Modeling Data Flow and Control Flow for High Level Memory Management *

Michael F.X.B. van Swaaij       Frank H.M. Franssen       Francky V.M. Catthoor
Hugo J. De Man †
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

The goal of this paper is to advocate a control flow independent modeling of data flow in applicative algorithm specifications. The model is utilized in the synthesis of ASIC architectures for real-time signal processing applications. It allows for a generalization of control flow transformations which are used to optimize the memory organization at an early stage in the synthesis trajectory. Arguments supporting the inherent amenity of this type of model for use in efficacious memory management optimization schemes will be adduced. A CAD tool is reported which extracts all information related to the model from an applicative algorithm description. Its use is demonstrated on a real-life test vehicle.

1 Introduction

In real-time signal processing systems large quantities of data are processed. These data are most often expressed as multi-dimensional signals. Processing multi-dimensional signals in real-time poses not only computational problems but also storage problems. In studies of memory management methods [2, 14] it has been recognized that the efficiency of a chosen storage scheme is mainly dependent on the relative ordering of the execution of computations, which from now will be called ordering of computations. We believe that the task of organizing the multi-dimensional signals in background memories is crucial in the synthesis of efficient ASIC architectures. Therefore, it should precede data path synthesis. The task of high-level memory management (HLMM) in time multiplexed architecture synthesis is to devise an efficient scheme for multi-dimensional signal storage and retrieval. It will only produce constraints on the ordering of computations related to multi-dimensional signals. This is different from schedulers which fix absolute cycle instances [4, 10]. The PRIDEO compiler [6, 7] forms an exception to the mentioned scheduling approaches because it uses the concept of 1-dimensional data streams. In this paper, a multi-dimensional data model is proposed which is more general than the data stream model, leading to less restrictions on the ordering of computations.

The term data flow is defined here as the combination of operations and dependencies which form the algorithm. Control flow is defined as a (partial) ordering of computations meeting the restrictions of their dependencies. None of the possible control flows should be excluded in advance from the search space in the memory management task, except those incoherent with a given data flow. In other words, the syntactical structure of an algorithm description should not be used to limit the set of possible control flows. Nevertheless, this structure is often used in architecture synthesis methods to derive a control flow, even for non-procedurally described algorithms. In fact, this structure is still often regarded as a part of the algorithm specification which has to be implemented instead of seeing it for what it is: a syntactical convenient way to describe a data flow.

Recently, methods have been proposed to alter loop structures in algorithm descriptions (e.g. [7, 13, 14]) within the context of ASIC architecture synthesis. The used transformations are based on those applied in optimizing software compilers [9, 11]. It has been found difficult to formalize these transformations in terms of the global optimization criteria to be used. The reason for the difficulties in the formalization of transformations, leading to these suboptimal methods, is the following. By deriving control flow directly from the syntactical structure of an algorithm description, this structure determines both data flow and control flow. The global relation between syntactical structure, data flow and control flow can be quite intricate. Therefore, if local transformations are expressed in terms of syntactical structure then both the data flow directed constraints on these transformations as well as the quantitative effects of them on implementation aspects, like storage efficiency and potential parallelism, will be hard to extract.

These problems motivate control flow independent
data modeling (section 2). The data flow is given solely in terms of operations consuming and producing signal instances and not in terms of loop structures and iterators. A signal instance is a single signal of a multi-dimensional signal. Control flow is defined in terms of placement of the operations in a common node space and an ordering vector defined over that space. Control flow transformations can then be easily expressed as a re-indexing (repositioning) of operations in the common node space and/or as a change in the ordering vector. The applicability of the proposed modeling method in HLMM is motivated in section 3. Using this model, control flow transformations can be formalized, thereby generalizing traditional loop transformations. As a result, control flow optimization techniques can be devised which are more general, controllable and efficacious than those based on syntactical structure transformations. We believe that this novel approach will change the way synthesis researchers can incorporate the effect of control flow transformations and is a real step forward in coping with the complexity of the exploration of control flow alterations. In section 4, a CAD tool is reported which extracts all information related to the model from an applicative algorithm description. Its use is demonstrated on a real-life test vehicle in section 5.

2 Modeling data flow and control flow

Given an initial Signal Flow Graph (SFG) [3], atomic operations are extracted by clustering SFG operations into groups. Each atomic operation is considered a single multiple-input/output operation. A typical clustering is for example a sequence of a shift operation and an add operation. It should be noted that the proposed model only supports worst case linear approximations of data dependent or non-linear dependencies. It will be extended in the near future to support extra information on e.g. data dependent dependencies [2]. The data/control flow model defines per atomic operation (to be called operations from now on) three different algorithmic spaces (index spaces). Within each space, polytopes are used to encapsulate sets of grid points [8]. Each different space holds different information.

- The Node space of a SFG-node defines the number of signal instance definitions belonging to all operations in its polytope.
- The Operation space of a signal defines which operation using which operands produces which signal instance.
- The Operand space of a signal defines which operation consumes which signal instance as an operand.

Figure 1: Relations between the various entities in the signal dependency model.

The placement of operations in the operation/operand spaces defines the data flow and fully determines the production and consumption of signal instances. In figure 1, the relation between the various spaces is given by means of a small example for a single SFG node. The relation between the polytopes of different spaces, in which w and v are vectors, is given by affine mappings of the form w = Aw + b (shown as [A|b] in figure 1), with A a matrix and b a column vector derived from the indices of the multi-dimensional signals. Suppose we label the SFG node in which signal a is defined as 'x', and the SFG node in which signal b is defined as 'y'. The function f_{ax}(w) = A_{ax}w + b_{ax} links the node space 'x' to the operand space of b, see figure 2. Therefore, it indicates which operation in node space 'x' consumes which signal instance of b. The function f_{yb}(w) = A_{yb}w + b_{yb} links the node space 'y' to the operation space of b. Therefore, it indicates which operation in node space 'y' produces which signal instance of b. Given the fact that the link between an operand space and an operation space of the same signal is always the identity function, this leads to the following relation between node space 'x' and node space 'y': w = A_{yb}^{-1}A_{ax}w + b_{ax}b_{ax}^{-1}A_{yb}b_{yb}. In this way dependencies can be found in terms of complete domains between different node spaces, instead of single computations. This is a prerequisite for efficiently handling large multi-dimensional signals.

Dependencies between operations can always be extracted whether or not the computations have been defined in the same node space [12].

3 Control flow transformations for high-level memory management

The goal of HLMM is to minimise the total circuit area, which is related to multi-dimensional signal storage, under a timing constraint. The total cost related to storage is dependent on all of the following
cesses and complexity of address calculations. In this section, examples are used to explain the use of the proposed model and its link with HLMM. For clarity, two different types of algorithmic description are used in this text.

- An applicative description, i.e. implying no ordering of operations, except for those inferred by the data dependencies. This type of description is a syntactical short hand notation for specifying nothing more than a dependency graph. The iteration syntax is defined by: '(index-name: range ) ::' and a unit step size (similar to the SILAGE-language style [2]).

- A procedural description, i.e. implying a sequential ordering of operations. The iteration syntax is defined as: 'for index-name = range do .. od' and a unit step size (similar to the FORTRAN style).

Example 1

\( k : 1..n :: \)
\( b(k) = g(c(k)); \)
\( l : 1..n :: \)
\( a(l) = f(b(l)); \)

An operation ordering is defined here as an ordering of computations of signal instances. Suppose that the operation ordering is such that it corresponds to a \textit{procedural execution} of the given algorithm. This creates a \textit{global relative ordering}, not an absolute schedule in terms of cycles, of the computations of the signal instances as follows: \( T(b(1)) < \ldots < T(b(n)) < T(a(1)) < \ldots < T(a(n)) \). The function \( T() \) assigns the computation of a signal instance to a point in the ordering of all computations.

In example 1, the computations are defined over two distinct node spaces. In each node space, individual operations are distributed according to the indices used in the SFG node. For example, the computation defining \( a(3) \) is placed at point (3) in its node space which is set up by \( i \). Coincidently, the placement of this operation in the operation space of \( a() \) is also at point (3). But note that the base vectors of the operation space correspond to the index dimensions of the signal. We can now express a procedural ordering of the computations of signal instances \( a(i) \) by a linear function \( (e.g. t_1(l) = l) \) on the points at which the operations are located in the node space.

A \textit{global} relative ordering can be expressed by placing all individual computations of signal instances in a single common node space and defining a linear ordering function. Figure 3 shows just two of many possible ways to do this. The arrows in this figure represent the dependencies. In figure 3a, all signal instances are placed within a common 1-dimensional node space 's1' with ordering function \( t_1(e) = e \). An alternative placement is given in figure 3b. For this placement, signal instance \( a(3) \) is defined by an operation placed in the common node space 's2' at position (3, 1). The ordering function \( t_2(i, j) = i + n j \) would lead to the same ordering as indicated by procedural interpretation. The line \( i + n j = c \) in the figure represents the \textit{equi-ordering} line, i.e. operations on this line are ordered at the same ordering point. The order increases in the direction perpendicular to this line, as indicated by the dashed arrow, called the \textit{ordering vector}, denoted by a row vector \( \Pi \). It is clear that for a given placement there exist only a small set of relevant linear orderings. An affine function could be used, but the additional constant is irrelevant since the function determines an ordering and not a schedule. The operations (or signal assignments) on two points \( p, q \) in the common node space have an execution ordering given by \( \Pi p \) and \( \Pi q \). If \( \Pi p < \Pi q \) then \( p \) will be executed before \( q \). The constraints on \( \Pi \) are the same as used for regular array synthesis [12]: if there is a dependency \( d_{pq} \) between point \( p \) and \( q \) then \( \Pi \) must be chosen such that \( \Pi d_{pq} \geq 0 \). The dependencies between operations can be readily extracted from the model. Recall that in the proposed model all signal instances within a
single polytope are placed in a common node space by the same affine placement function. Therefore, the use of a linear ordering function poses a certain restriction on the possible orderings of computations. However, we have experienced that for practical cases this is not a significant restriction.

All classical loop transformations (loop merging, folding, splitting, etc) can be described by the model. Loop merging will be used here as an example. Loop merging is performed by placing the operations of both nodes in a common node space. How such a placement is to be done (constraints and criteria) is explained in [12]. The resulting placement is identical to the one in figure 3b. The importance of this placement is that a linear ordering function can be found by selecting a different ordering vector, see figure 4, which requires only 1 signal instance to be stored between any two computations instead of $n$ signal instances for $I_1 = (1, n)$, see figure 3b. Thus, by selecting a different ordering vector the storage size has been reduced. Furthermore, the single storage location can be assigned to a foreground register instead of a background memory. As a result of this access distribution, the background memory bandwidth has also been reduced. A linear address scheme can be directly deduced from the selected ordering function.

In general, a specific ordering between operations of various signals is equivalent to a placement of the signals in a common node space together with the definition of a linear ordering function in this space. Ways to tackle this general problem have already been investigated for regular array synthesis [12]. We believe that many of the techniques developed there can be used to optimise control flow for other synthesis tasks; in particular memory management for time multiplexed architectures.

4 A CAD tool for model extraction

The extraction of the information related to the proposed model is supported by a CAD tool. The techniques used in the core routines of this tool are mainly based on results from polyhedral theory, as described in e.g. [8], and on conventional graph theory.

![Diagram](image)

Figure 4: Loop merging represented as a change in ordering vector.

<table>
<thead>
<tr>
<th>Test vehicle</th>
<th>Signal inst.</th>
<th>n-dim. signals</th>
<th>Node spaces</th>
<th>Polytopes</th>
<th>CPU time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRD</td>
<td>±136M</td>
<td>55</td>
<td>69</td>
<td>227</td>
<td>20</td>
</tr>
<tr>
<td>CRD-out</td>
<td>±68M</td>
<td>26</td>
<td>20</td>
<td>66</td>
<td>4.6</td>
</tr>
<tr>
<td>ex. 2</td>
<td>±9M</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Lev-Dur</td>
<td>±0.38M</td>
<td>4</td>
<td>11</td>
<td>17</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Performance of model extraction tool.

CPU complexity is linear in the number of dependencies, but exponential in the dimensionality of the node spaces. However, in most real-life signal processing algorithms this dimensionality is low enough to ensure fast model extraction. Table 1 presents some performance results of the implemented model extraction tool for a DECstation 3100. The first test vehicle is a complete version of a real-life image contour regularity detector (CRD) algorithm [5]. This test vehicle has been chosen to demonstrate that even for the complete algorithm the model extraction requires little CPU time. However, techniques are available to prune the original SFG, leading to a substantial complexity reduction [2]. The second test vehicle corresponds to the output of this pruning task in HLMM. The third test vehicle is the simplification of the CRD algorithm (example 2), which will be used as an example in the following section. The core of the Levinson-Durbin algorithm [14] is the fourth test vehicle.

5 Control flow transformations in practice

In this section, the theory of control flow modeling and transformation, as outlined in this paper, will be applied in the design of a memory management scheme for the simplified CRD algorithm (example 2).

Example 2

```
func main(tangent: word[4096]) path: bool[] =
begin
  (i : 0..511);
  (j : 0..511);
begin
  (k : 0..63) :: diff[i][j][k] = f1(tangent[], i, j, k);
  mean[i][j] = f2(diff[i][j][]);
end;
  (i : 0..511);
  (j : 0..511);
  grid_diff[i][j] = f3(diff[i][j][], mean[i][j]);
  (i : 0..511);
  (j : 0..511);
  path[i][j] = f4(path[i-1][j-1], grid_diff[i][j]);
end;
```

The input of the algorithm is a multi-dimensional signal tangent with 4096 signal instances. This large
Figure 6: Polytope placement with the control-flow-equivalent procedural code for ordering vector $\Pi = (-34237, 34304, 1)$ in the $i,j,z$ space.

The result of the placement, which is in fact a series of affine polyhedral transformations, can be found in figure 6. In this figure, the polytopes, but not their placement, correspond to those extracted from the initial applicative code. Their dependencies are indicated for a single $i,j$-pair by arrows. The bold arrows indicate the dependencies on 64 signal instances $\text{diff}[i][j]$. It is clear from the way the polytopes have been placed that minimizing dependency lengths is an important placement criterion. Notice the dependencies within the polytope of path which are the only ones which are not parallel to the $z$-axis.

The selection of an ordering vector is done by successively adding constraints to the ordering vector. Each constraint has the form of a linear equation, set up by a vector inner product $\Pi v = c$, in which $v$ denotes a direction corresponding to a preferred ordering of operations. The constant $c$ is a result of the previously given constraints and the placement of the polytopes. The optimal ordering vector for the placement depicted in figure 6, in terms of minimizing memory size requirements, is determined by 3 linearly independent equations: $\Pi (0 0 1)^T = 1$, $\Pi (1 1 0)^T = 64 + 3$ and $\Pi (0 1 0)^T = 512 * 67$. The unique solution is $\Pi = (-34237, 34304, 1)$. A procedural code with a control flow equivalent to this placement and ordering vector is also given in figure 6. Notice the differences in loop structure and indexing of signals compared to the applicative code of figure 5. It is difficult to fully describe the applied polyhedral transformations in terms of classic loop transformations. It is clear that some
sort of loop merging has occurred, but the re-indexing of signals in this example is not covered by any of those loop transformations. In the graph of figure 5, each arc carries two numbers. The numbers in the graph indicate memory requirements in terms of the number of signal instances. Those above the dashed lines apply when the code of example 2 is interpreted procedurally. Those below the dashed lines correspond to the optimized control flow. Note the extraordinary reduction in total memory requirements from 8.5M to 67 signal instances.

6 Conclusions

In the design of ASIC's for real-time signal processing systems the efficient management of signal storage and retrieval is a requisite for a cost effective realization. It has been indicated that control flow transformations are instrumental in designing efficient memory management schemes. For this purpose a data flow and control flow model has been presented with the following properties:

- The data flow model allows control flow selection independent from the syntactical structure of an algorithm description.
- Control flow alternatives are expressed as a combination of operation placement in a common node space and the selection of a linear ordering function in this space.
- Amenity to a more formal and general approach to control flow optimization than conventional syntactical structure transformations.

It has been shown that conventional loop transformations can be expressed in terms of the proposed model. Effective CAD techniques to extract the model and to place operations in a common node space have already been developed [12]. These techniques were initially developed to minimize processor interconnection complexity in processor array architectures, but are applied here to minimize background memory bandwidths. The complexity of the extraction task is very low and CPU times for real-life test vehicles show the feasibility of using this model in practice. Although the complete control flow of an algorithm description can be optimized by using this model, it is clear that within such a description many special data flow constructs can be identified for which a known optimal control flow can be chosen directly [1,2]. By pruning the data flow in such a way, the complexity of the remaining control flow optimization problem can be significantly reduced. After such a pre-processing step, a formal and general optimization of the control flow can be performed by use of the new model and a set of appropriate optimization techniques.

References