Computer-aided design and visualization of regular algorithm dependence graphs and processor array architectures

Oleg Maslennikow, Natalia Maslennikowa, Przemysław Soltan
Department of Electronics
Technical University of Koszalin
Poland

Abstract

In this paper, the part of a program environment for computer-aided design of VLSI processor array architectures is proposed. This part, named JGEN, is destined for construction and visualization of dependence graphs of algorithms described by nested loops, as well as for visualization of the corresponding processor array architectures. Program JGEN is based on the (designed with authors contribution) method for construction of dependence graphs for regular algorithms and their mapping in the processor array architectures, and generates information about target graph nodes and arcs in the XML-document format. It allows to transform the derived output XML files in SVG format (by means corresponding XSLT styles) and to use, for example, Adobe SVG Viewer 3.0 or even MS Windows Internet Explorer (with corresponding plug-in module) environments for visualization of algorithm dependence graphs, as well as the corresponding processor array architectures.

1. INTRODUCTION

Many real life problems in various fields of science, engineering and technology such as signal and image processing, statistical and numerical analysis, biomedical researches, physical experiments, etc. are finally reduced to operations over large size vectors or/and matrices of input data. Most of methods for solving mentioned problems are characterized by a high computational complexity (up to $O(N^3)$ multiply-add operations, where $N$ is the order of input data array) and regularity [1]. The regularity means here that identical calculations are carried out with nearly all elements of input data at each algorithm step, and the order of computations is independent from input data values. Therefore, effective real-time solution of such problems may be reached using application-specific parallel systems, which are destined to implementation of several applied algorithms and are adapted to their properties. The VLSI processor arrays (PA) [2] are typical examples of such architectures. Using massive pipelining, these arrays exploit the regularity of algorithms to achieve high performance while keeping local communications and low I/O requirements.

Processor array architectures can be designed systematically using linear space-time mappings to algorithms that are expressed as systems of recursive equations or, equivalently, by nested loops [2,3,4]. The most known mapping methods [2-5], as a rule, are based on the representation of applied regular algorithm by its dependence graph DG (may be not directly). Then, in a course of mapping, a given algorithm with the dependence graph $G$ is transformed into a set of arrays architectures $C = <S,T,Φ>$ implementing this algorithm (see fig.1), where $S$ is a directed graph called the array structure, $T$ is the synchronization function specifying the computation time of nodes in the DG, and $Φ$ is the set of execution algorithms of processor elements. However, for deriving of the algorithm DG it is needed to use the special methods, which are complicated, mostly operate with only uniform recursive algorithms [4] and are few feasible for the CAD implementation, as well as, for visualization of derived graphs. The last feature is very impotent, because the obtained PA architectures, as a rule, are strictly determined by algorithm DG properties, and are not always effective ones. Therefore, in order to deriving the array architecture with improved parameters, the purposive transformations of the basic algorithm DG may be performed before space-time mapping of the algorithm graph into array architecture [6, 10]. These transformations correspond to deriving of such algorithm version, which is better suitable to the parallel implementation. Therefore, the paper includes the brief descriptions of the proposed methods [7] of deriving dependence graphs of regular algorithms, and mapping of these graphs onto corresponding processor array architectures with desired parameters [3]. These methods are realized in the proposed program JGEN, which generates information about target graph nodes and arcs in the XML format (XML - eXtensible Markup Language). It allows to transform the derived output files in the Scalable Vector Graphics (SVG) format (by means XSLT-converter with corresponding XSL styles) and to use, for example, Adobe SVG Viewer 3.0 [8, 9] or even MS Windows Internet Explorer (with corresponding plug-in module) environments for visualization of algorithm dependence graphs, as well as the corresponding processor array architectures.
2. DERIVING OF REGULAR ALGORITHM DG AND MAPPING OVERVIEW

VLSI processor arrays can be designed systematically by applying linear (or affine) mappings to algorithms that are expressed as systems of recursive equations or, equivalently, by nested loops \([2, 4, 7]\). Such algorithms can be represented by regular or quasi-regular, lattice dependence graphs. Each node of such lattice DG corresponds to a certain iteration of the original algorithm, and is associated with an integer vector \(C = (I_1, \ldots, I_n)\) located in the vertices \(K\) of an integer lattice space \(K^n\). Arcs between nodes of this DG, or data dependencies between iterations of the algorithm, are represented by the dependence matrix \(D_j\), in which the \(j\)-th column is a dependence vector \(d_j\). If the iteration corresponding to a node \(K_j\) depends on the iteration corresponding to another node \(K_i\), this dependence is represented by the difference \(d = K_j - K_i\).

The method is based on the approach, in which the initial description of an algorithm is divided into components corresponding to different elementary loop nests (ELN). For each ELN, its DG is constructed, and the resulting DG of the whole algorithm is obtained as a composition of all elementary graphs, which are connected by short arcs (with the length of unity). The main idea of deriving the ELN DG is based on the simulation of its executing and saving of current values of indices of its variables in the loop body. We assume that ELN consists of a multilevel construction of nested for-statements including one another, and the corresponding loop body without any exit from it. The description of the nest in a Pascal-like notation is represented in fig.2,a.

\begin{verbatim}
for I1:=a1 to b1 step c1
for I2:=a2 to b2 step c2
 . . . . . . . .
for In:=an to bn step cn
{statements of the loop body}
endfor
 . . . . . . . .
endfor
endfor
\end{verbatim}

Here \(a_j\), \(b_j\), and \(c_j\) are expressions denoting the lower, upper limits and step of the loop at nesting level \(j\) \((j=1,2,\ldots,n)\). Moreover, \(a_1\), \(b_1\) and \(c_1\) are constants, while others of \(a_j\) and \(b_j\) are given by linear functions of \(I_j\). The statements of the loop body contain some indexed variables \(X_{a_1,\ldots,a_n}\), whose indices \(m\) are functions of \(I_j\). Each elementary nest is characterized by its dimension \(n\) (which is equal to the number of for-statements) and defines the corresponding iteration space. Each of its node represents a single execution of the loop body, and is defined by an iteration vector \(K=(I_1, I_2, \ldots, I_n)\), where \(i_j\) is equal to the value of \(I_j\) during the corresponding iteration. In order to derive the lattice DG of an elementary nest, the following parameters should be determined:

- the dimension \(n\) of its iteration space;
- the number of nodes in the DG and their coordinates;
- coordinates of all arcs (vectors) between the nodes.

The dimension \(n\) is equal to the number of for-statements in the nest. To find the number \(V\) of nodes in the DG and their coordinates, we introduce an auxiliary counter statement of the form \(C:=C+1\), into the loop body. After executing the nest, the value of \(C\) will be equal to \(V\). Furthermore, if while running the algorithm we store current values of iteration indices \(I_j\), which correspond to counter values \(C\), then each node of the DG will be identified not only by the value \(C\), but also by its coordinates \(k_1, \ldots, k_n\).

The following idea is used to determine arcs between the nodes and their direction (i.e. coordinates of the graph arcs). The presence of an arc between two nodes in the DG means that a certain indexed variable \(X_{a_1,\ldots,a_n}\) is transferred between these nodes. Hence, this arc will exists only if this variable has the same value of its indices \(I_1, \ldots, I_Q\), in the both nodes. The arc will be directed from the node with a less value of the variable \(C\) to the node with a greater value of \(C\). Consequently, in order to determine the arcs, while running the algorithm we should store current values of indices.
for all indexed variables of the loop body. Then for each variable, we find the set of nodes in which the variable has the same values of its indices. The nodes belonging to this set will be finally connected by arcs according to a linear order given by the counter variable \( C \), where an arc connects two nodes with nearest values of \( C \). The example of Gauss elimination algorithm DG corresponding to the construction showed in Fig.2,b is represented in the Fig. 3 (for input matrix \( A(4,4) \)).

One of most promising approaches to mapping recursive algorithms with regular dependencies into processor arrays \[3\] consists in finding of the linear mapping operator \( F \), which determines, for each node \( K \) of the algorithm DG, the corresponding node (i.e. processor element) in the PA structure:

\[
F : K^n \rightarrow K_{F}^{m+1}, \ F(K) = F.K, \ \forall K \in K^n,
\]

where \( m \) is the dimension of the PA structure \((m+1) \leq n\). Operator \( F \) represents the \((m+1)n\) matrix and composes of two components: space mapping \( F_S \), and time mapping \( F_T \). As a result, the arbitrary DG node \( K \in K^n \) will be carried out in the processor element PE) with coordinates \( F_S \cdot K \) at the time step number \( F_T \cdot K \). In accordance to the methodology \[3\], the set of all possible and nonequivalent allocation mappings \( F_S \) satisfying given constraints for links between PEs is firstly determined. For each of network topologies \( S \) corresponding to this set, an optimal schedule mapping which implements the algorithm correctly is determined then. This mapping is constructed as a linear (or affine) function \( F_T \). Note, that the geometric interpretation of space mapping function \( F_S \) is the coordinates of the hyperplane, where the DG must be projected. The schedule function \( F_T \) determines the coordinates of a vector, which finds the direction of computation propagation.

3. JGEN - PROGRAM ENVIRONMENT FOR DESIGN AND VISUALIZATION OF REGULAR ALGORITHM DEPENDENCE GRAPHS AND PA ARCHITECTURES

Program JGEN is designed in Java language. The input algorithm also should be represented as a fragment of Java program. This allows to use the specially designed library \texttt{jen.core.gzi.user}, which destined for generation of the output information about algorithm graph. This library consists of different classes (for example, \texttt{GziMatrix} and \texttt{GziVector}), which are used in the analyzed program. The example of the description of Gauss elimination algorithm and its dependence graph (for input matrix \( A(4,4) \)) are represented in the fig. 3,a and fig 3,b respectively.

```java
import jgen.core.gzi.user.*;

...  
Gzi gzi = new Gzi();
gzi.setName("gauss"); //opcjonalna nazwa grafu  
gzi.setSize(3);  //rozmiaru generowanego grafu  
int N = 4;
GziMatrix a = new GziMatrix("a", 1, N, 1, N);
GziMatrix m = new GziMatrix("m", 1, N, 1, N);
for (int k1 = 1; k1 <= N - 1; k1++) {
  for (int k2 = k1 + 1; k2 <= N; k2++) {
    gzi.setNode(2, k1, k2, k1);
    m.set(k1,k2,a.get(k2,k1) / a.get(k1,k1));
  }
  for (int k2 = k1 + 1; k2 <= N; k2++) {
    for (int k3 = k1 + 1; k3 <= N; k3++) {
      gzi.setNode(1, k1, k2, k3);
      a.set(k2,k3,a.get(k2,k3) - m.get(k1,k2)/a.get(k1,k3));
    }
  }
}
gzi.stop(); //finalizacja tworzenia grafu

...  
```

Fig. 3. Representation of Gauss elimination algorithm in JGEN program and its dependence graph
Procedure of the determination of DG node and arc coordinates is based on executing of methods setNode(...), and set() i get() respectively. Moreover, input file may be modified, for example, by means changing of order of executing in the input program operators. Due to, another algorithm dependence graph may be derived, and better PA architectures may be designed. Executing of input program causes, that the output XML file will be created, which consists the lists of graph nodes (with their types) and graph arcs (edges). The example of output file, which describes the Gauss elimination algorithm DG is represented in the fig.4.a.

a)  

```xml
<?xml version='1.0' encoding='UTF-8'?>
<gzi id='1' size='3' ver='2.37' name='gauss'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xsi:noNamespaceSchemaLocation='graph.xsd'>
  <nodeTypeList>
    ...
  </nodeTypeList>
  < nodeList>
    ...
    <node type='1' k1='1' k2='3' k3='2' id='7'/>
    ...
  </ nodeList>
  <edgeList>
    ...
    <edge from='2' to='7' id='6'/>
    ...
    <edge from='4' to='7' id='7'/>
    <edge from='7' to='8' id='8'/>
    ...
    <edge from='7' to='10' id='13'/>
    <edge from='7' to='13' id='18'/>
    ...
  </edgeList>
</gzi>
```

b)  

```xml
<?xml version="1.0" encoding="UTF-8"?>
<svg height="460" width="400">
  <defs>
    <marker ... />
    <radialGradient ... />
    <script language="Javascript">
      ...kod javaScript
    </script>
  </defs>
  <path d="M 150 170 L 230 150"/>
  ...
  <circle cx="150" cy="170" r="5"/>
  ...
  <circle cx="230" cy="150" r="5"/>
  ...
  ...
</svg>
```

Fig. 4. Fragment of XML file described of Gauss elimination DG (a) and corresponding fragment of SVG file (b)

The generation of resulting SVG document, destined for DG visualization, is based on transformations of several constructions in obtained XML file to corresponding graphic images represented in accordance to SVG standard. For example, graph nodes, which described by the construction

```xml
<node type="..." k1="..." k2="..." k3="..."/>
```

are transformed to the construction `circle cx="X" cy="Y" r="5">`. As a result, the coordinates, size and color of the circle representing the target node in the graph are determined.

Analogously, the construction `edge from="..." to="...">`, which represents a edge between two nodes in the graph, are transformed to the construction `path d="M X1 Y2 L Y1 Y2" marker-end="...">`. These transformations are carried out by means the XSLT style designed by authors (XSLT - eXtensible Stylesheet Language Transformations).

Note, that visualization of the designed processor array architectures is carried out in similar way. Fig. 5a-fig.5b and fig. 5c – fig.5d represents the examples of the one-dimensional and two-dimensional PA structures for realization of the Gauss elimination dependence graph.

Fig. 5. Examples of PA structures for realization of the Gauss elimination algorithm
The main quality criterion during designing of JGEN environment has been the using of standard mechanisms and formats for data transformation, transmission, saving and visualization. Therefore, Java and XML have been used as basic languages for description of the input algorithm, its dependence graph and the derived architectures of processor arrays. As a result, the program JGEN has been designed, which is based on the method for construction of dependence graphs for regular algorithms and their mapping in the PA architectures, and generates information about target graph nodes and arcs in the XML-document format. It allows to transform the derived output XML files in SVG format and to use, for example, Adobe SVG Viewer 3.0 or even MS Windows Internet Explorer (with corresponding plug-in module) environments for visualization of algorithm dependence graphs, as well as the corresponding processor array architectures.

CONCLUSIONS

BIBLIOGRAFIA
