Loop transformations for interface-based hierarchies IN SDF graphs

Jonathan Piat, Shuvra S. Bhattacharyya, Mickaël Raulet

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 [**?**], SynDEx [**?**] or PREESM [**?**] 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 [**?**]. In order to extend the expressivity of the SDF model, we propose a new hierarchy type more detailed in [**?**] based on interface. 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.

Section 2 explains the Synchronous Data-Flow graphs, and 3 present existing hierarchy and the new hierarchy representation. Section 4 presents loop analyze and optimization techniques. In 5 we investigate the application of the given optimization technique to the hierarchical SDF model. Section 6 provide results of such optimization on a given application. Finally, section 7 highlights the future work and concludes this paper.

2. SYNCHRONOUS DATA-FLOW GRAPH

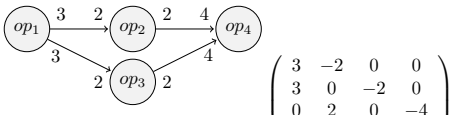
The Synchronous Data-Flow (SDF) graph [**?**] is used to simplify the application specification, by allowing the representation of the application behavior at a coarse grain. This data-flow model represents operations of the application and specifies data dependencies between the operations.

A Synchronous Data-Flow graph is a finite directed,

weighted graph $G =< V, E, d, p, c >$ where :

- V is the set of nodes; each node represents a computation that operates on one or more input data streams and outputs one or more output data streams.
- $E \subseteq V \times V$ is the edge set, representing channels which carry data streams.
- $d : E \rightarrow N \cup \{0\}$ $(N = 1, 2, \ldots)$ is a function with $d(e)$ the number of initial tokens on an edge e.
- $p : E \rightarrow N$ is a function with $p(e)$ representing the number of data tokens produced at e's source to be carried by e.
- $c : E \rightarrow N$ is a function with $c(e)$ representing the number of data tokens consumed from e by e's sink node.

The topology matrix is the matrix of size $|E| \times |V|$, in which each row corresponds to an edge e in the graph and each column corresponds to a node v. Each coefficient (i, j) of the matrix is positive and equal to N if N tokens are produced by the j^{th} node on the i^{th} edge. (i, j) coefficients are negative and equal to N if N tokens are consumed by the j^{th} node on the i^{th} edge. It was proved in [**?**] that a static schedule for graph G can be computed only if its topology matrix's rank is one less than the number of nodes in G. This necessary condition means that there is a Basic Repetition Vector (BRV) q of size $|V|$ in which each coefficient is the repetition factor for the j^{th} vertex of the graph. SDF graph representation allows use of hierarchy, meaning that for $v = G$, a vertex may be described as a graph. A vertex with no hierarchy is called an actor.



$$\begin{pmatrix} 3 & -2 & 0 & 0 \\ 3 & 0 & -2 & 0 \\ 0 & 2 & 0 & -4 \\ 0 & 0 & 2 & -4 \end{pmatrix}$$

Fig. 1. SDF and topology matrix

2.1. SDF to DAG translation

One common way to schedule SDF graphs onto multiple processors is to first convert the SDF graph into a precedence graph such that each vertex in the precedence graph corresponds to a single execution of an actor from the SDF graph. Thus each SDF graph actor A is "expanded into" q_A separate precedence graph vertices, where q_A is the component of the BRV that corresponds to A. In general, the SDF graph aims at exposing the potential parallelism of the algorithm; the precedence graph may reveal more functional parallelism, moreover it exposes the available data-parallelism. A valid precedence graph contains no cycle and is called DAG (Directed Acyclic Graph). Unfortunately, the graph expansion due to the repetition count of each SDF node can lead to an exponential growth of nodes in the DAG. Thus, precedence-graph-based multiprocessor scheduling techniques, such as those developed in [**?**] [**?**], in general have complexity that is not polynomially bounded in the size of the input SDF graph, and can result in prohibitively long scheduling times for certain kinds of graphs (e.g., see [**?**]).

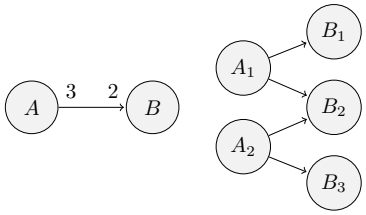


Fig. 2. SDF graph and its precedence graph

3. HIERARCHY TYPES IN SDF GRAPH

Hierarchy can be extracted from a graph in order to optimize application for the scheduling, but can also be used by user to describe an application at different grain level. The first type of hierarchy called clustering allows analysis to group vertices into a single vertex to group data and ease scheduling steps. The second hierarchy representation allows the user to design portion of a graph as a vertex which inner behavior can be determine at runtime using parameters from the data-flow. The interface-based hierarchy allows to design portion of an application to be instantiated at later time. Those fragment have fixed interface behavior that allows to perform local analysis (outside of the application), and generate cleaner C code.

3.1. Repetition-based SDF hierarchy

Hierarchy has been described in [**?**], as a mean of representing cluster of actor in a SDF graph. In [**?**] clustering is used as a pre-pass for the scheduling described in [**?**] that reduces the number of vertices in the DAG, minimizing synchronization overhead for multi-threaded implementation and maximizing the throughput by grouping buffers [**?**]. Given a consistent SDF graph, this approach first clusters strongly connected components to generate an acyclic graph. A set of clustering techniques are then applied based on topological and data-flow properties, to maximize throughput and minimize synchronization between clusters. This approach is a bottom-up approach, meaning that the starting point is a SDF graph with no hierarchy and it automatically outputs a hierarchical (clustered) graph. In order to ensure that clustering an actor may not cause the application to be deadlock, the authors (in [**?**]) describe five composition rules based on the data-flow properties.

3.2. Parameter-based SDF hierarchy

Parameter-based SDF hierarchy has been introduced in [**?**] where the authors introduce a new SDF model called *Parameterized SDF*. This model aims at increasing SDF expressivity while maintaining its compile time predictability properties. In this model a sub-system (sub-graph) behavior can be controlled by a set of parameters that can be configured dynamically. These parameters can either configure sub-system interface behavior by modifying production/consumption rate on interfaces, or configure behavior by passing parameters (values) to the sub-system actors. In this model each sub-system is composed by three graphs: the init graph ϕ_i, the sub-init graph ϕ_s, the body graph ϕ_b.

Each activation of the sub-system, is composed by an invocation of ϕ_s followed by an invocation of ϕ_b. The init graph is effectively decoupled from the data-flow specification of the parent graph and invoked once, at the beginning of each (minimal periodic) invocation (see [**?**]). The sub-init graph performs reconfiguration that does not affect sub-system interface behavior and is activated more frequently than the init-graph which can modify sub-system interface behavior. In order to maintain predictability, actors of ϕ_b are assigned a configuration which specifies parameters values. This value can either be a domain which specifies the set of valid parameter value combinations for the actor, or left unspecified, meaning that this parameter value will be determined at run-time.

3.3. 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. The rules defined in the composition rules ensure the graph to be deadlock free when verified, but are used to analyze a graph with no hierarchy. 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 the following a hierarchical vertex will refer to a vertex which embeds a hierarchy level, and a sub-graph will refer to the graph representing this hierarchy level.

3.3.1. Special nodes

Source node: A Source node is a bounded source of tokens which represents the tokens available for an iteration of the sub-graph. This node behaves as an interface to the outside world. A source port is defined by the four following rules:

A-1 Source production homogeneity: A source node $Source$ produces the same amount of tokens on all its outgoing connections $p(e) = n \quad \forall e \in \{Source(e) = Source\}$.

A-2 Interface Scope: The source node remains write-locked during an iteration of the sub-graph. This means that the interface cannot be filled by the outside world during the sub-graph execution.

A-3 Interface boundedness: A source node cannot be repeated, thus any node consuming more tokens than made available by the node will consume the same tokens multiple times (ring buffer). $c(e)\%p(e) = 0 \quad \forall e \in \{source(e) = source\}$.

A-4 SDF consistency: Every token made available by a source node must be consumed during an iteration of the sub-graph.

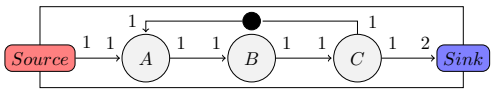


Fig. 3. Design of a sub-graph.

Sink node: A sink node is a bounded sink of tokens that represent the tokens to be produced by an iteration of the graph. This node behaves as an interface to the outside world. A sink node is defined by the four following rules:

B-1 Sink producer uniqueness: A sink node $Sink$ only has one incoming connection.

B-2 Interface Scope: The sink node remains read-locked during an iteration of the sub-graph. This means that the interface cannot be read by the outside world during the sub-graph execution.

B-3 Interface boundedness: A sink node cannot be repeated, thus any node producing more tokens than needed by the node will write the same tokens multiple times (ring buffer). $p(e)\%c(e) = 0 \quad \forall e \in \{target(e) = Sink\}$.

B-4 SDF consistency: Every token consumed by a sink node must be produced during an iteration of the sub-graph.

3.3.2. Hierarchy deadlock-freeness

Considering a consistent connected SDF graph $G = \{g, z\}$, $g = \{Source, x, y, Sink\}$ with $Source$ being a source node and $Sink$ being a sink node, and z being an actor. In the following we show how the hierarchy rules described above ensure the hierarchical vertex g to not introduce deadlocks in the graph G:

- if it exists a simple path going from x to y containing more than one arc, this path cannot introduce cycle since this path contains at least one interface, meaning that the cycle gets broken. User must take this into account to add delay to the top graph.

- Rules **A2-B2** ensure that all the data needed for an iteration of the sub-graph are available as soon as its execution starts, and that no external vertex can consume on the sink interface while the sub-graph is being executed. As a consequence no external vertex strongly connected with the hierarchical vertex can be executed concurrently. The interface ensures the sub-graph content to be independent to the outside world, as there is no edge $\alpha \in \left\{ \alpha' \| \begin{pmatrix} (src(\alpha') = x) \\ and \\ (snk(\alpha') \in C) \\ and \\ (snk(\alpha') \notin \{x, y\}) \end{pmatrix} \right\}$ considering that $snk(\alpha') \notin \{x, y\}$) cannot happen.
- The designing approach of the hierarchy cannot lead to an hidden delay since even if a delay is in the sub-graph, an iteration of the sub-graph cannot start if its input interfaces are not full.

Those rules also guarantee that the edges of the sub-graph have a local scope, since the interfaces make the inner graph independent from the outside world. This means that when an edge in the sub-graph creates a cycle (and contains a delay), if the sub-graph needs to be repeated this iterating edge will not link multiple instances of the sub-graph.

The given rules are sufficient to ensure a sub-graph to not create deadlocks when instantiated in a larger graph.

3.3.3. Hierarchy improvements

As said earlier, this hierarchy type eases the designer work, since he/she can design subsystems independently and may instantiate them in any application. Not only easing the designer work, this kind of hierarchy also improves the application with the same criteria than the clustering techniques. The proposed hierarchy is intended to be both a model and a user-friendly graphical representation, while PSDF and other abstract forms of data-flow should only be considered as models of computation.

This model also makes the application easier to describe for programmers who are for example more familiar with C, and less familiar with concepts such as repetitions vectors and sub-init graphs as seen in PSDF.

4. STATE OF THE ART ON NESTED-LOOPS AND PARTITIONING TECHNIQUE

4.1. Definition and representation

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.

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.

```
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. 4. Nested loop example.

Optimizing nested loops aims at extracting parallelism, by transforming the loops structure. Those transformations can be any of the five types described below.

- Loop distribution : this transformation aims at distributing the nested loop to extract at least one forall loop.
- Loop fusion: this transformation aims at fusionning several loops body into one unique loop
- Loop unrolling: this transformation aims at unrolling a loop to extract the inter-iteration parallelism
- Loop partitioning: this transformation aims at partitioning the loops to extract disjoint iteration domain.
- Unimodular Transformation: this transformation aims at modifying the iteration domain resulting in an out–of–order iteration.

In order to perform those transformations to optimize the loop execution, one must analyze the dependencies between the iterations of the loops. Three types of dependencies exist.

- Flow dependence: a statement S2 is flow dependent on S1 (written) if and only if S1 modifies a resource that S2 reads and S1 precedes S2 in execution.
- Anti-dependence: a statement S2 is anti-dependent on S1 (written) if and only if S2 modifies a resource that S1 reads and S1 precedes S2 in execution.
- Output dependence: a statement S2 is output dependent on S1 (written) if and only if S1 and S2 modify the same resource and S1 precedes S2 in execution.

Analyzing those dependencies can rely on a model which can be treated for optimization. In the following we will be using the "distance vector" as a dependency representation. A distance vector represents the flow dependency between two operation along the iteration domain. In nested loop of depth N, given two access to the same data in the flow order by two instruction S_1 and S_2 with $S_1 \Rightarrow S_2$ with respective index vector $\vec{p}$ and $\vec{q}$. The distance vector $\vec{\delta}$ is an N dimensional vector.For a flow dependency $S_1[\vec{p}] \Rightarrow S_2[\vec{q}]$ the distance vector

is $\vec{\delta} = \vec{p} - \vec{q}$. This specific representation allows to use linear algebra to perform analysis and optimisation. In the following we will investigate a loop partitioning technique based on this representation.

4.2. Loop partitioning by iteration domain projection

This nested loop partitioning technique was developed as a method for systolic array synthesis in [**?**]. 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.

To determine the projection vector we must first determine a time vector τ that satisfies the data dependencies. In [**?**] the author introduces a nested loops optimization technique which aims at transforming a nested loop in a nested loop for which some (all in the optimal case) of the internal loops can be computed in parallel. This goal is achieved by finding a set of parallel linear hyperplane in the iteration domain, such as the set of point being computed is $E(t) = H(t) \cap D$. This set of point can then be computed in parallel. Going from one hyperplane to another $H(t) \rightarrow H(t+1)$ is made by translating the hyperplane with a vector τ called time vector. In the case of uniform nested loops, dependency vector being all positive, it is easy to determine a valid τ vector. A valid τ vector always verify $\tau d \geq 1$ for any dependency vector d. For a given τ vector, the parallel execution time of the nested loop, can be determine by $t_{parallel} = \tau * p_{max}$, p_{max} being the coordinate of the last point of the domain. In [**?**] Lamport propose a solution to find one τ vector for which the parallel execution time is minimal.

Given a valid time vector, a valid projection vector s_n verify $\tau s_n \neq 0$. Let us now complete the s_n vectors into a unimodular matrix S_n, s_n being the first column of the matrix. The matrix S_n is the space base, so we project the computation domain along the first column of S_n onto the hyperplane generated by the $n-1$ other vectors of the matrix. Coordinates of a point p of the computation domain into the space base S_n are $S^{-1}p$. Thus, the allocation function is the lower $(n-1) \times n$ sub-matrix of S^{-1}, which correspond to the $n-1$ last coordinate in the space base.

Using this allocation matrix we can now determine how the domain points are allocated onto the computing network. Further analysis using the time vector and distance vectors also allows to figure out the communication activity over the network and the computation activity of each cell in the network.

By completing the vector τ into a unimodular matrix T^{-1}, T is the time base of the computation domain. The last $n-1$ column vector of T forms the base of the hyperplane of the point computed at time 0, while the first column, is the translation vector that allow to go from the hyperplane of the point computed at t to the hyperplane of the point computed at $t+1$. The activity translation vector correspond to the $n-1$ element of the first column vector of the product $S^{-1}.T$, and the $(n-1) \times (n-1)$ sub-matrix is the activity base.

5. 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.

5.1. 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 $d(e_{N-1})$.

Using the distance vector of the application we can now directly perform the analysis described in the method above and reveal the parallelism nested into the hierarchy. Using the analysis results, it is then possible to synthesize a new SDF network performing the same computation.

5.2. 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.

For a set of strongly connected components C and for each computing element $E_n \in C$ we can determine the hyperplane in the iteration domain containing the last computation performed by E_n. The element with the hyperplane that has the shortest distance to the hyperplane of points computed at $t = 0$, is the element that must be scheduled first. This means that its incoming edges of belonging to the strongly connected set must carry a delay.

The synthesized network can then be used with ring_buffer vertex, to ensure the output data to be the last. The computation performed by the network is strictly the same, with some vertices performing computation outside of the iteration domain (which should be considered null time).

6. 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.

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

from this set we can extract a the following system.

Initial state

$$\begin{cases} R_{i,k} = 0 & if \quad k = 0 \\ V_{i,k} = v_k & if \quad i = 0 \\ M_{i,k} = m_{i,k} \end{cases}$$

Calculus equations

$$\begin{cases} R_{i,k} = R_{i,k-1} + V_{i-1,k} M_{i,k} & i \quad 1 \leq k \leq N \\ V_{i,k} = V_{i-1,k} & i \quad 1 \leq k \leq N \\ M_{i,k} = m_{i,k} \end{cases}$$

Output equation

$Ri = R_{i,N}$

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.

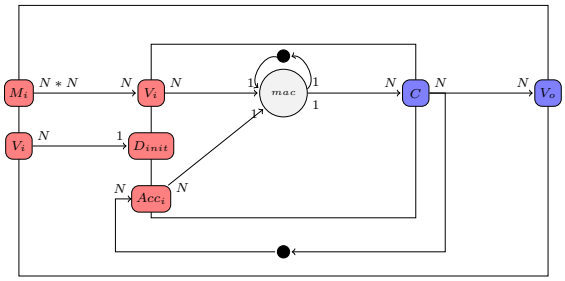


Fig. 5. Matrix vector product

6.1. 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.

6.2. 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 [**?**] 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 $N = 3$.

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

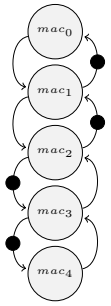


Fig. 6. Valid timed network

The resulting timed network need 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.

7. CONCLUSION AND FUTURE WORK

This paper introduces a new hierarchy type that involves the designer more in the application optimization process by al-

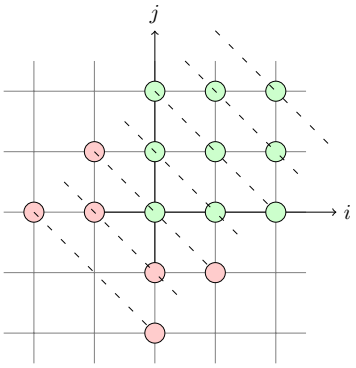


Fig. 7. Iteration domain after graph timing

lowing him/her to modify the application structural description.In particular, our hierarchy representation is closer to C code semantics, and provides a useful hybrid representation related to SDF and C. This representation makes the application easier to describe for programmers who are for example more familiar with C, and less familiar with concepts such as repetitions vectors and sub-init graphs. Our method allows reuse of graphs developed in other applications with no modifications, and offer more flexibility by allowing the description of execution patterns that do not map directly into conventional types of hierarchy.

The Interface-based hierarchy for Synchronous Data-Flow Graphs has been implemented as the algorithm specification model in the tool PREESM [**?**].

The optimization technique described in section 5 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 [**?**]. This paper shows that loop optimization method inherited from various computing environment can be used in the Synchronous Data-Flow and give relevant results.

8. 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] E.A Lee and D.G Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, sept 1987.

[7] H. W. Printz, *Automatic mapping of large signal processing systems to a parallel machine*, Ph.D. thesis, Pittsburgh, PA, USA, 1991.

[8] G.C. Sih and E.A. Lee, "Dynamic-level scheduling for heterogeneous processor networks," in *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, October 1990, pp. 42–49.

[9] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for DSP applications," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1995, pp. 122–126 vol.1.

[10] C. Hsu, J. L. Pino, and S. S. Bhattacharyya, "Multithreaded simulation for synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, Anaheim, California, June 2008, pp. 331–336.

[11] 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.

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

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