In Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, pages 341-344, Rennes, France, July 2010. DOI: 10.1109/ASAP.2010.5540954.

LOOP TRANSFORMATIONS FOR INTERFACE-BASED HIERARCHIES IN SDF GRAPHS

Jonathan Piat[1]*, Shuvra S. Bhattacharyya*[2]*, and Mickael Raulet*[1]

(1) IETR/INSA, UMR CNRS 6164
Image and Remote Sensing laboratory, F-35043 Rennes, France
email: {jonathan.piat@insa-rennes.fr, mickael.raulet@insa-rennes.fr}
(2)Department of Electrical and Computer Engineering,
University of Maryland, College Park, MD, 20742, USA
email: {ssb@umd.edu}

ABSTRACT

Data-flow has proven to be an attractive computation model for programming digital signal processing (DSP) applications. A restricted version of data-flow, termed synchronous data-flow (SDF), offers strong compile-time predictability properties, but has limited expressive power. A new type of hierarchy (Interface-based SDF) has been proposed allowing more expressivity while maintaining its predictability. One of the main problems with this hierarchical SDF model is the lack of trade-off between parallelism and network clustering. This paper presents a systematic method for applying an important class of loop transformation techniques in the context of interface-based SDF semantics.The resulting approach provides novel capabilities for integrating parallelism extraction properties of the targeted loop transformations with the useful modeling, analysis, and code reuse properties provided by SDF.

Index Terms— Data-Flow programming, SDF graph, Scheduling, Code Generation, Loop parallelization.

1. INTRODUCTION

Since applications such as video coding/decoding or digital communications with advanced features are becoming more complex, the need for computational power is rapidly increasing. In order to satisfy software requirements, the use of parallel architecture is a common answer. To reduce the software development effort for such architectures, it is necessary to provide the programmer with efficient tools capable of automatically solving communications and software partitioning/scheduling concerns. Most tools such as PeaCE [1], SynDEx [2] or PREESM [3] use as an entry point a model of the application associated to a model of the architecture. Data-flow model is indeed a natural representation for data-oriented applications since it represents data dependencies between the operations allowing to extract parallelism. In this model, the application is described as a graph in which nodes represent computations and edges carry the stream of data-tokens between operations. The Synchronous Data-Flow (SDF) model allows to specify the number of tokens produced/consumed on each outgoing/incoming edge for one firing of a node. Edges can also carry initialization tokens, called delay. That information allows to perform analysis on the graph to determine whether or not the graph is schedule-able, and if so to determine an execution order of the nodes and application's memory requirements.

In basic SDF representation, hierarchy is used either as a way to represent cluster of nodes in the SDF graph or as parameterized sub-system [4]. In order to extend the expressivity of the SDF model, we propose a new hierarchy type more detailed in [5] based on interfaces. This new representation allows the designer to describe sub-graphs in a top down approach, thus adding relevant information for later optimizations. In this paper, we introduce optimization techniques for this particular model based on regular loop transformations. This transformation allows to extract a given level of parallelism from the hierarchy while maintaining an average level of clustering.

2. INTERFACE-BASED SDF HIERARCHY

While designing an application, user might want to use hierarchy in a way to design independent graphs that can be instantiated in any design. From a programmer view it behaves as closures since it defines limits for a portion of an application. This kind of hierarchy must ensure that while a graph is instantiated, its behavior might not be modified by its parent graph, and that its behavior might not introduce deadlock in its parent graph. In order to allow the user to hierarchically design a graph, this hierarchy semantic must ensure that the composed graph will have no deadlock if every level of hierarchy is independently deadlock free. To ensure this rule we must integrate special nodes in the model that restrict the hierarchy semantic. In [5] we have introduced a new model of hierarchy for SDF graphs. This model is based on spe-

cial interface behavior that ensure every hierarchy level to be independent from the scheduling point of view.

3. NESTED LOOPS PARTITIONING BY ITERATION DOMAIN PROJECTION

Definition A nested loop of depth n is a structure composed of n nested loop for which each loop, excluded the n^{th} one, contains only a loop.

```
for i1 := l1 to u1 do
 for i2 := l2(i1) to u2(i1) do
  ...
  for in := ln(i1, i2, ..., in−1) to un(i1, i2, ..., in−1) do
   {Instruction1}
   ...
   {Instructionk}
  end
 end
end
```

Fig. 1. Nested loop example.

The iteration domain of the outer loop remains constant while the iteration domain of inner loops consists in maxima and minima of several affine functions.

This nested loop partitioning technique was developed as a method for systolic array synthesis in [6]. A systolic array is massively parallel computing network. This network is composed of a set of cells locally connected to their spatial neighbors. All the cells are synchronous to a unique clock. For each clock cycle, a cell takes data from its incoming edges, perform a computation and output data to its outgoing cells. This partitioning technique aims at finding a projection vector by analyzing the distance vector of a nested loop of depth N. When this projection vector has been determined, the iteration domain is projected along this vector resulting in an $N-1$ dimension systolic array.

4. APPLYING LOOP OPTIMIZATION TECHNIQUES TO INTERFACE-BASED HIERARCHY

Loop partitioning technique described in previous section reveals the parallelism nested into the loops by using basic linear algebra and gives a set of results. As seen previously, interface-based hierarchy suffers from a lack of parallelism. All the embedded parallelism remains unavailable for the scheduler, making an application hard to optimize on a parallel architecture. Interface-based hierarchy being close to code nesting, it seems appropriate to tap into nested loops partitioning techniques to extract parallelism. The nested loops code structure could be defined as follow in the Interface-based Synchronous Data-Flow model :

Definition A nested loop of depth n is a structure composed of n nested hierarchical actor with a repetition factor greater than one, for which each actor, excluded the n^{th} one, contains only one actor.

In order to exploit this optimization technique we must be able to extract the distance vector from the hierarchical description, thus allowing to have a relevant representation for the partitioning. Then having the different projection vector and their respective resulting execution domain, we must be able to map back this representation into a SDF graph.

4.1. Distance vector extraction from interface-based SDF

The Synchronous Data-flow paradigm brings some limitation to the representation.

- In the data-flow paradigm, actors produce tokens that can then be consumed. A data-flow representation cannot contain other dependencies than the flow dependency.
- In the SDF paradigm all the data are represented by edges. Thus all the data of a network are considered disjoint.
- In the SDF model, data are uni-dimensional and atomic (token). It means that you cannot have multi-dimensional access to a data.

The third limitation shows that, the basic SDF representation does not allow to extract distance vector. The hierarchical SDF allows factorized representation and therefore allows to represent edges as multi-dimensional data over the iteration domain. As data are being disjoint, only recursive edge ($source(e) = sink(e)$) can carry an inter iteration dependency. It means that the analyze only have to be carried out on this specific kind of edge. For our purpose, we will consider a recursive edge as an array of size $q(source(e)) + d(e)$.

Given a vertex a with $q(a) > 1$ and a recursive edge e_0 with $source(e_0) = target(e_0) = a$ and $d(e_0) > c(e_0)$. The index vector for the read accesses to the data carried by e_0 is $\vec{r} = [i_0 - d(e_0)]$, and the index vector for the write accesses to the data carried by e_0 is $\vec{w} = [i_0]$. Thus the distance vector between the iteration of a is $\vec{\tau} = \vec{w} - \vec{r} = [d(e_0)]$.

Let us now consider that a is a hierarchical actor that contains one actor b with $q(b) > 1$ and a recursive edge e_1 with $source(e_1) = target(e_1) = b$ and $d(e_1) > c(e_1)$. Given that edge has a local scope in a hierarchical representation, the data carried by e_1 can be represented as an array of size $(q(source(e_1))) + d(e_1)$ itself contained in an array of size $q(a)$. Thus the index vector for the read accesses to the data carried by e_1 is $\vec{r} = [i_0, i_1 - d(e_1)]$, and the index vector for the write accesses to the data carried by e_1 is $\vec{w} = [i_0, i_1]$. Thus the distance vector between iteration of b is $\vec{\tau} = \vec{w} - \vec{r} = [0, d(e_1)]$.

By extension the distance vector for a recursive edge at the N^{th} loop of a nested loops structure is a vector of size N with the $(N-1)^{th}$ element being $d(e_{N-1})$.

4.2. SDF network synthesis using analysis results

The network of computing element resulting from the projection is itself an SDF graph. Using information given by the allocation vector we can determine the points of the execution domains computed by each cell and consequently distribute the input data among the cells using explode and broadcast vertices. The output data can also be sorted out using implode vertices and circular buffers.

The SDF graph then needs to be timed using delay to ensure a proper execution of strongly connected components. In a systolic array all the cell are active synchronously. Thus in order to synchronize the computation on the cell network, the communication channel must consist in a register whose size allows to synchronize the computation. In the SDF paradigm, computations are synchronized by data, and actors are not triggered synchronously but sequentially if they share data. Thus if the resulting network contains strongly connected components, delays have to be added in order to time the graph. A proper execution guarantees that the last data available on a communication link is the valid one for the execution of the sub-graph.

5. THE MATRIX VECTOR PRODUCT EXAMPLE

In this section we will use the matrix vector product as a test case for the method described above.

Given a vector V and matrix M, the product $V \times M = R$ can be described using a set of recurrent equations.

$$\begin{cases} R_{i,k} = 0 & if \quad k = 0 \\ R_{i,k} = R_{i,k-1} + v_i m_{i,k} & if \quad 1 \leq k \leq N \\ r_i = R_{i,N} & 0 \end{cases}$$

The SDF representation extracted from those recurrent equations exposes two level of hierarchy . The first hierarchy level contains a $vector \times scalar$ product, and the second hierarchy level represents a $scalar \times scalar$ product.

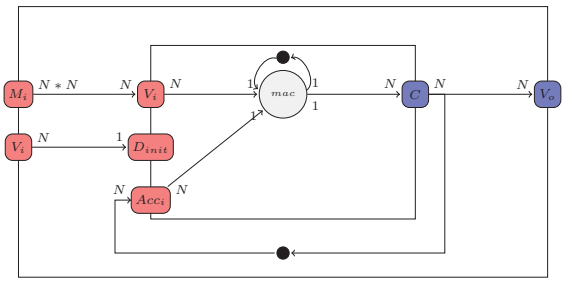

Fig. 2. Matrix vector product

5.1. Network description

The matrix vector product networks, takes a $N \times N$ matrix and a N vector as input, and outputs the result as a N vector. The M_i port is the matrix input and the V_i is the vector input. The V_o port is the vector output. The $vect_{scal}$ vertex takes two input, V_i being a line of the matrix, and D_{init} being an element of the vector. The element in D_{init} initialize the delay token on the recursive edge around the mac operation. The Acc_i port takes the vector in which the result is accumulated. The mac operation takes two scalar, one from the the matrix line one from the delay (being an element of the input vector), multiply them and adds the result with the input accumulating vector. The valid schedule for the graph is then:

$N \times \{N \times mac\}$

The schedule take advantage of the special behavior of the port V_o, which behaves as a ring buffer of size N. Thus the data contained in V_o at the end of the schedule, is the result of the last N^{th} iteration of the mac operation, that is the valid result.

5.2. Distance vector extraction

The index vector for the read operation on the top recursive edge is $\vec{r_o} = [i_0 - 1, i_1]$, and the index vector for the write operation on the top recursive edge is $\vec{w_o} = [i_0, i_1]$. Thus the distance vector is $\vec{\tau_0} = \vec{w_o} - \vec{r_o} = [1, 0]$. The index vector for the read operation on the inner recursive edge is $\vec{r_1} = [i_0, i_1 - 1]$, and the index vector for the write operation on the inner recursive edge is $\vec{w_1} = [i_0, i_1]$. Thus the distance vector is $\vec{\tau_1} = \vec{w_1} - \vec{r_1} = [0, 1]$.

Using Lamport's method [7] we can determine that the time vector minimizing parallel execution time for this application is $\tau = [1, 1]$. Based on this time vector, a set of projection vector can be determined :

$$s_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} s_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} s_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

The following analysis will be carried out using the projection vector s_3. The uni-modular matrix S_3 is

$$S_3 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

$$S_3^{-1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$$

The allocation function is then $A_3 = [-1, 1]$. Using this allocation function we can now determine how the points of the computation domain are allocated onto the execution domain. The extremes of the allocation function in the iteration domain are $-N$ and N meaning that the execution domain is of size $(2 \times N) - 1$. The end of the analysis will be performed with $N = 3$.

5.3. Network synthesis

N being three, the resulting network is composed of 5 vertices. Using the allocation function we can determine the point of the iteration domain that will be computed by each vertex.

- Vertex 0: computes the point $[1, 3]$
- Vertex 1: computes the points $[1, 2]$ and $[2, 3]$
- Vertex 2: computes the points $[1, 1]$, $[2, 2]$ $[3, 3]$
- Vertex 3: computes the points $[2, 1]$ and $[3, 2]$
- Vertex 4: computes the point $[3, 1]$

Computing the topology matrix of the network shows that the repetition factor for each of the actor is 3, as the computation load must be balanced in an SDF. Thus vertices $0, 1, 3, 4$, will compute points outside of the iteration domain. This means that we must consequently time the graph to get sure that the valid data, will be the last produced data. For the first strongly connected set $\{V_0, V_1\}$, the hyperplane containing the point $[1, 3]$ as a shorter distance to the hyplerplane containing $[0, 0]$, than the hyperplane containing $[2, 3]$. This means that V_0 must be scheduled before V_1. To consequently time the network we must add a delay on the arc going from V_1 to V_0. Timing all the strongly connected sets that way leads to a translation of the iteration domain for the vertices $0, 1, 3, 4$.

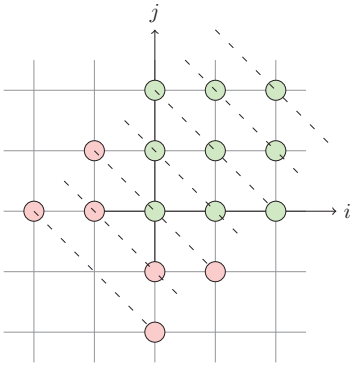

Fig. 3. Iteration domain after graph timing

The resulting timed network needs to be connected to input and output ports. The original hierarchical representation had no degree of parallelism, but the resulting representation after transformation reveals five degree out of nine available for the flat representation. The other available projection vectors would give less parallelism, with more regularity in the computation as the activity rate of cells would be homogeneous over the network.

6. CONCLUSION AND FUTURE WORK

The optimization technique described in this paper helps at improving the degree of potential parallelism in the application while keeping the network size at a low level. For large iteration domain, and when targeting architecture with a low level or parallelism, this optimization does not in general allow one to keep the network size optimized in relation to the architecture. Nevertheless the distance vector extraction can lead to further optimization, using technique such as the one described in [8]. This paper shows that loop optimization method inherited from various computing environment can be used in the Synchronous Data-Flow and give relevant results.

7. REFERENCES

[1] W. Sung, M. Oh, C. Im, and S. Ha, "Demonstration Of Codesign Workflow In PeaCE," in *in Proc. of International Conference of VLSI Circuit, Seoul, Koera*, 1997.

[2] T. Grandpierre and Y. Sorel, "From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[3] J. Piat, M. Raulet, M. Pelcat, P. Mu, and O. Déforges, "An extensible framework for fast prototyping of multiprocessor dataflow applications," in *IDT08: Proceedings of the 3rd International Design and Test Workshop*, Monastir, Tunisia, december 2008.

[4] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001.

[5] J. Piat, S. S. Bhattacharyya, and M. Raulet, "Interface-based hierarchy for Synchronous Data-Flow Graphs," *in Signal Processing Systems (SiPS)*, 2009.

[6] D.I. Moldovan and J.A.B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *Computers, IEEE Transactions on*, vol. C-35, no. 1, pp. 1–12, Jan. 1986.

[7] L. Lamport, "The parallel execution of DO loops," *Communications of the ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[8] P. Boulet, A. Darte, T. Risset, and Y. Robert, "(Pen)-Ultimate Tiling ?," *The VLSI Journal*, vol. 17, pp. 33–51, 1994.