A Graph Covering Algorithm for a Coarse Grain Reconfigurable System

Yuanqing Guo
yguo@cs.utwente.nl

Gerard J.M. Smit
smit@cs.utwente.nl

Hajo Broersma
broersma@cs.utwente.nl

Paul M. Heysters
heysters@cs.utwente.nl

Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente, P.O. Box 217, 7500AE Enschede, The Netherlands

ABSTRACT

The availability of high-level design entry tooling is crucial for the viability of any reconfigurable SoC architecture. This paper presents a graph covering algorithm. The graph covering is done in two steps: template generation and template selection. The objective of template generation step is to extract functional equivalent structures, i.e. templates, from a control data flow graph. By inspecting the graph, the algorithm generates all the possible templates and the corresponding matches. Using unique serial numbers and circle numbers, the algorithm can find all distinct templates with multiple outputs. The template selection algorithm shows how this information can be used in compilers for reconfigurable systems. The objective of the template selection algorithm is to find an efficient cover for an application graph with a minimal number of distinct templates and minimal number of matches.

Categories and Subject Descriptors

B.1.4 [**Hardware**]: CONTROL STRUCTURES AND MICROPROGRAMMING—*Microprogram Design Aids, Languages and compilers*; D.3.4 [**Software**]: PROGRAMMING LANGUAGES—*Processors, Code generation*

General Terms

Algorithms

Keywords

Reconfigurable system, template generation, compiler

LCTES'03, June 11–13, 2003, San Diego, California, USA.

1. INTRODUCTION

In the CHAMELEON/GECKO[1] project a heterogeneous system on chip (SoC) for handheld multimedia devices is being designed [16]. The structure contains a general-purpose processor (i.e. an ARM core), a fine-grained reconfigurable part (consisting of FPGA tiles) and a course-grained reconfigurable part. The latter comprises several MONTIUM processor tiles. The algorithm domain of the MONTIUM comprises 16-bit digital signal processing (DSP) algorithms that contain multiply accumulate (MAC) operations such as FFT, FIR and linear interpolation. However, the application domain of MONTIUM is not limited to these algorithms. A MONTIUM tile is designed to execute highly regular computational intensive DSP kernels. The irregular parts of the algorithm run on the general-purpose processor.

The availability of high-level design entry tooling is critical for the viability of any reconfigurable architecture. The architecture of the MONTIUM is kept simple and regular in order to bound the complexity of a compiler. Currently we are in the process of implementing a C compiler for the MONTIUM architecture. In our toolset the input language C is first translated into a Control Data Flow Graph (CDFG). In general, CDFGs are not acyclic. In the first phase we decompose the general CDFG into acyclic blocks and cyclic control information. In this paper we only consider acyclic graphs. A directed acyclic CDFG is a graph that represents the operations (for example C operators and function calls) and the dataflow between those operations. The data in the CDFG includes operands of mathematical operations, the state-space (abstraction of the memory) and control information which is used, to control the iteration and selection statements. After translating the C source code to a CDFG, this graph is minimized using a set of behaviour preserving transformations such as dependency analysis, common subexpression elimination. Next, the primitive operations (such as additions, subtractions, multiplications) are partitioned into clusters, such that each cluster can be executed by a MONTIUM-tile within one clock cycle. Due to the limitation of the configuration space, the number of configurations should be as small as possible [15]. After applying graph clustering, scheduling and allocation transformations on this

minimized graph, the configurations for the MONTIUM can be generated.

The clustering phase is implemented by a graph covering algorithm. The procedure of clustering is the procedure of finding a cover for a CDFG. The graph covering algorithm includes two steps: template generation and template selection. By analyzing the CDFGs, our template generation algorithm can indicate which templates are most frequently used. The template selection algorithm tries to find an efficient cover for an application graph with a minimal number of distinct templates and minimal number of matches. Distinct templates correspond to distinct ALU configurations and selected matches correspond to clusters.

Although our primary target architecture is the MONTIUM processor, we believe that these techniques can also be used for designing programming field programmable gate arrays (FPGA) or logic circuits as shown in [2][4][5][6][13][14].

In this paper first a short overview of the MONTIUM architecture is given. After that, the template generation and selection algorithms are presented.

2. TARGET ARCHITECTURE: MONTIUM

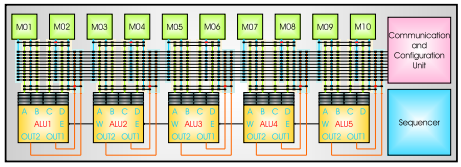

Figure 1: MONTIUM processor tile

In this section we give a brief overview of the MONTIUM architecture, because this architecture led to the research questions and the algorithms presented in this paper. Details of the MONTIUM architecture can be found in [16]. Figure 1 depicts a single MONTIUM processor tile. The hardware organisation within a tile is very regular and resembles a very long instruction word (VLIW) architecture. The five identical arithmetic and logic units (ALU1$\cdots$ALU5) in a tile can exploit spatial concurrency to enhance performance. This parallelism demands a very high memory bandwidth, which is obtained by having 10 local memories (M01$\cdots$M10) in parallel. The small local memories are also motivated by the locality of reference principle. The ALU input registers provide an even more local level of storage. Locality of reference is one of the guiding principles applied to obtain energy-efficiency in the MONTIUM. A vertical segment that contains one ALU together with its associated input register files, a part of the interconnect and two local memories is called a processing part (PP). The five processing parts together are called the processing part array (PPA). A relatively simple sequencer controls the entire PPA. The communication and configuration unit (CCU) implements the interface with the world outside the tile. The MONTIUM has a datapath width of 16-bits and supports both integer and fixed-point arithmetic. Each local SRAM is 16-bit wide and has a depth of 512 positions, which adds up to a storage capacity of 8 Kbit per local memory. A memory has only a single address port that is used for both reading and writing. A reconfigurable address generation unit (AGU) accompanies each memory. The AGU contains an address register that can be modified using base and modify registers.

It is also possible to use the memory as a lookup table for complicated functions that cannot be calculated using an ALU, such as sinus or division (with one constant). A memory can be used for both integer and fixed-point lookups. The interconnect provides flexible routing within a tile. The configuration of the interconnect can change every clock cycle. There are ten busses that are used for inter-PPA communication. Note that the span of these busses is only the PPA within a single tile. The CCU is also connected to the global busses. The CCU uses the global busses to access the local memories and to handle data in streaming algorithms. Communication within a PP uses the more energy-efficient local busses. A single ALU has four 16-bit inputs. Each input has a private input register file that can store up to four operands. The input register file cannot be bypassed, i.e., an operand is always read from an input register. Input registers can be written by various sources via a flexible interconnect. An ALU has two 16-bit outputs, which are connected to the interconnect. The ALU is entirely combinatorial and consequentially there are no pipeline registers within the ALU. The diagram of the MONTIUM ALU in Figure 2 identifies two different levels in the ALU. Level 1 contains four function units. A function unit implements the general arithmetic and logic operations that are available in languages like C (except multiplication and division). Level 2 contains the MAC unit and is optimised for algorithms such as FFT and FIR. Levels can be bypassed (in software) when they are not needed.

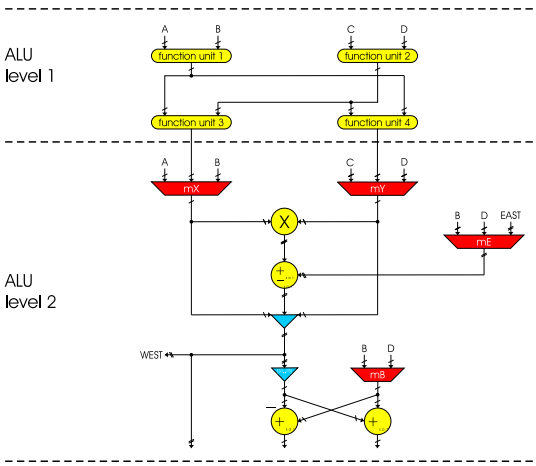

Figure 2: MONTIUM ALU

Neighboring ALUs can also communicate directly on level 2. The West-output of an ALU connects to the East-input of the ALU neighboring on the left (the West-output of the leftmost ALU is not connected and the East-input of the rightmost ALU is always zero). The 32-bit wide East-West connection makes it possible to accumulate the MAC result of the right neighbor to the multiplier result (note that this is also a MAC operation). This is particularly useful when performing a complex multiplication, or when adding up a large amount of numbers (up to 20 in one clock cycle). The East-West connection does not introduce a delay or pipeline, as it is not registered.

3. RELATED WORK

There have been published many related research efforts in the areas of high-level synthesis and FPGA logic synthesis.

In [5][7], a template library is assumed to be available and the template matching is the focus of their work. However, this assumption is not always valid, and hence an automatic compiler must determine the possible templates by itself before coming up with suitable matchings.

[2][13][14] give some methods to generate templates. These approaches choose one node as an initial template and subsequently add more operators to the template. There is no restriction on the shape of the templates. The drawback is that the generated templates are highly dependent on the choice of the initial template. The heuristic algorithm in [12] generates and maps templates simultaneously, but cannot avoid ill-fated decisions.

The algorithms in [4][6] provide all templates of a CDFG. The complete set of tree templates and single-PO (single principle output) templates are generated in [6] and all the single-sink templates (possibly multiple outputs) are found by the configuration profiling tool in [4]. The central problem for template generation algorithms is how to generate and enumerate all the (connected) subgraphs of a CDFG. The methods employed in [6] and [4] can only enumerate the subgraphs of specific shapes (tree shape, single output or single sink) and as a result, templates with multiple outputs or multiple sinks cannot be generated. In the MONTIUM architecture, each ALU has three outputs, so the existing algorithms cannot be used. In [3], an algorithm is presented to generate templates with multiple outputs. This method checks matches to find a new template. The computational complexity might be quite high when the number of matches is large. In this paper, we present an algorithm that use a labelling method to enumerate all the (connected) matches and then generate a template library.

4. PROBLEM FORMULATION

For the purpose of formulating our problem in a mathematical context, it is convenient to introduce a new type of graphs called **hydragraphs**[2] to model our directed acyclic CDFGs (CDFGs for short in this paper). This concept should capture and represent the operations, the inputs and outputs, as well as which inputs are used and which outputs are produced by the operations (and which outputs of a certain operation serve as inputs for one or more further operations).

A hydragraph $G = (N_G, P_G, A_G)$ consists of two finite non-empty sets of **nodes** N_G and **ports** P_G and a set A_G of so-called **hydra-arcs**; a hydra-arc $a = (t_a, H_a)$ has one **tail** $t_a \in N_G \cup P_G$ and a non-empty set of **heads** $H_a \subset N_G \cup P_G$. In our applications, N_G represents the operations of a CDFG, P_G represents the inputs and outputs of the CDFG, while the hydra-arc (t_a, H_a) either reflects that an input is used by an operation (if $t_a \in P_G$), or that an output of the operation represented by $t_a \in N_G$ is input of the operations represented by H_a, or that this output is just an output of the CDFG (if H_a contains a port of P_G).

See the example in Figure 3: The operation of each node

[2]These graphs are named after Hydra, a water-snake from Greek mythology with many heads that grew again if cut off.

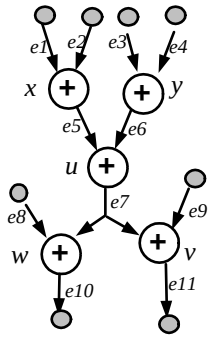

Figure 3: A small CDFG

is a basic computation such as addition (in this case), multiplication, or subtraction. Hydra-arcs are directed from their tail to their heads. Because an operand might be input for more than one operation, a hydra-arc is allowed to have multiple heads although it always has only one tail. The hydra-arc $e7$ in Figure 3, for instance, has two heads, w and v. The CDFG communicates with external systems through its ports represented by small grey circles in Figure 3.

Two distinct nodes u and v from N_G are called **neighbors** if there exists a hydra-arc (t, H) with $\{u, v\} \subset H \cup \{t\}$. These nodes are called **connected within a hydragraph** G if there exists a sequence $u_0, \ldots, u_k$ of nodes from N_G such that $u_0 = u$, $u_k = v$, and u_i and u_{i+1} are neighbors for all $i \in \{0, \ldots, k-1\}$. If u and v are connected within G, then the smallest k for which such a sequence exists is called the **distance of u and v within the hydragraph** G, denoted by $\mathrm{Dis}(u, v|G)$; the distance is 0 if $u = v$.

We call u and v are **connected within a subset** $S \subset N_G$, if there exists a sequence $u_0, \ldots, u_k$ of nodes from S such that $u_0 = u$, $u_k = v$, and u_i and u_{i+1} are neighbors for all $i \in \{0, \ldots, k-1\}$. Correspondingly, the **distance within a subset** S is defined, denoted by $\mathrm{Dis}(u, v|S)$.

A subset S of the nodes of a hydragraph is called **connected** if all pairs of distinct elements from S are connected within S. A hydragraph is called **connected** if all pairs of distinct elements from N_G are connected within G, i.e., if N_G is a connected set.

Let $S \subset N_G$ be a non-empty connected set of nodes of the hydragraph G. Then S generates a connected hydragraph in the following natural way:
For every $v \in S$ consider the following two types of hydra-arcs of G related to v:

- (t_v, H_v), so hydra-arcs with tail v: if $H_v \not\subset S$, we introduce a new port p_v and replace (t_v, H_v) by $(t_v, (H_v \cap S) \cup \{p_v\})$; otherwise, we keep (t_v, H_v) as it is.
- (t_u, H_u) with $v \in H_u$, so hydra-arcs for which v is one of the heads: if $t_u \notin S$, we introduce a new port t'_u and replace (t_u, H_u) by $(t'_u, H_u \cap S)$; otherwise we keep (t_u, H_u) as it is.

Doing so for all hydra-arcs, e.g. starting from the sources in S, we obtain a unique hydragraph which we will refer to as the **template** generated by S in G. We denote it by $T_G[S]$ and say that S is a **match** of the template $T_G[S]$. In the sequel we will only consider connected templates without always stating this explicitly.

For example, in Figure 4 we see two templates of the

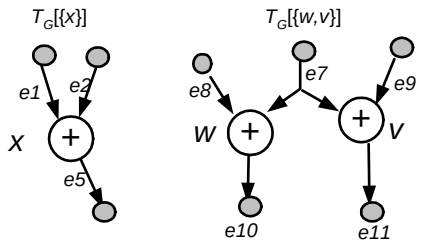

Figure 4: Two generated templates from Figure 3

CDFG from Figure 3: the left one is generated by the set $\{x\}$, the right one by $\{v, w\}$. Compared with the original CDFG from Figure 3, in the left one, the newly added port is a head for hydra-arc $e5$, while in the right one the newly added port is a tail for hydra-arc $e7$.

Two hydragraphs G and F are said to be **isomorphic** if there is a bijection $\phi : N_G \cup P_G \rightarrow N_F \cup P_F$ such that: $\phi(N_G) = N_F$, $\phi(P_G) = P_F$, and $(t_v, H_v) \in A_G$ if and only if $(\phi(t_v), \phi(H_v)) \in A_F$.
We use $G \cong F$ to denote that G and F are isomorphic.

We say that $S' \subset N_G$ is a **match for the template** $T_G[S]$ if $T_G[S'] \cong T_G[S]$. A hydragraph H is a **template of the hydragraph** G if, for some $S \subset N_G$, $T_G[S] \cong H$. Of course, the same template could have different matches in G. As an example, Figure 5 shows all the connected templates and matches of up to two nodes generated from Figure 3.

Note that, in general, a template is not a subhydragraph of a hydragraph, because some nodes may have been replaced by ports. Also note that every template of a proper hydragraph is again a proper hydragraph, since we introduce a (source or sink) port whenever a node of S was a source or sink in the subhydragraph of G induced by S.

The important property of templates of a CDFG is that they are themselves CDFGs that model part of the algorithm modeled by the whole CDFG: the template $T_G[S]$ models the part of the algorithm characterized by the operations represented by the nodes of S, together with the inputs and outputs of that part.

Because of this property, templates are the natural objects to consider if one wants to break up a large algorithm represented by a CDFG into smaller parts that have to be executed on ALUs.

In order to be able to schedule and execute the algorithm represented by the CDFG in as few clock cycles on the ALU-architecture as possible, in our applications it is considered preferable to break up the CDFG in as few parts as possible and with as few different parts as possible. So we like to use a small number of rather large templates, just fitting on an ALU, and a rather small number of matches to partition the nodes of the CDFG. It is clear that we cannot expect to solve this complex optimization problem easily. We would be quite happy with a solution concept that gives approximate solutions of a reasonable quality, and that is flexible enough to allow for several solutions to choose from. For these reasons, we propose to start the search for a good solution by first generating all different matches (up to a certain number of nodes because of the restrictions set by the ALU-architecture) of nonisomorphic templates for the CDFG. To avoid unnecessarily many computations we will use a clever labelling of the nodes during the generation process we will describe in more detail in the next section.

The set of all possible matches of admissible templates will be used to serve as a starting point for an approach to solving the following problem.

We say that a collection $(T_1, \ldots, T_k)$ of hydragraphs is a **k-tiling** of the hydragraph G if there exists a partition of N_G into mutually disjoint sets $S_1, \ldots, S_k$ such that $T_G[S_i] \cong T_i$ for all $i \in \{1, \ldots, k\}$. In that case we call $S_1, \ldots, S_k$ a **k-cover** of G.

Note that we allow multiple copies of isomorphic hydragraphs in the k-tiling. Also note that all ports of G are contained in at least one of the smaller hydragraphs.

A **(k, ℓ)-tiling** is a k-tiling in which at most ℓ nonisomorphic hydragraphs appear. Similarly, we define a **(k, ℓ)-cover**.

Problem 1: Hydragraph Covering Problem
Given a CDFG G, find an optimal (k, ℓ)-cover $S_1, S_2, \ldots, S_k$ of G.

Problem 1 cannot be properly defined here because of several reasons that have partly been discussed before.

First of all, the size of the sets S_i depends on the ALU-architecture as well as the operations represented by the nodes.

Secondly, the values of k and ℓ will depend on each other: an optimal value for one could imply a bad value for the other.

Thirdly, how should one define optimal in the formulation of Problem 1? The overall aim is to schedule and execute the algorithm represented by the CDFG on the ALU-architecture in as few clock cycles as possible. This aim cannot be translated in a simple statement about the values of k and ℓ in an optimal solution.

We postpone such questions to a later stage of our research project. In the sequel we first focus on generating all matches of nonisomorphic templates, i.e., on part A of the next set of problems. We will also describe a first approach to solving part B.

Problem A: Template Generation Problem
Given a CDFG, generate the complete set of nonisomorphic templates (that satisfy certain properties, e.g., which can be executed on the ALU-architecture in one clock cycle), and find all their corresponding matches.

Problem B: Template Selection Problem
Given a CDFG G and a set of (matches of) templates, find a 'good' (k, ℓ)-cover of G.

5. TEMPLATE GENERATION AND SELECTION ALGORITHMS

For convenience let us call a template (a match) an **i-template** (an **i-match**) if the number of its nodes is i. The objective of the template generation algorithm is to find all nonisomorphic i-templates with $1 \leq i \leq$*maxsize* for some predefined value *maxsize* depending on the application, and their corresponding matches from G. Given the input CDFG of Figure 3 and *maxsize*=2, the algorithm should find the templates and matches as shown in Figure 5.

A clear approach for the generating procedure is:

- Generate a set of connected i-matches by adding a neighbor node to the $(i-1)$-matches.
- For all i-matches, consider their generated i-templates. Choose the set of nonisomorphic i-templates and list

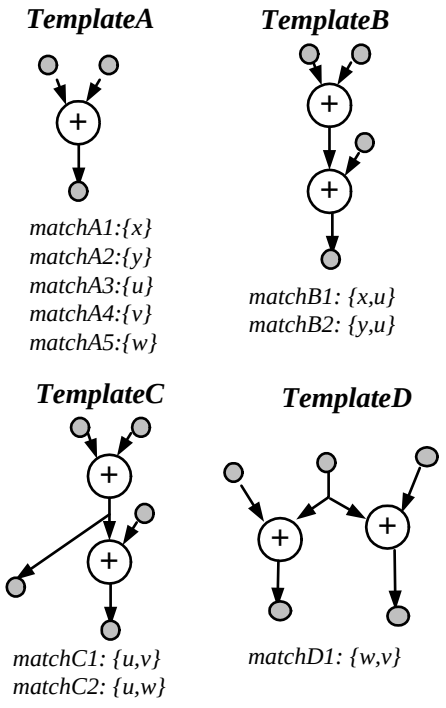

Figure 5: Templates and matches for the CDFG of Figure 3

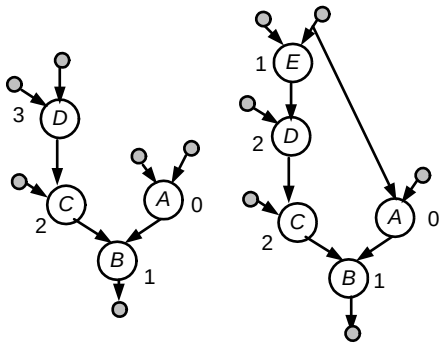

Figure 6: A node with different circle numbers in predecessor and successor matches.

all matches of each of them. (The graph isomorphism algorithm, Gemini in [8], is used here).

- Starting with the 1-templates, repeat the above steps until all templates and matches up to *maxsize* nodes have been generated.

By a clever use of a predetermined labelling of the nodes, a choice of a so-called **leading node** for each match, and using the distance of the newly added node to this leading node, we are able to save on execution time and memory during the generation procedure. Of course we cannot avoid that it is timeconsuming since G can have exponentially many (matches of) templates. We will give more details on the suggested procedure below.

The union of a connected match and one of its neighbor nodes is a larger connected set, which is a match for some larger template. The newly obtained match is called a **successor match** of the old match, and correspondingly, the old one is called a **predecessor match** of the new one. Each match has one **leading node**, which, for the case of a 1-match, is the unique hydragraph node. An i-match ($i > 1$) inherits the leading node from its predecessor match. In Figure 5 the set $\{x,u\}$, which is a match for *templateB*, is obtained when the match *matchA1:*$\{x\}$ absorbs its neighbor u. Thus, the match *matchB1:*$\{x,u\}$ is a successor match and *matchA1:*$\{x\}$ is a predecessor match of *matchB1*. The leading node for *matchA1:*$\{x\}$ is x, which is also the leading node for *matchB1:*$\{x,u\}$. On the other hand, *matchB1:*$\{x,u\}$ can be treated as a successor match for *matchA3:*$\{u\}$ (see Figure 5 and Figure 3). In this case, u is the leading node for *matchB1:*$\{x,u\}$. In short, a match might have different predecessor matches and the leading node for a match cannot be determined in a unique way without knowing all its predecessor matches. In section 5.1 we will introduce a technique to solve this.

Within a match S, each graph node $n \in S$ is given a **circle number**, denoted by Cir($n|S$), which is the distance between the leading node and n within S, i.e., Cir($n|S$) = Dis(S.LeadingNode, $n|S$). Except for the leading node, which has circle number 0, the circle numbers of other nodes might be different in the predecessor and successor matches. Suppose the match $S_1 = \{A, B, C, D\}$ is a predecessor match of the match $S_2 = \{A, B, C, D, E\}$ and their corresponding templates are those given in Figure 6. If A is the leading node of S_1 and S_2, for instance, then Cir($D|S_1$)=3 while Cir($D|S_2$)=2. In section 5.1 an algorithm is presented to skip those successor matches where the circle numbers of nodes are different in the predecessor match and the successor match. This is done to avoid the generation of multiple identical matches.

5.1 The Template Generation Algorithm

The template generation algorithm we propose is shown by the pseudo-code in Figure 7. The result of the algorithm (all the different matches of nonisomorphic templates) are stored in *TemplateList* array. The structure of *TemplateList[i]* is:

template1, match1, match2, match3,$\cdots$
template2, match1, match2, match3,$\cdots$
template3, match1, match2, match3,$\cdots$
$\cdots$

First, each node is given a unique serial number (see Figure 8). To distinguish between serial numbers and circle numbers within some specific match, the former ones are denoted by the numbers enclosed in small boxes.

1-templates are generated by single nodes of a CDFG (line 10 in Figure 7). The unions of all possible $(i-1)$-matches and all possible neighbors of those matches cover the whole i-matches space (lines 34, 35 in Figure 7). Each i-match generates a corresponding i-template (*tentativeTemplate*) (lines 11, 37 in Figure 7). All i-templates (*tentativeTemplate*s) are then checked against the templates in *TemplateList*[i] (lines 12, 38 in Figure 7). If an isomorphic template (*isomorphicTemplate*) does not exist in the list, a new template (*newTemplate*) is inserted into *TemplateList*[i] (lines 13-19, 39-45); otherwise a match (*newMatch*) for that isomorphic

```
    //Input: graph G.
    //Outputs: TemplateList[1], TemplateList[2],
      ···, TemplateList[maxsize].
    //TemplateList[i] stores i-templates and their matches.
1   Main(){
2     Give each node a unique serial number;
3     FindAllTemplatesWithOneNode(G).
4     for(i=2; i ≤ maxsize; i++){
5       FindAllTemplatesWithMoreNodes(i, G);
6     }
7   }

8   FindAllTemplatesWithOneNode(G){
9     TemplateList[1]=φ;
10    foreach node currentNode in graph G{
11      tentativeTemplate = new template (currentNode);
12      isomorphicTemplate
        =TemplateList[1].FindIsomorphicTemplate(
          tentativeTemplate);
13      if(isomorphicTemplate==φ){
14
15        newTemplate = tentativeTemplate;
16        newMatch = GenerateOneNodeMatch(currentNode);
17        add newMatch to newTemplate;
18        add newTemplate to TemplateList[1];
19      }
20      else{
21        newMatch = GenerateOneNodeMatch(currentNode);
22        add newMatch to isomorphicTemplate;
23      }
24    }
25  }

26  match GenerateOneNodeMatch (newNode){
27    newMatch ={newNode};
28    newMatch.leadingNode = newNode;
29    Cir(newNode|newMatch)=0;
30    return(newMatch);
31  }

32  FindAllTemplatesWithMoreNodes(nodes number: i, graph: G){
33    TemplateList[i]=φ;
34    foreach (i-1)-match currentMatch
35    and foreach neighbor currentNeighbor∈Nei(currentMatch) {
36      if(CanMatchTakeInNode(currentMatch, currentNeighbor)
          ==yes){
37        tentativeTemplate = new template (currentMatch
            "+" currentNeighbor);
38        isomorphicTemplate=
          TemplateList[i].FindIsomorphicTemplate(tentativeTemplate);
39        if(isomorphicTemplate== φ){
40
41          newTemplate = tentativeTemplate ;
42          newMatch
            =GenerateMoreNodesMatch(currentMatch,currentNeighbor);
43          add newMatch to newTemplate;
44          add newTemplate to TemplateList[i];
45        }
46        else{
47          newMatch
            =GenerateMoreNodesMatch(currentMatch,currentNeighbor);
48          add newMatch to isomorphicTemplate;
49        }
50      }
51    }
52  }

53  match GenerateMoreNodesMatch (oldMatch, newNode){
54    newMatch=oldMatch∪{newNode};
55    newMatch.LeadingNode=oldMatch.LeadingNode;
56    foreach n∈oldMatch, let Cir(n|newMatch)=Cir(n|oldMatch);
57    Cir(newNode|newMatch)=1+min(Cir(n|oldMatch),
        where n∈oldMatch and n is a neighbor of newNode.
58    return(newMatch);
59  }
```

Figure 7: Pseudo-code for template generation algorithm

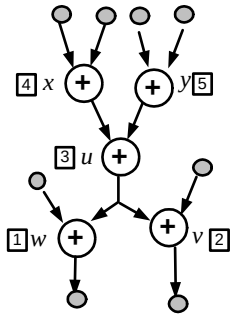

Figure 8: Give each node a unique serial number

template (*isomorphicTemplate*) is added (lines 20-23, 46-49). For a 1-match, the leading node is the only node (line 28) and its circle number is 0 (line 29). For an i-match ($i > 1$) *newMatch* (line 55), the leading node is inherited from the predecessor match *oldMatch*.

In this algorithm, multiple identical matches are avoided by the function "CanMatchTakeInNode(*oldMatch*, *newNode*)" (line 36), which, making use of the unique serial numbers and the circle numbers, filters out other predecessor matches except the one (*oldMatch*) that satisfies the following conditions:

1. *oldMatch*.LeadingNode.Serial<*newNode*.Serial;
2. Dis(*oldMatch*.LeadingNode, *newNode*| *oldMatch* ∪ {*newNode*}) is not smaller than Cir(n|*oldMatch*) for any $n \in$ *oldMatch*;
3. For every n which satisfies $n \in$*oldMatch* and Cir(n|*oldMatch*) = Dis(*oldMatch*.LeadingNode,*newNode*| *oldMatch*∪ {*newNode*}), n.Serial< *newNode*.Serial.

Condition 1 makes sure that the leading node is the one with the smallest serial number in a match; Condition 2 makes sure the newly added node *newNode* always has the largest possible circle number among all choices. The nodes for a predecessor match, as a result, keep the same circle number in the successor match (line 56); Condition 3 makes sure that *newNode* is the one with the largest serial number among the nodes with the same circle numbers. For a match *newMatch*, these conditions decide a unique pair (*oldMatch*, *newNode*) that satisfies (*newMatch*=*oldMatch*∪ {*newNode*}), where *oldMatch* corresponds with a connected $(i-1)$-template, and *newNode* is a neighbor of *oldMatch*.

In Table 1, the procedure of finding all the i-matches from $(i-1)$-matches of Figure 8 is given. The symbols in bold are the names of the leading nodes and the newly added nodes are underlined. In each row, the match in the left column is the predecessor match of the match in the right column. The matches that do not satisfy the conditions of the function "CanMatchTakeInNode" are discarded. The numbers next to the discarded matches indicate which of the above three conditions is violated.

THEOREM 1. *Given a CDFG (a hydragraph) G and an integer maxsize, the template generation algorithm of Figure 7 generates all the possible templates and their corresponding matches with at most maxsize nodes. None of the generated*

Table 1: Multiple copies of a match are filtered out by the function "CanMatchTakeInNode(*oldMatch*, *newNode*)".

1-matches	2-matches	3-matches	4-matches
{■}	~~{■,$\underline{u}$}~~ 1		
{■}	~~{■,$\underline{u}$}~~ 1		
{■}	~~{■,$\underline{w}$}~~ 1		
	~~{■,$\underline{v}$}~~ 1		
	{■,$\underline{x}$}	{■,x,$\underline{y}$}	~~{■,x,y,$\underline{w}$}~~ 1
			~~{■,x,y,$\underline{v}$}~~ 1
		~~{■,x,$\underline{w}$}~~ 1	
		~~{■,x,$\underline{v}$}~~ 1	
	{■,$\underline{y}$}	~~{■,y,$\underline{x}$}~~ 3	
		~~{■,y,$\underline{w}$}~~ 1	
		~~{■,y,$\underline{v}$}~~ 1	
{■}	{■,$\underline{u}$}	{■,u,$\underline{x}$}	{■,u,x,y}
			~~{■,u,x,$\underline{w}$}~~ 1
		{■,u,$\underline{y}$}	~~{■,u,y,$\underline{x}$}~~ 3
			~~{■,u,y,$\underline{w}$}~~ 1
		~~{■,u,$\underline{w}$}~~ 1	
	~~{■,$\underline{w}$}~~ 1		
{■}	{■,$\underline{u}$}	{■,u,$\underline{x}$}	{■,u,x,$\underline{y}$}
			~~{■,u,x,$\underline{v}$}~~ 2
		{■,u,$\underline{y}$}	~~{■,u,y,$\underline{x}$}~~ 3
			~~{■,u,y,$\underline{v}$}~~ 2
		~~{■,u,$\underline{v}$}~~ 3	
	{■,$\underline{v}$}	{■,v,$\underline{u}$}	{■,v,u,$\underline{x}$}
			{■,v,u,$\underline{y}$}

templates for G are isomorphic and no matches appear more than once.

PROOF. We use induction on the number of nodes of the matches. Clearly, the algorithms checks all 1-matches of G; the nonisomorphic 1-template are listed once, and for each 1-template, all of its 1-matches.

The induction hypothesis is that the algorithm finds all $(k-1)$-matches, grouped according to nonisomorphic $(k-1)$-templates. We want to prove that the algorithm also finds all k-matches, grouped according to nonisomorphic k-templates. For this purpose, let $S \subset N_G$ be a match of the k-template $T_G[S]$. Recall that S is supposed to be a connected set. Order its nodes $\{n_1, n_2, \cdots, n_k\}$ as follows: The first node n_1 is the node with the smallest serial number. The set of all the other nodes is ordered according to their distance to n_1, in order of increasing distance, where the nodes with the same distance to n_1 are sorted by their serial numbers, again in increasing order. It is obvious that this ordering of k nodes is unique once the serial numbers are fixed.

It is also clear that the set $S' = S \setminus \{n_k\}$ is connected and hence is a match of a $(k-1)$-template with leading node n_1. By the induction hypothesis, S' and a $(k-1)$-template isomorphic to $T_G[S']$ are found by our algorithm. Now, since S is a successor match of S', the pair (*oldMatch*=$\{n_1, \cdots, n_{k-1}\}$, *newNode*=n_k) will be passed on to the function "CanMatchTakeInNode(*oldMatch*, *newNode*)" and one easily checks that it satisfies the three conditions. Therefore $\{n_1, n_2, \cdots, n_k\}$ is a match, and either $T_G[S]$ will be listed by the algorithm as a new template or S will be listed as a match of an previously found isomorphic template. This shows that all connected sets at most *maxsize* nodes will be listed as a match of some template.

Next we prove that no isomorphic templates are listed more than once as a template by the algorithm. This is clear because for each potential template $T_G[S]$, the algorithm checks the already listed templates for isomorphism and either list $T_G[S]$ as a new template or lists S as a new match for an already listed template.

To complete the proof, we will show that S is not listed twice. Suppose that $S' = S$ are both listed as a match by the algorithm and suppose the nodes of S' have be ordered $\{n'_1, n'_2, \cdots, n'_k\}$, which is different from $\{n_1, n_2, \cdots, n_k\}$. Note that the order presents the sequence in which nodes enter a match. If the node in S' with the smallest serial number is not at the first position n'_1, i.e., n_1 is not the leading node, it will never be taken into a match due to condition 1 of the function 'CanMatchTakeInNode". So, we may assume that $n'_1 = n_1$. If a node with label n_b appears before a node with label n_a in the order $\{n'_1, n'_2, \cdots, n'_k\}$ of S' while n_b appears after n_a in the order $\{n_1, n_2, \cdots, n_k\}$ of S, this could have had two reasons: (a) the distance between n_b and the leading node in S is larger than the distance between n_a and the leading node; or (b) these distances are equal in S, and the serial number of n_b is larger than that of n_a. The former situation conflicts with condition 2 and the latter one conflicts with condition 3 of the function 'CanMatchTakeInNode". This completes the proof of Theorem 1. □

For a random CDFG, the number of templates and matches can be very large, i.e., exponential in the number of nodes of the CDFG [4]. Of course, if *maxsize* is fixed, the number of matches is always bounded by n^k, where $n = |N_G|$ is the total number of nodes and k =*maxsize*, so polynomial in n. In practice, the admissible match space can indeed be restricted significantly by adding constraints to the templates. In our MONTIUM tile, for instance, the number of inputs, outputs, and operations are limited. The templates that do not satisfy such limitations will be discarded. When DSP-like applications are considered, the size of the set of admissible matches can be further decreased.

5.2 The Template Selection Algorithm

So far, we have presented an algorithm to generate a set of templates Ω and their corresponding matches $M(\Omega)$ from G. Now we briefly present a heuristic to find an approximate solution to the template selection problem. The scheduling of the matches of the templates on the ALU-architecture in as few clock cycles as possible is not part of the solution concept given here. This last phase will be dealt with in future research.

Given G, $\Omega = \{T_1, T_2, \cdots, T_p\}$ and the matches $M(\Omega)$, the objective is to find a subset C of the set $M(\Omega)$ that forms a 'good' cover of G. Here by 'good' cover we mean a (k, ℓ)-cover with minimum k and ℓ.

Since the generated set $M(\Omega)$ can be quite large, the template and match selection problem is computationally intensive. We adopt a heuristic based on maximum independent set, and apply it to a conflict graph related to our problem, similarly as it was done in [12][14].

A **conflict graph** is an undirected graph $\tilde{G} = (V, E)$. Each match $S \in M(\Omega)$ for a template of the CDFG G is represented by a vertex v_S in the conflict graph $\tilde{G}$. If two

matches S_1 and S_2 have one or more nodes in common, there will be an edge between the two corresponding vertices v_{S_1} and v_{S_2} in the conflict graph $\tilde{G}$. The **weight** $w(v_S)$ of a conflict graph vertex v_S is the number of CDFG nodes $|S|$ within the corresponding match S. The vertex set of the conflict graph are partitioned into subsets, each of which corresponds to a certain template (see Figure 9). Therefore, on the conflict graph, vertices of the same subset have the same weight. The **maximum independent set (MIS) for a subset** $T \subset V(\tilde{G})$ is defined as the largest subset of vertices within T that are mutually nonadjacent. There might exist more than one MIS for T. Corresponding to each MIS for T on $\tilde{G}$, there exists a set of node-disjoint matches in G for the template corresponding to T; we call this set of matches a **maximum non-overlapping match set**(MNOMS). To

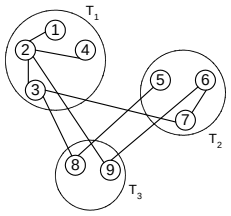

Figure 9: A conflict graph. The weight of each node is 4.

determine a cover of G with a small number of distinct templates, the templates should cover a rather large number of CDFG nodes, on average. The following theorem expresses the relationship between this goal and the concept of MNOMSs.

THEOREM 2. *Given a CDFG G and a template T of G, the template matches of a MNOMS (corresponding to a MIS in $\tilde{G}$) yield an optimal cover of G, where optimal is to be understood as covering a maximum number of nodes of G.*

The proof to this theorem is straightforward and can be found in [12] or [1].

An MNOMS corresponds to a MIS on the conflict graph. Finding a MIS in a general graph, however, is an NP-hard problem [9]. Fortunately, there are several heuristics for this problem that give reasonably good solutions in practical situations. One of these heuristics is a simple minimum-degree based algorithm used in [11], where it has been shown to give good results. Therefore, we adopted this algorithm as a first approach to finding 'good' coverings for the CDFGs within our research project.

For each template T, an **objective function** is defined by:

$$g(T) = g(w, s),$$

where w is the weight of each vertex and s is the size of an approximate solution for a MIS within the subset corresponding to T on the conflict graph. The outcome of our heuristic will highly depend on the choice of this objective function, as we will see later.

The pseudo-code of the selection algorithm is shown in Figure 10. This is an iterative procedure, similar to the methods in [4][6][14]. At each round, after computing an approximate solution for the MISs within each subset, out of all templates in Ω, the heuristic approach selects a template T with a maximum value of the objective function, depending on the weights and approximate solutions for the MISs. After that, on the conflict graph, the neighbor vertices of the selected approximate MIS and of the approximate MIS itself are deleted. This procedure is repeated until the set of matches C corresponding to the union of the chosen approximate MISs, covers the whole CDFG G.

1 Cover $C = \phi$;
2 Build the conflict graph;
3 Find a MIS for each group on the conflict graph;
4 Compute the value of objective funtion for each template;
5 The T with the largest value of objective funtion is the seleted template. Its MIS is the selected MIS. The MNOMS corresponding to the MIS are put into C.
6 On the conflict graph, delete the neighbor vertices of the selected MIS, and then delete the selected MIS;
7 Can C cover CDFG totally? If no, go back to 3; if yes, end the program.

Figure 10: Pseudo-code of the proposed template selection algorithm

THEOREM 3. *Using the template generation algorithm, at the moment that there is no vertex left on the conflict graph $\tilde{G}$ after removing the vertices and neighbors of the approximate MISs corresponding to the MNOMSs of C, the matches of the generated C cover the whole CDFG G and vice versa.*

PROOF. Every node in the CDFG G forms a 1-node match, which is represented by a vertex in the conflict graph $\tilde{G}$. Suppose a node n of G is not covered. Then this node n is still part of a match (or several matches) that are represented by vertices in the graph that results from $\tilde{G}$ after removing the vertices and neighbors of the approximate MISs. (At least the vertex v_n representing the 1-match n will not have been removed, because it does only conflict with matches that contain n.) Hence there is at least one vertex left in the remaining graph.

On the other hand, if there is a vertex v of the conflict graph left, then v corresponds to a match S, such that none of the nodes of S is a node of a match of a MNOMS corresponding to an already chosen approximate MIS (otherwise v would have been removed as (a neighbor vertex of) a vertex contained in one of the chosen approximate MISs). Therefore S is uncovered. □

We currently used the following objective function:

$$g(T) = w^{1.2} \cdot s = ws \cdot w^{0.2}. \quad (1)$$

In this function, for a template T, ws equals the total number of CDFG nodes covered by a MNOMS, which expresses a preference for the template whose MNOMS covers the largest number of nodes. Furthermore, due to the extra $w^{0.2}$ factor, the larger templates, i.e., the templates with more template nodes, are more likely to be chosen than the smaller templates. As described before, the template selection algorithm should be developed to find a cover with, in some sense, a minimum number of (matches of) templates. It is obvious that this is more likely the more nodes each

match (template) contains, but it could conflict with finding suitable covers; the latter seems to be easier if the matches (templates) contain fewer nodes.

An example is shown in Figure 11, in which ports are omitted. For this CDFG, the following holds:

$$w_A s_A = 3 \cdot 5 = 15,$$

$$w_B s_B = 1 \cdot 15 = 15.$$

We prefer *Cover A* to *Cover B* since it consists of fewer matches of templates, which is caused by the factor $w^{0.2}$.

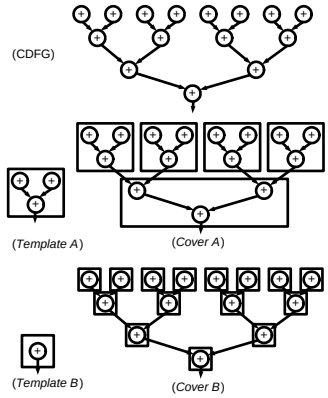

Figure 11: Give the template with more CDFG nodes more preference

Example: Suppose every vertex of the conflict graph in Figure 9 represents a 4-match on the CDFG. The sets {1,3,4}, {5,7} and {8,9} are MISs for T_1, T_2, T_3 respectively. Furthermore, the objective function is given by equation 1. We get:

$$g(T_1) = 4^{1.2} \cdot 3,$$

$$g(T_2) = 4^{1.2} \cdot 2,$$

$$g(T_3) = 4^{1.2} \cdot 2.$$

T_1 has the largest value of the objective function, so the selected MIS is {1,3,4}. The matches corresponding to vertex 1, 3 and 4 are chosen to be put into the cover C first. After that, we remove the neighbor vertices of the selected MIS and the vertices of the selected MIS itself. The newly obtained conflict graph is shown in Figure 12, which is the input for the next round. This time {5,6} will be selected and the algorithm terminates.

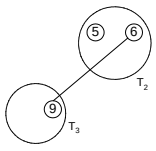

Figure 12: The conflict graph after the first round

After the template selection, the selected matches will be scheduled and mapped onto MONTIUM structure and each match can be executed by one ALU in one clock-cycle [15].

6. EXPERIMENTS

We tested the template generation and selection algorithms using some test CDFGs that are typical DSP functions, namely:
FFT4: 4-point FFT;
FFT8: 8-point FFT;
FFT16: 16-point FFT;
FIR: FIR filter with 128 taps;
DCT: 8-point discrete cosine transform;
TEA: tiny encryption algorithm.

For each test CDFG, a set of matches of templates was generated and a cover was produced using the generated matches. Equation 1 in section 5.2 was taken as the objective function. All the generated templates can be executed by one MONTIUM ALU (Figure 2), unlike the experimental result in [10], where the generated templates fitted on one tile (Figure 1).

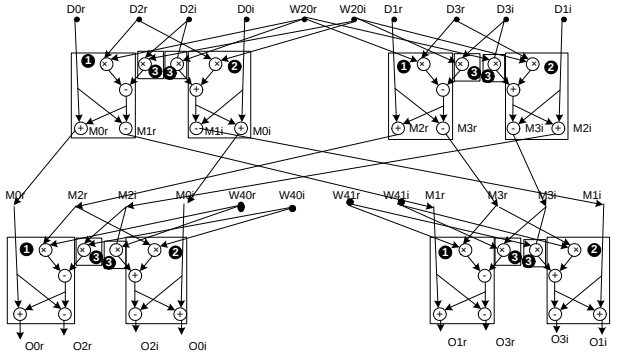

Figure 13: The CDFG for a 4-bit FFT-algorithm

As an example Figure 13 presents the produced cover for FFT4. The letters inside the dark circles indicate the templates. For this CDFG, among all the templates, three have the highest objective function value. The hydragraph is completely covered by them. This result is the same as our manual solution. The same templates are chosen for n-point FFT ($n = 2^d$).

The other hydragraphs are too large to be presented here. The results are summarized in TABLE 2.

Application	Number of nodes	Number of generated matches	Number of generated templates	Number of selected matches	Number of selected templates
FFT4	40	≈ 500	≈ 200	16	3
FFT8	120	≈ 1500	≈ 200	48	3
FFT16	320	≈ 4400	≈ 200	128	3
FIR	255	≈ 3700	≈ 35	139	4
DCT	43	≈ 1000	≈ 500	21	7
TEA	80	≈ 700	≈ 80	42	5

Table 2: Using template coverings for DSP-like applications

From the table, it can be seen that:

- Compared to the total number of nodes in the hydragraph, the number of templates needed to cover a CDFG of a DSP function is small. This is due to the highly regular structure of the CDFGs of DSP functions.
- The number of templates for FFT4, FFT8 and FFT16 are almost the same although they have different number of fundamental operators. This is because of the similarity in the structures of their CDFGs.
- The number of templates will not increase unlimited with the total number of nodes. This is due to limitation of the *maxsize* of templates and the ALU structure.
- The more regular a CDFG is, the fewer templates are generated by the algorithm.

7. CONCLUSION

The major contributions of this paper are the algorithms developed to generate all possible nonisomorphic templates and the corresponding matches. For this purpose we introduced a new type of graph (hydragraph) that can cope with multiple outputs. Therefore, the generated templates are not limited in shapes (as opposed to the approaches in [4][6]), i.e., we can deal with templates with multiple outputs or multiple sinks. By applying a clever labelling technique, using unique serial numbers and so-called circle numbers, we were able to save on execution time and memory, and avoid listing multiple copies of isomorphic templates or matches. This approach is applicable to control data flow graphs as well as general netlists in circuit design.

Using the templates and corresponding matches, an efficient cover for the control data flow graphs can be found, which tries to minimize the number of distinct templates that are used as well as the number of instances of each template. The experiments showed promising results.

In future work, we plan to test more CDFGs, and then compare the results with the manual solutions. We will further improve the template selection algorithm by manipulating and optimizing the objective function. In a later stage, we will deal with the scheduling of the solutions on the ALU-architecture in as few clock cycles as possible.

8. ACKNOWLEDGMENTS

This research is supported by PROGram for Research on Embedded Systems & Software (PROGRESS) of the Netherlands Organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the technology foundation STW.

9. REFERENCES

[1] Instruction generation for hybrid reconfigurable systems. http://citeseer.nj.nec.com/446997.html.

[2] Srinivasa R. Arikati and Ravi Varadarajan. A signature based approach to regularity extraction. In *Proceedings of Internaltional Conference on Computer-Aided Design (ICCAD)*, pages 542–545, 1997.

[3] Marnix Arnold and Henk Corporaal. Automatic detection of recurring operation patterns. In *Proceedings of the seventh International Workshop on Hardware/software Codesign*, pages 22–26, Rome, Italy, March 1999.

[4] Srihari Cadambi and Seth Copen Goldstein. Cpr: A configuration profiling tool. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 1999.

[5] Timothy J. Callahan, Philip Chong, Andre DeHon, and John Wawrzynek. Fast module mapping and placement for datapaths in fpgas. In *Proceedings of International Symposium of Field Programmable Gate Arrays*, 1998.

[6] Amit Chowdhary, Sudhakar Kale, Phani Saripella, Naresh Sehgal, and Rajesh Gupta. A general approach for regularity extraction in datapath circuits. In *Proceedings of Internaltional Conference on Computer-Aided Design (ICCAD)*, pages 332–339, 1998.

[7] Miguel R. Corazao, Marwan A. Khalaf, Lisa M.Guerra, Miodrag Potkonjak, and Jan M. Rabaey. Performance optimization using templete mapping for datapath-intensive high-level synthesis. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems*, 15(8):877–888, August 1996.

[8] C. Ebeling and O. Zajicek. Validating vlsi circuit layout by wirelist comparison. In *Proceedings of Internaltional Conference on Computer-Aided Design (ICCAD)*, pages 172–173, 1983.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[10] Yuanqing Guo, Gerard J.M. Smit, and Paul M. Heysters. Template generation and selection algorithms for high level synthesis. accepted for publication in *the 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, 2003.

[11] Magnús M. Halldórsson and Jaikumar Radhakrishnan. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. In *Proceedings of ACM Symposium on the Theory of Computing*, 1994.

[12] Ryan Kastner, Seda Ogrenci-Memik, Elaheh Bozorgzadeh, and Majid Sarrafzadeh. Instruction generation for hybrid reconfigurable systems. In *Proceedings of Internaltional Conference on Computer-Aided Design (ICCAD)*, 2001.

[13] Thomas Kutzschebauch. Efficient logic optimization using regularity extraction. In *Proceedings of Internaltional Workshop on Logic Synthesis*, 1999.

[14] D. Sreenivasa Rao and Fadi J. Kurdahi. On clustering for maximal regularity extraction. *IEEE Transactions on Computer-Aided Design*, 12(8):1198–1208, August 1993.

[15] Michel A.J. Rosien, Yuanqing Guo, Gerard J.M. Smit, and Thijs Krol. Mapping applications to an fpfa tile. In *Proceedings of Design, Automation and Test in Europe*, March 2003.

[16] Gerard J.M. Smit, Paul J.M. Havinga, Lodewijk T. Smit, Paul M. Heysters, and Michel A.J. Rosien. Dynamic reconfiguration in mobile systems. In *Proceedings of FPL2002*, pages 171–181, September 2002.