Template Choice and Template Hierarchy Graph construction during the Partition Phase in reconfigurable environment.

Alessandro Febretti - Alessandro Frossi

20th June 2006

Contents

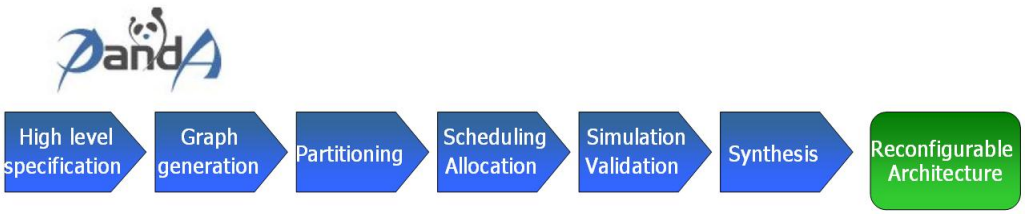

Figure 1: DRESD WorkFlow

1 Introduction

This project is intended to be used within the PandA Project, developed in Politecnico di Milano. The primary objective of the PandA project is to develop a usable framework that will enable the research of new ideas in the HW-SW Co-Design field; the current research program aims at defining an efficient high-level synthesis tool that starting from C, C++ or SystemC system descriptions generates synthesizable VHDL RTL descriptions. This work, in particular, deals with partial dynamic reconfiguration (see Section 2) on FPGAs (*Field Programmable Gate Array*) and it is therefore strictly related to (and developed within) the DRESD Project. The vision of the DRESD Project is the research and development of methodology and tools tailored to design, test and implement reconfigurable architectures; from a less abstract point of view its mission is presented in 1: starting from an High Level Specification (C or SystemC) the aim is to arrive at the actual realization of a Reconfigurable Architecture that can be programmed on a device. This procedure follows some well-defined steps: after the graph generation, the DFG just created is partitioned and the templates are identified (along with their template hierarchy graph). These are then used to perform scheduling of the application and the relative allocation on the device; the solution is then simulated and validated and, at last, synthetized.

The problem that is addressed in this report is the Template Choice and Candidate Evaluation: given as input a set of possible templates that cover a Data Flow Graph, we want to identify isomorphic subgraphs common to at least two of those templates, and classify them using given thresholds and metrics.

Section 2 describes the state of the art and provides the definitions and concept used during this work, while Section 3 outlines the problem and the resolution methodology.

2 State of the art

In this section is presented the environment and the starting point of our work, as well as the main concepts encountered during the presentation of the proposed methodology.

2.1 Reconfiguration

The concept of reconfigurable computing has been around since the 1960s, when Gerald Estrin's landmark paper proposed the concept of a computer consisting of a standard processor and an array of "'reconfigurable"' hardware. The main processor would control the behavior of the reconfigurable hardware. The reconfigurable hardware would then be tailored to perform a specific task, such as image processing or pattern matching, as quickly as a dedicated piece of hardware. Once the task was done, the hardware could be adjusted to do some other task.

Therefore, a possible definition for reconfiguration can be:

Definition 2.1 (Reconfiguration) *The process of physically altering the location or functionality of network or system elements. Automatic configuration describes the way sophisticated networks can readjust themselves in the event of a link or device failing, enabling the network to continue operation.*

In our scenario, reconfiguration is intended from a slight different point of view: we're not interested in performing different tasks (where *different* means *belonging to completely different applications*) but we're more concerned with fitting applications into a programmable device. Usually, in fact, applications are too big to fit entirely in a single device; one solution is to use bigger devices but this can be very expensive. Another approach can be to split the application in modules and program them on the FPGA one (or, usually, more) at a time, allowing part of the computation to be performed as if the whole logic was already written on the device. When other modules are needed they're dynamically programmed and computation can continue with only a minimum loss in terms of computational time. The main drawback of such an approach is that we introduce in our design latencies not related to the application itself but to the way it is implemented on the FPGA: those are the represented by the time spent on the reconfiguration phase, which is very relevant with respect to execution time. Another drawback is that it requires additional offline (on computers, for example) preparation, since those modules have to be identified and properly scheduled.
The main pro is instead that with reconfiguration we can implement applications that would be big and would need bigger and more expensive devices.

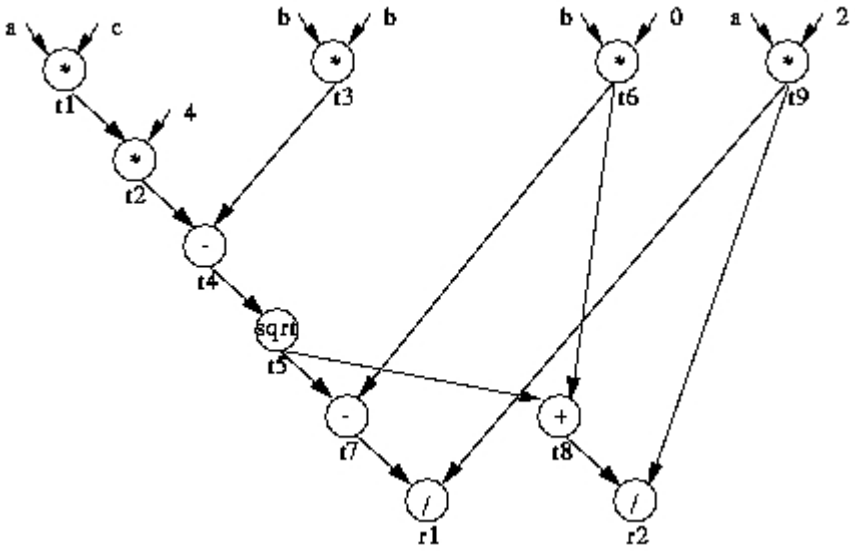

Figure 2: An example of a DFG

In order to do that we need a way to outline the operations that have to be performed and the dependencies (mainly RAW - read after write - dependencies) among instructions; graphs seem the most easy and adapt instruments we can use for this concern.

2.2 Data Flow Graph

A Data Flow Graph (DFG) is defined as

$$G = \langle O, P \rangle \tag{1}$$

where O is the set of operations present in the original specification and P are the edges connecting the various nodes in the graph and represent data dependencies between the two instructions they connect. Tha DFGs we're going to consider are DAGs (*Directed Acyclic Graphs*); they must be directed because dependencies are not commutative while for simplicity reasons where assuming them to be also acyclic, for the time being. Moreover, each DFG has an entry node (from which input parameters are derived) and an exit node where the return data is sent; in this way we cannot obtained not connected graphs, even when no depencies are present.

In Figure 2 is presented an example of a Data Flow Graph.

In addition to this, a function α has been defined as follow:

$$\alpha : O \rightarrow A \tag{2}$$

where A is the set of functional unit types, such that $\alpha(o)$ provides the functional unit type, k associated to a given vertex $o \in O$; the cardinality of A is:

$$|A| = \sum_{1}^{n} FU_k \tag{3}$$

where

- k is the index used to identify a functional unit type.
- FU_k represents the number of functional unity of type k, $FU_k \in N$.

Equation (3) is correct if holds $\bigcap_{i...n} FU_k$ otherwise it becomes

$$|A| = \left| \bigcup_{i...n} FU_k \right| \tag{4}$$

The scenario behind Equation (4) is shown in Figure 3

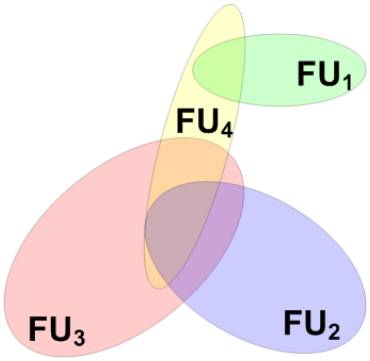

Figure 3: Scenario Overview: Overlapping Functional Unit Type

Moreover, the numbering of inputs is modeled as a function

$$\mu : P \rightarrow N \tag{5}$$

which associates each edge $(o \rightarrow o')$, where $\alpha(o')$ is a non-commutative operation, to the index of the input which it provides. It will be assumed that this information is also accessible in a way that lets us find the ancestor of a node providing the input at a given index.

Due to the nature of current reconfigurable devices, it is not technologically feasible, nor probably it would be advantageous, to reconfigure logic portions as small as those needed to implement a *single* operation from the

specification. We then want to identify portions of the specification, i.e. *subgraphs*, which will constitute our processing elements, or *cores*.

Let us then introduce the concept of *node-induced subgraph*:

Definition 2.2 (node-induced subgraph) *A subgraph*

$$S = \langle O_S, P_S \rangle \tag{6}$$

of the graph, defined in (1),

$$G = \langle O, P \rangle$$

is defined by its vertex set $O_S \subset O$, with its edge set subsequently defined as $P_S = P \cap (O_S \times O_S)$.

As a result of our partitioning phase, we want to obtain a collection of n subgraphs $S_1, \ldots, S_n$, such that

$$\left(\bigcup_{i \ldots n} S_i \right) = O \tag{7}$$

.

Each of these subgraphs represents a *core*, i.e. a processing unit that will run independently of others to execute the operations represented by the nodes contained in the subset.

Our next step is to *collapse* the original graph so that nodes belonging to the same subgraph will now be aggregated as a single new vertex. By doing so, we come up with a new intermediate representation, which takes the form of a task graph T, having the previously identified subgraphs as its vertices, defined as follows:

$$T = \langle O_T = \{S_1, \ldots, S_n\}, P_T \subset (O_T \times O_T) \rangle \tag{8}$$

with the edge set constructed as

$$\forall\, (o \rightarrow o') \in P, ((o \in S_j) \wedge (o' \in S_k) \wedge j \neq k \iff (S_j \rightarrow S_k) \in P_T) \tag{9}$$

Dependencies in the original specification are all respected if we execute the task graph nodes according to the precedences among them that we have just defined. In fact, this ensures the correctness of the execution but might force a suboptimal schedule by enforcing *unnecessary* precedences, depending on the structure of the identified subgraphs [**?**].

It easy to see that there is not only one way to obtain the T graph defined in (8). There should be more than one policy that can be used to drive the partitioner into the identification of a collection of the n subgraphs $S_1, \ldots, S_n$ subject to (7), that means that given an input specification G there are several, i.e. m, feasible solution $T_1, T_2, ..., T_m$ for the some partitioning problem. According to this observation it is necessary to define an objective function or, a methodology, that can be used to measure the goodness of a solution T_i given an input specification G.

2.3 Isomorphic Partitioning

As already said it would not be technologically feasible nor advantageous to reconfigure every single node of the DFG; the idea is therefore to identify common subgraphs in the original Data Flow Graph: this is called *Partitioning*. There exists many algorithms and metrics to perform graph partitioning (such as *temporal partitioning* or *spatial partitioning*) but the one used in this project is based on isomorphism. Let's now give some definitions:

Definition 2.3 (Isomorphic graphs) *Two directed graphs* $G_1 = \langle O_1, P_1 \rangle$ *and* $G_2 = \langle O_2, P_2 \rangle$ *are said to be* isomorphic *if there exists a bijection* $p : O_1 \to O_2$ *s.t.*

$$(o \to o') \in P_1 \iff (p(o) \to p(o')) \in P_2$$

Definition 2.4 (Isomorphic data flow graphs) *Two data flow graphs* $G_1 = \langle O_1, P_1 \rangle$ *and* $G_2 = \langle O_2, P_2 \rangle$ *are isomorphic if they satisfy definition 2.3 and, in addition, the following hold:*

$$\forall o \in O_1, \alpha(o) = \alpha(p(o))$$

and

$$\forall (o \to o') \in P_1, \mu(o \to o') = \mu(p(o) \to p(o'))$$

where α *and* μ *are defined as in (2) and in (5).*

Now that we have defined what information we want to extract from our specification, we need to find an efficient way of doing it. In literature, the problem we are trying to tackle, i.e. recognizing isomorphic subgraphs inside a single graph, is named the Isomorphic Subgraphs problem, and is defined as follows:

Definition 2.5 (Isomorphic Subgraphs) *Given a graph* G*, find two disjoing isomorphic subgraphs* G_1*,* G_2 *of* G*.*

The practical way to do this is not part of this project.

2.4 Occupation and Evaluation Metrics

Once we have identified all the isomorphic subgraphs within a given DFG, we need a way to choose which, among them, are the best candidates to perform reconfiguration. It means that need a practical rule to assign a score to every template.
The most common and simple idea is to compute the total space that would be occupied by each template and take a decision on which of them is the most suitable for reconfiguration (*Template Choice*). In Figure 4 is presented a list of possible metrics that can be used to estimate the validity of a template. The one we're going to consider is a subset of the *Frame Occupancy Estimation Metrics*: with this metric every template is associated to a value indicating the number of CLBs (or frames, but we can easily move from one unit of measurement to the other, depending on the device) that it will use in the target FPGA.
As depicted in the figure, each of those estimators (CLB Usage Estimation and Communication Overhead Estimation) has three different behaviors that can be adopted: M1 and M2 estimators use some algorithm to compute the frame occupancy while Const simply sums the occupancy of each component needed.
This means, for example, that an adder is split into its logic ports and all the occupations are simply summed. In order to make this estimation less "'rough"' an overhead due to the communication infrastructure is added (this can be a constant factor or a more precise value). This kind of estimation, even if rough, is widely used but the frame occupation of each operation is not calculated in the way described above, since it would be a long and error prone process, tough it would be more precise. The idea is instead to get those values by experimental results, as performed in the thesis work by Marco Magnone. Those results are reported in Table 1, where N is the dimension in bytes of the data, i is the number of inputs (of the multiplexer) and A is the FAN-IN: the CLB estimation is the one that is going to be used throughout the whole project.

Small nodes are not the only ones that can lead to inefficiency if reconfigured: also nodes that are too big (over a given threshold, usually) are not the best candidates. The reason is simple: if programmed on the FPGS, they are likely to prevent other modules to be on the device at same time, because of their excessive occupation. Those modules can be split into smaller blocks so that they fit on the FPGA leaving enough space for other cores. It would also be good if the sub-modules now identified were common to more than one template, in order to have less reconfiguration stages. This step is called *Candidate Choice*.

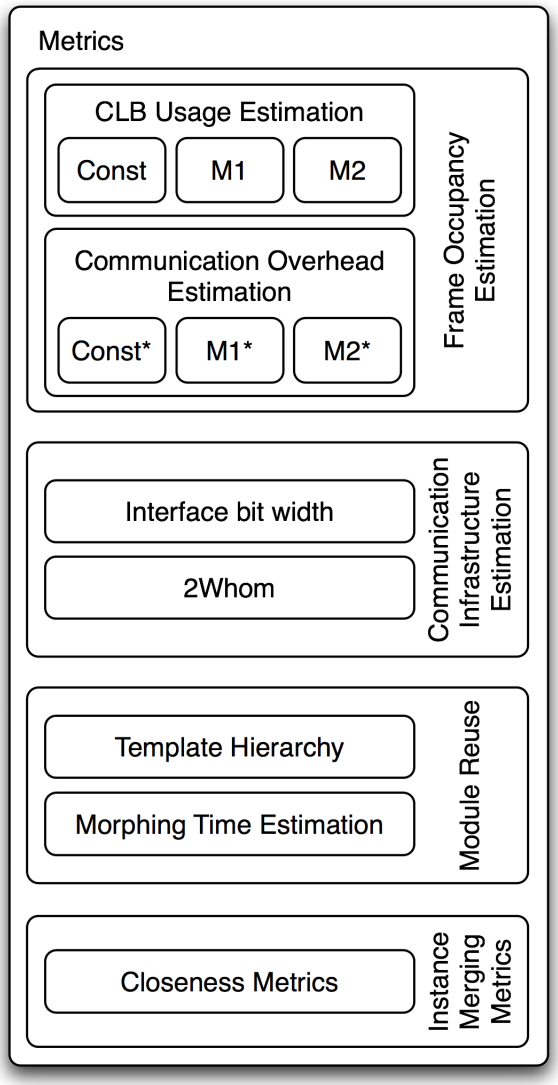

Figure 4: Metrics Definition

Component	CLB
Register (w/o Reset)	$\frac{N}{2}$
Register (w/ Reset)	$\frac{N}{2}$
Multiplexer	$\frac{A+N}{4}$
Sum / Sub	N
Counter	$\frac{N}{2}$
Comparator ($<$ or $>$)	$\frac{N}{2}$
Comparator ($\leq$ or $\geq$)	$\frac{N}{2}$+1
Comparator ($=$)	$\frac{N}{4}$
Comparator ($\neq$)	$\frac{N}{4}$+1
Multiplier	$\frac{N2}{2}$ if (N1=1) N1*N2 if (1<N1<16) $\frac{(N1+N2)^2}{4}$ if (N1>16, 1≤N2<50) N1*N2 otherwise
Logical operator (AND, OR, NAND, NOR)	$\frac{N}{2}$
Logical operator (XOR, XNOR)	$\frac{N}{2}$ if (2≤N≤ 4) N otherwise

Table 1: Operators Occupancy

3 The proposed methodology

As previously said in Section 1 this work was developed within the DRESD project, whose workflow is represented in Figure 1. To be precise, it is inserted in the *Partitioning Phase*, outlined in Figure 5.
The partitioning procedure starts from a DFG graph (output of the previous phase of *Graph Generation*) and performs three operations, in order to give as output a set of templates with informations on space occupation, graph coverage and density.
The three phases are:

Template Generation : finds all the templates available in the graph with no informations regarding their relationship

Template Choice : a Template Hierarchy Graph is built, also using metrics to compute the space occupancy of each template

Template Growth : graph coverage is performed and templates are expanded to cover all the nodes of the graph

In Section 3.1 the problem definition is presented, while Sections 3.2 and 3.3 describe the methodology adopted to solve respectively the Template

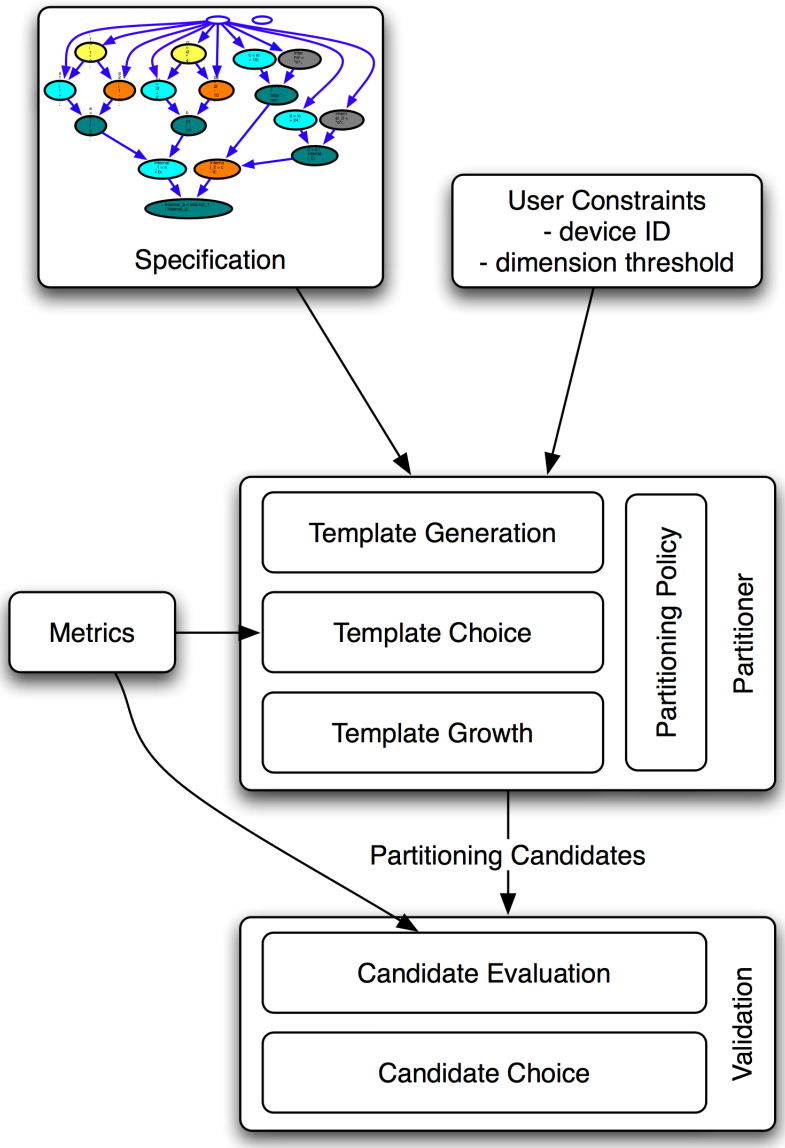

Figure 5: Partitioner Working Scheme

Choice and Template Hierarchy Graph constuction problems. Section 3.4 finally describes the updates needed to make this project properly implemented and working; at last some final remarks are added.

3.1 Problem Definition

The aim of this project is to solve the Template and Candidate Choice.
The software solution has to be integrated in the Isomorphic Partitioner created by Matteo Giani and receives as input a set of possible templates for a given Data Flow Graph (those template are of size two or more). It has to perform Template Choice evaluating the frame occupations (in percentage, that is over the total number of frames) of every template and discard those whose occupation is below a given threshold; this limit represent the percentage of total frames below which nodes are too small to be good candidates for reconfiguration (because of high reconfiguration latencies and the problems depicted before). At this point Candidate Choice has to be performed, keeping in mind the improvements described above: for every couple of templates we have to identify a possible common isomorphic subgraph that satisfies a second condition. This condition is that they're size must be over another threshold, calculated multiplying the higher threshold by a constant a ($0<a<1$) given as input; the reason for this is that we don't need subgraphs that are too small, in order to avoid too many reconfigurations phases and consequent losses of performances and increasing latency timings. In the next subsections is described in detail how the two phases described above have been implemented. Throughout the description of this work we'll use the following source code:

```
01.   int test_code( int io, int * o1, int i1, int i2 )
02.   {
03.       int a1, b1, a2, b2, a, b, c, d;
04.
05.       i1 *= 2;
06.       a1 = i1 + 1;
07.       a2 = i1 - 4;
08.       a = a1 / a2;
09.
10.       i2 *= 4;
11.       b1 = i2 + 2;
12.       b2 = i2 - 10;
13.       b = b1 / b2;
14.
```

```
15.       c = io + 16;
16.       c = c / *o1;
17.
18.       d = io + 24;
19.       d = d / *o1;
20.
21.       return (a+b) / (c-d);
22.   }
```

It gets 4 parameters as input and declares 8 other integers (internal vars). Notice that in the code there isn't any loop but it is quite linear, to avoid cycles in the Data Flow Graph. Let's now try to identify all the RAW data dependencies that will be outlined by the resulting DFG:

```
i1 -> a1, a2
a1, a2 -> a

i2 -> b1, b2
b1, b2 -> b

c -> c

d -> d
```

In lines 15 and 16 (as well as in 18 and 19) there is a WAW -write after write- dependency: this is solved by simply renaming the variables and using Internal names to have them defined (Internal_#, usually). Once this WAW dependency has been solved, only the RAW one remains and it's pointed out in the graph. The resulting Data Flow Graph will be the one in Figure 6: the Entry node is the common ancestor of the tree while the Exit node gets the result of the function (it works exactly as an exit point).

Running the IsoMorphic Partitioner by Matteo Giani on the previous example we obtain a set of templates ordered by cardinality (they're obtained incrementally): there are three different templates of size 2, five of size 3 and one of size 4, as it can be seen in the following caption from the output of the partitioner itself. These are the templates we're going to work on.

```
[AGGR]: ********************** STATS ***************************
[AGGR]: Number of identified template cardinality sets: 3
[AGGR]: Number of instances for each template, by size:
[AGGR]: Size 2 ( 10.5263% of total vertices ) - 3 templates:
```

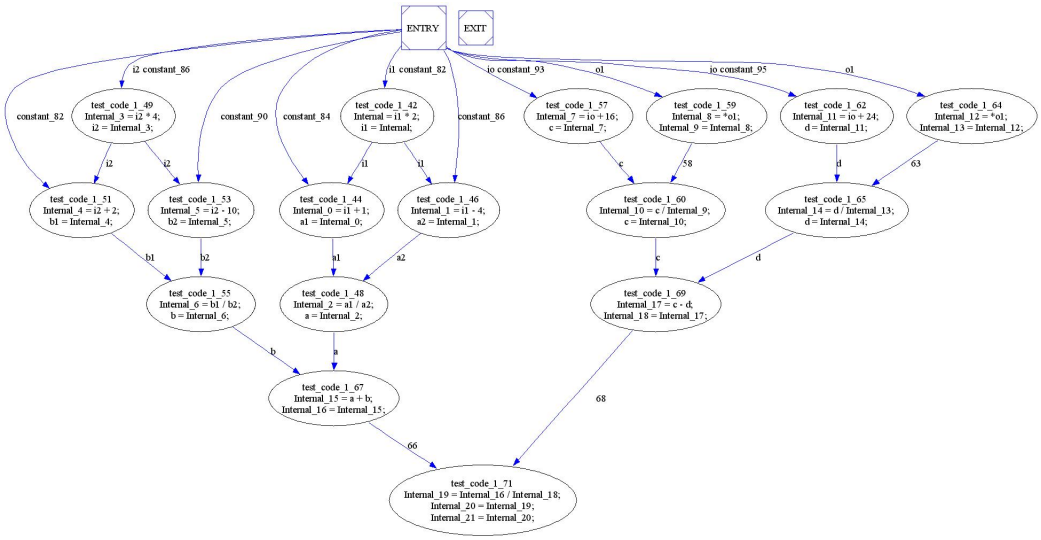

Figure 6: Data Flow Graph of the example code

```
[...]
[AGGR]: Size 3 ( 15.7895% of total vertices ) - 5 templates:
[...]
[AGGR]: Size 4 ( 21.0526% of total vertices ) - 1 templates:
```

3.2 Template Choice

The first phase of our work is to choose which templates, among all the one that have been identified, are more suitable for reconfiguration. So we have to use the metrics described in Section 2 to compute the occupation of each template; this value is then used to give a percentage of the total area of the FPGA that will be taken. The total area available on a device is a parameter depending on the device itself and it's expressed in frames. This is consequence of the architecture of the FPGA: the logical unit on the FPGA is the CLB (configurable logic blocks) and it has a fixed dimension in terms of frame. Here are some of the parameters available for each device (in the example below we're referring to the XC2VP7):

```
device_name XC2VP7;
number_of_frame 1920;
CLBxColumn 40;
CLBxRow 34;
slicexCLB 4;
LUTxslice 2;
```

```
TBUFxCLB 2;
```

We're only concerned with number_of_frame, CLBxColumn and CLBxRow: the first is the total number of frames on the FPGA (used to calculate the percentage), the second and the third represent the number of CLB present respectively on a column and a row (imagine the programmable area as a grid). In order to convert CLBs into frames we need another information, common to all the devices: a column is 48 frames wide; this means that a CLB has a width of exactly 48 frames. Now that we have all the information needed, we can estimate (it's always an estimation) the total number of frames needed by every template: we simply have to count the total CLBs used, referring to the operation implemented in the node and to the occupancies in Table 1, convert them into frames and compute the total ratio. The first step, therefore, is to get a template and, for each node it contains, sum up all the CLB occupancies of the operations. This procedure returns a CLB occupation estimation: this value, anyway, is quite "'rough"' since we're not taking into account yet the space needed for communication purposes, or to interconnect the various operators. That is, we're discarding what in Figure 4 is called *Communication Overhead Estimation*. We can bypass this problem by simply multiplying the total CLB count by a constant factor (usually 1.25 is used) to better approximate the real occupation.

Now we simply have to find out how many columns we need and multiply that value by the number of frames for each column (48) to get the total number of frames used. Getting the percentage from here is trivial. Running the estimator on the same graph as before we get the following results:

```
Occupancy of templates of size: 2
Template CLB count:7
Template occupancy: 2.5%
Template CLB count:7
Template occupancy: 2.5%
Template CLB count:2
Template occupancy: 2.5%
Occupancy of templates of size: 3
Template CLB count:12
Template occupancy: 2.5%
Template CLB count:10
Template occupancy: 2.5%
Template CLB count:7
Template occupancy: 2.5%
Template CLB count:10
```

```
Template occupancy: 2.5%
Template CLB count:12
Template occupancy: 2.5%
Occupancy of templates of size: 4
Template CLB count:15
Template occupancy: 2.5%
```

Another value that will be important in evaluating a template and choosing which are the best candidates is the *density*. In reconfiguring a device the least unit that can be written is a column (48 frames wide); assuming therefore that a column is composed of 34 CLBs, we have that cores built on 36 and 60 CLBs have the same occupancy on the FPGA.
It is anyway better to optimize how the space is used and justify the time spent on reconfiguration: the former of those template has a density lower than the latter and it's therefore more suitable for expansion during the template growth, since the aim is avoiding the use of another column on the device for reconfiguration. A way to decide on this issue is assigning a value to each template stating how much of the last column will be taken: this value is called *density*.
In our code it will be computed by simply dividing the number of used CLBs in the last column by the total number of CLBs in a column; this gives the percentage of space occupied in the last column needed.

3.3 Template Hierarchy Graph

Now that a way to compute the graph / module occupancy has been defined, it is possible to decide (using a given threshold) what are the best candidate templates for reconfiguration. Before explaining how Candidate Evaluation and Choice work, we have to look at the existing partitioner to see where to fill in our code. The isomorphic partitioner by Matteo Giani has two main work phases:
1. *templates identification*
2. *template growth* (also known as *graph coverage*)

The template choice and THG construction phases should be performed between those two steps since its output will be needed to have informations on how to optimize the template growth and graph coverage; in this sense some data on the scheduling would be very useful as well.
The *Template Hierarchy Graph* construction is performed in the three following steps, which will be shortly explained afterwards:

1. THG filling

2. Dependency identification

3. Postprocessing and pruning

3.3.1 THG Filling

The aim of this phase is to simply populate the THG with all the templates identified by the partitioner and stored in its own data structure; no dependencies and relationship among the newly inserted nodes are added, so our graph will contain only nodes with no edges.
The reason for this phase to be performed is to detach as much as possible the construction of the THG from the data structures used and created by the partitioner, so that the following phases will be completely independent by them. Another pro of this approach is that informations on templates such as occupancy, graph coverage, density, etc are computed only once at this point and are efficiently stored in the nodes in the THG. The way this operation is performed is quite trivial: the set of templates created in the *Template Identification* phase is simply visited and each entry is copied in the THG structure.

3.3.2 Dependency Identification

Now that the THG is completely filled and contains all the nodes it needs, dependencies have to be found. A dependency, as said before, is a parent-children relation in which the child derives from the parent by expansion; that is, the children contain all the nodes of the parents and at least one more. The algorithm works as follows: for each template couple (T_{Big}, T_{Small}) some conditions are first checked. Those conditions are:

- T_{Big}'s number of instances must be less or equal than T_{Small}'s.
- The number of vertices of one of T_{Big}'s instances must be strictly less than the number of vertices of one of T_{Small}'s instances.

If one or both the previous conditions is not met, it is impossible for T_{Small} to be an ancestor of T_{Big} in the THG so the algorithm will not process the specified couple.
Otherwise it works as follows:

At this point the THG will contain all the nodes and relative edges but with some redundancies that have to pruned. The next section will describe how to detect and solve them.

Algorithm 1 $ProcessTHGPair(T_{Big}, T_{Small})$

$I_{Big} \leftarrow instance\ of\ T_{Big}$
for all $instance\ I_h \in T_{Big}$ **do**
  $found\_vertices \leftarrow 0$
  for all $vertex\ V_k \in I_h$ **do**
    if $V_k \in I_{Big}$ **then**
      $found\_vertices \leftarrow found\_vertices + 1$
    end if
  end for
  if $cardinality\ of\ I_h == found\_vertices$ **then**
    $create\ edge\ T_{Small} \rightarrow T_{Big}$
    return
  end if
end for

3.3.3 Postprocessing and Pruning

There are two different problems in the THG as it is now.
The first one is the presence of *Transitive Edges*: as shown in Figure 7(a), during *Dependency Identification*, if the dependency T1 ← T2 ← T3 is found, the dependency T1 ← T3 will be found as well. The latter is obviously redundant and can thus be removed; the result of such an operation is depicted in Figure 7(b).

Algorithm 2 $RemoveTransitiveEdges(THG)$

$pruned\_edges \leftarrow \oslash$
for all $vertex\ V_h \in THG$ **do**
  for all $vertex\ V_k \in adjacent\_vertices(V_h)$ **do**
    for all $vertex\ V_j \in adjacent\_vertices(V_k)$ **do**
      if $\exists edge\ V_h \rightarrow V_j$ **then**
        $add\ \langle V_h, V_j \rangle\ to\ pruned\_edges$
      end if
    end for
  end for
end for
for all $\langle V_h, V_j \rangle \in pruned\_edges$ **do**
  $remove\ edge\ V_h \rightarrow V_j$
end for

The second kind of redundancy is due to nodes with just one outgoing edge, as shown in Figure 8(a). These nodes represent templates with just

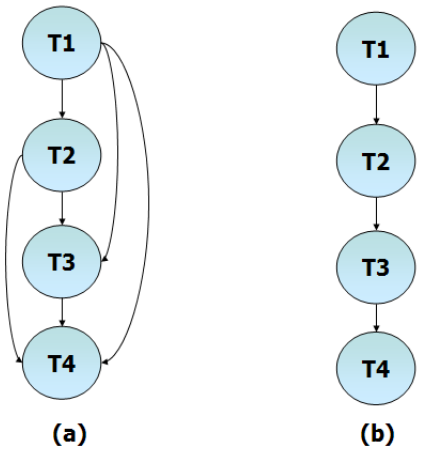

Figure 7: Transitive Edges identification and elimination

one child relation, that is, they do not represent subgraphs common to two or more templates. Due to this reason, this kind of nodes does not gain any additional or useful information in the THG and can be pruned away, as Figure 8(b) shows.

Algorithm 3 RemoveNodeChains(THG)

repeat
  $done \leftarrow TRUE$
  for all *vertex* $V_h \in THG$ **do**
    if $|adjacent\_vertices(V_h)| == 1$ **then**
      $done \leftarrow FALSE$
      $V_{next} \leftarrow adjacent\_vertex(V_h)$
      for all *vertex* $V_k \in inverse\_adjacent\_vertices(V_h)$ **do**
        create edge $V_k \rightarrow V_{next}$
      end for
      remove vertex V_h *from* THG
    end if
  end for
until *not done*

3.4 DRESD Updates

As said when defining the problem, some parameters are given as input in order to make some choices on which templates can be discarded and, on

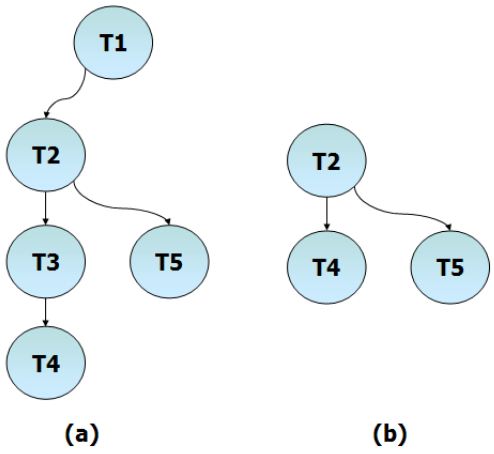

Figure 8: Single-dependency nodes elimination

the other hand, what are the best candidates for reconfiguration; these parameters are the *Occupancy Threshold*, the constant factor to get the lower threshold and the device used. The user must therefore be made able to provide these values at prompt line, exactly in the same way he can decide on what partition algorithm to use: some modifications to the original executable file are so required. First of all we have to choose the proper strings for the options to be defined: for the Occupancy Threshold the option $--$*occupancy-threshold* (-o) can be good, as well as $--$*lower-threshold-constant* (-l) and $--$*device-name* (-d) for the other two parameters. The following code fragment shows how these options were made available in the executable file:

```
const struct option long_options[] =
{
        [...]
        {"occupancy-threshold", required_argument, 0, 'o'},
        {"lower-threshold-constant" , required_argument, 0, 'l'},
        {"device-name", required_argument, 0, 'n'},
        [...]
        { 0, 0, 0, 0}
};
```

Those input parameters are then parsed and converted into integers and floating point numbers where necessary; if an error occurs or their values are out of the defined boundaries ((0, 100] for the occupancy and (0, 1] for the constant) they're simply put to their default values. They are 80%, 0.5 and

"'XC2VP7"' respectively. Now when calling the DRESD executable file the three new parameters can be specified and the output looks as follows:

```
$ ./dresd -a bachl -p test_code -o 72 -l .3 -n XC2VP7 [file]
User requested partitioning algorithm: "bachl"
Reconfigurable system partitioner
User requested template choice occupancy threshold: 72%
User requested candidate choice occupancy constant: .3
User requested device: XC2VP7
Using device XC2VP7 with occupancy threshold = 72% and
                              lower threshold constant = 0.3
...
```

3.5 Final Remarks

The aim of this project was to create and implement the instruments that will help the designer in taking some decisions during the following phases of the design of the reconfigurable architecture. The occupancy metrics and the THG can prove useful when having to choose what are the templates that can minimize the reconfiguration time, that is in the *Scheduling and Allocation* phase.
Therefore it would be difficult to show the results of just this work, except for some generated THGs, since every result must be properly evaluated in the following phases and no considerations can be done for the time being. One of the improvement that will be probably implemented in the near future is the study and refinement of the metrics adopted, maybe taking into account not only the operations present in a template but also the operation dependencies among different templates (to maximize the reuse of the resources over time, for example).