Improving processor architecture exploitation by genetic algorithm based algebraic optimization

Birger Landwehr
University of Dortmund, Germany
landwehr@ls12.cs.uni-dortmund.de

Abstract

This report presents a new approach for the algebraic optimization of computationally intensive applications. The presented approach is based upon the paradigm of simulated evolution which has been proven for solving large non-linear optimization problems. We introduce a chromosomal representation of data-flow graphs which ensures that the correctness of algebraic transformations realized by the genetic operators *recombination*, *mutation*, and *selection* is always preserved. We also present different fitness functions allowing to simply adapt the algorithm to different processor architectures in order to produce the best feasible solution concerning the given target architecture. The presented method has been integrated as a C-to-C converter within a versatile compiler framework and has proven its efficiency for several DSP applications.

Chapter 1

Introduction and Motivation

Due to the rapid progress in the design of powerful processors including general purpose and especially digital signal processors (DSP) during the past years, there has been an increasing demand for high optimizing compilers. However, the architectural variety of available DSP processors makes it difficult for the compiler industry to immediately respond to the market. One possibility to cope with this problem is the use of retargetable compilers which has been one of the major research topics in the recent years.

Considering the structure of any compiler for a high-level language (e.g. C/C++) we can identify at least four individual parts: The language specific *front end* which is used to parse the given source code, the *intermediate representation* that serves as a basis for control and data flow transformations, the different *optimization algorithms*, and the *code generator* which performs code selection, instruction scheduling and register allocation. In the area of retargetable code generation, significant contributions have been published recently (see e.g. [Leu97] for an overview). Concerning optimization techniques (including algebraic optimizations), there is almost no comparable work presented in the literature so far which takes into account a detailed processor specification.

VLIW processors in particular offer the possibility to perform several arithmetic operations in parallel which make them ideal for real time applications. Considering the source code of most DSP applications it becomes clear, however, that even the computationally intensive sections can not be used directly for code generation without optimizations.

Since the architectures of modern processors differ concerning their topologies as well as the number and functionality of functional units, the increasing demand for adaptable optimization methods which take a detailed processor specification into account is clearly visible.

The remainder of the report is organized as follows: In the next section we give a short overview of the related work which originates from both the traditional research area of software compilers and recently the domain of CAD tools for behavioral synthesis of integrated circuits (also known as "silicon compilers"). After discussing the different paradigms of target specific and target independent optimizations in section 3, we describe the actual genetic algorithm based optimization approach in section 4. This includes the chromosomal representation of data-flow graphs, the genetic operators, and especially the individual fitness functions required for different processor architectures. In Section 5 we briefly present the entire compiler framework the optimization approach was integrated into in form of a stand alone C-to-C converter. Section 6 summarizes the presented work.

Chapter 2

Related work

The use of algebraic transformations has been established in several domains: In classical computer-algebra systems such as MAPLE [GGC82] or MATHEMATICA [Wol88] they are indispensable for the transformation and simplification of algebraic expressions.

In the domain of high-level-language compilers (see [BGS94] for a comprehensive overview) algebraic transformations are used along with control flow optimizations for both minimization of machine code length and, particularly for real time applications, maximization of the execution speed. Some of the known standard techniques are constant folding, constant propagation, common subexpression elimination, algebraic simplification, strength reduction, reassociation, and expression tree balancing. Although the latter is well suited to optimize a given source code for VLIW architectures due to its property of increasing the level of parallelism of a data-flow graph, it usually lacks an adequate consideration of the underlying architecture.

In the area of behavioral synthesis, algebraic transformations are being used for improving resource utilization [PR94] [PD94], tree-height minimization [HC89] [HC94], the maximization of data throughput [HR94] [IPDP93], and minimization of power consumption [CPM$^+$95]. Recent research for high-level address optimization and operation cost minimization has been published in [Jan00].

A genetic algorithm based approach presented in [LM97] allows for the first time to incorporate detailed hardware information into algebraic optimization and aims at improving resource exploitation and critical path minimization. Another approach based upon evolutionary programming published in [ZP96] is used for an area efficient design of Application Specific Programmable Processors (ASPP) and also takes the underlying hardware into account. The presented approach is based on a genetic algorithm for transforming a set of data-flow graphs concurrently so that a given *behavioral kernel* (defined by a set of RT-level components) is optimally exploited by the algorithms.

Especially the latter two approaches have proven the applicability of simulated evolution for algebraic optimization due their distinct properties: Genetic algorithms are robust and powerful methods for searching vast solutions spaces. They are capable of overcoming local optima so that solutions found by genetic algorithms are usually close to the global optimum. Finally, different objective functions can be simply incorporated by choosing an appropriate fitness function. Consequently, genetic algorithms seem to be an adequate tool even for multi-objective algebraic optimization in the domain of high-level language compilers.

Chapter 3

Target-specific vs. target independent optimizations

As already mentioned in the introduction, the underlying processor architecture has a large impact on the source code formulation of a given application. This means that both the order of operations within the DFG[1] and the potential range for parallelization, i.e. the “width” of the DFG are fundamentally affected by a certain formulation.

The following example shows the necessity of incorporating the target architecture into the optimization process: Consider the expression $f = a - b - c + d + e - f$ and a general purpose processor with a single functional unit as target architecture.

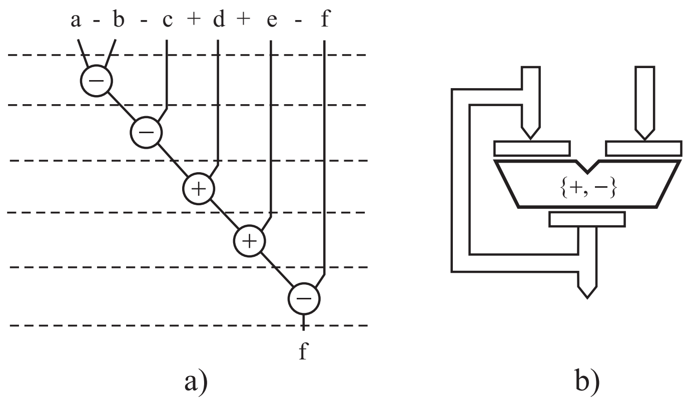

Figure 3.1: Given DFG (a) and processor data-path (b)

Obviously, the structure of the data-flow graph depicted in fig. 3.1 is suitable for the depicted archi-

[1]The DFG shows data dependencies which can be changed by algebraic transformations, in contrast to the execution order determined by instruction scheduling.

tecture since no operations can be performed in parallel (only one functional unit is available) and, due to the given sequential order of operations, register moves of intermediate results can be avoided. Apart from reducing memory accesses, no additional algebraic optimizations can be performed in this case[2].

In contrast to standard processors, VLIW architectures offer a much higher potential for optimization due to their high degree of parallelism. In order to take the given distinctive architectural features into consideration during optimization we furthermore distinguish between the following two classes:

Multiple Instruction Multiple Data (MIMD)

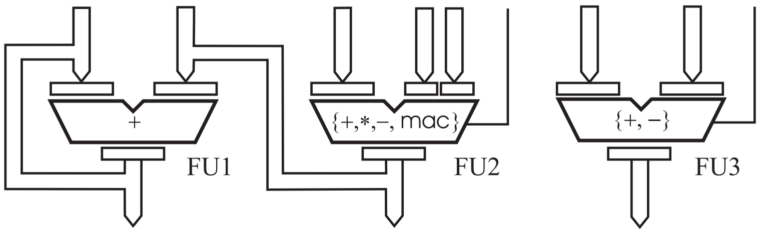

Figure 3.2: MIMD architecture

The MIMD-architecture given in fig. 3.2 consists of three functional units each capable of performing a distinct set of operations in parallel independent of each other. The interconnection $FU2 \longrightarrow FU1$ allows to forward the result of $FU2$ to $FU1$ in the subsequent execution cycle and hence avoids time consuming data transfers of temporary values.

[2]The only possible situation in which a reduction of the resulting number of arithmetical operations can be expected is the exploitation of the distributivity $(a*c)+(b*c) \rightarrow (a+b)*c$. In this case however, this rule can obviously not be applied.

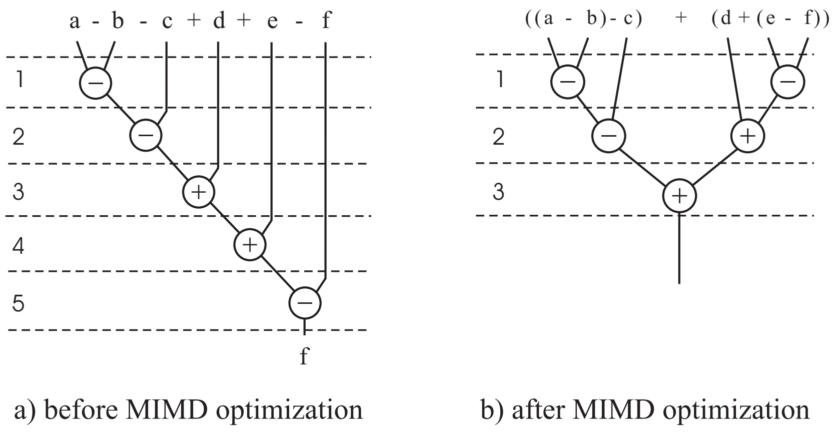
a - b - c + d + e - f
1
2
3
4
5
f
a) before MIMD optimization
((a - b) - c) + (d + (e - f))
1
2
3
b) after MIMD optimization

Figure 3.3: Algebraic optimization for SIMD architectures

Since up to three operations can be performed in parallel using the shown architecture, the DFG depicted in fig.3.1 does not yet have an adequate structure to exploit the available parallelism. Obviously, only a sequence of transformations leading to a DFG such as given in fig.3.3b allows to reduce the execution time from five to three cycles.

Single Instruction Multiple Data (SIMD):

In contrast to MIMD-architectures, all functional units perform the same operation type during one cycle (e.g. only additions or only multiplications).

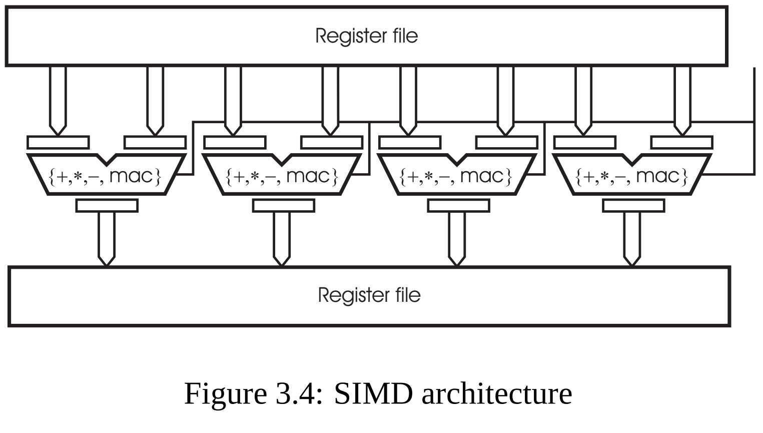

Figure 3.4: SIMD architecture

In this case, improving the architecture exploitation means to rearrange the operations of the DFG such that the set of operation types are maximized for each cycle. We call this *operation type homogenization*.

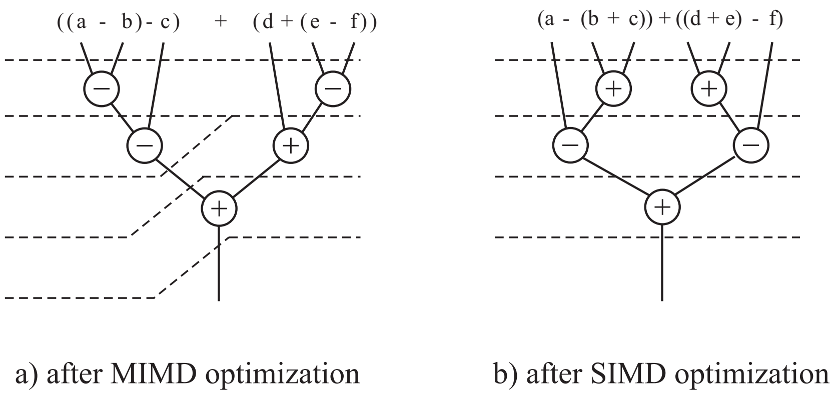
((a - b) - c) + (d + (e - f))
(a - (b + c)) + ((d + e) - f)
a) after MIMD optimization
b) after SIMD optimization

Figure 3.5: Algebraic optimization for SIMD architectures

The DFG given in figure 3.5a would result in four cycles since the subtraction and addition cannot be executed in parallel due to the underlying architecture. Only restructuring the DFG shown in figure 3.5b allows reducing the execution time to three cycles.

Apart from general algebraic transformation rules it should be emphasized that *architecture specific transformations* may lead to a further improvement of the architecture exploitation. For instance, the use of multiplier-accumulators can be supported by rearranging expressions such that multiplications and additions are ordered appropriately.

As we have seen, different processor architectures require different optimization strategies, i.e. the incorporation of an additional architecture specification is indispensable for an effective optimization.

Chapter 4

GA-based algebraic optimization

Genetic algorithms (GA) have been proven to be very powerful in solving NP-hard optimization problems usually characterized by non-linear, non-steady functions. GAs mimic the natural principle of evolution by representing a set of possible solutions of a given problem by a population of individuals. In each generation, the best individuals are selected in order to create the next offspring by recombining their genetic material. Due to the fact that only those individuals with a better fitness as compared to their competitors are able to transmit their genetic material to the offspring ("survival of the fittest"), a convergence of the population's fitness to the global optimum can be expected in the course of the generations.

4.1 Chromosomal representation

One major task in adapting a genetic algorithm to a certain problem is to establish an appropriate chromosomal representation. The actual problem of representing an algebraic expression by a chromosome is to find a suitable mapping of its tree-like structure onto a sequence of genes. Although it is in fact possible to define tree-like chromosomes, the realization of correctness preserving genetic operators (*crossover, mutation*) would become very complicated if not impossible. Repair mechanisms are also not appropriate here due to their computational complexity.

The chromosomal representation can be directly derived from the given C-code which is, for the sake of clarity, shown without variable and constant assignments or control flow information (jumps or labels). All those statements are also represented by the chromosome but are not considered during optimization.

```
gene  0:  t1 = a - b;     [·, ·]
      1:  t2 = t1 - c;    [→0, ·]
      2:  t3 = e - f;     [·, ·]
      3:  t4 = d + t3;    [·, →2]
      4:  t5 = t2 + t4;   [→1, →3]
```

Figure 4.1: Chromsomal representation of $t5 = ((a-b)-c)+(d+(e-f))$

Fig. 4.1 represents the *chromosome* for the expression presented in fig. 3.5. We define each statement (represented by a 3-address-code) of the basic block as a particular *gene*. A gene corresponds to a single function (in the pure mathematical meaning) that can be computed in one step by an appropriate functional unit whereas an *allele* represents its actual algebraic formulation. In other words, a certain gene describes for example that a variable x is multiplied with the constant 2, the allele describes its actual representation by an algebraic expression (e.g. $x*2$, $x+x$ or $x \ll 1$). Additionally, each allele contains references (depicted in brackets in fig. 4.1) to those genes which represent the predecessor in the DFG. For instance, gene 1 ($t2 = t1 - c$) contains a reference to gene 0 where $t1$ is represented as well as the variable c, denoted by a null reference "·".

Thus, we can represent each DFG by the chromosome $(a_0, \ldots, a_{n-1})$ of length n consisting of an enumeration of selected alleles $a_0 \leq a_i \leq a_{n-1}$ for each gene position i.

4.1.1 Representation and handling of non-arithmetical statements

Among algebraic expressions, the source code of a certain application usually contains C-constructs which must be treated appropriately in order to preserve the original semantic:

- **Variable and constant assignments**
  Basically, variable and constant assignments are not affected by algebraic transformations except the possibility of constant folding and unfolding. However, read/write dependencies between variables must be preserved. They are represented (analogous to the data dependencies) by additional gene references at each allele. The same applies to memory accesses and pointers, such that the order of statements consisting of expressions with array variables and pointers are not changed by any transformation.

- **Function calls**
  Function calls must be considered in two different respects:

  First, C-functions may cause side effects especially if they write to global variables or contain input/output statements (e.g. `printf`). Hence, the evaluation order of expressions containing such functions must not be changed by transformations. This means that the algebraic properties of operations relating to function calls are very restricted. For instance, the expression $f(x)+g(y)$ is not commutative since the evaluation order of $f(x)$ and $g(y)$ would be changed.

Second, in contrast to arithmetical expressions without C-function calls where common subexpression elimination can be applied without problems, expressions which do contain function calls must not be transformed: Consider the expression $(a*f(x))+(b*f(x))$ which contains $f(x)$ both in the left and in the right addend. A transformation into $(a+b)*f(x)$ is obviously not permitted if $f(x)$ causes side effects.

Consequently, it must be assured that if a certain C-function can not be clearly identified to be free of side effects, all subgraphs of the DFG which contain those function calls must be excluded from any transformations.

4.2 Gene pool expansion

Since there exists only one algebraic formulation (i.e. an allele) for each gene at the beginning, the initial gene pool for our example consists of only five genes so far. In order to create new variants of DFGs, the gene pool is first expanded which results in an extension of the chromosome length. During optimization, the expansion of the gene pool is always performed in an intermediate step (between so-called “epochs”). This is done after a sufficient number of generations by applying transformation rules to the DFG subgraphs which are represented by singular genes or short gene sequences. In order to avoid a certain transformation rule being applied more than once at a certain gene position, additional information concerning the type of the rule and involved genes are stored.

	Gene	[0]	[1]	[2]	[3]
	0 : $t1 =$	**a - b [·, ·]**			
	1 : $t2 =$	**t1 - c [→0, ·]**	$a - t6$ [·, →5]		
	2 : $t3 =$	**e - f [·, ·]**			
	3 : $t4 =$	$d + t3$ [·, →2]	**t7 - f [→6, ·]**		
output gene →	4 : $t5 =$	$t2 + t4$ [→1, →3]	$t8 + d$ [→7, ·]	**t9 - c [→8, ·]**	$t17 - f$ [→16, ·]
	5: $t6 =$	**b + c [·, ·]**			
	6: $t7 =$	**d + e [·, ·]**			
	7: $t8 =$	$t3 + t2$ [→2, →1]	**t12 - c [→ 11, ·]**		
	8: $t9 =$	$t4 + t1$ [→3, →0]	**t10 + d [→9, ·]**	$t11 - b$ [→10, ·]	
	9: $t10 =$	**t3 + t1 [→2, →0]**			
	10: $t11 =$	**t4 + a [→3, ·]**			
	11: $t12 =$	**t13 - b [→12, ·]**	$t14 - t15$ [→13, →14]		
	12: $t13 =$	$t3 + a$ [→2, ·]	**t14 - f [→13, ·]**		
	13: $t14 =$	**a + e [·, ·]**			
	14: $t15 =$	**f + b [·, ·]**			
	15: $t16 =$	**t2 + d [→1, ·]**			
	16: $t17 =$	**t16 + e [→15, ·]**	$t2 + t7$ [→1, →7]		

Table 4.1: Gene pool for the example

Table 4.1 presents the expanded gene pool for the example given in fig. 3.5a. Genes 0–4 (allele 0) correspond to the initial gene pool and hence represent the original DFG. Genes 5–14 (including the accompanying alleles) as well as additional alleles for genes 0–4 were added later by applying algebraic transformations to the original DFG: for instance, the application of the associativity rule $a + (b + c) \to (a + b) + c$ to the subexpression $t5 = t2 + \underbrace{(d + t3)}_{t4}$ (represented by genes 4 and 3) in combination with the commutativity rule leads to the equivalent formulation $t5 = \underbrace{(t2 + t3)}_{t8} + d$. The latter is represented by an additional allele at gene position 4 (allele 1: $t5 = t8 + d$) and the newly created gene 7 (allele 0: $t8 = t2 + t3$).

4.3 Synthesis of new DFG variants

In our example, gene 4 represents the only output node of the basic block (variable $t5$). If the DFG contains several outputs, all corresponding genes are marked accordingly. For a given chromosome $C = (0,0,0,1,2,0,0,1,1,0,0,0,1,0,0,0,0)$ the resulting DFG is synthesized from the output nodes to the input nodes (leafs) of the graph: starting with gene 4 allele 2 ($t9 - c$), all subgraphs referenced by the selected allele are traversed recursively until all leafs have been processed, i.e. the genes are visited in the following order[1]: $4^{(2)} \to 8^{(1)} \to 9^{(0)} \to 2^{(0)} \to 0^{(0)}$. Thus, the resulting DFG corresponds to

[1]We use $g^{(a)}$ to denote gene g with its current allele a at this place.

$t5 = ((e - f) + (a - b)) + d - c.$

It should be emphasized that not all genes are required for the DFG construction; typically there are several genes in the gene pool which are temporarily redundant. For instance, the resulting DFG of the given chromosome only consists of the genes $4, 8, 9, 2,$ and 0 (enumerated from the root to the leafs of the DFG). All other genes are redundant for this DFG. Nevertheless, such genes may crucially influence the quality, i.e. the fitness of the resulting DFG, since they can be activated at any time by the genetic operators *crossover* and *mutation* as well.

4.3.1 Further DFG transformations

Apart from the application of general algebraic rules, there are two further DFG transformations which can be employed to improve the applicability of standard optimizations such as common subexpression elimination and constant folding.

Improving common subexpression elimination

Even though common subexpression elimination is a standard optimization technique available in all commercial compilers, simple approaches could fail in situations in which common subexpressions (CS) are ”invisible” to the compiler and hence cannot be removed:

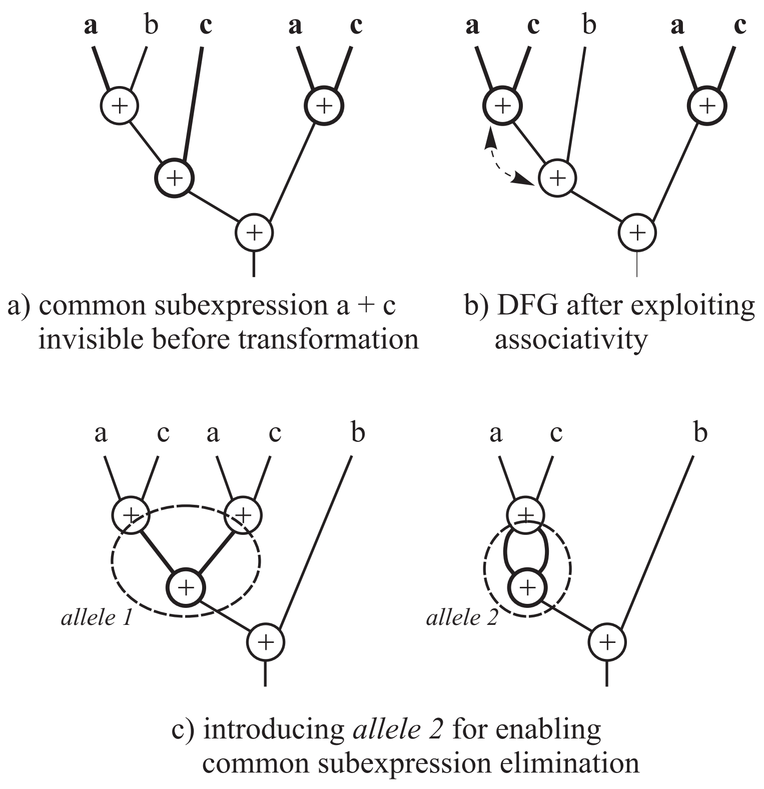
a b c a c
a c b a c
a) common subexpression a + c invisible before transformation
b) DFG after exploiting associativity
a c a c b
a c b
allele 1
allele 2
c) introducing allele 2 for enabling common subexpression elimination

Figure 4.2: Making common subexpressions visible to the compiler

As shown in fig. 4.2, algebraic transformations can also be used as a preprocessing step for enabling the compiler to find and eliminate common subexpressions.

Constant folding/unfolding

Constant folding as another standard optimization technique minimizes the number of operations by evaluating constant expressions at compile time. Similar to the scenario presented in fig. 4.2 it is possible that constant folding is enabled only after a sequence of transformations.

During algebraic optimization, however, it is also advisable to perform such transformations in the opposite direction: Constant unfolding means to split a certain value into a sum of constants and can be useful in at least the following situations:

- It is well known that constants which are a power of two may lead to very efficient implementations. This means that multiplications in particular can be replaced by a corresponding shift operation.
- Some processors require that the bitwidth of constants does not exceed a certain value, especially for algebraic operations with immediate values (i.e. values are part of the instruction

word). In this case, splitting a value into a sum of constants, each not exceeding a certain bitwidth, can enable those instructions and can hence improve the final code size and execution time.

Since the different alternatives are represented as alleles for a certain gene, performed transformations are not necessarily definite. Thus, single or even sequences of transformation steps can simply be canceled by the genetic operators discussed in the following section.

4.4 Genetic operators

Algebraic transformations are directly realized by the application of genetic operators. Since alternatives are available for a set of expressions after gene pool expansion, the application of transformation rules can be realized by selecting a certain allele at each gene position. This is done by crossover and mutation.

4.4.1 Crossover

The main task of crossover is to recombine the genetical information of two parent chromosomes aiming at recombining their (positive) properties and transmitting them to the offspring. In our case, the individual's properties are characterized by a more or less efficient implementation of the DFG or at least a part of it. Obviously, the fitness of an individual is strongly dependent on the cost model (e.g. the execution time of the generated code) of the inherited algebraic expressions. Crossover recombines the properties of the predecessor generation and transmits them to the offspring. Even though this step is purely probabilistic, the periodical selection process ensures that only individuals with a higher fitness as compared to the competitors are able to produce descendants.

Several crossover variants have been investigated in the past (see e.g. [Dav91] for an overview). Especially the uniform crossover has been proven to be powerful in many applications and can moreover be implemented very efficiently. In our case, uniform crossover can be compared with a mutual exchange of subexpressions between the parental chromosomes:

$$\begin{array}{lcl}
C_1 = (0,\mathbf{0},0,\mathbf{1},\mathbf{2},0,0,1,1,0,0,0,1,0,0,0,\mathbf{0}) & \equiv & ((e-f)+(a-b))+d-c \\
C_2 = (0,\mathbf{1},0,\mathbf{0},\mathbf{0},0,0,0,2,0,0,1,0,0,0,0,\mathbf{1}) & \equiv & (e-f)+(a-(b+c))+d \\
\cline{1-1}\cline{3-3}
C_1' = (0,\mathbf{1},0,\mathbf{0},\mathbf{0},0,0,1,1,0,0,0,1,0,0,0,\mathbf{1}) & \equiv & (a-(b+c))+(d+(e-f)) \\
C_2' = (0,\mathbf{0},0,\mathbf{1},\mathbf{2},0,0,0,2,0,0,1,0,0,0,0,\mathbf{0}) & \equiv & ((d+e)-f)+(a-b))-c
\end{array}$$

In this example, genes 1, 3, 4, and 16 (marked in bold) have been probabilistically selected for crossover which leads to the new chromosomes C_1' and C_2'. Depending on the underlying architecture, C_1' and C_2' may result in better or worse resource exploitation.

4.4.2 Mutation

In contrast to the crossover operator which is necessary for recombining the parental genetic information, mutation aims at introducing new material to an individual meaning that the resulting DFG includes expressions from the gene pool which have not been inherited from the parents. In general, the mutation operator is realized by modifying a probabilistically selected gene, or to be precise, exchanging the current allele by another one. The following example shows chromosome C_2'' as mutation of C_2' at gene position 8, i.e. allele 2 has been replaced by allele 0.

$$C_2'' = (0,0,0,1,2,0,0,0,\mathbf{0},0,0,1,0,0,0,0,0) \quad \equiv \quad ((d+(e-f))+(a-b))-c$$

4.4.3 Architecture specific fitness functions

We have seen that crossover and mutation are appropriate operators for creating new variants of the original DFG. They are distinguished by an efficient implementation and their correctness preserving property and can hence avoid computationally intensive repair mechanisms. In order to determine which individual is selected for recombination, the fitness of each created data-flow graph must be computed first. In order to produce an *exact* estimation of a generated DFG a complete compiler run including instruction selection, instruction scheduling and register allocation must be performed in principle. However, this approach would lead to unacceptable running times of the genetic algorithm which are directly dependent on the basic block lengths, population sizes, and the number of generations. Due to this fact, the fitness functions only take that architecture specific information into account (provided by an external processor specification) which is essential for algebraic optimization. In particular, the number, functionality and interconnectivity of available FUs are considered. Generally, all fitness functions described below are based upon certain scheduling strategies which differ according to the underlying architecture.

- general purpose processors (SISD-architectures):
  A simple ASAP-scheduling with a consideration of execution times of individual operations is sufficient since operations cannot be executed in parallel. The scheduling of operations is performed in a depth first manner in order to minimize data transfers of temporary results.

- VLIW-architectures:
  Since VLIW architectures are distinguished by their capability of performing a set of operations in parallel, the fitness function should be formulated so that the computed schedule is as

close as possible to the result of instruction scheduling (which is part of the code generator). The fitness computation is based upon an extension of the standard list-scheduling algorithm [Bak74]. One essential part of this algorithm is the determination of the ready set containing those operations of the entire DFG which can potentially be performed next on the available FUs. The actual selection and mapping of an operation to a certain FU is controlled by a distinct priority function which, in our case, is based on the longest path heuristic. For the formulation of the detailed architectural features we introduce additional priority functions for the following two classes:

MIMD: In principle, the list scheduling approach described above is sufficient to achieve a high utilization of the function units in each clock cycle and hence to minimize the overall execution time of the given application. Further improvements of the heuristic can be obtained by considering the immediate successors of the ready set: we define the *look-ahead ready set* as set a of operations whose immediate predecessors are contained in the ready set and which can potentially be executed in the subsequent step without data transfers. Concerning the underlying architecture this means that the involved functional units must be directly connected (e.g. $FU2 \longrightarrow FU1$ in fig. 3.2). The priority function prefers those operations of the ready set whose successors are contained in the look-ahead ready set such that data transfers of intermediate results can be avoided.

SIMD: In contrast to MIMD architectures, all functional units perform the same type of operation during the same clock cycle. In principle, this means a large restriction concerning the scheduling of operations which must be adequately formulated in the priority function: During scheduling, the ready set usually consists of operations with different types (e.g. additions, multiplications). In order to maximize the set of operations scheduled in a certain cycle, those operations are selected whose types have the majority within the ready set.

It should be re-emphasized that the purpose of the fitness function is to select a certain individual out of the population and consequently has the largest impact on the quality of the optimization result. Since the fitness function corresponds to the execution time of the compiled source code on the given processor, it should be able to suitably “emulate” the scheduling method of the code generator. The fitness functions described above avoid time intensive compiler runs for the estimation of the individual’s fitness but are able to take all necessary architectural properties into account. Hence, they are a reasonable compromise between the running time of the genetic algorithm and the accuracy of the estimation of individuals.

4.4.4 Preservation of numerical stability

Ensuring the numerical stability of DSP applications is a crucial aspect which must not be neglected during algebraic optimization. Due to the implementation of real values by fix point and floating point numbers, certain transformations can potentially lead to a corruption of computed results caused by e.g. rounding errors and overflows. In order to avoid such variations or at least to keep them within a certain range, a “simulation phase” can be simply integrated into the fitness function: for a given set of input values[2], the transformed basic blocks are evaluated and the computed values are compared to the desired values. If an evaluated value of a certain individual is beyond a given deviation $\varepsilon \geq 0$ a penalty p is added to its fitness value and hence decreases the probability to transmit its genetic material to the next generation.

4.5 Outline of the genetic algorithm

After presenting the genetic operators and the fitness function in the previous sections, we now show how all parts work together in the genetic algorithm.

```
initialize individuals of the population $p$
FOR EACH epoch $e$ DO
    apply transformation rules to the current population $p$
    FOR EACH generation $g$ DO
        compute fitness of all individuals
        select individuals according to their fitness
        create offspring by crossover
        mutate offspring
        replace individuals of the current population by the offspring
        exit loop, if criterion $T_G$ is fulfilled
    END
    terminate, if criterion $T_E$ is fulfilled
END
```

The algorithm presented above is an extension of a standard genetic algorithm [Dav91] by introducing the concept of epochs which are required to introduce new genetic material to the gene pool by applying algebraic transformations to the individuals.

The inner loop represents the basic steps of the standard genetic algorithm, namely *fitness computation* for each individual, *selection* of individuals according to their fitness, recombination by *crossover* and *mutation*, and finally the *replacement* of individuals of the current population by the offspring. The termination criterion T_G is usually defined by maximum number of generations. The outer loop represents a sequence of epochs in which the existing gene pool is expanded in order to increase the number of possible DFG variants. If the termination criterion T_E for the epoch loop is fulfilled, the

[2]These input values are comparable with “test vectors” used for error analysis of integrated circuits.

genetic algorithm stops and generates C-code for the best DFG variant. T_E is either controlled by an epoch counter or fulfilled if the gene pool has reached its maximum size.

Experiments have shown that population sizes of 100–200 individuals and 50–100 generations (dependent of the application) are sufficient to produce high quality code.

Chapter 5

Integration into a compiler framework

The presented optimization method has been implemented in form of a C-to-C transformation tool embedded within a compiler framework shown in fig. 5.1.

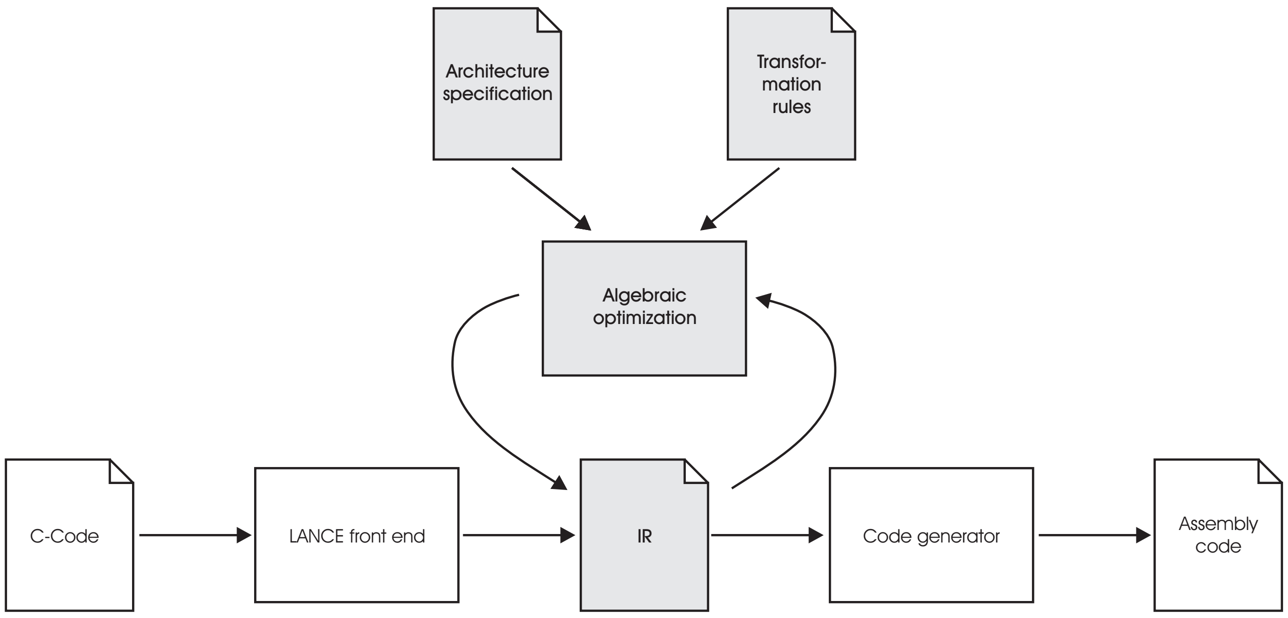

Figure 5.1: Compiler framework

The compiler framework basically consists of a processor independent part, namely the LANCE front end [Lan] which transforms the given C-code into an intermediate representation (IR). The IR serves as an interface between the front end and the processor specific code generator and forms the basis for all optimization steps including standard techniques (dead code elimination, common subexpression elimination, constant folding and propagation etc.) as well as the presented algebraic optimization. Even though the IR itself is in principle processor independent, the external architecture specification still provides the system with the required information about the topology as well as the functionality and number of FUs. The additional transformation rule library enables the designer to specify architecture specific rules among the standard transformations. After algebraic optimization the IR is specifically adapted for the underlying processor for which the code generator is hence able to produce highly efficient assembly code.

Chapter 6

Conclusion

The genetic algorithm approach presented in this report has been proven to be a powerful instrument for algebraic source code optimization especially for VLIW-processors. It has been integrated into a compiler framework and is realized as a C-to-C converter. Since both the transformation rules and the description of the underlying processor architecture are specified in external libraries, the available hardware resources can be exploited very efficiently, leading to high quality machine code. In contrast to the heuristical methods published so far, the presented probabilistical approach is capable of overcoming local optima and is hence able to produce results very close to the optimum. Experiments with our optimization tool have shown that the presented approach can compete with optimization techniques embedded in modern DSP compilers which, however, do not offer the same flexibility as our method.

Bibliography

[Bak74] K. R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[BGS94] 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.

[CPM$^+$95] 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.

[Dav91] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[GGC82] 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.

[HC89] 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.

[HC94] 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.

[HR94] 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.

[IPDP93] 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.

[Jan00] Martin Janssen. Word Level Algebraic Optimization Techniques for Accelerator Data-Paths and Custom Address Generators. *Ph. D. Thesis*, 2000.

[Lan] Lance front end. Available via http://ls12sr.cs.uni-dortmund.de/∼leupers.

[Leu97] R. Leupers. *Retargetable Codegeneration for Digital Signal Processors.* Kluwer Academic Publisher, 1997.

[LM97] B. Landwehr and P. Marwedel. A New Optimization Technique for Improving Resource Exploitation and Critical Path Minimization. *10th International Symposium on System Synthesis*, 1997.

[PD94] 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.

[PR94] M. Potkonjak and J. Rabaey. Optimizing Resource Utilization by Transformations. *IEEE Transactions on CAD, Vol. 13, No. 3,* pages 277–292, 1994.

[Wol88] S. Wolfram. *Mathematica, A System for Doing Mathematics by Computer*. Addison Wesley, 1988.

[ZP96] 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.