CORE IDENTIFICATION FOR RECONFIGURABLE SYSTEMS DRIVEN BY SPECIFICATION SELF-SIMILARITY

BY

MATTEO GIANI
Bachelor in Computer Engineering, Politecnico di Milano, 2003

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2006

Chicago, Illinois

Copyright by

Matteo Giani

2006

TABLE OF CONTENTS

CHAPTER			PAGE
1	INTRODUCTION		1
	1.1	Setting	2
	1.2	Structure of the thesis	7
2	PREVIOUS WORK		8
	2.1	Partitioning for Reconfigurable Architectures	8
	2.1.1	Temporal Partitioning approaches	8
	2.1.2	Spatial and Temporal Partitioning Approaches	16
	2.2	Regularity extraction	17
	2.3	Other Partitioning approaches	19
3	BASIC CONCEPTS		21
	3.1	Specification languages	21
	3.2	Abstract representation	22
	3.3	Intermediate phases	27
	3.4	Reconfigurable unit size	29
4	THE PROPOSED APPROACH		30
	4.1	Self-Similarity: rationale	30
	4.2	Partitioning algorithms	33
	4.2.1	Tree-shaped templates	35
	4.2.2	Unconstrained shape templates	42
	4.2.3	Graph covering	52
5	IMPLEMENTATION		57
	5.1	Data Structures	57
	5.1.1	The PandA framework	58
	5.1.2	The Boost Graph Library	61
	5.2	The Partitioning Phase	63
6	RESULTS		66
7	CONCLUSIONS		73
	7.1	Accomplished goals	73
	7.2	Future development	73
	CITED LITERATURE		75

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

VITA 80

LIST OF TABLES

TABLE		PAGE
I	Template Identification: Number of templates, maximum sizes	66
II	Graph Covering: Results with MFF, LFF heuristics	68
III	Graph Covering: using internal/boundary edge ratio as a metric for template choice	72

LIST OF FIGURES

FIGURE		PAGE
1	The Area-Latency trade-off	3
2	Area-Latency solution space: area and time constraints	4
3	Area-Latency solution space: the effect of RTR	6
4	Temporal partitioning: the sequence of configurations	9
5	A simple piece of code we will use as a working example.	25
6	DFG for the code in Figure 5, with color-coded operation types	26
7	Example of template with two instances in a DFG.	34
8	Tree-shaped template generation: example	40
9	Unconstrained Shape Template generation: example	48
10	Local Isomorphism Check: example	49
11	PandA: behavioral description layer	59
12	PandA: Data Flow Graph example	64
13	Partitioning phase implementation: class diagram	65
14	`AES - encryptblock`: template size vs. number of templates	69
15	`AES - encryptblock`: template size vs. number of instances	70
16	`AES - encryptblock`: template size vs. internal/boundary edge ratio	71

LIST OF ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
CTR	Compile-Time Reconfiguration
DFG	Data Flow Graph
FPGA	Field-Programmable Gate Array
HLS	High Level Synthesis
ILP	Integer Linear Programming
IC	Integrated Circuit
PDG	Program Dependency Graph
RTL	Register-Transfer Level
RTR	Run-Time Reconfiguration
SDG	System Dependency Graph
VHDL	Very high speed IC Hardware Description Language

SUMMARY

An issue in the field of reconfigurable hardware systems is that there is no clear way to partition a given system specification in order to implement it on such an architecture.

Many existing approaches rely on the designer to carry out the partitioning according to his knowledge of the desired behaviour, which is really not a desirable assumption for allowing designers to take advantage of such architectures in a straightforward way.

We propose a way to partition a system specification based on the detection of recurrent structures in the specification itself, with the objective of identifying modules that can be used more than once during the application's lifetime, thus saving device resources and reconfiguration time.

Two approaches have been defined to this end, which differ in the structure of the recurrent patterns they identify. The two algorithms were implemented as an extension to an existing framework for research in the field of hardware/software codesign being developed at Politecnico di Milano.

A comparison with regard to the amount and size of the identified recurrent structures was carried out using working examples from widely used algorithms as input data.

Finally, some ideas for future improvement of this work and further development are given.

CHAPTER 1

INTRODUCTION

Over the last years, considerable interest has risen towards the field of reconfigurable hardware systems, with noteworthy research efforts in this direction such as (49; 33; 20; 18). These are, generally speaking, systems in which the hardware itself is able to adapt to the application's changing needs through its lifetime. Especially interesting is the scenario in which this *reconfiguration* of the hardware is carried out without necessarily having to cease execution of those parts of the system that are not involved.

This work is particularly concerned about the implementation of such behaviour on a class of user-programmable hardware components called *Field-Programmable Gate Arrays*, or FPGAs in short. Such devices were first conceived, and used, as prototyping platforms on which to test designs that were later to be implemented on *Application Specific Integrated Circuits*, ASICs. Their attractiveness for this purpose is clear: FPGAs allow the designer to test the hardware implementation of their desired application much sooner than it would take to have an actual ASIC prototype and, since an FPGA is reusable, at a much smaller expense. However, as FPGA technology grew more and more refined and cost-effective, it became not so unrealistic to actually think about using these devices in production systems.

The feature of these devices that most concerns this work is the way they they allow their configuration to be modified. The simplest scenario is that in which the whole device is configured *"at compile time"*, which is to say its configuration *does not change* once the application

starts running. A far more challenging one, however, is a setting in which the device allows part of its logic to be reconfigured *"at run time"*. These two configuration models are often referred to as *Compile-Time Reconfiguration* (CTR) and *Run-Time (or Dynamical) Reconfiguration* (RTR). If, in addition, the device allows parts of its logic to be reconfigured without requiring the rest of the device to stop its computation, it is said to be *partially dynamically reconfigurable*.

While dynamical reconfiguration is technically feasible, we still have to pinpoint the situation in which it can be usefully and advantageously exploited.

1.1 Setting

Let us begin with a simple diagram, Figure 1 which depicts, in a simplified way, the trade-off between used device area and latency with which we are faced when mapping a specification onto a reconfigurable device such as a FPGA, *in a static scenario that does not make use of RTR*. In this simple graph, where the used area varies along the horizontal axis, each area usage has the length of the best obtainable schedule lying exactly on the border between the green and the greyed out area: the more resources we have at our disposal, the shorter time we will be able to obtain. One extreme case is the one in which an *infinite resource scheduling* of the specification is implemented on the device, so that the only limiting factor on the time we are able to obtain is given by the actual structure of the specification. On the other hand, we have the solution in which only the minimal set of resources needed to realize the specification is implemented, i.e. *only one functional unit per type*: the time we will obtain is longer since all the operations of a certain type will necessarily have to be executed not more than one

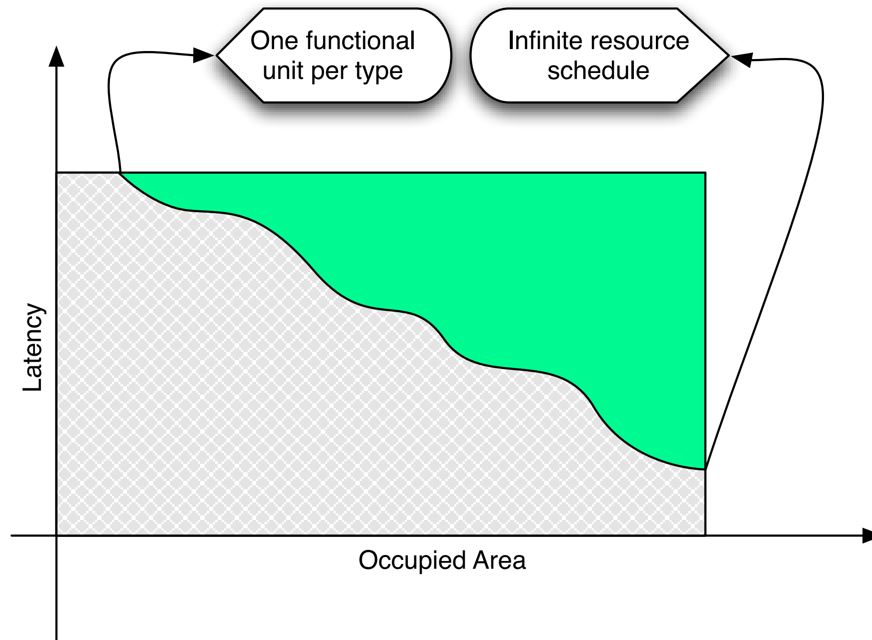

Figure 1. The Area-Latency trade-off

at a time. Solutions represented by points in the green area are feasible but not optimal[1], while solutions in the greyed-out area are not feasible being shorter than the respective optimal latency schedule for a given occupied area.

In Figure 1 we haven't represented, though, a fundamental element in the selection of a solution: the presence of the constraints given by both the designer, in the form of timing

[1]For that matter, solutions *above* the green area, i.e. longer latencies than the single functional unit one, are feasible too, albeit not interesting.

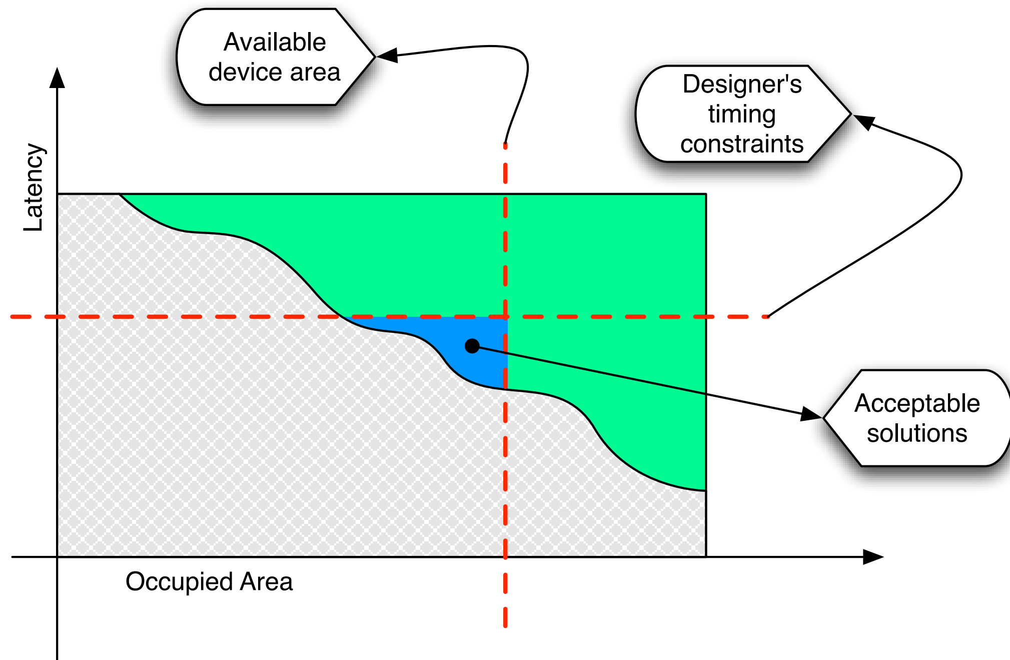

Figure 2. Area-Latency solution space: area and time constraints

constraints, and by technical constraints of the device, i.e. the total usable area offered by it. These are represented by the two dashed lines in Figure 2: the horizontal one represents the maximum schedule length that the designer deems acceptable, and may be dictated by various external factors such as real-time requisites of the system being engineered; the vertical one is, in turn, a physical constraint that represents the *maximum amount of logic* that the device is able to implement. The range of acceptable solutions is the blue area, i.e. the feasible solutions that satisfy both time and area constraints.

As it appears clear from the graph, the introduction of a time constraint implicitly inserts the requirement of a *minimum* device area occupied, i.e. the minimum area needed to allow a schedule at least as short as the designer's constraint specifies. If this minimum area is less than the area offered by the device, as it happens in Figure 2, we are in principle able to obtain an acceptable solution. What happens, though, if the designer's time constraint is so stringent[1] that the area offered by the device is *not enough* to allow for a short enough schedule? This is one of the scenarios where we believe dynamic reconfiguration could prove to be useful.

One way to model the effect of reconfiguring our device dynamically, in an approach that has been explored in previous works as well (32; 15), is to show a *virtual* area larger than the physical one, much like operating systems do with memory management. In our diagram, such area enlargement is represented as a relaxation of the area constraint. Figure 3, in which part of the previous graph is magnified for clarity, portrays such situation. Dynamical reconfiguration, though, comes at a price: *it takes some time.* As is shown in the diagram, the schedule length is worsened by some amount, which we will call *reconfiguration overhead* or simply *reconfiguration time*, that is spent performing the reconfiguration of the device's logic which, in our scenario, takes place at run-time. If however, that amount of time is small enough, we might nevertheless obtain a region of solutions which satisfy the constraints and are feasible, i.e. the blue area in the diagram.

[1]but still greater than the length of the infinite resource schedule, since it would make no sense otherwise

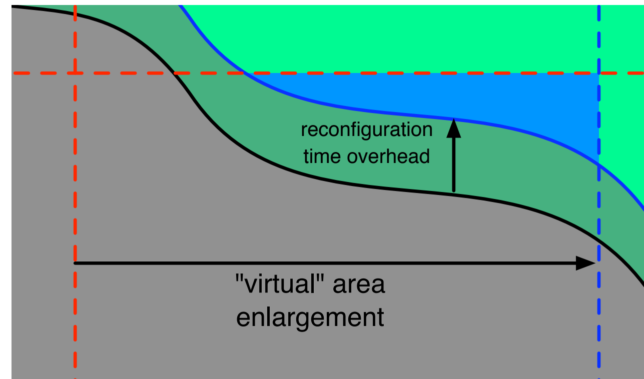

Figure 3. Area-Latency solution space: the effect of using Dynamical Reconfiguration

Reconfiguration time is indeed one of the reasons why *partial* dynamical reconfiguration is attractive: if we manage to have part of the device carry on its computation while we reconfigure other regions of it, this will effectively contribute to lessen the impact of the reconfiguration time. Achieving this goal, however, is not a trivial matter: the system specification has to be somehow processed and adapted to this kind of implementation. The goal of this processing is to maximise the advantage offered by dynamical reconfiguration. This thesis work aims at defining and implementing a method for breaking a system specification into modules, possibly to be implemented as reconfigurable units.

As has been mentioned, reconfiguring any amount of the device's logic *takes time*. Intuitively, then, it is advantageous to spend configuration time for activating a unit that implements *a significant fraction* of the specification, i.e. will spend a significant amount of time running its computation, over choosing one that will only be used for a short time, thus *"wasting"* the

spent configuration time. This is the rationale behind the choice of the approach followed in this work, i.e. the recognition of recurrent patterns in the system specification.

1.2 Structure of the thesis

The remainder of this thesis is organized as follows:

- Chapter 2 presents a review of existing works in the fields of reconfigurable systems and regularity extraction;
- chapter 3 establishes some basic concepts which will be useful through the remainder of the thesis;
- chapter 4 describes the approach we chose to undertake, detailing the mechanisms involved in the proposed algorithms for our partitioning purposes;
- chapter 5 contains a more detailed description of the implementative details, including a presentation of the data structures of which the implementation of this work makes use;
- chapter 6 summarizes the obtained results, while
- chapter 7 draws some conclusions about the objectives achieved, and hints at possible future developments.

CHAPTER 2

PREVIOUS WORK

This chapter reviews some existing literature in the reconfigurable architectures field, and presents some previous work regarding partitioning and regularity detection.

2.1 Partitioning for Reconfigurable Architectures

Different approaches to the exploitation of the reconfiguration capabilities of devices such as FPGAs have been explored in literature, among which we find both approaches which only carry out *temporal* partitioning of the device's resources, i.e. the whole device is reconfigured at once when the application needs so, and approaches which also perform *spatial* partitioning of the device into reconfigurable units of smaller size. The latter approaches are those that are closest to our point of view, and are elsewhere referred to as *dynamically reconfigurable* architectures. Let us first focus on some of the works that propose total reconfiguration.

2.1.1 Temporal Partitioning approaches

A quite straightforward approach is proposed in (39), in which the authors have the objective of partitioning a specification that would not fit on the FPGA all at once into a set of *subprograms*, to be configured sequentially on the device. The proposed solution involves two steps:

- as a first step, an acyclic data flow graph is partitioned to produce a set of configurations, $\{c_1, c_2, \ldots, c_n\}$, each of which fits singularly on the FPGA

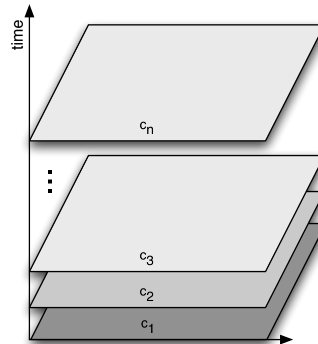

Figure 4. Temporal partitioning: the sequence of configurations

- the partitions thus obtained are configured one after the other onto the device, and executed serially. Figure 4 depicts the series of configurations loaded onto the device: all of the device is reconfigured for the execution of each partition.

Two approaches are proposed for the partitioning of the specification, one of which aims at minimizing the temporal latency, while the other tries to minimize the communication overhead generated by the partitioning. Both algorithms start out by making sure that the data dependencies encoded in the data flow graph are respected in the partition that will be generated, and they do so essentially by traversing the graph in a breadth-first fashion, and assigning a so-called ASAP-level to each vertex v, defined as:

$$\lambda(v) = 1 + \max_{w \in FanIn(v)} \lambda(w)$$

The first approach simply orders the vertices by increasing value of λ, and adds vertices to a partition until no more would fit on the device, at which point it creates another partition and continues similarly until all the specification has been considered. The second approach, on the other hand, aims at minimizing the overhead due to data exchange between different partitions by trying, when possible, to assign successors of a given node to its same partition.

An estimate of the total execution time was provided in this work as $T_{total} = (n \cdot T_{reconf}) + T_{exec}$, simply stating that the total time required for executing the partitioned specification is given by the total reconfiguration time needed added to the total execution time needed. The authors themselves, however, noted how the reconfiguration time was, for their hardware devices, *orders of magnitude* larger than the execution time, and suggested improvements to tackle this disadvantage, such as applying the method to applications in which each configuration can be used multiple times before being swapped out for the next one, such as applying the first phase of a JPEG encoder to all the input images before going on to configuring the subsequent stages.

A similar approach is exposed in (22), which also considers some amount of spatial partitioning while still considering total reconfiguration of the physical devices due to the use of multiple FPGAs in a single system.

A series of papers (36; 26; 37; 25; 27; 17; 19) presents an approach to the temporal partitioning of a system specification, exploring different ideas and improvements. In (25), an Integer Linear Programming (ILP) model is proposed describing the temporal partitioning problem. It takes into account the dependencies between different parts of the specification and the resource

availability on the FPGA, and also models the constraints due to memory needs for exchanging data between the different partitions identified.

As its input, this approach uses a task graph, i.e. a graph in which each node is not a single operation, but a set of operations instead. This constrains the algorithm to miss a great amount of the possible solutions, since the minimum working granularity is set at the task level. Also, many of the parameters needed by the model to function properly are to be supplied by the designer, apparently with no general guideline for choosing reasonable values, e.g., for the maximum number of partitions or the maximum number of functional units to be instantiated for each type.

The work proposed here is extended in (36) with the application of a genetic algorithm that adds spatial partitioning in order to split the specification onto multiple interconnected reconfigurable devices. The lack of indications towards the choice of the ILP model parameters is tackled in (26), which revises some aspects of the model itself: each task is associated to a set of possible implementations, each characterized with an estimate of its latency and area usage. The ILP solver chooses among them which to employ. Choice of the ILP parameters is tackled here in an enumerative fashion: the ILP solver is run multiple times for each partition number bound in order to find which latency constraints yield the best solution, which is repeated until an upper bound on the partition number constraint is reached.

A clear disadvantage of this way of operating, however, is that the ILP model is solved multiple times, thus increasing the computational burden.

In (37), a further modification of the approach is introduced: the operations inside each task are now allowed to be assigned to different partitions, thus refining the granularity at which the scheduling is performed. The authors stress here that, given the unbalance between the configuration and execution times, in order to obtain a good overall latency it is necessary to minimize the need for reconfiguration, and thus the number of temporal partitions identified.

The approach that they propose to this end is the *sharing* of the functional units instantiated in each temporal partition between different operations allocated to the same partition. The authors present as an example the application of their approach to the description of a matrix multiplier, showing that making use of said sharing can yield a gain in execution latency.

The approach proposed therein employs a resource constrained force-directed list scheduling, the resource set of which is modified to signify the change in the chosen temporal partitioning in a local-search like fashion.

A major issue that the authors are faced with is, once again, the remarkable unbalance between the times spent in computation and reconfiguration, the latter being orders of magnitude more relevant than the former - several seconds spent in reconfiguration vs. just microseconds needed for execution.

This observation leads the authors, in a further work (27), to explore a possibility of lessening the impact of reconfiguration time on total latency. They observe here that many data-intensive application can be modeled as the subsequent application of some repeating tasks, in a loop-like fashion. If each of the tasks were to be implemented as a temporal partition, then a reconfiguration would be necessary each time the application needed to begin execution of a

new task. The possibility explored here is that, depending on the structure of the application, it might be possible to configure the device for each task type only once, by running the task on all of the data it needs to process before going on to the next configuration.

As an example, the JPEG compression algorithm is internally composed of four well defined tasks (`DCT`, quantization, zig-zag and Huffman encoding), each of which is executed independently on all of the image data to be compressed - or on the output of the previous task.

A problem foreseen by the authors with this approach is that, for some classes of algorithms, the intermediate data which the application would need to save to ensure proper communication between a task and the subsequent one could have a significant size. Thus, they propose two ways to cope with this issue: the first one is called *Final Data to Host*, in which each task is only run as many times as the on-board memory of the embedded system allows it to, after being swapped out for the next one, while the other is *Intermediate Data to Host*, which has each task run on all of the input data and generate all the output before being overwritten with the next temporal partition, even if needing costly data accesses to external memory.

Experimental data provided by the autors suggests, as expected, that the reconfiguration times can only be effectively hidden when the data intensiveness of the application is appropriately stressed, e.g. for the JPEG case only when the input image exceeds a certain size.

Other approaches, like the one proposed in (7; 11), deal with the possibility of using high-level languages as the means of writing system specifications. More specifically, the authors here focus on defining a way to implement a specification given as Java bytecode on a reconfigurable hardware platform. The specification's data flow graph is first traversed in order to produce an

ordering of the nodes based on topological levels, with mobility of the nodes (ASAP-ALAP) used to break ties between nodes on the same level. From here on, the approach is pretty similar to the one seen in (39). Unfortunately, little depth is devoted to the discussion of reconfiguration time.

Further work from the same authors, (9; 10), try to apply a concept that we have already exposed above for other approaches: the sharing between different operations of the functional units present in each temporal partition. The proposed way of proceeding here is to first create a temporal partition for each node on a critical path of the specification's data flow graph, and then trying to fit the other nodes in a partition according to their dependencies and resource usage, taking advantage where feasible of resource sharing. In a subsequent step, an effort is made to try and merge bordering partitions.

In (8), the authors also propose a novel way to split a loop into several time partitions, which is useful when its body cannot fit onto the resources offered by the reconfigurable device. This however, requires a large number of reconfiguration steps, and thus greatly affects the system's latency, making it really only advantageous when there is no other way of fitting the specification onto the device.

A series of works related the MorphoSys platform (33; 34) try to approach dynamic reconfiguration with respect to the particular features of this architecture. The core of the architecture is made of an 8x8 array of reconfigurable cells, each of which resembles a miroprocessor's datapath and is provided control information by a dedicated control register, which in turn loads its

contents from the context memory. The task of controlling the behaviour of the reconfigurable hardware is delegated to a dedicated `RISC` control processor.

The approach proposed uses a library of *kernels* as its starting point, i.e. a collection of subprograms written in C each of which has its counterpart implemented as reconfigurable hardware. It is also assumed that a generic specification can be expressed as a loop on a certain sequence of said kernels. Note that this assumption has also been encountered in other works. The reason this approach is considered as only performing temporal partitioning is that each kernel is assumed to require the use of the whole processing array at once.

Having said this, it appears clear that the key to optimizing the implementation is an appropriate way to manage the loops that the specification is composed of. Not unexpectedly, the first approach proposed is the modification of the loops' schedulings in order to execute all necessary runs of each kernel before reconfiguring the hardware to execute the next one, with the objective of minimizing the number of required reconfigurations. Different ways of rescheduling the kernels to this end are proposed in (34), taking into account the limitations imposed by the hardware architecture.

Once the loops have been scheduled, the authors focus in turn on the generation of the control code, which states when context changes take place, and how data is transferred between subsequent partitions. A primary objective is the maximization of the overlapping between computation on part of the reconfigurable array and data transfer or context loading, along with optimal management of the memory in order to simplify the control code.

2.1.2 Spatial and Temporal Partitioning Approaches

In (47; 48; 46), Vasilko et al. deal with reconfigurable architectures that have, ideally, a reconfiguration time of less than a clock cycle. Their approach (47) is to sort the tasks composing the specification according to their `ASAP` and `ALAP` labels, and then using a normal list-based scheduler constrained in resource usage to the area available on the device. When the area is used up, functional units configured to implements tasks that are no longer needed are removed from the device, and replaced with new ones. The fragmentation that might arise from such usage of the device is unfortunately not dealt with. (46) is devoted to the presentation of a graphical tool to allow the designer to manually *pack* the functional units, both regarding their placement on the device and across partitions, and thus explore the design space.

An automatic approach is instead proposed in (48), where the authors try to explore the design space by taking advantage of a genetic algorithm, tackling task allocation to functional units, scheduling, and floorplanning at once.

The authors of (36; 26; 37; 25; 27; 17; 19), cited above, also explore the possibility of extending their work to take advantage of partially reconfigurable architectures. Their main concern (17) is to hide the reconfiguration times: their approach is to divide the reconfigurable device into two partitions of fixed sizes. At a given time, one of the two parts is computing while the other one is being reconfigured and getting ready for its next task. It is assumed that the configuration of one of the two parts starts synchronously with the execution of the other one, which implies that in order to properly hide the reconfiguration times, the task assigned to a part has to be long enough in execution time to last at least as long as the configuration

of the other part. In order to ensure this, the now well known idea of embedding a loop inside each partition is proposed.

This is made harder, clearly, by the fact that each partition can only occupy half of the resources on the device, and thus can only fit smaller loop bodies.

The authors' focus changes in (19), where they try to minimize the computational effort needed in the design phase as opposed to optimizing the execution further, by exploiting a library of pre-placed and routed macros as functional units, that the design phase thus only has to place on the device.

2.2 Regularity extraction

The extraction of regular structures from a given input graph is a valuable step in different contexts, ranging from software engineering, e.g. in detecting duplicate code (28) both for plagiarism detection and for optimization purposes, to hardware areas such as VLSI design (30; 1; 40; 12).

In (30) and (1), both of which work at a lower level of abstraction than the one we are concerned with, propose *signature-based* approaches to describe the local structure of the design and thus recognize regular structures.

The authors of (1) try to recognize recurrent interconnection patterns between circuit elements from a known library, in order to optimize placement with respect to area and wire length. At the core of their proposed approach is the usage of a hash function which, given a circuit element, takes into account its type and the types of the neighbors connected, in order, to each of its ports. The described approach aims at identifying blocks of regular functions

among random instances which are not part of the datapath, in order to place them into the bit-slices to which they show the closest relation, thus saving communication costs.

The work presented in (30), also in the VLSI design field, has as its objective the identification of functionally equivalent *slices*, i.e. subgraphs of the netlist generated by the system's HDL description, in order to ease the subsequent logic optimization step. The proposed approach starts with the identification of seed sets, i.e. initial groups of known equivalent gates, to be used as a starting point.

Seed sets are built considering heuristics closely related to the application field, such as considering successors of high-fanout nodes, or entities with similar names as generated by the HDL tool. At the core of the described algorithm is an expansion of the already known functionally equivalent slices driven by the comparison of the so-called *regularity signatures* between the corresponding entities in each of the slices. These signatures are essentially subgraphs, including the neighbors which are candidates for being attached to the slices and the edges which their inclusion would add to the slice, in order to ensure functional equivalence of the enlarged slices.

Another way to approach the regularity extraction problem is presented in (40), which also deals with integrated circuit design. Here, the focus of the work is on recognizing in the system graph instances of a given set of templates. The authors propose an approach which involves using a linear representation of the graphs, in form of strings called *K-formulas* (5). By representing the templates to be recognized as K-formulas, the authors formulate the matching problem as a string matching one, and propose ways to restrict the K-formula representation,

which in itself would allow a given graph to have multiple legal string encodings, to obtain a unique representation for a given directed graph.

This work, however, assumes that the template library is given. The template generation problem is dealt with only to a shallow depth, not being the main focus of the paper.

2.3 Other Partitioning approaches

Work related to specification partitioning can be found in different fields as well: in (24), the authors propose an approach to the partitioning of a software program into threads, in order to run it efficiently on a single-chip multiprocessor architecture. Even if not directly related to the problem we are tackling, an interesting idea is proposed here: the minimization of the impact that inter-partition dependencies have on the performance of the partitioned system.

In this approach, the partitioning problem is solved using a heuristic at the heart of which is the solution of a *min-cut* problem over a graph encoding the application: the cut generated this way splits the application into two partitions, severing some dependency edges across the cut. Each of the edges in the graph is labeled with a weight representing how bad cutting the edge is likely to affect the latency of the partitioned application, due to different considerations about the way the author's architecture works. After performing an initial minimum cut, the algorithm performs a local search by moving the vertices bordering the cut across the partition, one at a time, and keeps the version with the best performance metric, i.e. an estimate of the runtime and the penalty due to the cut dependencies. When no more performance can be gained through these "balancing" steps, the algorithm stops. If the cut obtained this way yields a performance improvement over not cutting at all, then the original thread is split and

two new threads are generated, which in turn will be considered for partitioning in the same fashion.

As the authors themselves point out, this is a heuristic to a NP-complete problem, and as such could suffer from getting stuck into local optima. As a last step to their partitioning algorithm, then, they propose an *edge perturbation* step, which ensures that each edge in the application's graph has at least once been considered for being cut.

CHAPTER 3

BASIC CONCEPTS

Let us establish some background concepts and assumptions for the remainder of this work. First of all, our purpose is to take the system specification as written by the designer - in a suitable description language, as we will see shortly in 3.1 - and process it, making it suitable for implementation on a reconfigurable hardware system by first providing a suitable abstract representation (3.2) and then transforming it (3.3) towards our goal of implementing the system on a dynamically reconfigurable device.

3.1 Specification languages

Popular ways to specify the desired behaviour of an embedded system include languages such as `VHDL` and `Verilog`, well established in the area of hardware design, but a growing interest has been recently developing towards executable specification languages, such as `SystemC` (45), `JHDL` (4). The former two are "traditional" hardware description languages: as such, they require the designer to design his specification keeping well in mind that the semantics of what he is writing are not quite the same of what one would expect of a software programming language. This, for an experienced designer, could be seen as an advantage: the ability of accessing the low-level details of his design could mean *tweaking* it performance-wise in ways not accessible from a higher level of abstraction, much like it happens with programming languages. On the other hand, however, the majority of people approaching embedded systems design nowadays already

have quite some software programming experience. For these people, the need of learning a whole new way of *writing code* is a discouraging perspective to say the least. This is where languages such as `SystemC` come into play[1].

`SystemC` is a `C++` class library offering hardware-oriented constructs. Its intended use spans over both the design and implementation phases, with a big advantage towards design verification: *the specification itself is executable* being written in `C++`. Another advantage from the point of view of ease of adoption is that experienced software programmers will feel more at home with `SystemC` since it allows them to exploit their existing knowledge of object-oriented approaches. These are the reasons why, throughout the remainder of this thesis, the examples will refer to a C-like language for the specifications: the increasing popularity of languages such as `SystemC`, along with the appearance of commercial offers of FPGA design tools using C as their specification language (23), allows us to say this is an acceptable assumption.

3.2 Abstract representation

Having to deal directly with the designer's description is clearly not a viable approach, we therefore have to resort to some kind of abstract representation of it that enables us to process it in a convenient way. Following an ubiquitous approach, which for years has been common in several fields even other than our own (42) such as compiler theory (21; 35), debugging techniques (44; 31), and program optimization (29; 14), the most advantageous way of representing

---

[1]It is yet unclear whether a philosophical debate of remarkable breadth, as the one that took place between long-time FORTRAN developers and the new Pascal *"quiche-eating"* sustainers in the early 1980s (38), will originate.

the desired behaviour of the system being designed is through a *graph*. An appropriate representation of the specification in form of a graph allows us to manipulate it conveniently, and enables us to take advantage of the work done in the well-established field of graph theory.

Several kinds of graph-like abstract code representations were defined in different contexts, each focused towards exposing different aspects of the modeled specification. Nonetheless, the common idea underlying the representations interesting for our purpose is that:

- nodes in the graph represent *operations* in the specification. In our assumption of dealing with a C-like code, this means we will have a node in the graph for each operation like a sum, a subtraction, or the dereferentiation of a pointer.
- edges of the graph represent *precedences*, or *dependencies*, between operations. In the most general form, these can be precedences that arise due to data exchange between operations, or due to control flow issues.

For our purpose, however, it is necessary to somehow augment the information offered by such graph, in order to keep track of what kind of operation each node performs: each vertex in the graph, which corresponds to an operation in the original code, will have to be *tagged* with the kind of computation it represents. We will thus have a *colored* graph.

The representation chosen for this work is the *Data Flow Graph*, or DFG in short. In this graph, the edges represent precedences that arise due to actual *exchange of data* between the operations they connect. As such, it will also be necessary to keep track of the *numbering of inputs* for nodes that represent non-commutative operations in order to have a correct representation of the original specification.

Let us then formalise the kind of graph we will be dealing with, referring to a similar notation to the one used in (41). Our data flow graph is

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

where O is the set of operations present in the original specification, and $P \in O \times O$ is the set of precedences among them. Graph G is assumed to be acyclic. In addition to this, we define a function

$$\alpha : O \rightarrow A$$

where A is the set of operation types, such that $\alpha(o)$ is the operation type associated to a given vertex o.

Moreover, the numbering of inputs is modeled as a function

$$\mu : P \rightarrow \mathbb{N}$$

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.

Figure 5 contains a simple C function, which we will use as a working example of a specification in the following chapters. The use of a "traditional" programming language as our input makes sense since, as we have discussed before, it nowadays becoming more and more common

```
int test_code( int io, int * o1, int i1, int i2 )
{
    int a1, b1, a2, b2, a, b, c, d;

    i1 *= 2;
    a1 = i1 + 1;
    a2 = i1 - 4;
    a = a1 / a2;

    i2 *= 4;
    b1 = i2 + 2;
    b2 = i2 - 10;
    b = b1 / b2;

    c = io + 16;
    c = c / *o1;

    d = io + 24;
    d = d / *o1;

    return (a+b) / (c-d);
}
```

Figure 5. A simple piece of code we will use as a working example.

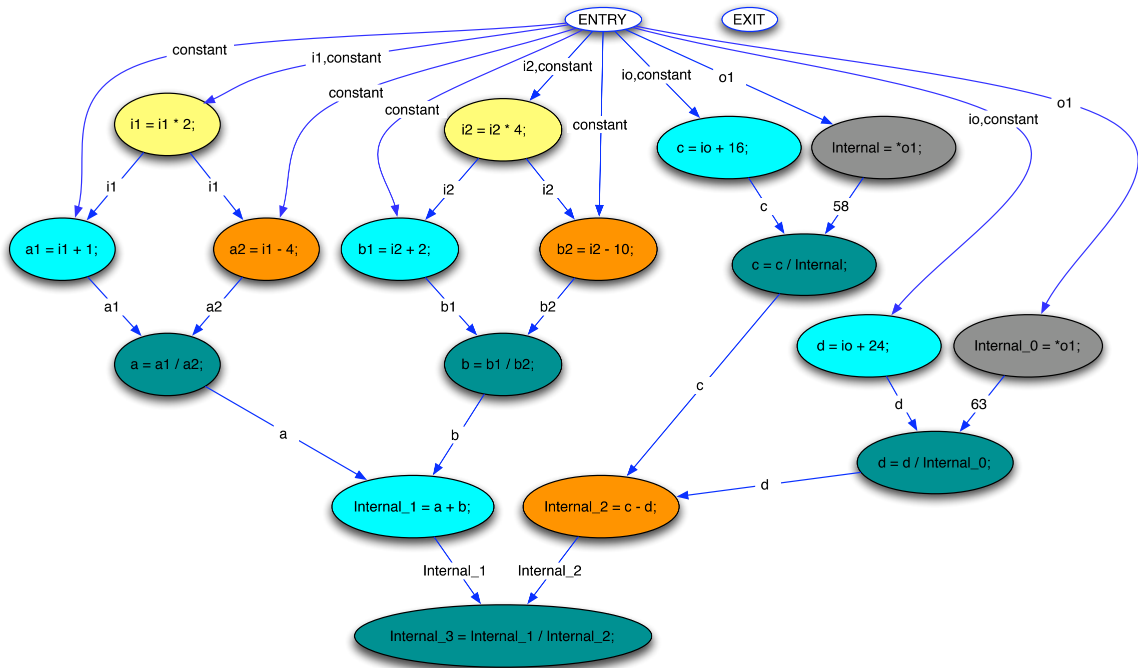

Figure 6. Data flow graph representing the code in Figure 5. Colors of the nodes represent the operation type.

to use such languages for writing embedded systems specifications. The corresponding data flow graph, in which each node is labeled with the line of code it represents, is in Figure 6. In this graph, the nodes are colored according to the operation they perform; in other words, the color of a node o graphically represents $\alpha(o)$. Note that some apparently meaningless variable names appear in the DFG: they are "dummy" variables created by the infrastructure that automatically generates the graph in order to split statements into elementary operations. More about the data structures used in the actual implementation will be discussed in chapter 5.

3.3 Intermediate phases

We have now established the starting point for our processing: a directed, acyclic graph as defined in the previous section. It is now time to define exactly what information we want to extract from this graph.

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 units, or *cores*.

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

$$S = \langle O_S, P_S \rangle$$

of the graph

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

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 $(\bigcup_{i \ldots n} S_i) = O$. 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 interme-

diate representation, which takes the form of a task graph T, having the previously identified subgraphs as its vertices, defined as follows:

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

with the edge set constructed as

$$\forall \left(o \rightarrow o'\right) \in P, \left((o \in S_j) \wedge \left(o' \in S_k\right) \wedge j \neq k \iff (S_j \rightarrow S_k) \in P_T \right)$$

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 clusters (41).

The nodes of T will afterwards have to be scheduled according to their dependencies, and at the same time *allocated* on the reconfigurable device to a suitable location. This scheduling and allocation phase will have to take into account that each of the identified tasks will have to be allowed some *configuration time* before being able to start execution: remember we are working on a reconfigurable device and, as such, we have to *program* it before it can carry out any computation. As has been shown in other work, the time needed to configure some part of such a device can be modeled with satisfactory accuracy as being affine in the size of the area affected by the reconfiguration (13). Special attention will have to be paid to issues such as minimizing the need to spend time reconfiguring the device, e.g. avoiding, in having to place

a new core on the device, to use the same area currently allocated to another core if it is still needed for unexecuted tasks; on the other hand, reclaiming the device area used by cores for which all the tasks that could be run on them have already been executed; trying to minimize fragmentation of the device area in order to be able to fit larger cores in the unused space.

3.4 Reconfigurable unit size

The class of current reconfigurable hardware devices which this work takes into consideration, i.e. FPGAs, typically present a structure in which the behaviour that the designer wants to obtain is implemented by mapping it onto a set of configurable blocks which, depending on the particular device family, might show a different level of granularity (e.g. number of I/O bits each of the blocks has) and specialization (e.g. only having a set of identical simple logic blocks such as lookup tables, or also having some dedicated blocks for commonly used functionalities such as multipliers).

An important aspect to be kept in mind however is that, in general, the device does not allow each of the logic elements to be configured *individually*, i.e. the smallest configurable unit has a size larger than that of a single logic block.

The Xilinx Virtex-II FPGA family, for instance, presents a reconfigurable area modeled as a rectangle, in which reconfiguration takes place *by columns*: a whole column has to be reconfigured even if the designer only wishes to modify one of the logic blocks contained in it.

In this work, we will take this aspect into consideration by referring to $Size_{rec}$ as the size of the reconfigurable unit. This will be useful in the definition of metrics to guide the partitioning phase.

CHAPTER 4

THE PROPOSED APPROACH

This chapter presents the approach to the partitioning problem that was chosen. As is explained in 4.1, privileging the identification of clusters showing a great deal of similarity among them appeared attractive. This problem, however, is computationally intensive and demands the definition of heuristics for us to be able to treat it. 4.2 offers some insight into the two algorithms that were applied.

4.1 Self-Similarity: rationale

As was briefly noted in 3.3, each of the modules that are identified by the partitioning phase in our approach will have to be *configured* on the device prior to being ready for execution. Moreover, we have to keep into account that if a module is *overwritten* by another one, and later it is needed for further computations, it will have to be configured back onto the device.

Unfortunately, the reconfiguration time issue is not of little importance: current FPGA technology requires reconfiguration times which are typically *orders of magnitudes* larger than the actual time which will be spent in the execution of the configured functionality.

These reasons compel us to try and minimize the need for reconfiguration during the execution of our application. A promising way of doing this is the recognition of recurring structures in the system specification: if we manage to detect the repetition of *similar computation pat-*

terns in different parts of the specification, we can then try to identify a *single module* that performs the recurrent computation, and *reuse it* more than once in the application's lifetime.

On a higher level of granularity, it's as if the task graph T defined in 3.3 became a new data flow graph in which the operation types, instead of being elementary operations as in the original one, are now whole tasks, comprising sets of the original operations. Our aim is to detect recurrent structures so that the same "type" will be associated to more than task in the graph. Tasks of the same type can be executed on the *same core* on the device, which we will need to configure only once thus saving reconfiguration time.

We now have to define what we mean by *similar*, though. We are looking, for our purpose, for a *very strict* kind of similarity. Being able to use the *same* core, without needing to reconfigure it in any way, implies that the parts of the specification we consider have to be not only *similar*, they have to be *exactly the same*.

In our context, the translation in graph-theoretic terms of *exactly the same* is *isomorphic*. The definition of isomorphism that we take into consideration is somewhat extended to take into account the peculiar features of our DFG, such as the association of each vertex to its operation type and the numbering of inputs. Let us first consider the usual definition of isomorphism for directed graphs:

Definition 1 (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.*

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

In order to extend the above definition to take into account our "augmented" graph, we add a couple more constraints:

Definition 2 (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 1 and, in addition, the following hold:*

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

and

$$\forall \left(o \rightarrow o'\right) \in P_1, \mu\left(o \rightarrow o'\right) = \mu\left(p(o) \rightarrow p\left(o'\right)\right)$$

where α *and* μ *are defined as in 3.2.*

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 (2; 3), and is defined as follows:

Definition 3 (Isomorphic Subgraphs) *Given a graph* G*, find two disjoing isomorphic subgraphs* G_1*,* G_2 *of* G*.*

Unfortunately, this problem is known to be NP-complete (2) in its general form, so that we are compelled to either resort to some heuristic, or place some restriction on the structure of the subgraphs we are looking for in order to be able to make some simplifying assumptions.

4.2 Partitioning algorithms

As a general approach, it was chosen to clearly separate the partitioning phase into two steps:

- *template generation*: detecting isomorphic structures inside the graph can in general produce overlapping clusters, clusters contained into one another. It becomes then a goal of primary importance to properly organize and structure the data produced in this phase in order to ease the process of choosing which clusters to choose for the implementation of the reconfigurable systems.
- *graph covering*: once the previous phase has produced a *set of templates*, each with its associated set of instances in the original graph, we have to decide which of these to use as cores in our reconfigurable system. Methods for choosing the most suitable of the identified subgraphs range from simply trying to reuse the most area, and thus use the biggest clusters or the most often repeated ones, to more subtle considerations of device area fragmentation due to the technological structure of the reconfigurable hardware.

As an illustrative example, Figure 7 shows a template, consisting of three nodes, and its two instances in the same DFG shown in Figure 6.

At the current state of this work, two different algorithms for regularity extraction have been implemented:

- The first one copes with the inherent complexity of isomorphism extraction by restricting the structure of the subgraphs it identifies to reversed trees, i.e. subgraphs with only one

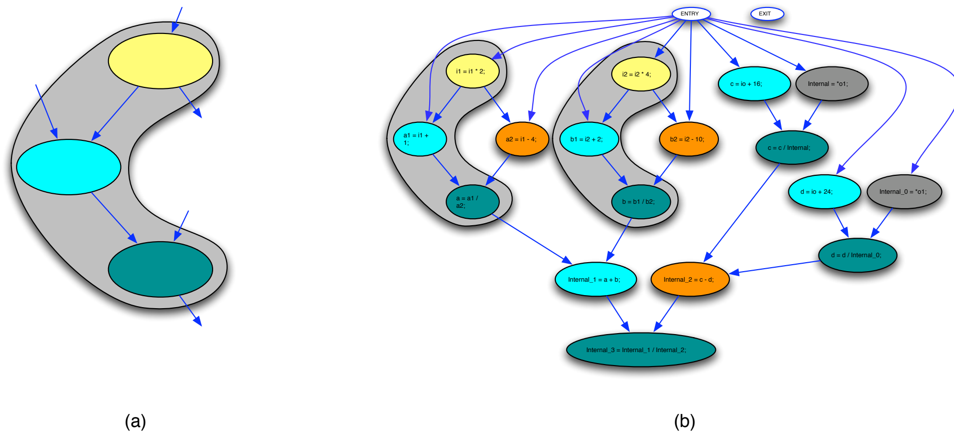

Figure 7. A template (a) with two instances in a data flow graph (b). This is the same graph as in Figure 6.

output, in which each internal node has only one output as well. This algorithm was first developed with the goal of recognizing regular structures in datapath circuits (12).

- The second algorithm, which originated in a purely graph-theoretical setting (3), adopts a heuristic to try and recognize *pairs* of isomorphic subgraphs[1], i.e. solving the Isomorphic Subgraphs problem as defined above, without restricting their structure. The core of the algorithm is based on the expansion of known isomorphic subgraphs by adding nodes from their neighborhood, and extending the bijection defining the isomorphism by means of a bipartite matching.

---

[1]For each run of the algorithm, only a pair of subgraphs is produced. Since we are interested in generating templates with a class of instances, we will have to extend the algorithm in this sense, as we will see later.

The following sections give more insight into the two approaches.

4.2.1 Tree-shaped templates

This algorithm, as we will see, works based on the assumption that the input graph is *acyclic*, which as we already mentioned is a feature of our data flow graphs. The need for the graph to be acyclic stems from the fact that this approach requires the nodes to be topologically sorted in order to achieve its complexity advantage over the general approach. Let us first of all explain how the algorithm identifies templates in the graph. Algorithm 1 represents the main loop which goes through pairs of vertices, where the second always follows the first in topological order, and relies on the function $LargestTemplate$ to detect whether the two are roots of a common template.

The data structures involved in this algorithm are the following:

- G is the input data-flow graph
- S is the output that the template generation produces, i.e. a set of templates
- $template_{i,j}$ stores, for each two vertices i and j, the common template rooted at the two vertices, if any
- $rootNodes_{template_i}$ is the set of nodes at which the instances of $template_i$ are stored

Remember we are trying to detect tree-shaped templates here. The rationale behind the way this algorithm works is that, at any iteration, we are trying to see whether a common template exists that has respectively i and j as its *roots*, i.e. the nodes that generate the *only output* of the tree.

Algorithm 1 $identifyTemplates\,(G)$

```
Require: G is a DAG
  S ← ∅
  topologically sort the nodes of G
  for each node i of G do
    for each node j of G following i in topological order do
      currentTemplate ← LargestTemplate (i, j)
      if  template ≠ nil then
        template_{i,j} ← currentTemplate
        if template ∈ S then
          rootNodes_{currentTemplate} ← rootNodes_{currentTemplate} ∪ {i, j}
        else
          S ← S ∪ currentTemplate
          rootNodes_{currentTemplate} ← {i, j}
        end if
      end if
    end for
  end for
  return S
```

A point worth noting here is that the graph is traversed according to the topological ordering of its vertices. This means, at any given iteration of the loop the ancestors of the current nodes, if any, *will already have been visited*. This is a fundamental point, as we will see, on which the $LargestTemplate$ function is based.

Once the $LargestTemplate$ function has been computed, if it has detected no common template rooted at the two current nodes then the pair is simply discarded and we proceed to the next one. If instead a common template has been detected, then as a first step we perform some bookkeeping: we update the $template_{i,j}$ variable to account for the fact that the current

template has two instances rooted at nodes i and j. This information will be used later on, when the algorithm considers children of the current nodes.

At this point, all that remains to do for the current pair is to see whether the template they have in common is a known one or if, instead, it is the first time we encounter such a template. In either case, at the end of the computation for the current pair we will have the template inserted in set S, and the current pair of nodes will have been added to the $rootNodes_{template}$ set.

Let us now consider how the *LargestTemplate* function (Algorithm 2) carries out its task. In order to understand its workings, we first have to clarify how a template is treated in this approach to the problem. Thanks to the fact that we are identifying tree-shaped templates, we can efficiently store them by memorising the type of the root node, and keeping track of which other templates are *children* of a given one.

A template is as such composed of two elements[1]:

- *rootFn* is simply the kind of operation performed by the root of the template
- *childrenTemplates* is a list of children templates, i.e. templates rooted at ancestors[2] of the root node of the current template.

---

[1]in the pseudocode explaining the algorithm, this is represented with a C-like dot notation for the sake of readability

[2]This might be somewhat confusing, but we must keep in mind here that the term "tree-shaped" is used in a somewhat stretched way: we are in fact identifying templates which are shaped like reversed trees, i.e. their root is a node that has no children inside the template.

It is of basic importance to note here that, since the graph is visited in topological order, then the templates which are candidates to be children of the one we are building have, at any time, *already been generated*.

Other data structures accessed in this function are the same used in algorithm 1, with the exception of $\alpha(\cdot)$ which we had defined as the operation type associated with a given node (see 3.2). Moreover, we implicitly access $\mu(\cdot)$ when ordering the ancestors according to the index of the input they provide.

Given the rather strong assumptions about the kind of templates we wish to generate, and the restriction to a DAG of the input graph, the way $LargestTemplate$ works is rather straightforward. First of all, if the two nodes currently being considered represent different operation types, then we can be sure that they do not have any common template rooted at them. Otherwise, they do have a common template, even if just of just a node's size: $rootFn$ is then set to the common operation type of the two current nodes. Next, the ancestors of the current nodes are taken into consideration, in order to see whether they have any previously identified template in common. Once again, keep in mind that due to the topological ordering of the graph we can assure that these ancestors have already been visited at this time. We also have to take into account the ordering of the inputs here: the template we are identifying can be expanded to include two ancestors only if these ancestors provide the same input for the current nodes. Here is where the $template_{i,j}$ variables are actually used: since templates for the ancestors have surely already been detected, then we can be sure that, if two ancestors providing inputs at the same index share a template, we will find it in the respective $template_{i,j}$

variable. If such a template is found, then it is added to the *childrenTemplates* set for the template currently being generated.

An important point to note is that *LargestTemplate* only adds a children template *if the ancestors at which it is rooted have a single output.* This is done to ensure that the identified instances of a given template are nonoverlapping, and to restrict the algorithm to maintain the tree structure when generating templates.

Once all the ancestors have been considered, the current template is ready and can be returned.

Algorithm 2 $LargestTemplate\,(u, v)$

```
if α(u) ≠ α(v) then
  return nil
else
  sharedTemplate.rootFn ← α(u)
  for each pair (a_{u,k}, a_{v,k}) of ancestors of u and v providing inputs at the same index k do
    if (a_{u,k}, a_{v,k} only have one child) ∧ template_{a_{u,k},a_{v,k}} ≠ nil then
      add template_{a_{u,k},a_{v,k}} to sharedTemplate.childrenTemplates
    end if
  end for
  return sharedTemplate
end if
```

As an example of how this procedure for template identification works, consider Figure 8. Note that the figure is intentionally incomplete, i.e. it does not show *all* of the generated templates, for the sake of clarity. The templates are numbered in the same order as they would

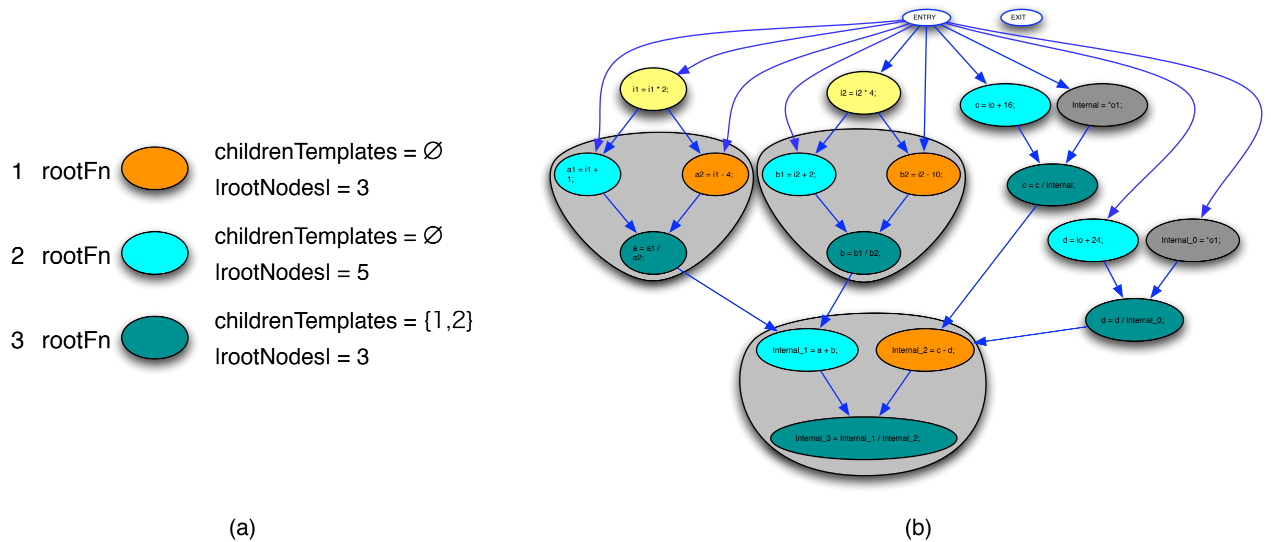

Figure 8. Example of how tree-shaped templates are generated. In (a) we show the way the generated templates are stored, while (b) shows where the identified templates have their instances in the graph.

be generated, i.e. ancestors first. Once it has generated templates 1 and 2, the algorithm would then generate template 3 and, noticing that ancestors of the root nodes of template 3 are roots of the same previously identified templates, would add them to the *childrenTemplates* for template 3.

We have now explained how this algorithm identifies templates in the graph. We still have to look into how it performs the second step, i.e. the covering of the graph using the generated templates. Algorithm 3 shows the adopted approach to this task: first of all, we perform the template identification step as outlined in algorithm 1. We then choose one out of the

identified templates according to a heuristic, represented here as a *chooseTemplate* function, which represents the *fitness* a template has to being implemented as one of the configurable units. Some ideas about how to define this heuristic will be discussed later. Once a template has been identified, we simply consider all of its instances one by one (by going through the root nodes of the chosen template) and, if they do not overlap with other instances, we identify a cluster in our partitioned graph containing the vertices belonging to the current instance. The last step in the algorithm is the removal of the vertices covered by the current chosen template from the input graph. The new G, lacking the vertices already covered, is then passed through another template identification step, and so forth until no more templates can be identified.

Algorithm 3 Graph Covering using Tree Templates

repeat
  $templates \leftarrow identifyTemplates\,(G)$
  $chosenTemplate \leftarrow chooseTemplate\,(templates)$
  $coveredVertices \leftarrow \emptyset$
  for each node n in $rootNodes_{chosenTemplate}$ **do**
    if $vertexSet\,(chosenTemplate, n) \cap coveredVertices = \emptyset$ **then**
      create a new cluster containing $vertexSet\,(chosenTemplate, n)$ in the partitioned graph
      add $vertexSet\,(chosenTemplate, n)$ to $coveredVertices$
    end if
  end for
  remove the nodes in $coveredVertices$ from G
until $templates = \emptyset$

4.2.2 Unconstrained shape templates

Let us now move on to the second approach to template generation that was implemented. This method generates templates of unconstrained size, and employs a heuristic at the heart of which is the solution of a bipartite matching problem in order to extend the isomorphism bijection between two knowingly isomorphic clusters to new nodes chosen from their neighborhoods (3; 41). As a starting point, the original definition of the algorithm used single nodes. In our setting, it appears viable to start from sets of vertices of cardinality two[1].

Our first step, then, is to generate initial sets of subgraphs which are isomorphic *by construction*. As we can see in algorithm 4, we iterate over the edges in the input graph, and classify them in sets according to the operation types of source and destination, and in addition according to the input number provided by the edge. This way, the sets we build will actually contain a class of isomorphic subgraphs, of size 2 vertices. These are the isomorphic *"kernels"* which we will use as starting points for the proper isomorphism detection algorithm. Once we have generated the initial sets, so that all the edges have been classified, we sort them by decreasing order of cardinality. The rationale behind this step is that the most numerous sets will be the best candidates to produce templates with the most instances, so that if in a later step we decide to stop the template generation at a certain point, we will already have considered the most promising starting sets.

---

[1]as we will see later in chapter 5, this is also due to the data structure used for the implementation

The following statement, on the other hand, decides which sets are to be discarded *a priori*, e.g. the ones that only have one edge in them are useless from our point of view.

At this point, we are ready to actually start running the core algorithm. In this first exposition of our approach, we will assume the algorithm is run, in the context of each starting set I, for every possible pair of edges. This means that, for each initial set I, $\frac{|I|*(|I|-1)}{2}$ invocations of the algorithm will occur.

Algorithm 4 Template generation - unconstrained shape

---

for each edge $(s \rightarrow d)$ in graph G **do**
    $\alpha_s \leftarrow \alpha(s)$
    $\alpha_d \leftarrow \alpha(d)$
    $\mu \leftarrow \mu(s \rightarrow d)$
    add $(s \rightarrow d)$ to set $I^{\alpha_s}_{\alpha_d,\mu}$
end for
sort the I sets in decreasing cardinality order
prune the less promising I sets
for each set $I^{\alpha_s}_{\alpha_d,\mu}$ to be considered **do**
    for each pair $((s_1, d_1), (s_2, d_2))$ of edges in $I^{\alpha_s}_{\alpha_d,\mu}$ **do**
        expandTemplate(current edge pair)
    end for
end for

Let us now consider how the template expansion step, which constitutes the core of this algorithm, works in detail. The first operations, as is shown in algorithm 5, are the initialisation of V_1 and V_2, i.e. the sets that contain the vertices belonging to the two clusters being expanded. V, i.e. the bijection that defines the isomorphism between the two clusters, is also initialized.

P holds a queue of pairs of isomorphic vertices, and acts as a "work list" for the algorithm: if a pair of nodes is in P, then neighbors of those two nodes[1] will be considered for expansion of the isomorphism relation and, thus, of the clusters themselves.

Algorithm 5 $expandTemplate\left(\left(s_1, d_1\right), \left(s_2, d_2\right)\right)$

1: $V_1 \leftarrow \{s_1, d_1\}$
2: $V_2 \leftarrow \{s_2, d_2\}$
3: $V \leftarrow \{\left(s_1, s_2\right), \left(d_1, d_2\right)\}$
4: empty queue P
5: enqueue $\left(s_1, s_2\right), \left(d_1, d_2\right)$ into P
6: **while** P is not empty **do**
7: $\left(c_1, c_2\right) \leftarrow$ dequeue from P
8: $N_{left} \leftarrow enumerateNeighbors\left(c_1\right)$
9: $N_{right} \leftarrow enumerateNeighbors\left(c_2\right)$
10: formulate the matching problem between the elements of N_{left} and N_{right}
11: **for** each pair $\left(n_l, n_r\right)$ in $N_{left} \times N_{right}$ **do**
12: $w_{n_l,n_r} \leftarrow computeMatchingWeight\left(n_l, n_r\right)$
13: **end for**
14: $matching \leftarrow solveMatching()$
15: **for** each match $\left(m_l, m_r\right)$ in the solution **do**
16: **if** $isomorphismCheck() = true$ **then**
17: $V_1 \leftarrow V_1 \cup \{m_l\}$
18: $V_2 \leftarrow V_2 \cup \{m_r\}$
19: $V \leftarrow V \cup \{\left(m_l, m_r\right)\}$
20: enqueue $\left(m_l, m_r\right)$ into P
21: **else**
22: discard the current match
23: **end if**
24: **end for**
25: $classifyTemplate\left(V_1, V_2, V\right)$
26: **end while**

[1] which do not already belong to the clusters

We then get to the main loop of the algorithm, which is run until the queue P is empty. The first element in queue is considered, and two sets N_{left} and N_{right}[1] are filled with, respectively, the neighbors of the first and second nodes in the pair dequeued from P. It is worth noting here that, in our version of this algorithm, both neighbors encountered along edges *leaving* the current node, i.e. successors, and edges *entering* the current node, i.e. ancestors, are taken into consideration.

Once the two neighbor sets are built, we compute the weight, i.e. the "goodness" of every possible match between the two. This is represented in algorithm 5, as a call to the $computeMatchingWeight$ function. In the currently implemented version of the algorithm, the considerations involved in the computation of the matching weights are:

- a negative weight is assigned if a matching would surely not produce an isomorphic expansion of the clusters, i.e.
    - the two nodes involved in the match implement different operation types
    - the edges leading to[2] the neighbors involved in the match provide different input numbers
- the matching weight is increased if the nodes being matched have the same fanout and fanin, this is likely to increase the probability that further expansions will be possible starting from the new nodes.

---

[1] "left" and "right" with reference to the bipartite matching problem they will be used to define

[2] or coming from, since we also consider ancestors

Once we have assigned the matching weights, we proceed to solve the matching problem. In the current implementation, the bipartite matching problem is formulated as an ILP problem and solved that way.

With the solution to the matching problem ready, we still have to verify if adding the nodes that ended up being matched actually does yield subgraphs that are isomorphic. We then perform a *local isomorphism check* to the nodes candidates for being added to our clusters, which is represented by the *isomorphismCheck* function. If the new nodes pass the isomorphism check, they are added to the clusters, and the needed updates to the isomorphism bijection and to the queue are made. Otherwise, the match is discarded since it has proven to not produce a useful addition to our clusters.

At this point, neighbors of the current node pair dequeued from P have been considered, and the useful ones have been added to the clusters. We now proceed to classify the new pair of isomorphic graphs that we have obtained, i.e. the node-induced isomorphic subgraphs which have V_1 and V_2 as their vertex sets, in order to find out whether they belong to any already identified isomorphism class and keep track of it. This is represented by the $classifyTemplate$ call, and is the last operation carried out for a given pair extracted from the queue P.

When the current clusters have been expanded to a point where no more neighbors can be added to produce bigger isomorphic subgraphs, the queue will empty and the algorithm will stop.

A simple example of how the algorithm works can be seen in Figure 9. For each step of the algorithm the current pair of clusters is shown, along with the contents of the expansion queue

P, the matching that is performed between the neighbors of the first pair of nodes in the queue, and the nodes added to the clusters in this step, if any. The first step is simply the situation at the initialization of the algorithm: the clusters that are being expanded still include only two vertices each, and both pairs of vertices are present in the queue. The dark green vertices are considered first, and the matching between their neighbors produces the addition of the orange nodes. The light blue node is not added in order to keep the subgraphs from overlapping. Similarly, in the next step the yellow nodes are added to the two subgraphs. Every other pair of nodes in the expansion queue would, at this point, only produce overlapping subgraphs, so that no further expansions are carried out and the algorithm stops.

Let us now consider the local isomorphism check step, which was mentioned above. At a first sight, it may seem that making sure the operation types for two neighbors match is enough to ensure the isomorphism of the clusters with the neighbors added to them. This is not quite true: we must keep in mind, in fact, that we are considering *node-induced subgraphs*, i.e. subgraphs which include the vertices in their vertex set and *all the edges from the original graph that have both source and destination in the subgraph.* Figure 10 illustrates this problem with a simple example. In the figure, the six nodes surrounded by the grey area are already part of the two isomorphic clusters, which consist of three nodes each. The nodes labeled c_1 and c_2, as in the algorithm's pseudocode, are part of the pair that has been dequeued from P and whose neighbors are being considered for addition to the clusters. After the matching problem has been solved, the nodes labeled m_l and m_r have been matched. Their operation

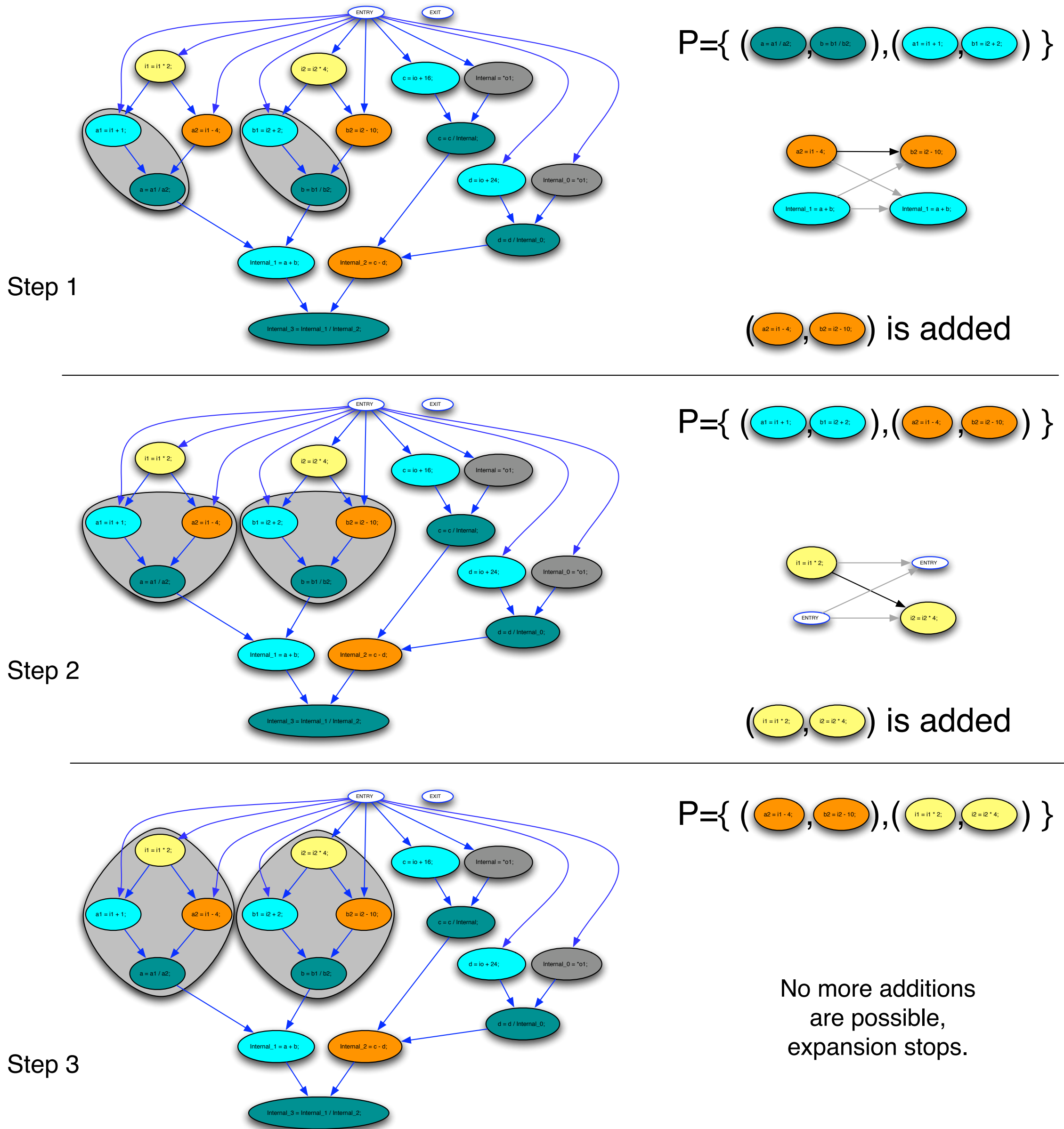

Figure 9. Unconstrained Shape Template generation: example

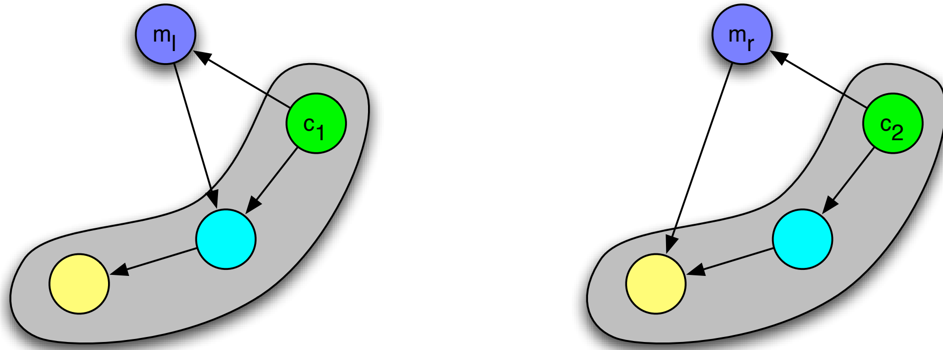

Figure 10. Local Isomorphism Check: example

type is the same, but adding them to the clusters would *not* produce isomorphic clusters of size 4!

Notice, in fact, how adding m_l and m_r to their respective subgraphs would also cause the addition of the edges from m_l to the light blue node and from m_r to the yellow node. The resulting subgraphs would then not be isomorphic.

In order to avoid this kind of issue, the local isomorphism check has been implemented as is shown in algorithm 6. The idea behind the check that is performed is indeed pretty simple: we build two sets, e_l and e_r, which represent respectively the edges that would be added in the left and right clusters[1], if we added the node currently being checked. Each edge is represented by a triple, $\langle inout, pair, \mu \rangle$, in which:

[1] "left" and "right" are still intended as the left and right sets of the bipartite matching

- *inout* can take two values: *in* if the edge being described has the newly added node as its destination, or *out* if the opposite happens
- *pair* is the pair of nodes in the isomorphism bijection to which the node already in the cluster which is the source (or sink) of the current edge belongs
- μ is the input number provided by the edge

The algorithm for local isomorphism check simply populates the two sets, with edges to and from the nodes being considered for addition to the clusters. When the sets have been populated, a simple comparison is performed: if the contents of the two sets are the same, then the local isomorphic check is passed. Otherwise, addition of the two new nodes is rejected.

One last step remains to be taken into consideration to complete our explanation of this template generation method: the $classifyTemplate$ step. We store information about the classes of isomorphic subgraphs that we have identified up to now in the $class_i$ sets. In other words, every index i identifies a template, which has as its instances the members of the class $class_i$. The template classification step works (as is shown in algorithm 7) by first checking whether classes already exist that contain either the first or the second cluster.

As a reminder, at the time when this procedure is performed we have two subgraphs, based on the vertex sets V_1 and V_2, which are known to be isomorphic. Due to the way we invoke $expandTemplate$ multiple times, there will be instances in which one of the two, or both, current subgraphs have already been generated, even if from a different sequence of expansions.

We now behave differently according to whether V_1 or V_2 already have been classified or not:

Algorithm 6 *isomorphismCheck*

$e_l \leftarrow \emptyset$
$e_r \leftarrow \emptyset$
for each edge $(m_l \rightarrow o) \in edges\,(G)$, with $o \in V_1$ **do**
  $e_l \leftarrow e_l \cup \langle out, (o, V\,(o))\,, \mu\,(m_l \rightarrow o) \rangle$
end for
for each edge $(m_r \rightarrow o) \in edges\,(G)$, with $o \in V_2$ **do**
  $e_r \leftarrow e_r \cup \left\langle out, \left(V^{-1}\,(o)\,, o\right)\,, \mu\,(m_r \rightarrow o) \right\rangle$
end for
for each edge $(o \rightarrow m_l) \in edges\,(G)$, with $o \in V_1$ **do**
  $e_l \leftarrow e_l \cup \langle in, (o, V\,(o))\,, \mu\,(o \rightarrow m_l) \rangle$
end for
for each edge $(o \rightarrow m_r) \in edges\,(G)$, with $o \in V_2$ **do**
  $e_r \leftarrow e_r \cup \left\langle in, \left(V^{-1}\,(o)\,, o\right)\,, \mu\,(o \rightarrow m_r) \right\rangle$
end for
if $e_l = e_r$ **then**
  return $true$
else
  return $false$
end if

- if neither V_1 nor V_2 belong to a class yet, then we simply create a new one for the two of them
- if one of V_1 and V_2 belongs to a class, but the other does not, then we simply add the one that still isn't found in any class to the class which already contains the other. Isomorphism is a transitive relation.
- if both V_1 and V_2 already belong to a class, then we behave as follows:

– if they belong to *two different* classes, then it means that the two classes are actually the same, due to the transitivity of the isomorphism relation. We then proceed to merge them.

– if they belong to *the same* class, then it means they have already been identified and expanded further, so that we can stop considering the present pair of clusters.[1]

After the templates have been generated and classified into equivalence classes as was presented above, we have to choose which ones to use to produce our task graph. An idea of how to proceed with the data produced in this fashion is outlined in algorithm 8: we simply apply a certain heuristic in order to select the template to use - more on this in 4.2.3 - and use all of its instances to cover the graph, and then proceed to removing all the instances of other templates that overlap with any of the clusters just covered by instances of the selected template. Templates that after this step are left with less than two instances are removed, since at this point they do not expose any additional regularity in the specification.

4.2.3 Graph covering

As we have seen above, different ways can be devised to extract regular patterns from a given system specification in the form of a data-flow graph. The bottom line is, however, that once we have performed the so-called *template identification* step, we are left with a set of equivalence classes, each containing two or more isomorphic subgraphs. These classes potentially show

---

[1]In doing so, however, we might prevent the algorithm from exploring some of the possible patterns. This is due to the fact that the two clusters, even if they already are in the same class, have probably been generated from a different sequence of expansions than the current one and, as such, will probably also have different queues.

Algorithm 7 $classifyTemplate\,(V_1, V_2, V)$

```
i_1 ← findClassIndex (V_1)
i_2 ← findClassIndex (V_2)
if i_1, i_2 = nil then
  create a new class class_k
  class_k ← {V_1, V_2}
end if
if i_1 = nil ∧ i_2 ≠ nil then
  add V_1 to class_{i_2}
end if
if i_1 ≠ nil ∧ i_2 = nil then
  add V_2 to class_{i_1}
end if
if i_1 ≠ nil ∧ i_2 ≠ nil then
  if i_1 ≠ i_2 then
    class_{i_1} ← class_{i_1} ∪ classi_2
    class_{i_2} ← ∅
  else
    abort expansion of the current clusters: P ← ∅
  end if
end if
```

Algorithm 8 Graph covering step

```
while there still are usable instances in class_i for at least one i do
  k ← selectTemplate()
  generate a task graph node for each of the instances in class_k
  for each i do
    for each instance u ∈ class_i do
      if u ∩ u_k ≠ ∅ for any u_k ∈ class_k then
        remove u from class_i
      end if
    end for
    if |class_i| < 2 then
      delete class_i
    end if
  end for
end while
```

very different characteristics from one another, both in size of the subgraphs (be it simply the number of vertices or a more refined estimate of the hardware resources needed to implement the given pattern) and in their structure (e.g. the number of connections that nodes inside the pattern have to and from vertices not belonging to it).

We then have to define a way to choose which of the generated templates is *better*, i.e. more suited to being efficiently implemented as a configurable hardware unit. Different metrics to this end have been considered, among which are some very straightforward ones like the following:

- Consider the template that has the largest size first, also known as *LFF (Largest Fit First) heuristic* (12). The point of such choice is to allow greater optimization possibilities to later stages of the design flow, i.e. intra-module placement and routing, but has the potential adverse effect of choosing templates so large that they could use up a big part of the device, thus leaving little room for configuring more modules at the same time.
- Consider templates with the most instances first, also known as the *MFF (Most frequent Fit First) heuristic* (12). Conversely, this approach allows for more moderate area usage in that it tends, in principle, to maximise reuse of the identified units possibly at the expense of a worse schedule length in the implemented system.

Due to the properties of our target hardware devices and to the way we plan to implement later stages of our workflow, though, we can propose some more specific metrics to tackle the template selection problem.

As we pointed out in 3.4, the size of units which the devices allow to be reconfigured is, in general, not a single logic block. Our goal to obtain a dynamically reconfigurable system in which a core can keep working regardless of other parts of the device being reconfigured, it is clear that this can only be achieved if the sets of reconfigurable blocks occupied by each core do not overlap at any given time.

This also means that, if a given core does not use a whole number of said reconfigurable units, some device area will be wasted. We model this by writing

$$A_{used} = \left\lceil \frac{A_{core}}{Size_{rec}} \right\rceil Size_{rec}$$

, i.e. the actual area used on the device by a given core is the size effectively occupied by the core's logic rounded up to the next reconfigurable unit.

Let us now get back to the actual goal we are trying to reach: the scenario in which dynamical reconfiguration is interesting is one in which the area offered by the device is not enough to implement an optimal scheduling for the whole specification. In this respect, one of our objective has to be the optimization of area usage. One of the aspects we might take into consideration when choosing which templates to employ as cores, then, is *how much of the entire specification is potentially implemented by that template* with its instances, which we will denote as $\frac{|\bigcup_i instance_i|}{|O|}$, i.e. what fraction of the total vertices of the data flow graph are covered by the template's instances.

We could, under these assumptions, consider such a metric to choose which template to consider:

$$\frac{A_{used}}{A_{device}} \frac{|\bigcup_i instance_i|}{|O|}$$

which privileges templates covering a large part of the graph while occupying a small fraction of the device.

Yet another consideration could be trying to minimize the cost of inter-core communication, which could be evaluated in our representation as the number of edges crossing a subgraph's borders, i.e. for a subgraph with vertex set V defined in a graph $G = \langle O, P \rangle$:

$$|\{(o_1 \to o_2) \in P \mid o_1 \in V \wedge o_2 \notin V\} \cup \{(o_1 \to o_2) \in P \mid o_1 \notin V \wedge o_2 \in V\}|$$

Note that this approach is similar to that used for other partitioning scenarios, such as in (24).

CHAPTER 5

IMPLEMENTATION

This chapter provides some detail about the actual implementation that was carried out for this thesis work, including some insight on the structure of PandA, a research software project which provided the data structures for the implementation of the current work (5.1.1), and the Boost Graph Libraries (6; 43), extensively used in the development of this project (5.1.2).

5.1 Data Structures

The implementation for this thesis work was developed in the context of PandA, a research project currently in active development by the microarchitecture research team at DEI, Politecnico di Milano, Italy. The primary objective of PandA is the development of a practically usable *framework* to enable the exploration of new ideas in the field of *hardware-software co-design*. One of the goals being currently pursued is the definition of an efficient high-level synthesis tool that, starting from `C`, `C++`, or `SystemC` system descriptions generates synthesizable `VHDL RTL` descriptions.

The software framework being produced by the PandA project is, however, designed with extensibility and modularity in mind: it defines data structures that allow easy access to the system specification by means of abstract representations suited to being used for several purposes, among which the experimentation of new approaches in the reconfigurable hardware field.

5.1.1 The PandA framework

As was briefly noted above, the PandA framework was structured in a modular fashion; more precisely, it is composed of several sub-projects which tightly interact together. The most relevant of these subprojects include:

- *behavioral description layer*, whose responsibility is that of translating a behavioral description of a system, which in our case is a `C`, `C++`, or `SystemC` description, into an intermediate representation usable by the rest of the framework.
- *graph, technology, and circuit layers*: this subproject introduces a level of abstraction between the behavioral description layer and the alrogithms that make use of the system specification, such as the synthesis algorithms, with the aim of defining a common intermediate representation independent from the original encoding of the specification. Among the information managed by this subproject, we have:
  - the system specification itself, encoded in several formats emphasizing different aspects of it such as Control Flow Graph (CFG), Data Flow Graph (DFG), System Dependency Graph (SDG).
  - a description of the technological resources available for the subsequent synthesis steps, especially useful for the HLS scenario.
  - a schematic description of the system, i.e. the circuit structure given for example by the `SystemC` description.

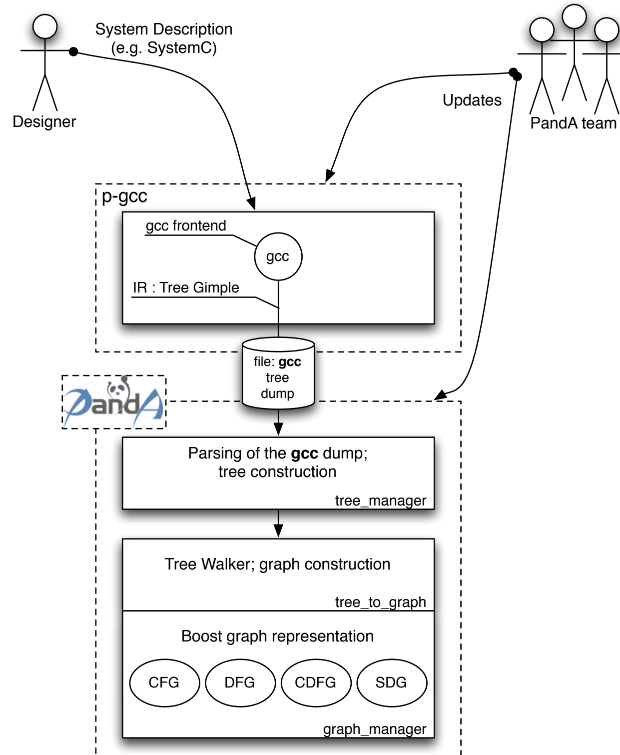

Figure 11. PandA: behavioral description layer

- *high-level synthesis layer*, with the aim of developing a high-level synthesizer producing the controller and data-path from a given input specification. This subproject includes an infrastructure for the development of scheduling algorithms, a fundamental phase of the high-level synthesis flow. Among the currently implemented scheduling algorithms, we have both heuristic (e.g. list-based) and exact (i.e. integer linear programming-based) approaches.

The areas of this infrastructure that interact most closely with the software developed for this thesis are the graph layer and, necessarily, the behavioral description layer that generates the graph data from high-level system specifications.

Let us briefly consider how they are organised. The behavioral description layer, as we noted above, is devoted to translating the system specifications from a high-level language to a representation usable by the rest of the framework. The first step in such a task is undoubtedly the parsing of the input code, certainly not a trivial task for a language of the complexity of `C++`. The choice was made to use a readily-available parser of proven stability: the one included in the `GNU GCC` compiler suite. A patch set is maintained[1] by the PandA team in order to add to `GCC` the ability to produce a dump of its internal representation containing all the needed information for our purposes. Using `GCC` as a parser allows us to delegate to it such compiler optimizations as loop unrolling and constant propagation. An overview of how this subproject is organized can be seen in Figure 11: the system specification as written by the designer is first fed into the patched version of `GCC`, labeled as `p-gcc` in the figure. After having been parsed and applied basic optimizations, the specification is dumped into a file, which substantially contains a syntax tree-like representation of the specification. This is the input format that the actual PandA code understands.

This representation is then read by a subsequent parser, which reconstructs the tree structure and extracts the hierarchy information from `SystemC` specifications (`tree_manager` in Fig-

[1] a the time of writing, patches are available against the tree-ssa branch of `GCC 3.5` and against `GCC 4.0`.

ure 11). Once this tree has been constructed from the input file, it is visited in order to build the graph representations. These representations are kept consistent by the `graph_manager`, shown as the bottom layer in the figure, and are actually managed as different views of the same information.

We have thus summarized how PandA constructs its internal representation starting from a high-level system description. The PandA framework is slated to be released in the summer of 2006 under the *GNU General Public License*(16).

The next section presents a brief overview of the way the system specification is represented and accessible inside PandA, especially for what concerns the aspects most relevant to the implementation that was carried out for this work.

5.1.2 The Boost Graph Library

Boost provides a set of free, portable `C++` source libraries. The Boost developers group, some members of which are also involved in the official `C++` standard committee, aim at maintaining libraries that work seamlessly with the `C++` Standard Template Library, extending it, and encourage usage in both commercial and non-commercial settings. Some of the libraries proviced by the Boost project are already being taken into consideration for inclusion into the `C++` standard. One of the libraries provided by the `boost` project, which is heavily exploited by the PandA framework, is the *Boost Graph Library* (BGL). This library offers a standardized, generic interface for building and traversing graphs, thus enabling the reuse of algorithms and data structures that would otherwise have to be adapted to the particular graph representation adopted. This is one of the main contributions of the BGL, which to this end provides different

graph representation models ready for use, among which *adjacency lists* and *adjacency matrices*, the latter especially desirable for dense graphs.

The library offers a great degree of customizability, allowing the user to simply specify whether graphs should be directed or undirected, whether they should be traversable only along the edges direction or in the reverse direction as well, or to which degree the representation should be tuned towards access speed at the expense of memory usage.

The Boost Graph Library is also fully interoperable with the Boost Property Map Library, thus allowing us to *label* objects in our graphs, i.e. vertices and edges, with custom data elements to carry whichever additional information we need. This is of particular relevance in this work, since it enables us to implement the needed *"coloring"* of the nodes according to their operation type in a straightforward way.

Other features meaningful to this work that are offered by the Boost Graph Library include the possibility of representing *subgraphs*, and of defining *filtered views* of graphs, which selectively hide some of the features of the graph according to a specifiable criterion. The former feature allows the user of the library to create a hierarchical structure of subgraphs, organized in a tree in which the original graph is the root, and each subgraph can in turn contain further subgraphs. The abstraction offered to the programmer is that subgraphs can be manipulated as regular graphs when it comes to visitor algorithms and such, yet their features such as vertex and edge labels are shared with their parent graph so that modifications to a (sub)graph are propagated to its ancestors and descendants in the subgraph tree. This feature is clearly useful

for our purposes in that it allows us to represent the partitioned Data Flow Graph, i.e. the final result of the partitioning phase.

Finally, the possibility of defining *filtered views* enables us, by means of a filter implemented as a `C++` functor object, to selectively hide parts of a graph. This is most useful in those cases in which the algorithms require to e.g. remove nodes from the DFG, in that by exploiting this feature we can accomplish the same effect without having to first create a separate copy of the graph, thus saving memory and time.

As an example of the data structure produced by PandA from `C` code, Figure 12 contains the Data Flow Graph automatically generated from the code in Figure 5, i.e. the same graph used in previous examples.

5.2 The Partitioning Phase

The partitioning algorithms described in 4 were implemented as part of the PandA framework, as such they take advantage of the features of the graph representation built by the behavioral description layer described above.

In order to allow for some degree of extensibility, an abstract interface was defined for the partitioning stage; this sets a clear point of contact between the partitioning phase and the subsequent scheduling step while allowing us to easily add other partitioning algorithms to be used by our workflow keeping the needed changes to a minimum. A class diagram depicting this hierarchy can be seen in Figure 13. The picture also shows a `TemplateEvaluator` class: it is an interface that allows different heuristics for template selection to be "plugged in" to be used in the graph covering phase.

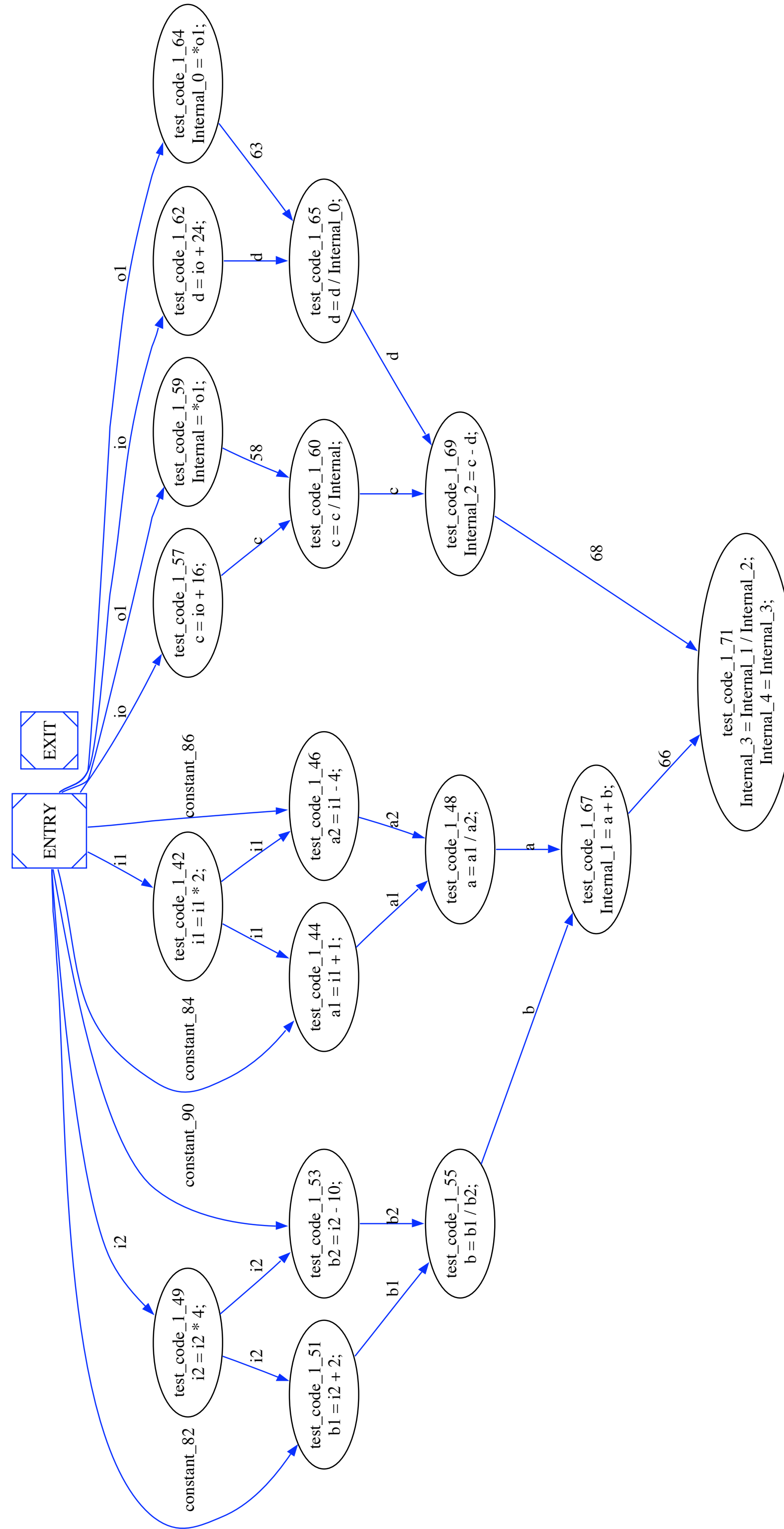

Figure 12. PandA: example of an automatically generated Data Flow Graph, representing the specification in Figure 5

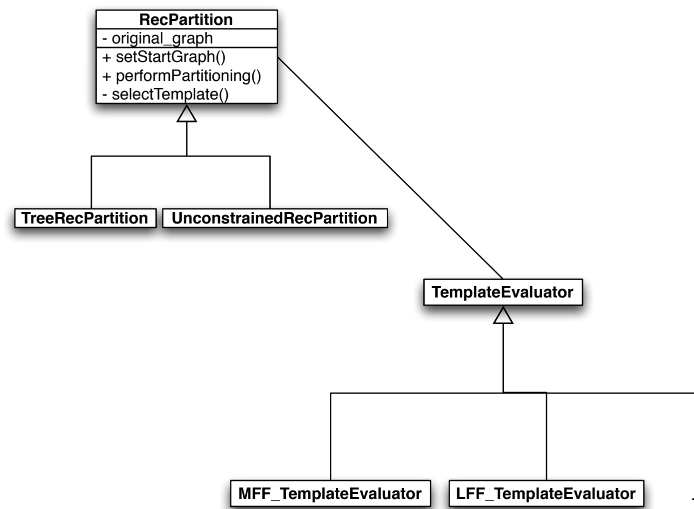

Figure 13. Partitioning phase implementation: class diagram

CHAPTER 6

RESULTS

The two regularity extraction approaches described in chapter 4 were implemented, and tests were carried out on some significant benchmarks.

Table I shows the results of the template identification phase run on some example applications. For each application, the number of nodes and edges in the generated DFG is reported. For the two algorithms, the table shows the number of nodes contained in an instance of the largest template identified, along with the number of instances of said template detected in the graph. The total number of identified templates, along with the time taken for computation, is also reported. As was expected, the algorithm detecting templates of unconstrained shapes is able to extract a greater amount of recurrent structures, which translates in detection of a much larger number of templates. The biggest templates identified are also larger, with the drawback of having a longer computation time.

TABLE I

Template Identification: Number of templates, maximum sizes

			Tree templates				Free Shape Templates			
Benchmark	$\lvert V\rvert$	$\lvert E\rvert$	Largest	#Inst	#Tmpl	Time	Largest	#Inst	#Tmpl	Time
AES - `encryptblock`	440	826	16	3	151	6.7 sec	132	2	6790	19.1 sec
AES - `decryptblock`	504	954	19	3	162	8.8 sec	147	2	11006	35.9 sec
DES - `des_encrypt`	287	509	38	4	57	2.8 sec	100	2	1802	6.1 sec
FDCT	232	416	6	6	40	1.7 sec	62	2	1470	4.7 sec

The reported computation times include all the processing performed by the PandA framework, i.e. the parsing of the dump produced by `gcc` and the construction of the graph representation, along with the actual time taken by the algorithms implemented for this thesis work. The results were generated by running a *single* template identification run, with no graph covering step following it.

Let us now consider some data about the whole partitioning process, including the graph covering phase. Table II summarizes the results of the partitioning step, including the graph covering phase which was excluded from the data in Table I, using the same input specifications. For the generation of this data, the partitioner was instructed not to use templates of size smaller than five nodes for covering the graph. For each of the specifications considered, partitioning was carried out using both template generation approaches, and for each of these two template choice heuristics were applied: LFF (i.e. Largest Fit First) and MFF (i.e. Most Frequent fit First).

The results reported here include the percentage of the graph that the partitioner was able to cover with instances of identified templates, in terms of number of nodes covered over total number of nodes in the specification graph, the total number of template instances that were used for covering said part of the graph, and the computation time taken to carry out the partitioning.

A few observations can be made about these results: one could be surprised seeing how the free-shape template algorithm does not always reach better coverage percentages than the tree-based one, which intuitively should happen seeing as it identifies a greater variety of isomorphic

TABLE II

Graph Covering: Results with MFF, LFF heuristics

	Tree Templates						Free Shape Templates					
	LFF			MFF			LFF			MFF		
Benchmark	%	#Inst	Time	%	#Inst	Time	%	#Inst	Time	%	#Inst	Time
AES - `encryptblock`	85.2%	35	11.9 sec	57.5%	45	12.0 sec	74.3%	6	32.4 sec	32.7%	32	32.9 sec
AES - `decryptblock`	90.28%	42	16.8 sec	62.6%	58	15.53 sec	85.31%	12	61.5 sec	51.7%	52	63.9 sec
DES - `des_encrypt`	67.9%	11	3.7 sec	57.8%	32	3.6 sec	90.5%	4	8.3 sec	59.6%	31	8.3 sec
FDCT	35.3%	14	2.1 sec	35.3%	14	2.1 sec	76.7%	4	6.4 sec	53.8%	25	6.2 sec

subgraphs, thus giving more freedom to the graph covering phase. The reason for this is that the covering phase only looks for nonoverlapping graph coverings, i.e. solutions in which the clusters do not overlap with each other. As such, the choice of a template can greatly influence the alternatives that are available for choosing the subsequent ones. Since the considered heuristics for template choice only consider the templates themselves or their number of instances, with no consideration of the impact that each choice could have on the availability of the remaining templates, such results are not unlikely.

However, we can observe how the free-shape algorithm, especially when used with the LFF metric for template choice, produces covers composed of much fewer templates, which makes sense accdording to the data in Table I, which showed how the free-shape algorithm generally identifies templates of much bigger size.

In order to gain some further insight on the kind of data we are able to extract from the structures that were implemented for this thesis work, let us consider the `AES - encryptblock` example at a deeper level of detail, when processed by the free-shape template algorithm. Figure 6 shows how the identified templates are distributed among the different sizes, in terms

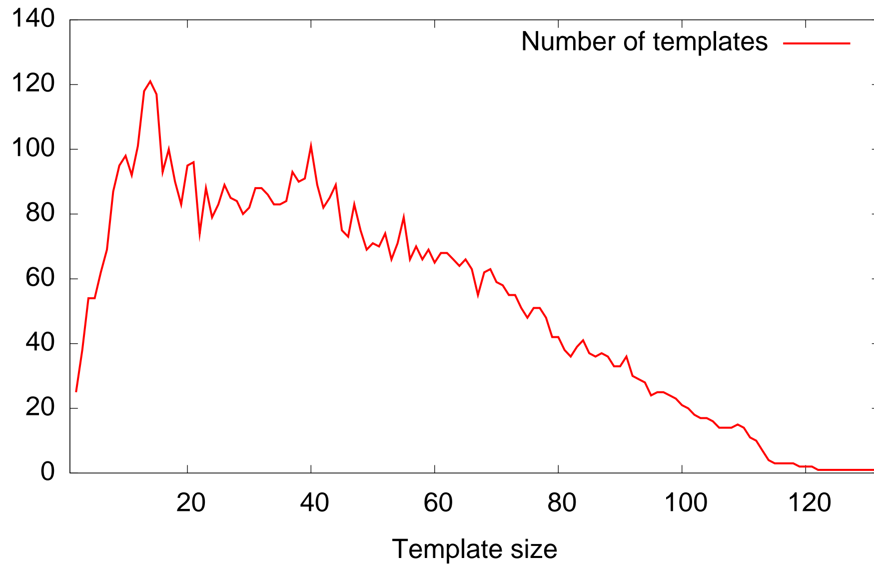

Figure 14. `AES - encryptblock`: template size vs. number of templates

of nodes enclosed in a single cluster. It is rather interesting to see how the number of identified templates rises to a maximum for medium sizes, and then decreases to just 1 for the biggest template. This behavior, it is worth mentioning, can be explained with a closer look at the code that specifies the input for this example: the biggest template accounts for a rather large block of `C` code repeated twice, which on another note is a good example of the kind of features we are trying to identify.

Another perspective is the one considered in figure 6, where we show the maximum number of instances for templates of a given size. Quite expectedly, templates of small sizes tend to

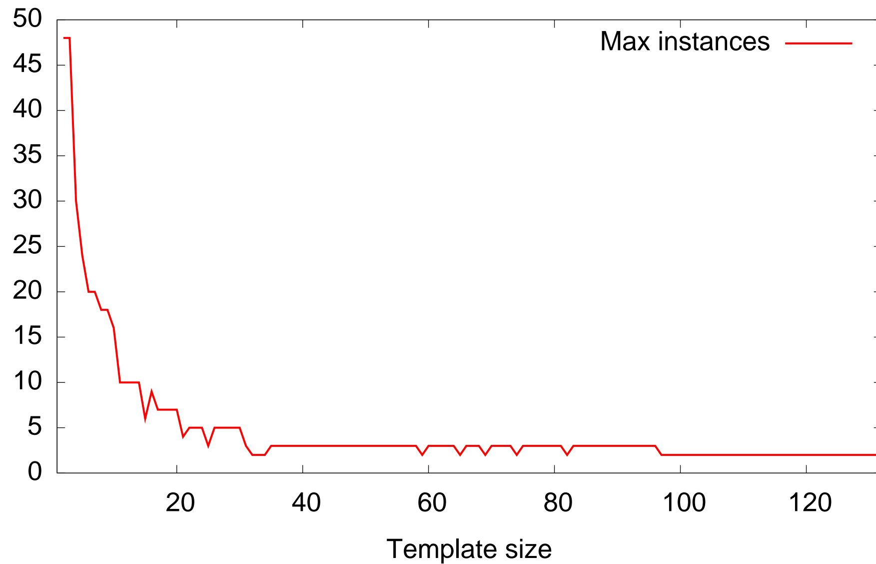

Figure 15. `AES - encryptblock`: template size vs. number of instances

have a larger number of instances, which decreases as the template size goes up. After a certain point, only two instances are found in the graph for every template.

To further analyze the behaviour of this approach on the same example, consider figure 6. In this graph, we show the ratio between the number of edges internal to the clusters and the number of edges crossing the cluster's border, for a single instance. The rationale behind extracting such data is on the one hand to estimate the communication burden that would be necessary to implement a given template as a reconfigurable module, and on the other hand to have an idea of how well the algorithm does at identifying clusters which are indeed "processing units" with tightly interdependent operations inside of them but loose dependencies with other

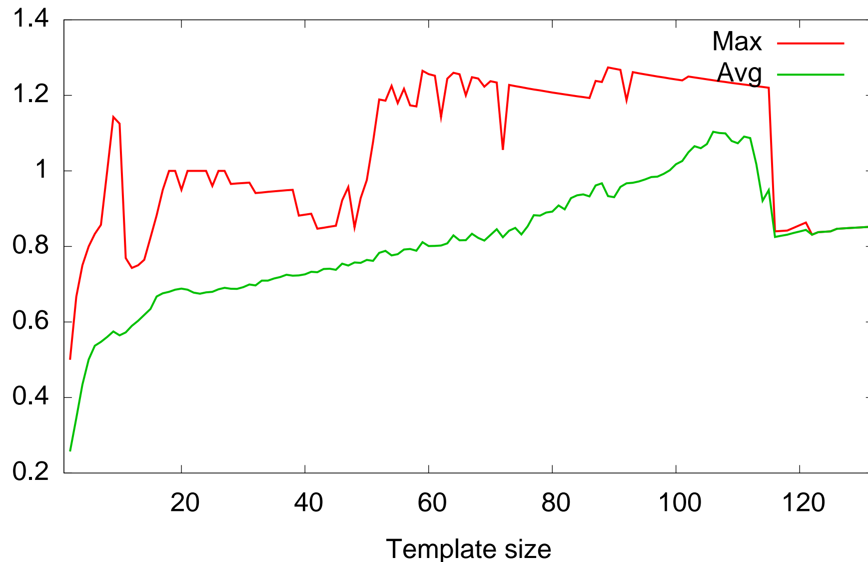

Figure 16. `AES - encryptblock`: template size vs. internal/boundary edge ratio

parts of the specification. For each template size, we show both the "best" template, i.e. the one with the highest ratio, and the average ratio computed among all the templates of that size. The two lines tend to collapse onto each other at the far right side of the graph, since there is just one template of maximum size.

The same internal/boundary edge ratio shown in figure 6 for the `AES - encryptblock` benchmark was adopted as a metric for template choice, in order to privilege templates with a tight communication pattern between operations included in their instances whilst being loosely tied to external nodes. The results for the same four benchmark specifications used before, for the free-shape template approach, are shown in Table III.

TABLE III

Graph Covering: using internal/boundary edge ratio as a metric for template choice

Benchmark	%	#Inst
AES - `encryptblock`	74.1%	22
AES - `decryptblock`	70.8%	20
DES - `des_encrypt`	87.8%	8
FDCT	73.3%	4

CHAPTER 7

CONCLUSIONS

7.1 Accomplished goals

A partitioning approach was defined and implemented which, starting from a system specification written in a high-level language, detects recurrent structures and produces a partitioned specification in which the identified clusters are instances of repeating templates in the original graph.

Two algorithms for regularity extraction were implemented, one of which was found in literature (12). The implementation work for this thesis was carried out using the data structures provided by the PandA framework, extending it appropriately. The algorithms thus had to be adapted and extended to work in our setting. Their performance was compared with regard to the size and number of regular structures identified by the two approaches, as well as the time taken to perform their task.

7.2 Future development

This thesis work is part of a larger effort, the final goal of which is a complete workflow for implementing a given system specification as a dynamically reconfigurable system. As was outlined in 3, this involves not only partitioning the specification into cores to be implemented as (re)configurable modules, but also a scheduling and placement step in order to decide when

each module should be configured on the device, and its position as well. This latter part of the approach we propose is the natural next step towards the achievement of our ultimate goal.

Another way of improving this work is the definition of more refined metrics for the choice of the templates used in the final partition: as of now, we consider the size of a given template or the number of its instances, i.e. we privilege templates that cover a bigger fraction of the original graph. Some ideas for more refined metrics have been suggested in 4.2.3, which again are based on the structure of the identified templates and on technological considerations such as the reconfiguration granularity offered by the device. Moreover, some kind of *feedback* mechanism from the subsequent scheduling and placement phase could be defined: the scheduler could e.g. request the partitioner to produce templates of a given size due to the area availability that it is facing.

CITED LITERATURE

1. Arikati, S. R. and Varadarajan, R.: A signature based approach to regularity extraction. In ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design, pages 542–545, Washington, DC, USA, 1997. IEEE Computer Society.

2. Bachl, S.: Isomorphic subgraphs. In Graph Drawing, ed. J. Kratochvíl, volume 1731 of Lecture Notes in Computer Science, pages 286–296. Springer, 1999.

3. Bachl, S. and Brandenburg, F.-J.: Computing and drawing isomorphic subgraphs. In Graph Drawing, eds, S. G. Kobourov and M. T. Goodrich, volume 2528 of Lecture Notes in Computer Science, pages 74–85. Springer, 2002.

4. Bellows, P. and Hutchings, B. L.: JHDL - an HDL for Reconfigurable Systems. In FCCM, pages 175–. IEEE Computer Society, 1998.

5. Berztiss, A. T.: Data Structures Theory and Practice. Academic Press, second edition, 1975.

6. Boost C++ Libraries: `http://www.boost.org`.

7. Cardoso, J. M. P. and Neto, H. C.: Macro-based hardware compilation of java bytecodes into a dynamic reconfigurable computing system. In Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, eds, K. L. Pocek and J. M. Arnold, page n/a, Napa, CA, 1999. IEEE.

8. Cardoso, J. M. P.: Loop dissevering: A technique for temporally partitioning loops in dynamically reconfigurable computing platforms. In IPDPS, page 181. IEEE Computer Society, 2003.

9. Cardoso, J. M. P.: On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures. IEEE Trans. Computers, 52(10):1362–1375, 2003.

10. Cardoso, J. M. P. and Neto, H. C.: Compilation for fpga-based reconfigurable hardware. IEEE Design & Test of Computers, 20(2):65–75, 2003.

11. Cardoso, J. M. and et al.: Fast hardware compilation of behaviors into an fpga-based dynamic reconfigurable computing system.

12. Chowdary, A., Kale, S., Saripella, P. K., Sehgal, N. K., and Gupta, R. K.: Extraction of functional regularity in datapath circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 18(9):1279–1296, 1999.

13. Donato, A., Ferrandi, F., Redaelli, M., Santambrogio, M. D., and Sciuto, D.: Caronte: A complete methodology for the implementation of partially dynamically self-reconfiguring systems on fpga platforms. In FCCM, pages 321–322. IEEE Computer Society, 2005.

14. Ferrante, J., Ottenstein, K. J., and Warren, J. D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9(3):319–349, 1987.

15. Fornaciari, W. and Piuri, V.: Virtual FPGAs: Some steps behind the physical barriers. In IPPS/SPDP Workshops, pages 7–12, 1998.

16. Free Software Foundation, Inc.: The GNU General Public License. `http://www.gnu.org/licenses/gpl.html`.

17. Ganesan, S. and Vemuri, R.: An integrated temporal partitioning and partial reconfiguration technique for design latency improvement, 2000.

18. Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., and Taylor, R.: PipeRench: A reconfigurable architecture and compiler. IEEE Computer, 33(4):70–77, April 2000.

19. Handa, M., Radhakrishnan, R., Mukherjee, M., and Vemuri, R.: A fast macro based compilation methodology for partially reconfigurable fpga designs, 2003.

20. Hauser, J. R. and Wawrzynek, J.: Garp: A MIPS processor with a reconfigurable coprocessor. In IEEE Symposium on FPGAs for Custom Computing Machines, eds, K. L. Pocek and J. Arnold, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

21. Horwitz, S., Prins, J., and Reps, T.: On the adequacy of program dependence graphs for representing programs. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 146–157. ACM Press, 1988.

22. Hudson, R. D., Lehn, D., Hess, J., Atwell, J., Moye, D., Shiring, K., and Athanas, P.: Spatio-temporal partitioning of computational structures onto configurable computing machines. In Configurable Computing: Technology and Applications, Proc. SPIE 3526, ed. J. Schewel, pages 62–71, Bellingham, WA, 1998. SPIE – The International Society for Optical Engineering.

23. Impulse Accelerated Technologies: CoDeveloper. `http://www.impulsec.com/`.

24. Johnson, T. A., Eigenmann, R., and Vijaykumar, T. N.: Min-cut program decomposition for thread-level speculation. In PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, pages 59–70, New York, NY, USA, 2004. ACM Press.

25. Kaul, M. and Vemuri, R.: Optimal temporal partitioning and synthesis for reconfigurable architectures, 1998.

26. Kaul, M. and Vemuri, R.: Temporal partitioning combined with design space exploration for latency minimization of run-time reconfigured designs, 1999.

27. Kaul, M., Vemuri, R., Govindarajan, S., and Ouaiss, I.: An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. In Design Automation Conference, pages 616–622, 1999.

28. Krinke, J.: Identifying similar code with program dependence graphs. In Proc. Eigth Working Conference on Reverse Engineering, pages 301–309, 2001.

29. Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M.: Dependence graphs and compiler optimizations. In Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 207–218. ACM Press, 1981.

30. Kutzschebauch, T.: Efficient logic optimization using regularity extraction. In ICCD, pages 487–493, 2000.

31. Liang, D. and Harrold, M. J.: Slicing objects using system dependence graphs. In Proceedings of the International Conference on Software Maintenance, page 358. IEEE Computer Society, 1998.

32. Ling, X. and Amano, H.: Performance evaluation of wasmii: a data driven computer on a virtual hardware. In PARLE, eds, A. Bode, M. Reeve, and G. Wolf, volume 694 of Lecture Notes in Computer Science, pages 610–621. Springer, 1993.

33. Lu, G., Singh, H., Lee, M.-H., Bagherzadeh, N., Kurdahi, F. J., and Filho, E. M. C.: The morphosys parallel reconfigurable system. In European Conference on Parallel Processing, pages 727–734, 1999.

34. Maestre, R., Fernández, M., Hermida, R., and Bagherzadeh, N.: A framework for scheduling and context allocation in reconfigurable computing. In ISSS, pages 134–140, 1999.

35. Ottenstein, K. J. and Ellcey, S. J.: Experience compiling fortran to program dependence graphs. Softw. Pract. Exper., 22(1):41–62, 1992.

36. Ouaiss, I., Govindarajan, S., Srinivasan, V., Kaul, M., and Vemuri, R.: An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures. In IPPS/SPDP Workshops, pages 31–36, 1998.

37. Pandey, A. and Vemuri, R.: Combined temporal partitioning and scheduling for reconfigurable architectures. In Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844, eds, J. Schewel, P. M. Athanas, S. A. Guccione, S. Ludwig, and J. T. McHenry, pages 93–103, Bellingham, WA, 1999. SPIE – The International Society for Optical Engineering.

38. Post, E.: Real programmers don't use Pascal. Datamation, 29(7), July 1983.

39. Purna, K. M. G. and Bhatia, D.: Temporal partitioning and scheduling data flow graphs for reconfigurable computers. IEEE Trans. Comput., 48(6):579–590, 1999.

40. Rao, D. and Kurdahi, F.: On clustering for maximal regularity extraction. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 12(8):1198–1208, 1993.

41. Redaelli, M.: Task partitioning for the scheduling on partially dynamically reconfigurable architectures. Master's thesis, University of Illinois at Chicago, 2005.

42. Santambrogio, M. D.: A methodology for dynamic reconfigurability in embedded system design. Master's thesis, Politecnico di Milano, 2004.

43. Siek, J. G., Lee, L.-Q., and Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, 2002.

44. Sinha, S., Harrold, M. J., and Rothermel, G.: System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In Proceedings of the 21st international conference on Software engineering, pages 432–441. IEEE Computer Society Press, 1999.

45. The Open SystemC Initiative: `http://www.systemc.org`.

46. Vasilko, M.: Dynasty: A temporal floorplanning based cad framework for dynamically reconfigurable logic systems. In FPL, eds, P. Lysaght, J. Irvine, and R. W. Hartenstein, volume 1673 of Lecture Notes in Computer Science, pages 124–133. Springer, 1999.

47. Vasilko, M. and Ait-Boudaoud, D.: Architectural synthesis techniques for dynamically reconfigurable logic. In FPL, eds, R. W. Hartenstein and M. Glesner, volume 1142 of Lecture Notes in Computer Science, pages 290–296. Springer, 1996.

48. Vasilko, M. and Benyon-Tinker, G.: Automatic temporal floorplanning with guaranteed solution feasibility. In FPL, eds, R. W. Hartenstein and H. Grünbacher, volume 1896 of Lecture Notes in Computer Science, pages 656–664. Springer, 2000.

49. Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., and Agarwal, A.: Baring it all to software: Raw machines. Computer, 30(9):86–93, 1997.

VITA

Personal data

- Full name: Matteo Giani
- Place and date of birth: Cuggiono (Mi), Italy. October 16th, 1981
- Nationality: Italian
- Address: via Cavour 61, 20020 Vanzaghello (Mi), Italy
- E-mail address(es): mgiani1@uic.edu matteo.giani@gmail.com

Undergraduate and graduate studies.

- Bachelor's degree: Politecnico di Milano, 2003, under the advising of Professor Luca Ferrarini.

Publications.

- Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P.: **Mobile Data Collection in Sensor Networks: The TinyLime Middleware**, *Elsevier Journal of Pervasive and Mobile Computing, Volume 1, Issue 4, pages 446-469, December 2005*
- Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P.: **TinyLime: Bridging Mobile and Sensor Networks through Middleware**, in *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications*

(PerCom 2005), Kauai Island (Hawaii), March 8-12, 2005, pp. 61-72, IEEE Computer Society Press. Selected among the conference's best papers and invited for publication in the Elsevier Journal of Pervasive and Mobile Computing.

- Giani, M., and Lazzarotto, M.: Design and Implementation of a software architecture for control of a model of a manufacturing plant on top of Linux/RTAI, *BS project*, original title: *"Progettazione e sviluppo di una architettura software per il controllo di un impianto didattico in ambiente Linux/RTAI"*, Advisor: Prof. Luca Ferrarini, Co-Advisor: Ing. Carlo Veber
- Curino, C., Giani, M., Giorgetta, M., Giusti, A., Trincavelli, M.: MIPS implementation of some very small databases data structures, *Technical report 2003.45, DEI, Politecnico di Milano, Italy*

Languages

- Written and spoken Italian.
- Written and spoken English.