1990.

[17] M. Potkonjak and S. Dey. Optimizing Resource Utilization and Testability using Hot Potato Techniques. *Proceedings of the 31st Design Automation Conference*, pages 201–205, 1994.

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

[19] A. Sharma and R. Jain. Estimating Architectural Resources and Performance for High-Level Synthesis Applications. *IEEE Transactions on VLSI Systems, Vol. 1, No. 2, June 1993*, 1993.

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

[21] S. Wolfram. *Mathematica, A System for Doing Mathematics by Computer*. Addison Wesley, 1988.

[22] W. Zhao and C. A. Papachristou. An Evolution Programming Approach on Multiple Behaviors for the Design of Application Specific Programmable Processors. *Proceedings of ED & TC*, 1996.