A Method for Avoiding Loops while Decomposing the Task Description Graph in System-Level Synthesis

Péter Arató, Dániel András Drexler, Gábor Kocza
Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest, Hungary
Emails: arato@iit.bme.hu, drexler@iit.bme.hu, koczagbr@gmail.com

Abstract—In system-level synthesis, the graph describing the task may consist of a great number of vertices, thus the design algorithms (e.g. hardware-software partitioning, pipeline synthesis, etc.) may become extremely complicated. This difficulty is relaxed by decomposing the task description graph that is usually unavoidable in system-level synthesis. The decomposing algorithms unite certain vertices of the graph, thus the resulting graph consists of less vertices. However, loops may appear in the decomposed graph, even if the original graph was loop-free, that endangers the efficiency of the design algorithms. We propose an algorithm that generates allowable cuts. We prove that any decomposition made along these cuts always yields a loop-free graph. The method is demonstrated on a simple example. Incorporating optimization criteria in the cut generation is also discussed.

Index Terms—loop-free decomposition, graph cutting, directed acyclic graph

I. INTRODUCTION

The system-level synthesis of complex hardware or multiprocessing systems starts from some kind of a task description formalized usually in a high-level programming language. For the further steps of the design procedure, a dataflow-like task description graph (TDG) is generated from the task description. In most cases, the decomposition of the TDG is necessary in attempting to approach an optimal implementation. The decomposing algorithms yield graph segments representing subtasks of the system to be designed. The graph segments are handled as vertices of a segment graph (SG) in the further design steps. However, during the decomposition, it is a crucial problem that the resulting SG may contain loops (cycles), even if the TDG is loop-free (acyclic). Such loops may cause dead-locks in the task execution formalized by the SG and can be disadvantageous in designing pipeline systems. As the decomposing algorithms alone cannot guarantee loop-free SGs, subsequent checking and "trial and error" loop elimination steps should be executed.

In order to avoid these difficulties, a straightforward systematic method is presented. We define the list of vertices of the TDG as an allowable cutting list (ACL), if the vertices can be divided into two disjoint subsets \( A \) and \( B \) with the following properties:

- for any \( i < N \), \(|A| = i \) and \(|B| = N – i \), where \( N \) is the number of elements in the ACL;
- \( A \) contains the first \( i \) elements of the ACL;
- There is no such vertex in \( A \) that has a successor in the set \( B \);
- There is at least one vertex in \( B \) that has successor in \( A \).

By choosing the cuts from the ACL during the decomposition, it is guaranteed that the resulting SG is loop-free. This statement is proved in Section III. The ACL generating algorithm is extended to handle weighted graphs, for incorporating optimization criteria into the ACL generating algorithm.

The paper is organized as follows. The properties of the TDG and the problem formulation is discussed in Section II. The ACL generating algorithm (ACLGEN) is presented and analyzed in Section III. The paper ends with the conclusion in Section IV.

II. THE TASK DESCRIPTION GRAPH AND THE DECOMPOSITION PROBLEM

We suppose that for the system to be designed, a task description graph (TDG) is available. A TDG is a directed acyclic graph. Each vertex of the graph represents an elementary operation of the task, that has an input and an output. The directed edges between the vertices represent the communication (data transfer). If the TDG is created from a program code, the vertices are commands, and the edges represent the datapath. The graph is supposed to be acyclic, such that there is no feedback in the graph (there is no directed cycle).

The TDG can be compiled directly from a program code [1], [2], therefore the raw TDG usually may consist of a great number of vertices. In the typical applications, the TDG is decomposed into segments for distributing them onto different resources or to be processed by a pipelining tool. An example for distribution of the segments is the hardware-software partitioning [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. Examples for pipelining can be found in [13], [14], [15], [16], [17].
In either cases, the high number of vertices causes rather inconvenient computational burden, thus decomposing the TDG into a graph consisting of much less vertices (by uniting certain vertices) is a practical procedure. There exist various decomposition techniques in the literature (see e.g. [15], [4], [18]), however they all suffer from a great disadvantage, namely there is no guarantee that the resulting graph will be acyclic. If the result is not acyclic, then controlling the execution time of the whole task may become impossible, thus those decompositions are desirable, that result in acyclic graphs.

The problem is illustrated on an example in Figs. 1 and 2 (a similar example is used in [4] to illustrate this problem). The TDG consists of nine vertices, there are two input vertices and two output vertices (i.e. vertices that have edges from the environment, and edges pointing to the environment, respectively). Fig. 1 shows a decomposition that results in a graph that is not acyclic, while Fig. 2 shows a decomposition that results in a graph that is acyclic. In the next section we present an algorithm that guarantees that the decomposition yields an acyclic graph.

III. Generating the Allowable Cutting List

A. Finding a possible set of allowable cuttings

Suppose that the TDG has the following properties:
- It has finite number of vertices;
- It has no cycles, i.e. it is a directed acyclic graph (DAG).

For the formal handling of the TDG, let the set of all vertices of the graph be denoted by $G$, and let $|G| = N$. Let $x \in G$ be a vertex of the graph, then we denote by $s(x)$ the set of successors of the vertex $x$. Note that since the graph is acyclic, there always exists $x \in G$ such that $s(x) = \emptyset$ (e.g. the vertices 8 and 9 in Figs. 1 and 2). These vertices are called the output vertices of the graph. Let $O$ denote the set of output vertices, i.e. $O = \{ y \in G : s(y) = \emptyset \}$.

**Lemma 1.** If $G$ is the set of vertices of a directed acyclic graph with finite number of vertices, and the set of output vertices is $O$, then $\forall x \in G \setminus O$, there exists a directed path from $x$ to $O$.

**Proof:** The proof is constructive. Starting with $x$, build a path by adding any successor of $x$, say $x_1$ (i.e. $x_1 \in s(x)$), then adding any successor of $x_2$, say $x_3$ (i.e. $x_3 \in s(x_2)$), and so on. Since $G$ is acyclic, the path can only contain each vertex once, and since the graph has finite vertices, at the $n$th step we reach a vertex $x_n$ that has no successor for some $n \in \mathbb{N}$. By definition $x_n \in O$, so we have constructed a path between $x$ and $O$.

Unite the output set into one vertex $o$ by the following way: remove all the vertices in $O$ from $G$, add the vertex $o$, and for all $x \in G \setminus O$, that satisfies $s(x) \cap O \neq \emptyset$, add $o$ to the set $s(x)$.

The resulting TDG is weakly connected (i.e. if we replace the directed edges with undirected ones, the resulting graph is connected). If it is not the case, that means that the TDG contains a disjoint graph that has no output vertex, which can not happen if the graph is acyclic.

As a consequence of Lemma 1, there exists a connected path between any vertex of the resulting graph and the output vertex $o$.

Let the set of vertices of $G$ be partitioned into the disjoint sets $A$ and $B$ such that $G = A \cup B$, and $\forall a \in A, s(a) \cap B = \emptyset$, so there is no directed edge from $A$ to $B$. Note that this definition implies that $o \in A$. As a consequence of Lemma 1, there always exists an element $b \in B$ with the property $s(b) \cap A \neq \emptyset$, i.e. it has a directed edge from $B$ to $A$, since there is a directed path between every element of $B$ and the vertex $o \in A$.
Definition 1. We call the partitioning \( G = A \cup B \) defined in the previous paragraph a proper partitioning of \( G \).

Let us partition the set \( B \) into three parts:
1) \( X = \{ x \in B : s(x) \cap A \neq \emptyset, s(x) \cap B = \emptyset \} \), so the set of vertices in \( B \) that has successors only in the set \( A \);
2) \( Y = \{ y \in B : s(y) \cap A \neq \emptyset, s(y) \cap B \neq \emptyset \} \), so the set of vertices in \( B \) that has successors both in the set \( A \) and \( B \);
3) \( Z = \{ z \in B : s(z) \cap A = \emptyset, s(z) \cap B \neq \emptyset \} \), so the set of vertices that has successors only in the set \( B \).

It is trivial that \( X, Y \) and \( Z \) are disjoint sets, and \( B = X \cup Y \cup Z \). We can say more about these sets.

Lemma 2. Suppose that \( G \) is the set of vertices of a directed acyclic graph, and \( G \) is partitioned into the sets \( A, B, X, Y, Z \) as defined above. Then \( X = \emptyset \) if and only if \( B = \emptyset \).

Proof: Since \( X \subseteq B \), \( B = \emptyset \) trivially implies \( X = \emptyset \). We prove that \( X = \emptyset \) implies \( B = \emptyset \). The proof is indirect. Suppose that \( X = \emptyset \), but \( B \neq \emptyset \). It means that \( B = Y \cup Z \). Remove the edges pointing from \( B \) to \( A \). The result is two disjoint graphs, the graph represented by the vertices in \( B \) and the graph represented by the vertices in \( A \). Now examine the graph represented by the vertices in \( B \). Since \( X = \emptyset \), all the vertices in \( B \) have successors in \( B \), so there is no vertex in \( w \in B \) such that \( s(w) = \emptyset \). Since a directed acyclic graph always has a vertex \( w \) with the property \( s(w) = \emptyset \), the graph represented by the vertices in \( B \) cannot be acyclic, which means that it contains a directed cycle, that is a contradiction, since the original graph was a directed acyclic graph. So \( X = \emptyset \) implies \( B = \emptyset \).

Definition 2. We call a cut an allowable cut if it partitions \( G \) into the subsets \( A \) and \( B \) such that it is a proper partitioning of \( G \).

Definition 3. The Allowable Cutting List (ACL) is the list of the vertices from the set \( G \), such that the first element of the list is the output vertex \( o \), and for each \( 1 \leq i < N \), setting \( A \) be the set formed by the first \( i \) vertices in the list, and \( B \) be the set formed by the last \( N - i \) vertices of the list, \( A, B \) is a proper partitioning of \( G \).

Thus by definition each element of the ACL is an allowable cut, so it partitions \( G \) into two parts such that there is no feedback.

Next, we propose an algorithm that results in an ACL in Fig. 3. The algorithm creates a list of the vertices of the graph. The first element is the output vertex \( o \), and the next elements are always chosen from the set \( X \). We show that the algorithm results in an ACL.

Theorem 1. The algorithm ACLGEN defined in Fig. 3 is complete and results in an ACL.

Proof: The proof is done by induction on the steps of the algorithm. At the first step, the list contains only the output vertex \( o \), so by setting \( A := o \), and \( B := G \setminus o \), we get a proper partitioning of \( G \), since the output vertex trivially does not have a successor.

Suppose, that the \( k \)th step of the algorithm resulted in a proper partitioning, i.e. by setting \( A \) to be the first \( k \) vertices in the list, and \( B \) the remaining vertices of the graph, then \( A, B \) is a proper partitioning of \( G \) (the vertices in \( A \) have no directed edges to any vertex in \( B \)).

At the \((k + 1)\)th step there are two cases:
1) The set \( B \) is empty. In this case the algorithm is ready.
2) The set \( B \) is not empty. In this case the algorithm chooses some vertex \( x \) from the set \( X \). Since \( x \) has no directed edge to any vertex in \( B \), the new partitioning is also proper. As a consequence of Lemma 2, \( X \) is not empty if \( B \) is not empty, ensuring that such \( x \) always exists. This also yields that the algorithm is complete.

Note that the output vertices need not be united into one vertex, they can be handled separately, however in the algorithm ACLGEN the first few elements should contain the output vertices. The output vertices are united only for simplicity.

The decomposition of the task into \( K \) segments can be characterized by the ACL and a list of integers \( 1 \leq c_1 < c_2 < \ldots < c_{K - 1} < N \), such that the first segment is composed of the vertices defined in the first \( c_1 \) elements of the ACL, i.e. by the elements \( C(1 : c_1) \), the second segment is composed of the vertices defined by the elements \( C(c_1 + 1 : c_2) \), the \( k \)th segment is composed of the vertices defined by the elements \( C(c_k + 1 : c_{k + 1}) \), while the last segment is composed of the vertices defined by the elements \( C(c_{K - 1} + 1 : N) \).

As an example, consider the problem given in Fig. 2. A possible result of the algorithm ACLGEN is the list

\[
C = \{(8, 9), 7, 5, 6, 3, 4, 1, 2\}
\]  \hspace{1cm} (1)

and a possible result of the decomposition into three segments characterized by the integers \( 1 \leq c_1 = 3 < c_2 = 7 < 9 \) is

\[
C = \begin{cases} (8, 9), 7, 5, 6, 3, 4, 1, 2 \end{cases} \hspace{1cm} \text{II}  \\
\begin{cases} 5, 6, 3, 4, 1, 2 \end{cases} \hspace{1cm} \text{II}  \\
\begin{cases} 3, 4, 1, 2 \end{cases} \hspace{1cm} \text{I}
\]  \hspace{1cm} (2)
**Require:** A TDG, that is a weakly connected, directed acyclic graph, with its output vertices united into the vertex \( o \), and weighted edges.

**Ensure:** A list \( C \) of the vertices of the graph generated in light of weighting minimization along the cuts. 

\[
C := [o].
\]

Calculate the sets \( A, B \) and \( X \).

**while** \( B \neq \emptyset \) **do**

\[
\text{choose } x^* = \arg \max_{x \in X} \{\overline{w}(x) - w(x)\}.
\]

**remove** \( x^* \) from \( B \) and **add** \( x^* \) to \( A \).

**add** \( x^* \) to the list \( C \), i.e. \( C := [C, x^*] \)

**update** the set \( X \).

**end while**

Fig. 4. The extended algorithm ACLGEN to generate an ACL

Fig. 2 also demonstrates that the result is loop-free, such that segment III has no directed edge to segments II and I, and segment II has no directed edge to segment I.

**IV. COST FUNCTION GENERATION**

The ACL for a given problem is not unique, however the order of the vertices in the list determines the set of vertices that may be in one segment after the decomposition. In each step of the algorithm, the set \( X \) is not empty (except if the algorithm is finished), and may contain more than one elements. In that case there are more possible ways to choose an element from the set \( X \) to be replaced from the set \( B \) to \( A \). The element we choose affects the order of the vertices in the ACL, but does not affect the properties of the algorithm established in Theorem 1.

There are numerous strategies to choose a vertex from the set \( X \). In this subsection, we define a strategy that handles weighted TDGs, and attempts to minimize the total weight of the cut edges. This weighting is only an illustration, but in general any user-defined priority relation between vertices can be handled by using this weighting strategy.

Suppose, that the TDG is a weighted directed acyclic graph. Let \( \overline{\pi}(x) \) denote the sum of the weights of the output edges of vertex \( x \) and \( w(x) \) the sum of the weights of the input edges of vertex \( x \).

Suppose, that the goal is to minimize the sum of the weights of the edges selected for cutting. In this case, in each iteration, we shall move the vertex \( x \in X \) that reduces the total weight the most (has great \( \overline{\pi}(x) \)) and increases the total weight the least (has low \( w(x) \)), i.e. the difference of \( \overline{\pi}(x) \) and \( w(x) \) is maximal. Since we still choose from the set \( x \), the feedback-free property of the result is guaranteed.

A possible application of this weighting procedure is e.g. the case when the weights represent communication burden between the vertices. In this case, the algorithm attempts to generate the cuts that have the minimal communication burden between the resulting segments. The extended algorithm that minimizes weight along the cut is in Fig. 4.

The algorithm is illustrated on the example in Fig. 5. The graph is the same as the initial graph in Figs. 1 and 2, however the edges have weights (black numbers). The ACL of the current problem generated by the extended algorithm is

\[
C = \{(8, 9), 7, 6, 4, 5, 2, 3, 1\}.
\]  

(3)

and the resulting cutting places are depicted as gray dotted lines in Fig. 5, while the sum of the weights along each cut are depicted as grey numbers. Note that this application is just an example, and this modified algorithm is only a local search algorithm.

**IV. CONCLUSION**

The presented method generates the list of all such edges (ACL) of an acyclic graph, which are allowed to be cut by decomposing the graph in order to guarantee the loop-free property of the decomposed graph. It is proved that the decomposed graph is always loop-free if any kind of decomposing algorithms does not cut edges which are not in the ACL. The decomposing algorithms may be adapted easily to this constraint by limiting the cuts according to the ACL. In the applications, it is advantageous that the ACL generating algorithm can handle user-defined priority relations between vertices. As an example for this priority relation, a weighting of the edges based on communication burden has been presented. One of the further research aims is to adapt the method for handling the cases when the initial graph to be decomposed is not acyclic.

**V. ACKNOWLEDGMENT**

The research work presented in this paper has been supported by the Hungarian Scientific Research Fund OTKA.
REFERENCES


