FINE GRAIN CODE SYNTHESIS WITHIN A BLOCK DIAGRAM ORIENTED CODE GENERATION ENVIRONMENT

Markus Willems, Matthias Pankert, Sebastian Ritz*
611810 ISS, RWTH, 52056 Aachen, Germany

ABSTRACT

Code generation for a system specified by a block diagram facilitates the fast and efficient evaluation of the design space. As a drawback, automatically generated code includes a certain amount of data management overhead compared to handwritten code, especially when the block diagram includes fine granular structures. Within this paper we present a strategy how to overcome certain types of overhead by introducing a novel code generation approach. While traditional tools are based on a one-to-one correspondence between a block on the block diagram level and a functional kernel on the code synthesis level, now one new functional kernel for a group of blocks is generated automatically. Doing so, a maximum of dataflow information available from the block diagram level is employed to organize the kernel in an efficient way, regarding to the designer's criterion. As a result, reduction in memory consumption and an increased throughput can be achieved jointly.

1. INTRODUCTION

Code synthesis for digital signal processors (DSPs) of a system specified by a block diagram is becoming more and more popular. This is motivated by the possibility provided by advanced tools to evaluate the design space (spanned by throughput, memory consumption, latency) in an efficient way [1,2]. Additionally, the integration of a simulation environment allows to generate a code correct by construction, matching the system design criterions. Flexibility and speed up of the system design have to be paid for by a certain amount of code inefficiency compared to handwritten code. These inefficiencies can be interpreted as a compiler overhead. Within this paper we deal with overcoming specific types of these inefficiencies, which are inherent to the proposed model of code synthesis.

Code generation for systems specified by a block diagram matching the SDF paradigm [3] can be separated into two major task, taking place on different levels of abstraction:

1. Establishing a schedule by extracting the information on the block diagram level (precedence relations, signal rates) by dataflow analysis.
2. Building up the correct functionality in the target language by employing the information available from step 1. In the sequel we will address this as synthesis on the code level

Obviously, the way task 2 is performed influences the scheduling strategies implemented for task 1. Several optimization strategies located on level 1 have been presented in the literature, e.g. [4], [5]. They have all been driven by a specific code synthesis model, and calling different strategies as 'optimum' simply results from differences in the proposed synthesis models.

The outline of this paper is as follows: starting with a brief review on dataflow and its notation, the reference synthesis model on the code level is presented. This leads to an identification of shortcomings inherent to the system, followed by the general idea how to overcome a certain amount of data management overhead by automatically generating more complex functional kernels (section 2). In order to extract as much information from the dataflow description as possible, the necessary dataflow analysis is addressed in section 3. Section 4 covers the transformation of the block diagram information into an intermediate format and the process of kernel generation. Finally, section 5 provides some numbers for a simple block diagram, expressing the potential that comes with the presented approach.

2. THE CODE GENERATION ENVIRONMENT

We suppose that a digital signal processing system is specified by means of a scalable synchronous block diagram $F = (B, S, A, E, D_0)$. A block $b_j \in B$ specifies a signal processing component of arbitrary granularity. The signals $s_i \in S$ specify the data flow between the blocks. The topology of the block diagram is represented by the functions $A()$ and $E()$ defined on the signal set S. $A(s_i)$ is the block producing samples which are written in s_i and $E(s_i)$ is the block consuming samples which are stored in s_i and have been produced by $A(s_i)$. The number of initial samples on a signal s_i are specified by $D_0(s_i)$. According to the synchronous data flow semantics a block $b_j = A(s_i)$ produces $O(s_i)$ samples written into signal s_i and consumes $I(s_k)$ samples from the signals s_k, where $b_j = E(s_k)$. Additionally in the SSDF domain [2] a block may consume and produce any integer multiple $n(b_j)$ of the predefined rates, where $n(b_j)$ denotes the local blocking factor associated with block b_j.

The schedule Φ (the result of task 1) is an activation sequence $\Phi = a_1, a_2, ..., a_i$, where an activation $a_i = (b_k, n(b_k))$ either denotes the activation of block b_k $n(b_k)$ times or the activation of a subschedule Φ_l k_l times (with k_l the looping factor), $a_i = (\Phi_l, k_l)$. The latter describes a hierarchical schedule. Scheduling and hierarchization are based on dataflow analysis, while not considering the block's functionality.

We consider the code synthesis model of fig. 1 as our reference model for task 2 in the sequel. This synthesis model is applied by the Descartes system [2].

*Matthias Pankert is now with Philips Research, P.O. Box 1980, 52021 Aachen, Germany

0-7803-2431-5/95 $4.00

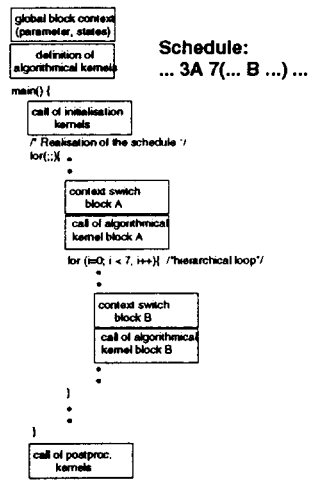

Figure 1. Software structure

Fig. 1 shows a one-to-one correspondence among a block on the block diagram level and a functional kernel on the code level. Signals correspond to FIFOs realized in memory. A local blocking factor is not transferred into a loop but handled by the algorithmical kernel, whereas a looping factor results in an explicit including all kernels corresponding to blocks of the hierarchy level. The time necessary to execute the code can be separated into two categories:
- time necessary to perform the algorithmical computations and
- time necessary for data management operations.
Latter includes context switching as well as storage operations between the execution of different algorithmical kernels. With regard to code efficiency, data management can be seen as an overhead, increasing memory consumption and runtime. Scheduling strategies (task 1) established to increase throughput try to increase the 'algorithm-to-data management time' ratio by sequencing the blocks in a suitable way. In general, this comes with an increase in data memory consumption. Obviously, the efficiency of these scheduling strategies is limited by the kernel's granularity.

The proposed strategy is to increase the kernels granularity by breaking up the one-to-one correspondence among block diagram and code level. Rather we express the functionality of multiple functional kernels of the reference system by one new kernel in our concept. This reduces the number of context switches and data storage operations (since less data exchange via memory is required) and therefore reduces data management overhead.

Another view to this concept is virtually replacing multiple blocks in the block diagram by one new block, representing the new kernel's functionality. There are several reasons why an explicit replacement of a block is inferior to an automatic kernel generation:
- design flexibility is reduced. Functional analysis is required prior to the system specification
- a block representing a very special functionality is not reusable, lacking an important argument for using a code generation environment
- when defining a block's kernel, dataflow information in general is not available. The kernel is not tailored to its dataflow environment.

In this paper, kernels that are candidates for combination are so called fine grain kernels, identified by a low functional complexity, like adders, subtractors, forks, dividers, multipliers, constant sources. This is motivated by the large relative gain that can be achieved while combining complexities that are easy to handle. Fine grain structures occur frequently in system specifications, either to glue certain coarse grain blocks or to build up functionalities that are not represented by coarse grain blocks.

The following questions come with the proposed concept:
a) which kernels are allowed to be combined?
b) which information relevant for the new kernels' synthesis is available from dataflow analysis?
c) how to transfer the original kernel functionalities into the new kernel functionality?
d) which performance gain can be achieved?

3. DATAFLOW ANALYSIS

To visualize the argumentation, a simple example shall be utilized throughout the following sections, presented in fig.2. Here all rates are equal to one except of the output rate of 'SOURCE', producing 10 data samples per execution. Signal 'd' contains two initial data samples.

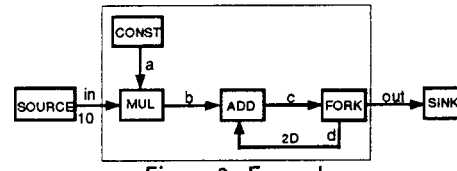

Figure 2. Example

As stated above, on the block diagram level the combination of dedicated functional kernels to one functional kernel corresponds to a hierarchization of the corresponding dedicated subdiagram. Constraints on the hierarchization of a dedicated subdiagram have been identified in [6]. Additionally, if not the complete subdiagram is allowed to be hierarchized, the procedure presented in [6] offers all possible sub-subdiagrams (so called groups) which are allowed to be hierarchized. The kernels corresponding to blocks of a group are allowed to be combined. For the example of fig.2, the kernels of the complete encircled subdiagram of fine grain blocks are candidates for combination. On the block diagram level this corresponds to replacing the complete group by one block b_{new}. This constellation is presented in fig.3.

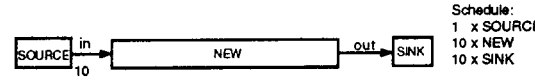

Figure 3. 'new' block diagram

An efficient generation of the kernel represented by b_{new} requires the information on the number of samples $d(s_i)$ available from every input signal s_i of b_{new} at kernel activation time. Therefore, the 'new' block diagram is scheduled (a block diagram level operation), following the guidelines set up by the designer. This 'external' schedule allows to identify $d(s_i) = n(b_{new})I(s_i)$. In the following we assume a single appearance 'external' schedule. Nevertheless, the proposed concept is easily extended to general constellations, exceeding the scope of this paper.

Since for the proposed fine grain constellations only homogeneous blocks are included ($I(s_j) = O(s_k) = 1, \quad \forall s_j, s_k \in S$), $I(s_i)$ is not effected by hierarchization. Notice that all optimization techniques performed on the block diagram level (e.g. [4],[5]) can be applied to the 'external' schedule. The generation of a new kernel simply corresponds to a dedicated hierarchization that has to be

obeyed by the scheduler. As an important consequence, the automatic generation of a new kernel does not destroy the systematic design space evaluation capabilities of the code generation environment.

The 'external' schedule is included in fig.3, indicating that 10 input samples are available at kernel activation time, a vectorization with factor 10 becomes possible. This information might be utilized at kernel generation time.

Dataflow information includes precedence and looping facilities, information crucial for the kernel generation process. Therefore the dataflow information inherent to the group represented by b_{new} has to be extracted as well. This is done by an 'internal' schedule which is optimized with regard to maximum vectorization, motivated by the way of transforming the kernel's functionality into the new functionality (section 4). To achieve maximum vectorization, the external scheduling information might be utilized. Following the principles of scalable static dataflow (SSDF, [2]), $n(b_{new})$ becomes the global blocking factor for this internal schedule. Hence, all relevant dataflow information is included in the internal schedule.

For the example, fig.4 presents the internal schedule, considering the information that 10 samples are available at activation time. The feedback structure results in a hierarchical schedule.

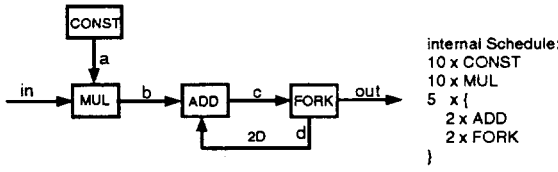

Figure 4. 'internal' schedule

4. TRANSFORMATION TO THE FOREST LEVEL

In order to allow an evaluation of the design space, the kernel shall be synthesized regarding to the designer's criterions. As well, the concept shall be open to extensions of the fine grain alphabet. Therefore an internal format is introduced. This internal format, named the forest level, is an extension of the classical expression tree representation [7], considering additional information coming from the block diagram representation.

4.1. From kernel to expression tree

An expression tree contains a root node, operational nodes and leaf nodes. There is exactly one topological predecessor to a root, a leaf possesses exactly one topological successor. Operator nodes possess exactly one topological successor, a minimum of one and a maximum of two topological predecessors. The limitation on two predecessors avoids problems coming with the precedence of operations. Operational nodes considered in this paper are: ADDITION, MULTIPLICATION, SUBTRACTION, DIVISION. For the transformation, only basic trees are allowed, containing a maximum of one operational node. A basic tree without operational node is denoted an assignment tree.

The transformation into the internal format comes as a combination of block diagram and code level information. Operational nodes are determined by the functionality inherent to the kernel. Each input signal of the corresponding block corresponds to a leaf node, each output signal to a root. The signal type provides the node's type, ensuring equivalence of the original and the transformed representation. A root that corresponds to a signal s_i is a topological predecessor to the leaf that corresponds to the same signal s_i. A block may be transferred to more than one tree (e.g. Fork). A constant source is transferred into an assignment tree, with the leaf a constant leaf.

Some extensions come with additional information available from the block diagram level:

A signal incident to only one block of the group corresponds to an external node, otherwise it is internal. An internal node is a state node if the corresponding signal contains initial data, otherwise it is temporary. A state node contains the information regarding the number of initial data.

The 'internal' vectorization information is provided to the internal format: a local blocking factor $n(b_{int})$ becomes the tree factor $t(T_i)$ for all trees T_i corresponding to block b_{int}. $t(T_i)$ indicates the number of executions of the functionality of T_i prior to the execution of another tree. A hierarchical schedule corresponds to a forest F. F includes all trees $T_1, \ldots, T_n$ corresponding to the blocks of the hierarchical loop. The loop factor k_l is transferred to a forest factor $t(F) = t(T_1, \ldots, T_n)$.

The knowledge of execution precedence inherent to the signals is expressed as well. A tree T_1 containing a root that corresponds to a signal s_i is a predecessor of tree T_2 containing a leaf corresponding to s_i as well. This precedence shall be represented by a solid arrow from T_1 to T_2. Note that execution precedence is a precedence among trees. Nevertheless, if a precedence arrow crosses the border of a forest, there is implicit precedence even among trees and forests. An exception is possible for a signal containing initial data, depending on the number of data consumed by the topological successor. So execution precedence information is a conditional precedence information.

See the transformation of fig.2 into the forest level, presented in fig.5. Dotted arrows represent topological precedence, solid arrows execution precedence.

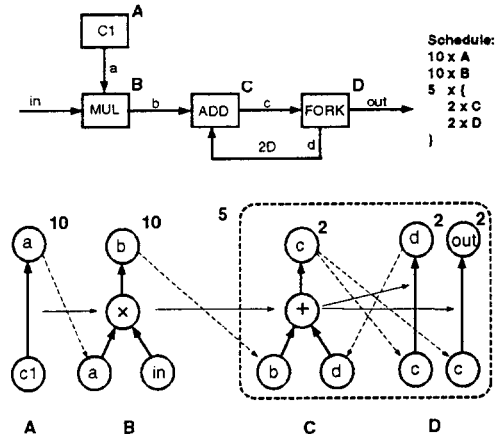

Figure 5. Transformation to the forest level

Notice the effect of signal 'd': Tree 'D' is a predecessor of tree 'C' with regard to execution precedence, since signal 'd' contains two initial samples. Of course, root 'd' of 'D' is a topological predecessor of leaf 'd' of 'C'. No additional effort has to be spent on execution precedence since this is completely determined by the information available from the internal schedule.

4.2. Valid Operations on the Forest Level

Without loosing any information inherent to the forest level representation, several operations are possible:

- Separation of a tree

A tree T with $t(T) > 1$ can be separated in two trees T_1, T_2, with $t(T_1) + t(T_2) = t(T)$
- Construction of a forest
Two trees T_1, T_2 can build a forest if no execution precedence is violated. The forest factor $t(T_1, T_2)$ has to be a common factor of the tree factors $t(T_1)$ and $t(T_2)$.
- Inclusion of a tree T_1 to a forest F
A necessary condition is $t(T_1)/t(F) = c$, with c an integer and no violation of execution precedence.
- Combination of two trees T_1, T_2
Necessary condition: Leaf L_2 of T_2 temporary and only successor of root R_1 of T_1, $t(T_1) = t(T_2)$; T_1, T_2 are allowed to build a forest. Combination by removing R_1 and L_2, making the predecessor of R_1 the predecessor of the successor of L_2 (and vice versa).

Here the impact of maximum vectorization becomes visible, since all types of combination can be achieved from a maximum vectorization. This does not hold the other way since on the forest level there is no information why the precedence relation has been established that way. In other words: one can always move a tree into a forest (if not violating the above constraints), but never move a tree out of a forest.

4.3. Target Independent Optimizations

Since basic trees can be seen as the most general representation of the functionality and independent from a specific code generation backend, all target independent optimizations shall result in these basic trees. All proposed concepts correspond to well known techniques in compiler technology [7].They are illustrated by references to the constellations of fig.5.
- Combination of T_1, T_2, with at least one T_i an assignment tree. This results in reducing the number of trees by one: Combination of 'A' and 'B', resulting in replacing leaf a by a constant leaf c_1.
- Arithmetic optimizations

multiplication with zero: all leaves succeeding the tree become constant leaves with value zero: if $c_1 = 0$, tree 'C' becomes an assignment tree with input d only.

addition of zero, multiplication with one: the contents of the second leaf is transferred to the root, the tree degenerates to an assignment tree: if $c_1 = 1$, tree 'B' becomes an assignment tree with 'in' as its leaf.

4.4. C-code-backend

Each basic expression tree matches a C-expression. A tree factor $t(T) > 1$ is transferred into a loop over the expression resulting from T. A forest factor $f(F) > 1$ results in a loop over the expressions of trees that belong to the forest. The freedom inherent to the forest level transformations becomes obvious here, since it results in different realizations of the same functionality. It allows to synthesize the code with regard to the designer's criterions.

C-expressions allow additional transformations on the forest level:
- more than one operation is allowed, therefore the combination of non-basic trees is allowed: combination of 'AB' with 'C', removing variable b. This implies to include 'AB' into the forest first, resulting in a tree factor of 2 (10/5)
- assigning a result to more than one variable. Therefore trees containing more than one root are allowed: combination of 'ABC' and both trees of 'D', resulting in removing c and assigning two roots 'd' and 'out', therefore only one tree remains.

5. RESULTS

Four different C-code versions for the system specified in fig.2 are compared with regard to throughput, data memory and program memory consumption. The first version is the classical approach with a one-to-one correspondence between block diagram and code level, with the functional kernels activated by function calls. The second version is classical as well, with the kernels inlined. The third and fourth version follow the new strategy. Here the third version results from an immediate C-code generation from the forest level without C-specific transformations with 'AB' outside the forest. The fourth version finally considers all C-specific transformations, resulting in one expression only. The state information included in state nodes (for the example: nodes 'd') result in additional instructions . The code has been profiled on a TMS320C40.

code version	throughput (Ksamp/s)	program mem. (words)	data mem. (words)
function call	129	271	24
inlined	156	157	24
basic tree	321	127	20
C-transform.	452	120	20

The improvement even for a simple constellation that is supposed to be best suited for inlining the functional kernels results in a gain in both, throughput (290%) and memory consumption (-24% program, -17% data memory), always compared to the best 'traditional' version.

6. CONCLUSION

We have presented a systematic approach of how to overcome a significant amount of data management overhead inherent to code automatically generated from a block diagram system description. Effort has been put on fine grain structures since these come with low functional complexity and large data management overhead. A procedure how to extract the dataflow specific information has been presented, as well as how to employ this information for a system specific code generation. Even simple examples show the potential of the proposed strategy, expanding the design space by reducing memory requirements while enhancing the throughput

REFERENCES

[1] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: A platform for heterogenous simulation and prototyping," in *Proc. 1991 European Simulation Conf.*, (Copenhagen, Denmark), June 1991.

[2] S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyr, "High level software synthesis for the design of communication sytems," *IEEE Journal on Selected Areas in Communications*, pp. 348 -358, Apr. 1993.

[3] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. of the IEEE*, vol. 75, pp. 1235–1245, September 1987.

[4] S. Ritz, M. Pankert, V. Živojnović, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Intl. Conf. on Application-Specific Array Processors*, pp. 285–296, Prentice Hall, IEEE Computer Society, 1993.

[5] S. Bhattacharyya and E. Lee, "Scheduling synchronous dataflow graphs for efficient looping," *Journal of VLSI Processing, vol. 6*, pp. 271 - 288, Dec. 1993.

[6] M. Willems, "Hierarchisation of Dedicated Subdiagrams," *Internal Report, Aachen University of Technology*, 1994.

[7] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.