TASK PARTITIONING FOR THE SCHEDULING ON PARTIALLY DYNAMICALLY RECONFIGURABLE ARCHITECTURES

BY

MASSIMO REDAELLI
Bachelor in Computer Engineering, Politecnico di Milano, 2005

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2005

Chicago, Illinois

TABLE OF CONTENTS

CHAPTER			PAGE
1	**INTRODUCTION**		1
2	**STATE OF THE ART**		7
	2.1	Temporal partitioning	7
	2.1.1	General approaches	7
	2.1.1.1	Sparc architecture	11
	2.1.1.2	Java^{TM} for Hardware	19
	2.1.2	Specific approaches	22
	2.2	Time and Space Partitioning	26
	2.2.1	An approach for fast context–switching architectures	26
	2.2.2	Sparc, again	29
3	**MODEL**		33
	3.1	Reconfigurable architectures	33
	3.1.1	Granularity	33
	3.1.2	Host coupling	34
	3.1.3	Reconfiguration Methodology	35
	3.1.3.1	Configuration memory	35
	3.1.3.2	Logic reconfigurability	36
	3.1.3.3	Configuration time	36
	3.1.4	Memory Organization	37
	3.2	The reference board	37
	3.3	The reconfigurable architecture model	40
	3.3.1	The chip	40
	3.3.2	The specification	45
	3.3.3	Technology binding and mapping	46
	3.3.3.0.1	Precedence	49
	3.3.4	Constraints	52
	3.3.4.0.2	Unknowns	52
	3.3.4.0.3	Partition	54
	3.3.4.0.4	Uniqueness	54
	3.3.4.0.5	Reconfiguration	54
	3.3.4.0.6	Duration	56
	3.3.4.0.7	Precedence	56
	3.3.4.0.8	Size	56
	3.3.4.0.9	Adjacency	57
	3.3.5	Aim	57

TABLE OF CONTENTS (Continued)

CHAPTER			PAGE
4	**PARTITIONING**		60
	4.1	Shape description	61
	4.1.1	The induced dependencies	61
	4.2	Candidate Generation	66
	4.2.1	Reuse	68
	4.2.2	Graph Isomorphism	73
	4.2.2.1	Problem definition	74
	4.2.2.2	Algorithms	76
	4.2.2.3	Specialization to our case	80
	4.3	Pruning	81
5	**ANALYSIS AND RESULTS**		84
	5.1	Choice of initial points	84
	5.2	Weighting function	86
	5.3	Matching algorithm	87
	5.4	Tests	88
6	**CONCLUSIONS**		89
	APPENDIX		92
	CITED LITERATURE		101

LIST OF FIGURES

FIGURE		PAGE
1	A program specification (**a**), the time partitioning schedule (**b**) and the optimal schedule (**c**).	3
2	Graphical view of temporal partitioning.	8
3	The data flow graph of a 2x2 matrix multiplier.	15
4	Two different schedulings of the data flow graph in Figure 3.	15
5	Loop restructuring.	18
6	The M1 implementation of the MorphoSys architecture.	23
7	Different possible scheduling of loops.	24
8	3-D Floorplanning — exploiting *partial* dynamic reconfiguration.	28
9	Alternated loading and executing of the partitioned specification.	30
10	The testing board used in MicroLab: Virtex–II Pro by Avnet	37
11	Structure of an FPGA	38
12	The structure of a logic block	39
13	A LUT	40
14	Example of specification	58
15	DAG specification becomes cyclic	59
16	Specification equivalent to that in Figure 17	62
17	Specification equivalent to that in Figure 16	62
18	The precedence is actually entering at the first node being executed	63

LIST OF FIGURES (Continued)

FIGURE **PAGE**

19 The precedence is actually leaving from the last node finishing execution 64

20 Counterexample to the easy solution of entering edge problem 65

21 A simple operation represented in a data flow graph 69

22 A different (?) simple operation represented in a data flow graph 70

23 A data flow graph representation of an operation *with numbered inputs* 71

24 Different operand bindings with the same operation 71

25 A single operation becomes operand for multiple operations 72

26 A specification G (colors represent the actions $\alpha(o)$) 94

27 The restriction to a tree of the graph in Figure 26 95

28 Not even a subgraph of the originally isomorphic subgraphs are isomorphic . . 96

29 The tree generated by a specification 99

LIST OF EXAMPLES

3.1 There is no “best” implementation 47

3.2 Suboptimality for general grouping of operations 50

3.3 From acyclic to cyclic specification 51

A.1 Isomorphism is not conserved under tree restriction 94

A.2 Not always the subgraphs of isomorphic subgraphs are isomorphic 95

LIST OF TABLES

TABLE **PAGE**

I EXAMPLE OF NON–EXISTENCE OF “BEST” IMPLEMENTATION . . 47

CHAPTER 1

INTRODUCTION

Reconfigurable hardware in general, and FPGAs in particular, have received much attention over the last years.

At first they have been employed as a cheap means of prototyping and testing hardware solutions without having to undergo the long and expensive process of ASIC design, thus allowing to drastically reduce the time-to-market. FPGAs have been so successful in this task that nowadays it is not uncommon to even directly *deploy* FPGA–based solutions.

In this scenario, that can be termed *compile–time reconfiguration*, the configuration of the FPGA is loaded at the end of the design phase, and it remains the same throughout the whole time the application is running. In order to change the configuration one has to stop the computation, reconfigure the chip resetting it, and then start the new application.

Compile–time reconfiguration was for some years the only kind of reconfiguration available for FPGAs. With the evolution of technology, though, it became possible to considerably reduce the time needed for the chip reconfiguration: this made it conceivable to reconfigure the FPGA *between* different stages of its computation, since the induced time overhead could be considered acceptable.
This process is called *run–time reconfiguration*, and the FPGA is said to be *Dynamically Reconfigurable*.

Run–time reconfiguration can be exploited by creating what has been termed "virtual hardware" [7, 8] in analogy with the concept of *virtual memory* in general computers. In the latter case the total

random access memory available is limited to a certain amount M, but the running applications $\{a_i\}_{i=1}^{n}$ require more than that — let us say $R > M$. What happens then is that the physical memory is *shared*: at a certain instant $t = t_0$ the memory is given to, say, $a_1, \ldots, a_m$ to use, while the other applications wait. After a certain time, say $t = t_1$, $a_1, \ldots, a_m$ are stopped, their intermediate results saved in a bigger memory (on the disk), and the physical memory is allocated to $a_{m+1}, \ldots, a_n$.

The same thing can happen with hardware. Consider an application that is too big to fit into a particular FPGA: one can *partition* it into n smaller tasks, each one fitting on the chip. Then it is possible to load task 1 on the chip, execute it, then reconfigure the FPGA for task 2 and execute it, and so on till task n is finished.

This idea is called *time partitioning*, and has been studied extensively in literature (see Chapter §2).

A further improvement in FPGA technology allows modern boards to reconfigure only *some* of the logic gates, leaving the other ones unchanged.

This *partial reconfiguration* is of course much faster in case only a small part of the FPGA logic needs to be changed.

When both these features are available, the FPGA is called *partially dynamically reconfigurable*.

Not much literature is devoted to the full exploitation of the features available in this kind of FPGAs, even if there are clear advantages over basic Dynamic Reconfiguration.

Consider for example a simple FPGA with 3 reconfigurable units and an application represented by the control flow graph shown in Figure 1a (where a node with label n/m has a delay of n cycles and needs m reconfigurable units). The time partitioning approach will at best identify the scheduling in Figure 1b, since the chip must be reconfigured completely each time. (The size of each task is

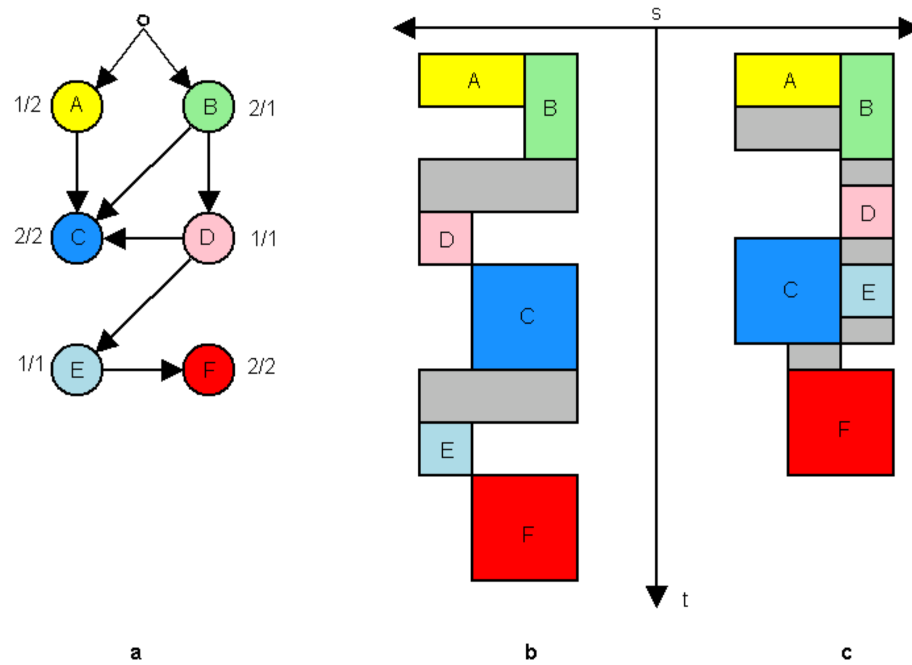

Figure 1. A program specification (**a**), the time partitioning schedule (**b**) and the optimal schedule (**c**).

represented horizontally, while time is shown increasing downwards. Moreover, gray areas represent reconfiguration time).

If instead one allows the selective reconfiguration of certain areas of the FPGA corresponding to a task (or to groups of tasks), we can obtain the better schedule shown in Figure 1c, which gives a much smaller total latency.

The idea is to hide the latency due to the reprogramming of the logic gates by reconfiguring some parts of the chip while others are performing useful computations. As shown in this very simple example, not only does one gain parallelism among reconfiguration and execution, but it is also possible to expose more parallelism among the tasks themselves (tasks C and E).

This allows a good exploitation of the power of partial dynamic reconfiguration.

FPGA OSs?

The problem of scheduling on FPGAs can be addressed from different points of view. Most of the early research was focused on *static scheduling*: given a completely specified specification, find a scheduling at compile time.

However the problem can be addressed also from a dynamical point of view, *i.e.* from the point of view of an operating system who is in charge of the physical allocation of the tasks on the resources they require. The tasks are *not* pre–determined, and must be dealt with in an online fashion.

The former problem allows the designer to employ better algorithms looking for optimal or near–optimal solutions since more time can be spent in solving the problem itself, but at the same time the final result cannot adapt to user inputs or other unforeseen factors.
On the other hand the latter problem guarantees more flexibility, but also requires very fast responses and has a much more restricted view of the overall scheduling problem — and thus has to be solved far less optimally.

In this work the on line management of task scheduling will not be addressed. This is due mainly to two reasons.

First of all, the time needed to reconfigure an FPGA is still quite high, and the amount of logic that fits on an FPGA is still somehow low, and in view of these technological limitations it seems a bit too early to devote much work on the study of an operating system for this kind of devices.

Secondly, being FPGAs mostly seen as a means of speeding up the execution of tasks running too slowly in software, it seems more reasonable to first solve in a satisfying way the actual static scheduling optimization problem.

Outline of the present work

This thesis work will focus on the scheduling of tasks on partially dynamically reconfigurable FPGAs in order to minimize the overall latency of the application.

In Chapter 2 the existing literature on the topic will be reviewed.

No reference to a particular FPGA will be made: in this work a simplified, but realistic, model of reconfigurable hardware will be assumed and presented in Chapter 3 in order to reduce the complexity of the analysis and to develop a consistent and general framework.

Based on this model, a mathematical description of the problem will be proposed.

In Chapter 4 the model built in Chapter 3 will be used to analyze the problem of partitioning the single operations in the specification.

Using this approach, the actual N tasks of the application (where N is “large”) will be grouped in n bigger partitions, with $n \ll N$. In this way it might be possible to solve the smaller problem of scheduling of the partitions near–optimally in a reasonable time — possibly using an algorithm otherwise too complicated and time–consuming if applied to of the original (larger) specification.

In Chapter 5 the proposed algorithm for the task partitioning will be analyzed. In particular, the sub–algorithms it uses to compute sub–steps of the whole process will be compared with those available in the literature.

Finally, the limitations of the proposed approach, as well as possible further improvements and developments will be addressed and suggested (Chapter 6).

CHAPTER 2

STATE OF THE ART

The first attempts at efficient scheduling for FPGAs were studied when partial reconfiguration was still not available or was being developed by board producers. For this reasons the early attempts that will be encountered always refer to *total reconfiguration*.

2.1 Temporal partitioning

There are at least two ways to look at the problem of scheduling in dynamically reconfigurable hardware.

One can look at it from a general point of view, making some reasonable assumptions about the hardware architecture but without binding oneself too much to a particular implementation — thus building a framework that can be applied with minor modifications on a wide range of platforms, but that will also probably not fully exploit the capability of the hardware.

On the other hand, one can choose to target a very particular architecture and tailor one's efforts to exploit all of its capabilities. Doing so, however, one must give up an easy portability of one's framework onto other platforms.

2.1.1 General approaches

One of the first general approaches is the one proposed in [9].

In this paper the authors address the problem of fitting on an FPGA a program that requires more logic than is actually available on the chip.

The solution is *temporal partitioning*: splitting the program into a set of subprograms to be executed serially, each of which fits on the FPGA. In this fashion one obtains, given an input specification S, an ordered list of configurations $(c_1, c_2, \ldots, c_n)$ which downloaded and executed onto the FPGA one after the other implement the original functionality of S (see Figure 2).

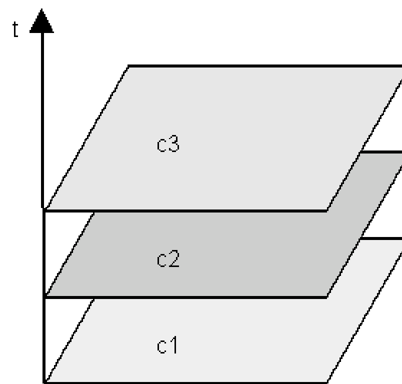

Figure 2. Graphical view of temporal partitioning.

The authors suggest two algorithms for temporal partitioning, both taking as an input the (acyclic) data flow graph of the program. These algorithms are very general and can be employed on virtually any reconfigurable hardware supporting total dynamic reconfiguration.

First of all, the so–called ASAP–*level* is assigned to each node of the data flow graph. The idea is that in order to ensure the correct execution of the program one must be sure that any node is run only after all the other nodes it depends on have terminated their execution.

The authors then assign a "level" to every node such that a node with level l can be executed only if all the nodes of levels $1, 2, \dots, l-1$ have terminated. In order to do so it is enough to traverse the data flow graph and assign to each node v a level $\lambda(v)$ equal to

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

where the set $\mathrm{FanIn}(v)$ is the set of all the nodes from which arcs entering in v depart.

Having done this, the first algorithm is intended to minimize the overall latency of the program. It simply creates a list of all the nodes, ordered by level. It then creates the first partition, p_1, and traverses the list assigning all the nodes it can in c_1, stopping when no more fit.[1] It then creates a new partition p_2 and resumes assigning the nodes of the graph, and so on till all the nodes have been assigned.
In this fashion it is obvious that the dependencies are satisfied.

The second algorithm is instead intended to minimize the overhead due to communication. In the simplified model this amounts to finding a partitioning of the data flow graph that minimizes the edges that are "cut" by the dividing lines identifying the partitions.
The idea here is to try and put nodes with common parent in the same partition. In order to obtain this behavior, every time a node is added to a partition it is also checked if its children nodes can be put in the same partition too (this is verified through a list of ready nodes).

---

[1]The algorithm also takes into account a rough estimate of the area overhead due to communication each time it adds a new node to a partition.

If the first algorithm was based on a breadth–first search, this modified approach amounts to adding a limited depth–first search every time a node is added.

The experimental results show that the approach works, but let us analyze the formula the authors give for the reconfiguration time:

$$T_{\text{exec}} = (k \cdot T_{\text{reconf}}) + T_{\text{hardexec}},$$

where k is the number of temporal partitions, T_{reconf} is the time it takes to reconfigure the FPGA, and T_{hardexec} is the time it takes to execute all the temporal partitions. In the case of the test architecture the authors used, $T_{\text{reconf}} \approx 242\text{ms}$, while the execution of a temporal partition takes $T_{\text{partexec}} \approx 250\text{ns}$.

As the authors themselves point out, there is a strong unbalance between reconfiguration and execution times: in the time required for one reconfiguration, the hardware partition can be executed $\frac{242\text{ms}}{250\text{ns}} \approx 10^6$ times.

This is why the authors suggest as an effective example a JPEG encoder, where the first partition (the DCT transform) can be repeatedly applied on all the input images with no need to reconfigure, thus drastically speeding up the overall computation.

This paper tackled the problem of scheduling on reconfigurable hardware in possibly the most obvious and straightforward way, thus achieving a simple and general algorithm. It is, however, far from being optimal — both because of the very simplified model of computation platform and because of the “ingenuity” of the search algorithm, which is a simple variation of the classical list–based scheduling.

It is worth to point out that, as we will see, a proper management of loops (which is here suggested) will be a recurring theme in trying to hide reconfiguration overhead.

A similar approach is taken in [10], where the authors give a bird–eye view of a possible implementation of a time–space partitioning algorithm for reconfigurable computers.

They take as input a specification written in GDL (Graph Description Language) and a description of the target architecture (which can be composed of n multiple interconnected chips), and:

1. partition it *temporally* so as to obtain time partitions $\{p_i\}_i$ that fit on the entire target machine;
2. partition each p_i *spatially* into n sub–partitions $\{p_{ij}\}_{j=1}^{n}, \forall i$ in order to place each of them on one of the n available chips.

The aim of the paper seems to be focused mainly on the proposal of GDL as a description language and on the possibility of automated time partitioning of a program on a reconfigurable computer — performance on the other hand appears to be neglected. There is in fact no data available to evaluate the proposed approach.

2.1.1.1 Sparc architecture

The same idea of temporal partitioning has been studied in a series of papers by R. Vemuri et al. [11, 12, 13, 14, 15, 16, 17].

In [14] the authors propose an ILP model that describes the problem of temporal partitioning, taking into account not only the dependency constraints and the available resources on the FPGA, but also allowing binding of multiple operations on the same functional unit and the constraint imposed by the amount of memory available for storing the values that must be passed from a partition to the next one.

The resulting model is then tightened with the introduction of additional constraints and then solved to find the optimum solution.

The input of the model is a task graph, where every task is composed of some operations.

This approach is shown to work in a reasonable time for some small examples (10 tasks and 70 operations, more or less).

It has a few drawbacks, though. First of all, the granularity being set at the task level, the solution is forced to place all the operations of a certain task in the same temporal partition, thus failing to explore a large part of the solution space.
Secondly, many parameters of the model are to be chosen by the user, and there is no guideline in doing so. For instance, one has to choose the maximum number of partitions and the maximum number of functional units per each type. These are all critical choices, since choosing too small a number will result in no feasible solutions to be found, while increasing any of the quantities will increase the number of the variables in the model, making it much more complicated to solve.

In [11] the authors extend the work in [14] by building a genetic algorithm that spatially partitions each of the time partitions obtained with the ILP model to fit on a reconfigurable architecture that is composed of several interconnected FPGAs. This new algorithm provides a better accuracy in the estimation of the resources being used, which was only rough in the previous work, and hence guarantees an actual fitting of the program on the board.

The problem of the estimation of the parameters to be employed in building the ILP model is tackled in [12]. Here the model itself is revised: the operations in the tasks are no more fed to the solver to be scheduled (thus simplifying the model) — it is instead assumed that each task can be implemented

in a finite number of ways $m_1, m_2, \ldots, m_n$ (called *models*), and the ILP solver simply chooses which of these models (characterized in terms of occupation and latency by a high–level synthesis tool) to actually use.

Then the authors suggest a heuristic to estimate reasonable bounds for the parameters of the model, and use a design–space exploration algorithm in order to obtain an near–optimal solution.
The approach is basically an iterative algorithm that:

1. finds a feasible solution of the ILP model with the estimated parameters N (lower bound on the number of partitions) and D_m, D_M (minimum and maximum latency);
2. replace the upper bound of the latency with $\frac{D_m+D_M}{2}$ and solves the model again, iterating till the best solution with the current N is found;
3. increases the current value of N and go to 2., until the upper bound for N is reached.

The final output of the method is the scheduling with the overall best latency.

This approach has the advantage of allowing the exploration of different implementations for every task — it may happen, however, that too many possible implementations for each task are available, thus increasing dramatically the number of variables in the model and making it much harder to solve. The authors cope with this difficulty by selecting only some of the possible implementations, but the selection might be sub–optimal.

Also, the run–time of the overall approach is increased with respect to the former papers, since now the solving of the ILP model is performed several times. For instance, the DCT example consists of a task graph made of 32 tasks, all of which belong to two categories. For each of the two kind of tasks three

possible implementations have been considered (which seems a pretty conservative selection). The time required for the total execution of the algorithm on this example took (with different settings) from 15 minutes to almost an hour.

A complete control over the scheduling of the operations inside the tasks (allowing them to be put also in different partitions) is finally considered in [13]. Here the authors stress the implicit trade–off between the latency of each single partition and the number of partitions. Which, in turn, since the time for reconfiguration is supposed to be much larger than that of execution, signifies a trade–off between the latency of each partition and the overall latency of the program.

The decrease in the number of temporal partitions is achieved by fitting more operations of the specification onto the same partition, and this is possible through *sharing*.

The authors give an example in which, unexpectedly, the sharing does *not* reduce the number of partitions. However, it does happen that through sharing the overall latency decreases. Consider the specification (given as a data flow graph) in Figure 3, and assume that the board has 400 CLBs and that the area occupation of a 16–bit–multiplier (*16) is 256 CLBs, of an 32–bit–adder (+32) is 32 CLBs, and of a n–input–multiplexer $(\mathrm{Mux}(n))$ is 8 times the number of its inputs then propose two different scheduling. Then at least the two scheduling in Figure 4 are possible.

In the first case (on the left) in the first two temporal partitions (the partitions are divided by a thicker horizontal line) there is only one multiplier shared by three operations. In the second arrangement, instead, the multiplier is shared by four operations.

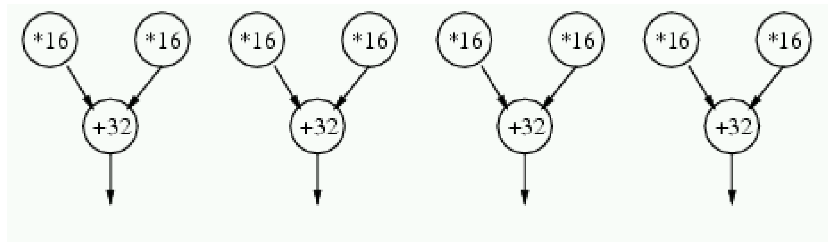

Figure 3. The data flow graph of a 2x2 matrix multiplier.

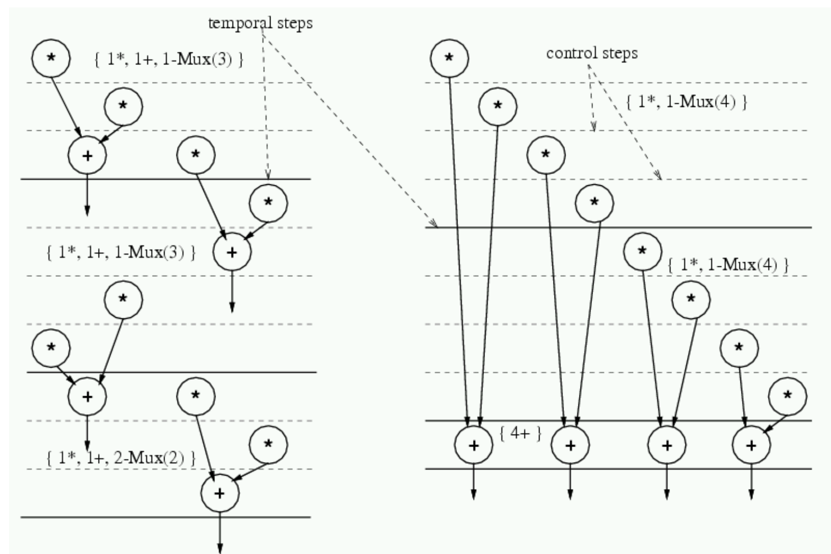

Figure 4. Two different schedulings of the data flow graph in Figure 3.

It can be noted that the increase in sharing of functional resources made it possible to “compact” more of the specification in earlier partitions, enabling the last one to perform in a massive parallel fashion the five final sums — thus gaining one control step in latency.

With this trade-off in mind, the authors propose a partitioning algorithm that encapsulates an enhanced resource–constrained force–directed list scheduling. As a pre–processing step, a list is created containing all the types of execution resources that are needed by the specification, sorted according to their “importance” (measured by the frequency with which they are used and by how “far in the future” they are needed).

Then the algorithm creates a minimal resource set (containing one resource of each type) and schedules on it as many operations as possible using the force–directed list scheduler. It then tries to “perturb” the solution modifying the resource set on which the algorithm schedules the operations — this perturbation can be either removing a low–priority resource from the resource set (thus moving some operations to the next temporal partition but at the same time freeing area for multiplexers and hence resource sharing), or increasing the number of resources available in the resource set (thus increasing the level of parallelism and hence reducing the latency).
Whether or not the perturbation is accepted depends on the estimation of the cost of the solution based on the total latency of the obtained implementation of the specification.

The analysis of this approach is difficult since the algorithm is not fully specified in the paper. However, if dealing with all the single operations (instead of with coarser tasks) is certainly advisable and is bound to give better solutions, the choice of a force–directed list–based scheduler is known to be non–optimal.

Also, the benchmarks offered in the paper appear not to be very helpful, since there is no data about the size of the specification of the two examples and there is no comparison with the results of other works.

We can also note how, again, the reconfiguration time is *orders of magnitude* higher than the execution time (in the FFT example, 66 seconds against 82μs). It is apparent here that a gain of one clock cycle in latency is absurdly small if compared to the reconfiguration overhead.

The authors noticed this problem, and addressed it in a further paper [15], where they study a method to make the reconfiguration overhead acceptable.

They observe that many data–intensive applications are structured as a certain number of iterations (say n) of a loop composed of some tasks. Usually these tasks don't fit on the FPGA, therefore it is necessary to split the specification in m time partitions. In this way, every time a loop is executed it will be necessary to perform m reconfigurations, and hence the overall program will require $m \cdot n$ reconfigurations.

The suggested approach then is to restructure the code in such a way that the need for reconfiguration is reduced as much as possible. One obvious way to achieve this is executing the tasks on each partition as many times as it is possible, under the assumption that the data dependencies allow this. For example, the JPEG compression algorithm: it is composed of fours tasks (DCT, quantization, zig–zag and Huffman encoding) that are all executed independently on all of the image (or on the output of a previous step).

In other words, consider the example in Figure 5. Assuming the three tasks in Figure 5a can be executed independently on all their input data (as soon as it is ready), and that the three tasks are all encoded in a different temporal partition, one can decide to execute as many times as possible (ideally n

times) each partition, then reconfigure to the next one, and so on. In this manner only m reconfigurations are necessary.

Iterating n times is not always possible or desirable, since each iteration of a temporal partition generates a result that has to be stored as input for the next partition, and the board might run out of memory. The authors then suggest two approaches: *Final Data to Host* (FDH, Figure 5b), in which each partition is executed only as many times as possible without running out of internal memory (if this number is k, it will be necessary to iterate the whole design $I_{sw} := \lfloor \frac{n}{k} \rfloor$ times, and hence $m \cdot \lfloor \frac{n}{k} \rfloor$ reconfigurations will be needed), and *Intermediate Data to Host* (IDH, Figure 5c), in which n iterations of all the partitions are always performed, and the temporary data that does not fit on the board are saved on the external RAM — even though it has lower performance.

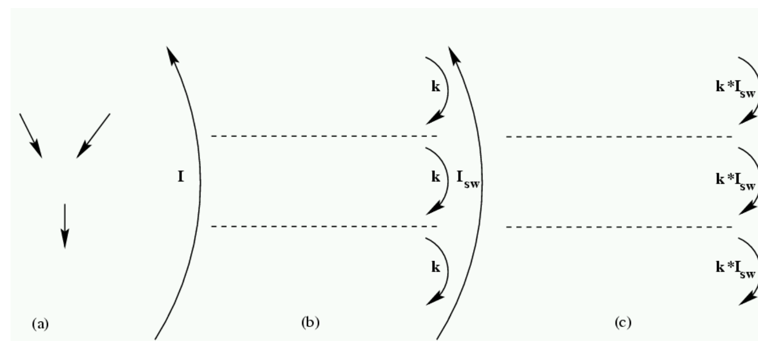

Figure 5. Loop restructuring.

The experimental results are based on two different hardware–software codesign implementations of the JPEG compression algorithm, in which only the DCT transform has been put in hardware. The reconfiguration time of the board was of 100ms.

The first implementation was a static one, that through sharing of resources fit all the DCT in one partition, thus avoiding reconfiguration overhead, and had a latency of 16μs.

The second implementation was a dynamic one, with three time partitions, but that exploiting parallelism only needed 8.44μs of actual computation.

The authors show that using the IDH policy on a "big enough" image (with approximately 10^5 DCT operations) the time saved in the execution hid the time lost in reconfiguration with respect to the static implementation — and actually gave speedups up to 50%.

The main questions about the method described in this paper rest on the fact that the loop restructuring appears to be done *after* the time partitioning (which is obtained through the usual ILP model), and that thus it is probably not optimal program–wise, and on the lack of an automated procedure to restructure the code — it appears that the problem has to be solved manually, which can be quite difficult and time–consuming if the task is less than trivial or the specification is not straightforward.

2.1.1.2 Java™ for Hardware

In two papers [18, 19] Cardoso and Neto propose and refine an approach to compilation of a specification, given in terms of Java™ bytecode, to a reconfigurable hardware architecture.

They perform a very detailed preprocessing step, also described in their previous work [20], which uses several data and control flow representations to extract all the parallelism present in the specification.

Using as input the results of this analysis, the authors first propose an algorithm to identify which macro–cell (*i.e.* which mapping to the logic level) is the best choice for every node of the data flow graph (assuming the required functionality can be implemented in different ways). Then they compute a priority list using, as in [9], topological levels and ASAP and ALAP. They do it twice, though: once "forward", taking level 0 and $\text{ASAP}_{\text{start}} = 0$ for the operations with no precedences, thus getting the maximum number of levels (L_M) and the maximum time (T_M) required; and once "backwards", setting the level of the nodes with no nodes depending on them to L_M, and their ALAP_{end} to T_M.

Finally, the nodes are sorted according to increasing level, and increasing mobility ($\text{ASAP} - \text{ALAP}$) is used to break ties.

Having defined the priority, the rest of the algorithm is a standard list–based scheduling.

In case the overall design does not fit on the board, the authors provide two strategies for temporal partitioning. The first one is a simple modification of [9], where the choice of the next node is defined, as explained before, by its level *and* subsequently by its mobility.
The second one tries to fit more operations in every partition by not stopping when the next node in the priority list doesn't fit, but trying to allocate other nodes with same level but different mobility.

The experimental results in both papers appear to fail in addressing the main issue of reconfigurable computing, that is, the reconfiguration overhead: only one example ends up being composed of more than one partition, and there is little discussion of the reconfiguration time.

The next paper, [21], still employs a standard list–scheduler but with a new function to compute the priority list, which gives good results but still doesn't address reconfiguration overhead.

The perspective changes in [22, 23], where the author strongly puts an emphasis on how resource sharing can reduce the number of time partitions, and hence greatly speed up the overall computation. The approach is based on a heuristic algorithm that creates a time partition for every node in a critical path of the data flow graph of the specification. It then iterates through the time partitions trying to fit in them the rest of the nodes, respecting the data and control dependencies and the size constraint of the board, opting for sharing of resources when possible and creating new partitions where necessary. When this phase is finished, the algorithm adopts some strategies to try and merge neighboring time partitions.

The experimental results show that sharing is a good means of reducing the number of time partitions (for instance a 4x4 Matrix multiplier with no sharing required 64 partitions, while only one sufficed with sharing).
However, the scheduling algorithm is strongly heuristic and gives little guarantees of near–optimality, also because of its “local” nature in the exploration of the design space.

In a more recent paper [24], the author investigates the scheduling of loops in reconfigurable architectures. He presents a new technique to split a loop into different time partitions, called *loop dissevering*.

Loop dissevering can be applied to loops of any kind, with any type of data dependencies, while the more traditional techniques of loop unrolling and loop distribution (see [15] and Figure 5) are not so general. Hence, in a setting where loop distribution cannot be employed loop dissevering can be an appropriate solution.

One problem arises, though: loop dissevering results in a larger number of partitions than loop distribution would, and also the number of reconfigurations needed to execute the original specification strongly grows. This heavily affects the performance of the overall system.

It would then seem that loop dissevering is useful only in emergency situations, when the body of a loop which cannot be decomposed with loop distribution is too large to fit on the FPGA.

2.1.2 Specific approaches

In [25, 26, 27] the authors tackle the problem of dynamic reconfiguration focusing their attention on the MorphoSys architecture, and specifically on its M1 implementation.

The structure of M1 is shown in 2.1.2. Its core is an 8x8 array of reconfigurable cells: each of these has a structure similar to the data path of a microprocessor, with registers, an ALU, multiplexers, *etc.* The control information for each cell is provided by a *context*, which is stored in the *context register* of the cell itself. The content of the context register is loaded from the (two port) *context memory*.
A RISC control processor is also present, and has the function of controlling the system operations (context changes, ...).

The architecture of the memory (with a two-set frame buffer and a DMA controller) is implemented in such a way that there are certain restrictions about who can access the memory, and when. In particular, it is possible to have the execution overlapping with data transfers, assuming the latter are from the external RAM to a set of the frame buffer that the computation is not using. Also, execution can sometimes overlap with context loading. However, context loading and data transfer cannot overlap.

As one can see, the limitations and peculiarities of the specific platform are completely taken into account in the authors' approach.

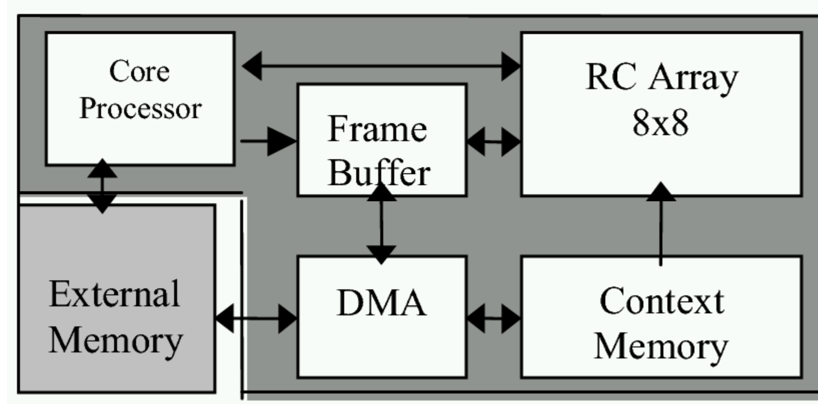

Figure 6. The M1 implementation of the MorphoSys architecture.

They then proceed with some further assumptions.

First of all it is assumed that the programs to be scheduled are written in terms of *kernels*, that is, small subprograms gathered in library, which contains not only the C implementation of the function considered, but also its mapping onto the reconfigurable cells. This library has to be built manually.

Furthermore, the authors assume that a generic program can be written as a loop on a certain number of iterations of a sequence of kernels (think for instance of an algorithm like JPEG compression).

Finally, it is assumed that only one kernel can be executed at a time, since it is supposed to be reasonable that all the kernels are in fact large enough to require all of the reconfigurable array to be executed.

In this setting it is clear that what matters is an optimal management of the loops in the program, and in fact the authors devote the most of the paper to this topic.

The first aim of the paper is to re–schedule the program not as a loop of a long sequence of kernels, but in a more efficient way (see Figure 7). With observations very similar to those made in [15] in §2.1.1.1, the authors point out that if m kernels have to be executed n times, one has to perform $m \cdot n$ reconfigurations (Figure 7a). If, however, it is possible to perform all the iterations of the first kernel independently of the rest of the kernels, and so on for all the rest of them, one can limit the number of reconfigurations to m, as in Figure 7b.

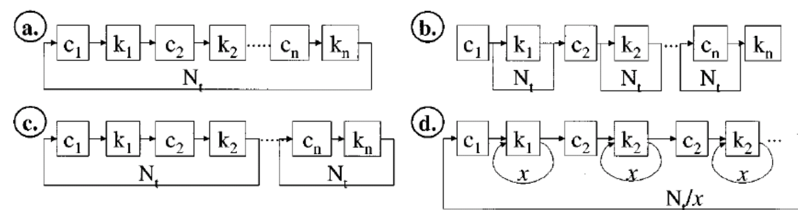

Figure 7. Different possible scheduling of loops.

In the general case this will not be possible (also because the internal memory to store the results of the execution of one kernel might not be big enough), but what can be usually achieved is to group some kernels in partitions that can be executed independently of the rest of the program (Figure 7c–d).

The authors provide in [26] a backtracking algorithm that explores all the possible partitions and evaluate their optimality using a detailed mathematical model representing the hardware architecture (in this process the data transfers and context loadings are supposed to be performed optimally).

Having obtained the partitions, which in this context also means having obtained the scheduling of the kernels, the next step is to schedule the data transfers and the context loading.

The authors solve this problem with three objectives in mind:

1. maximizing the overlapping of context loading and data transfer with computation;
2. minimizing the fragmentation of the memory;
3. obtaining a *periodic schedule* both in time and space, which means that
    - the code needed to manage the context and the data transfers is the same for every iteration (temporal);
    - the data required for the next iteration of a kernel is placed in the same position as the data for the last iteration was (spatial).

Objectives 2 and 3 are aimed at reducing the complexity of the control code, while the first one is intended to hide the reconfiguration times.

To accomplish these goals the authors extend the mathematical model they propose, formulating the problem of context scheduling in an exact form. The optimal solution found with this approach is then used as a validation for a fast heuristic they suggest [25, 26].

The problem of data scheduling is, instead, addressed in two later papers, in which two algorithms are developed that reduce the run–time through the minimization of the amount of data transfer — this is done via a careful exploitation of data reuse between iterations of the kernels [28, 29].

These papers do a very detailed analysis of the scheduling problem, obtaining also a fine–tuned mathematical model describing it. However, in order to do so the authors have to restrict the scope of

their work with a series of assumptions that are sometimes justified, but that limit its application and innovation.

First of all, as already pointed out, they focus on a very specific architecture. It is to be expected that their work needs some non–negligible tuning to be ported not only to a different architecture, but also to the next implementation of the MorphoSys architecture.

Secondly, it is assumed that all the programs can be represented as mainly composed of loops of kernels. This assumption is often true for data–intensive applications, but may fail to describe efficiently a control–intensive application.

Moreover, basing their approach on kernels not only the authors have to provide these kernels manually, but also they limit themselves to a coarse resolution in the task description and optimization.

Finally, the assumption that only one kernel is running at each instant (together with the further assumption that a kernel spans all the reconfigurable cells) implies that also this approach is in fact a simple time partitioning, and the reconfiguration, although dynamical, is not partial — while the architecture would be ready to support it.

2.2 Time and Space Partitioning

Oddly enough, there are only few papers addressing the exploitation of partial dynamic reconfiguration for scheduling purposes.

2.2.1 An approach for fast context–switching architectures

In [30, 31, 32, 33] Vasilko et al. propose their approach to dynamic reconfiguration.

In particular, in [30, 31] the authors formulate a (very general) mathematical model of the problem of scheduling on FPGAs and develop a very straightforward list–based algorithm to solve it, particularly targeted at the case in which the reconfiguration time is less than, or equal to, a clock cycle.

Basically, it sorts all the tasks in a priority list (based on the ALAP and ASAP labels), and then feeds this list to a standard list sub–scheduler with constrained resources (the area available on the FPGA) starting from control step $C = 1$.
When there is no more area available on the device, the redundant functional units from previous control steps are removed from the schedule and the next iteration begins. Through partial reconfiguration it is possible to execute one control step over two subsequent reconfigurations.

It appears to be difficult to fully appreciate the work described in these papers, since the mathematical model seems to have some obscure points, and the assumption that reconfiguration is so fast (especially in the years when the paper was written) seems a bit too optimistic.

The authors themselves are aware of the problem of reconfiguration overhead:

> Our experiments show, that for practical applications, the reconfiguration time has to be comparable to the system's clock period $(\tau \approx 1)$. Higher T_c reduces performances rapidly.[1]

Also, it is not clear how the authors intend to actually perform the partial reconfiguration and how to handle the problem of fragmentation of the available area, since all the algorithm seems to be concerned with is simply the unidimensional indication of "area".

---

[1] [31, pg. 294].

A broader point of view was then proposed in [33], where the authors describe a tool that allows the designer to explore *graphically* (but manually) a large part of the design space.

The basic idea here is to enlarge the actual search space allowing not only for time partitioning (which is a feature possible thanks to dynamic reconfiguration), but to temporal *and* spatial partitioning (possible thanks to the partial reconfigurability), and that the authors call *3–D floorplanning* (see Figure 8).

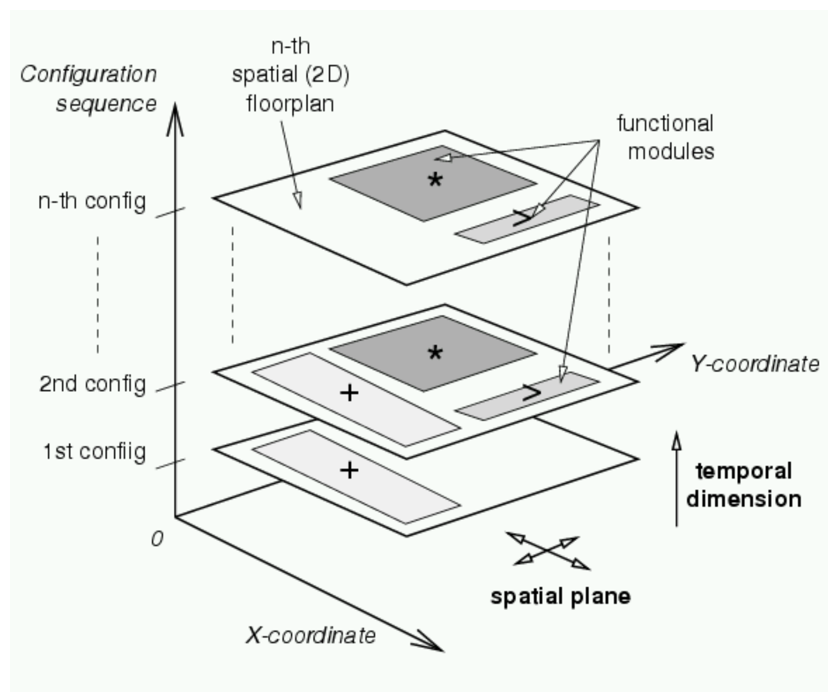

Figure 8. 3-D Floorplanning — exploiting *partial* dynamic reconfiguration.

This tool, called DYNASTY, is a framework that permits the manipulation of a design enabling the user to analyze it in many different (but integrated) views, such as *control data flow graph viewer*, *design hierarchy and structure browser*, *3–D floorplanner*, . . . , and to modify it directly from the graphical user interface.

The tool is undoubtedly interesting and useful for the designer. What it lacks is an automated tool that tries to optimize the design, instead of leaving the burden on the designer.

This necessity was addressed in a subsequent paper [32], where a genetic algorithm was contrived to try and minimize the total latency of the design, addressing several optimization problems at once:

- *module allocation* of every node of the control data flow graph of the specification to an execution unit in the library available in DYNASTY;
- *scheduling* of the nodes;
- *3–D floorplanning* of the modules, which can be seen as a modified (3–D) packing problem;
- *cost evaluation* of the obtained designs.

The algorithm was tested on three small design benchmarks, for which it gave solutions that are found to be sub–optimal with respect to hand–optimized ones of about 24%.

2.2.2 Sparc, again

The series of papers from Vemuri et al. that was considered in §2.1.1.1 never exploited the capability of modern boards to be *partially* reconfigured. In a subsequent paper [16] the authors suggest a first approach that tries to take advantage of this new feature.

The idea is to divide the reconfigurable area available on the FPGA in two parts. While part of the specification is being executed on one half of the board, the other one is reconfigured and thus prepared to execute the next partition (see Figure 9).

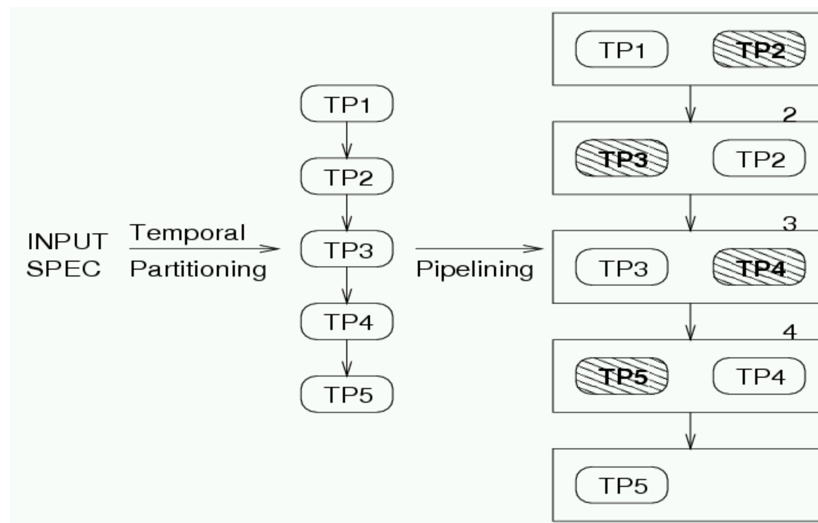

Figure 9. Alternated loading and executing of the partitioned specification.

Executions and reconfigurations are always in sync, meaning that what actually happens is the creation of time partitions, each containing an execution (let's say of task n, needing time E_n) and a reconfiguration (let's say of task $n+1$, requiring time R_{n+1}). They start working together at time t_0, and they are considered both busy until the time $t_0 + \max\{E_n, R_{n+1}\}$.

The best thing that can be hoped for is that all the time is actually spent in some computation, that is, there's no reconfiguration overhead. This means that in an optimal setting one would have

$$E_n \geq R_{n+1}, \qquad \forall n.$$

In order to achieve this the authors resort to the usual idea of encapsulating a loop inside any of the partitions.

Limitations of this approach seem to be the ability to find loops for every partition, and the question of whether loops will fit in the partition itself. The problems are emphasized by the fact that the partition size equals only half the reconfigurable area available on the FPGA (and therefore on one hand they are able to fit less operations, and on the other one one needs to find twice as many loops).

Also, because of the input representation used in the paper, only one thread is executing at any time in the running partition, thus somehow limiting parallelism.

In [17] the authors approach a similar problem but from a somewhat different perspective — that is, emphasizing the need for a reduction in the compilation time as opposed to a minimization of the total latency. They hence suggest a macro–based approach that eliminates the need for logical synthesis and technology mapping phases of the usual design flow, using a library of (relatively) pre–placed and pre–routed macros that can be mapped anywhere on the device via partial reconfiguration using the Jbits tool [34, 35].

Although this is certainly bound to decrease compilation time, the reconfiguration model is not very clear from the paper. It is not certain whether the proposed framework allows for multiple independent

tasks running on the board, and hence for the hiding of reconfiguration times like in the former paper, or if the partial reconfiguration is only a means to reduce the size of the data that has to be sent to the board at every reconfiguration (since on average each task to be mapped is smaller than the entire board).

It would however appear that the latter possibility is the case, since the authors talk about "initial configuration state" and "next configuration state", implying that the state is relative to *all the* FPGA, and hence that what actually happens is in reality a *temporal partitioning*, where the partitions are smaller than the entire board and hence require less reconfiguration time.

The result of this investigation into currently available literature on reconfigurable computing seems to suggest that only very few attempts have been made to fully exploit the most modern features of FPGAs, namely:

1. partial dynamic reconfigurability;
2. internal reconfigurability;
3. ability to host several different IP–Cores.

The approaches proposed so far tend to resort to time–partitioning (see §2.1) or to assume particularly "structured" layouts for the partial reconfiguration, thus restricting the degrees of freedom available to the designer in terms of functional mapping on the FPGA (see for instance §2.2.2).

We will try instead to maintain the level of freedom as far as placing on the chip is concerned that is currently available with modern FPGAs,. The approach, then, will try to cutting down the problem size through a smart choice of the grouping of the operations.

CHAPTER 3

MODEL

There are many kinds of FPGAs on the marker nowadays, all having different features and characteristics. Before building a model of a reconfigurable device it is worthwhile to take a look at the available architectures.

3.1 Reconfigurable architectures

Reconfigurable computing systems can be classified with respect to many different parameters. Here are some of the most important ones:

3.1.1 Granularity

By *granularity* one means the size of the smallest functional unit that can be reconfigured.

Typically FPGAs allow designers to define the computational structure of the board acting on as fine–grained elements as 4-input 3–output functional units (look–up tables, LUT). There are however coarser–grained boards, in which the reconfiguration defined the behaviour of arithmetic units of larger size (*e.g.* 32 bits).

Lower granularity provides more flexibility in adapting the hardware to the computation structure. However, lower granularity usually requires more time both statically, when in the compilation phase one must compose the required functions using many smaller components, and dynamically, since the size of the bitstream to download can be large.

Also, building complex (but common) functional units from simple LUTs requires a lot of resources and might imply un–optimal latency. That is why on many fine–grain FPGAs some common units are scattered, in order to avoid reimplementing them at high cost (*e.g.*, 18–bit multipliers in Xilinx Virtex–II Pro FPGAs).

3.1.2 Host coupling

All reconfigurable computing platforms need some kind of control unit to perform the control functions (configuring the logic, scheduling data input and output, interfacing with the external world, ...).

The interaction of this processor with the actual reconfigurable part of the board can be of different kinds:

- *Loose system-level coupling*, *i.e.* architectures which have reconfigurable logic communicating to the host through an I/O interface similar to a disk drive and other peripherals. A large number of first–generation FPGA–based boards are an example of this coupling (*e.g.* SPLASH [36]);
- *Loose chip-level coupling*, *i.e.* architectures where the overheads in communicating to the host is reduced by using direct communication between the host and the reconfigurable logic (*e.g.* PRISM [37]);
- *tight on-chip coupling i.e.* the processor is integrated on the same chip as the reconfigurable logic, significantly reducing the communication overheads between different components of the architecture (*e.g.* Garp [38]).

3.1.3 Reconfiguration Methodology

Typically, a reconfigurable device is configured by downloading configuration data, usually called *bitstream*, onto the device. The speed and means of download depend on the interface supported by the device.

3.1.3.1 Configuration memory

The bitstream to be loaded on the FPGA is stored in a local memory, called the *configuration memory*. This can be mainly managed in two ways:

- a *shared memory*, used for all the FPGA;
- a *context memory* (or possibly different context memories for different elements of the reconfigurable device).

In architectures with shared memory (*e.g.* Xilinx), while the new configuration of an area of the board is being loaded no computation can be executed on it. This means that reconfiguration and executions are two different and mutually exclusive states at any given time for any given reconfigurable area of the board.

Context memory, instead, allows the designer to load the data for the next configuration of part of the board *while* it is still performing a computation. There is then the operation of *context switching*, performing the actual reconfiguration, but it takes a negligible amount of time (especially if compared to shared memory architectures), of approximately one clock cycle.

On the other hand, this kind of reconfiguration tends to require more power and hence to generate much more heat than the shared memory approach.

Context memory is usually employed with coarse–grain architectures, where the memory of each reconfigurable unit and the reconfigurable unit itself can be more easily coupled.

3.1.3.2 Logic reconfigurability

The handling of the execution in presence of reconfiguration further partitions platforms in three categories.

Dynamic reconfiguration permits the board not to be reset during a reconfiguration, so that data stored in the internal registers is not erased and can be fed as input to the configuration to be loaded. This feature is important in that it allows time partitioning techniques.

Partial reconfiguration allows the change of the functionality of only a portion of the device, while the remaining portion retains its functionality.

Modern architectures are *partially dynamically reconfigurable*, *i.e.* they can be partially reconfigured[1] "on the fly", while the rest of the board continues its normal execution.

3.1.3.3 Configuration time

The time for configuration can be assumed to be affine in the size of the bitstream.[2]

This means that partial reconfiguration is a good way of reducing the reconfiguration overhead.

---

[1] With some limitations in placing accuracy.

[2] As already noted, coarse–grain devices typically need smaller configuration bitstreams and are thus in an advantage as far as reconfiguration time is concerned.

3.1.4 Memory Organization

The computation performed on the reconfigurable logic needs to access data from memory, and also intermediate results from computations also need to be stored before the logic can be reconfigured to perform the next computation.

The organization of the memory affects the data access cost and is a significant fraction of the actual execution time. Currently, most reconfigurable architectures include large memory on the reconfigurable logic device. This memory can be implemented as large blocks of memory (*e.g.* Virtex BlockRAMs) or as distributed memory blocks.

3.2 The reference board

In the context of the MicroLab, the research group this thesis has been developed in, the board used for testing is a Xilinx Virtex–II Pro (Figure 10).

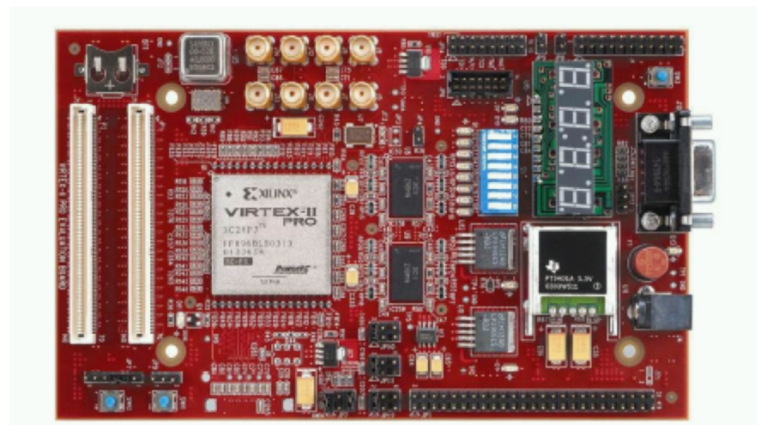

Figure 10. The testing board used in MicroLab: Virtex–II Pro by Avnet

The Virtex-II family is composed of different versions of fine–grained, partially dynamically reconfigurable, tight on-chip coupling FPGA boards with shared reconfiguration memory, which have capacities ranging from 50 thousand to 1 million system gates. The Virtex architecture is composed of an array of *configurable logic blocks* (CLBs), encircled by programmable I/O blocks, and dedicated block memories of 4096 bits each (Figure 11).

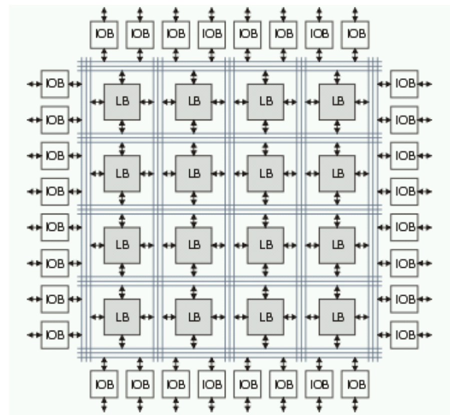

Figure 11. Structure of an FPGA

There is a hierarchical routing matrix with local routing and varying number of global routes of different lengths. There are 24 single length routes, 96 routes of length six and 12 long lines spanning the chip. There are additional I/O routing resources around the periphery of the logic blocks.

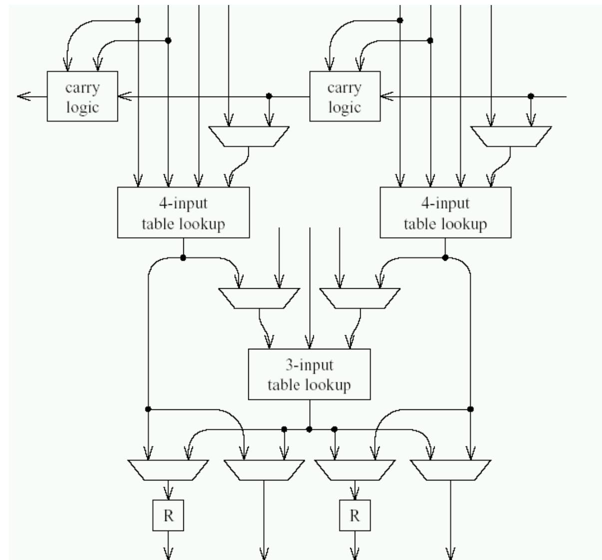

Figure 12. The structure of a logic block

The CLBs contain four logic cells each. Each logic cell (Figure 12) has a 4-input function generator (LUT), a flip-flop and some carry logic. The LUTs (Figure 13) can operate as function generators or they can be used as distributed RAMs.

Additional multiplexers and wires in a CLB provide flexible combination of different logic cell outputs and routing of input signals to CLB output. High speed arithmetic is facilitated by providing additional carry logic in each of the logic cells. A dedicated AND gate in each logic cell improves multiplier implementations.

On-chip local memories can be realized on the Virtex architecture in two different ways: the logic cells can be combined and configured as memory cells to obtain multiported RAM of required sized, but each Virtex also has large Block SelectRAM memories. These are organized vertically on the FPGA.

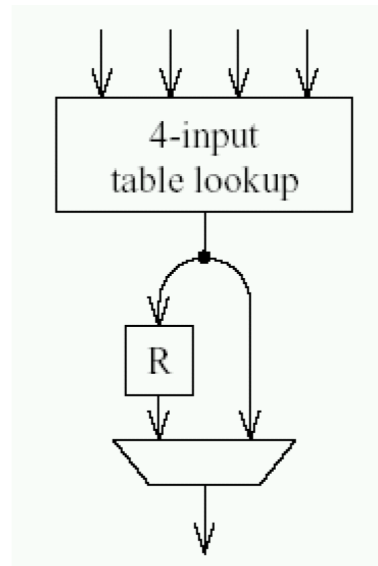

Figure 13. A LUT

Each memory block is four CLBs high.

Each such memory cell is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port and independent datawidths.

The Virtex–II Pro chips can integrate several hardware IP–cores, *e.g.* an ethernet interface for communication purposes, but most notably include a PowerPC microprocessor that enables it to be a self–sufficient, tight on-chip coupling FPGA.

3.3 The reconfigurable architecture model

3.3.1 The chip

The mathematical model of a reconfigurable device will be based on the (abstract) behavior of a Xilinx Virtex–II Pro.

As pointed out earlier, a reconfigurable device is basically made up of a set of computational units (that can be programmed to perform different operations), and of a set of routing resources (used to implement exchange of computed data).

In the reference chip the smallest reconfigurable element performing a well–defined computation is a logic cell. However, as preciously stated, logic cells are tightly coupled in groups of four inside CLBs. Hence, every time it will be needed to characterize the area (*i.e.*, computational resources) required for a function, this will be done by stating how many CLBs are necessary for its implementation.

However, given the current state of the technology and the available information released by Xilinx on its chips, it is not possible to arbitrarily reconfigure one CLB at a time. The chip is in fact arranged in columns (called *frames*), spanning multiple logic units inside multiple CLBs. The best that can be done so far is to reconfigure all the logic units inside a column with a width of one CLB (the so–called *reconfigurable unit*, RU).

Given a possible hardware execution unit $e \in E$, we will denote with $\rho(e) : E \mapsto \mathbb{N} : e \rightarrow \rho(e) =: r_e$ the number of CLBs necessary for the mapping of e on the FPGA.

As for the routing resources, it is customary in scheduling studies not to take them into account, thus assuming that the communication infrastructure provided around the CLBs is sufficient [9, 19, 13, 30, 14], or to model them in a very simplified way.

For instance a fixed overhead in terms of CLBs devoted to routing is sometimes assumed for every function that mapped on the chip. Alternatively, every execution unit e requires not only $\rho(e)$ CLBs for the implementation of its logic, but also other $\rho_r \rho(e)$ CLBs that will be used for routing purpouses, where ρ_r is fixed a coefficient (ideally 0) expressing CLBs routing overhead.

It is easy to see that the total number of CLBs needed for an execution unit is then

$$\rho(e) + \rho_r \rho(e) = (1 + \rho_r)\rho(e) =: \rho'(e),$$

so that this latter approach is virtually undistinguishable from the former one: one just needs to take into account the constant ρ_r in computing $\rho'(\cdot)$. For this reason it will be assumed that the routing overhead is included in ρ, to simplify the notation.

Given these premises, one can build a model of the chip. Let us assume that the FPGA is composed of a matrix of $n \times m$ CLBs: then the number of reconfigurable units $u \in U$ is given by

$$|U| := m,$$

so that we can write $U = \{u_1, u_2, \dots, u_{|U|}\}$, and each of these units is composed of

$$\rho(u) := n$$

CLBs.

The next thing to take into account is the reconfiguration process. It can be seen experimentally that the reconfiguration time can be assumed to be affine in the area that needs to be reconfigured

[39, 40]. Otherwise stated, if we write $\delta(e)$ to indicate the time needed for the reconfiguration of the reconfigurable unit e we can use an expression like

$$\widetilde{\delta}(e) : E \mapsto \mathbb{R} : e \rightarrow \widetilde{\delta}(e) := \delta_R \cdot \rho(e) + \Delta_R =: \widetilde{d}_e. \tag{3.1}$$

However, given the granularity of the reconfiguration process (each reconfigurable unit is reconfigured on its own and independently of the other ones) it makes sense to measure the reconfiguration time in terms of clock cycles. For this purpose we enumerate the set of the instants of the raising of the clock signal, so that the first clock cycle corresponds to 0, the second to 1 and so on. If we call this set $T = \{0, 1, \ldots, T_{max}\} \subseteq \mathbb{N}$ we have that the clock instants are $\{t_k \in \mathbb{R} \,|\, t_k = k\bar{t}, k \in T\}$, where $\bar{t}$ is the clock period.

Then we can define[1]

$$\delta(e) : E \mapsto T : e \rightarrow \left\lceil \widetilde{\delta}(e) \right\rceil =: d_e \tag{3.2}$$

Finally, one needs to measure the latency of each execution unit on the chip. For this purpose one introduces the function $\lambda(e) : E \mapsto T : e \rightarrow \lambda(e) =: l_e$.

The values of the functions λ and ρ over E will be estimated by the synthesizer (or in software through some metrics) for all the sets of operations.

---

[1] Please notice that we are asking T to do the double job of being the set of the instants *and* the set of the possible delays (differences between instants). It is somewhat of a notational stretch, but in order not to complicate the formulas we will proceed in this fashion.

Having modeled the FPGA chip we can now briefly describe the architecture. This work was born in the context of the efforts in MicroLab in Politecnico di Milano, where the Caronte architecture is being developed [41, 42, 39, 40].

The aim so far has been that of:

1. creating a methodology that enables an FPGA to act as a self–contained computer capable of running even applications that are too big to fit in the actual area available on the chip (through internal partial reconfiguration);
2. doing so without compromising the good speed achieved thanks to hardware execution (otherwise stated, hiding reconfiguration time as much as possible).

In order to obtain the first of these results an architecture was created, that divides the resources of the FPGA in two areas: a "static" one and a truly reconfigurable one. The static part is devoted to the controller of the reconfiguration and to all the rest of the logic needed to ensure the rest of the system works as needed, while the remaining area is actually used for hot–swapping functionalities that, one after the other, will implement the whole system.

In this framework it is clear that a certain number of execution units, say U_F have to be reserved for the fixed part of the architecture, so that only $|U| - U_F$ execution units will be available for actual reconfiguration.

Summing up, a reconfigurable architecture will be given as a tuple $\mathcal{D} = \langle E; T; U; \rho : \{E, U\} \mapsto N; \lambda, \delta : E \mapsto T \rangle$

3.3.2 The specification

It will be assumed that the specification to be mapped onto the FPGA is given in terms of a (directed) graph $\widetilde{G} = \langle \widetilde{O}, \widetilde{P} \rangle$, where $\widetilde{O} = \{o_i\}_i$ is the set of the operations to be performed and $\widetilde{P} = \{p_{ij} := o_i \rightarrow o_j := (o_i, o_j)\} \subseteq \widetilde{O}^2$ is a set of precedences among the operations.

Every operation performs some kind of computation. We describe this using a function

$$\alpha : O \mapsto A : o \rightarrow \alpha(o) =: a_o,$$

that associates each operation with one of the possible actions $a \in A$. The set A could contain for instance the actions "sum two 8–bit numbers" or "perform the logical AND of two bits" and so on.

Note that this simply amounts to stating that the graph G is *colored*.

The graph must not allow cycles with an unknown number of iterations (loops) or operations whose latency cannot be specified (such as user–interactions), since those situations can be dealt with only in an on–line fashion, while this work is addressing only the static computation of the schedule *before* the system execution starts.

For this reason we will require $\widetilde{G}$ to be a directed acyclic graph (DAG).

For simplicity three special operations will be added, o_0 ("null operation"), o_s ("start") and o_e ("end"), which act respectively as NOOP, and the source and sink nodes of the overall graph.

Therefore every operation o that doesn't have a predecessor will be assigned o_s as a predecessor, and every operation with no successor will have o_e as such. This amounts to considering the graph $G = \langle O, P \rangle$ where

$$\begin{aligned} O &:= \widetilde{O} & \cup \{o_s, o_e, o_0\} \\ P &:= \widetilde{P} & \cup \{o_s \to o, \forall o \in \widetilde{O} \,|\, \nexists\, o' \to o \in P\} \cup \\ & & \cup \{o \to o_e, \forall o \in \widetilde{O} \,|\, \nexists\, o \to o' \in P\} \end{aligned}$$

Thus the specification will be defined by the (augmented) graph $G = \langle O, P \rangle$.

3.3.3 Technology binding and mapping

It is necessary to find a way of implementing all the operations in O onto the FPGA. In order to do so one needs a synthesizer that from a specification o outputs an execution unit implementing o. We might represent this object with a function $\eta : O \mapsto E$, where E is the set of execution units.[1]

---

[1] What one should really do is, define a function $\widetilde{\eta} : A \mapsto E$, since it is the actual action that is implemented in hardware. However, one can assume to deal with the composition

$$\eta := \widetilde{\eta} \circ \alpha : O \mapsto A \mapsto E,$$

with no loss of generality.

The nature of η, though, requires more thinking. First of all, using ideas like *resource sharing* it is easy to see that sometimes it is worth looking for the execution unit that maps not only a single operation onto the FPGA, but a *series* of operations. What is worth looking for, then, is more something like

$$\eta : 2^O \supseteq \mathcal{O} \mapsto E : S \to \eta(S) =: e_S,$$

where S is a set of operations to be mapped together in one execution unit.

With this generalization one begins to wonder, however, that for complex sets of operations there are several different possible implementations. Among those two solutions it is not immediately obvious which one is the best. Or even better, the idea of "best" implementation is ill–posed, since it strongly depends on the overall system.

One easy example is the following.

Example 3.1 (There is no "best" implementation)

The chip has only one reconfigurable unit left available and the last set of operations to be executed, S, has two implementations e_1 and e_2 available. Assume the data in Table Table I for the e's and let's assume that $\rho(u) = 6$, $\delta_R = 4$ and $\Delta_R = 0$.

	$\rho(e)$	$\lambda(e)$
e_1	5	4
e_2	7	2

TABLE I

EXAMPLE OF NON–EXISTENCE OF "BEST" IMPLEMENTATION

If we use implementation e_1, it will fit in the remaining area of the FPGA ($\rho(e_1) = 5 < 6 = \rho(u)$) and will terminate its execution at $t_0 + \lambda(e_1) = 4$ (where t_0 is the clock cycle it starts at). If we use e_2 instead, it will not fit, requiring the reconfiguration of some area (specifically, two reconfigurable units). Hence the total time needed for the solution with e_2 would be

$$\lceil \delta_R \cdot \rho(e) + \Delta_R \rceil + \lambda(e_2) = \lceil .7 \times 5 \rceil + 0 + 2 = 4 + 2 = 6,$$

which is greater than that of e_2.

However, let us assume that in the scheduling process other choices were made, resulting in having S start at time $t_0 + 2$ but with more free space available, say two reconfigurable units. In this case choosing e_1 would result in finishing the computation in $t_0 + 2 + 4 = t_0 + 6$. Choosing e_2, instead, would not now require a reconfiguration, so that the computation would stop at $t_0 + 2 + 2 = t_0 + 4$, that is, earlier than with e_1. $\Diamond$

This example was meant to show how the choices of the implementations are intertwined and hence how it is unwise to consider only one execution unit for each set of operations. For this reason it is necessary to extend again the definition of η:

$$\eta : 2^O \supseteq \mathcal{O} \mapsto \mathcal{E} \subseteq 2^E : \{o_i\}_i \rightarrow \{e_j\}_j,$$

that is, η is a multivalued function.

The actual computation of η given a particular input can be done using a synthesis tool. Also, the set $\mathcal{E}$ can be defined to be the image itself of $\mathcal{O}$ under η. The real problem arises in defining $\mathcal{O}$.

In fact, $\mathcal{O}$ can be as huge as 2^O, but taking such a big set makes the search space enormous, so it will be necessary to construct $\mathcal{O}$ properly. The guidelines, however, are two:

- every operation must be present in at least one element of $\mathcal{O}$ (**completeness**):

$$\bigcup_{S \in \mathcal{O}} S = O;$$

- non–trivial sets should be present in $\mathcal{O}$ only if it is probable that they will result in improving the overall latency (**non–proliferation**).

We will see later that the first requirement is very easily taken care of by ensuring that

$$\forall o \in O, \quad \{o\} \in \mathcal{O}.$$

As for the second requirement, it will take a bit more work.

Finally, one defines the *null implementation* $e_0 \in E$ such that $\lambda(e_0) = 0$ and $\rho(e_0) = 0$, the sets $S_0 := \{o_0\}$, $S_s := \{o_s\}$, $S_e := \{o_e\}$ and defines $\eta(S_s) := \eta(S_e) := \eta(S_0) := \{e_0\}$.

3.3.3.0.1 Precedence

It must be noted that the precedence information we have through P is given in terms of the operations $o \in O$, while now we are working with the sets $S \in \mathcal{O}$. It is necessary, hence, to translate P in terms of the S's.

One can proceed as follows: for every $o_i \to o_j \in P$, consider all the sets S_i that contain o_i and all the sets S_j that contain o_j. Then add a dependency between every pair of the two sets (assuming they are not the same set). In formulas:

$$\mathcal{P} := \{(S_i, S_j) \,|\, \exists o_i \to o_j \in P, o_i \in S_i, o_j \in S_j, S_i \neq S_j\}. \tag{3.3}$$

This procedure has the disadvantage of loosing track of the clock cycle at which the dependency takes place.

Example 3.2 (Suboptimality for general grouping of operations)

Consider the specification in Figure 14.

The above procedure results in $\mathcal{P} = \{S_s \to S_1, S_s \to S_2, S_2 \to S_1, S_1 \to S_e, S_2 \to S_e\}$. This means that S_1 cannot start executing until S_2 terminates.

The problem lies in the fact that before the grouping in sets S, o_1 and o_3 could be executing *independently* of o_2, o_4 and o_6 — probably in parallel, speeding up the computation. The partitioning of the operations and the dealing with the precedences as defined in (Equation 3.3), instead, forces the scheduler to wait and hence wastes clock cycles.

This is caused by the fact that the precedence relation $o_4 \to o_5$ enters in S_1 in a node that is not the (one of the) first one(s) being executed, so that the dependence propagates backwards.

Similar problems arise when a precedence relation does not leave a set S from (one of the) last node(s) being executed. In this case, in fact, the dependency relation among sets gets satisfied only when *all the computation* in S is over. In our example, the relation $S_2 \to S_1$ is satisfied only when o_6 ends, while in the original dependence was satisfied earlier — namely when o_4 was over. $\Diamond$

One way of taking this into account is simply to enforce that each set S is chosen in such a way that dependency arcs leave from "last executed" nodes and enter from "first executed" nodes.

The latter request is easily taken into account stating that if a dependency enters in a node, then that node does not have any entering dependency with other nodes inside the set:

$$\forall S \in \mathcal{O}, \forall o \in S, \forall o' \in O \setminus S, \quad o' \to o \in P \Rightarrow \nexists \bar{o} \in S \,|\, \bar{o} \to o \in P. \tag{3.4}$$

The former case is not so easily handled. One can try with the dual formulation of (Equation 3.4), writing

$$\forall S \in \mathcal{O}, \forall o \in S, \forall o' \in O \setminus S, \quad o \to o' \in P \Rightarrow \nexists \bar{o} \in S \,|\, o \to \bar{o} \in P.$$

Unfortunately, this does not ensure that the dependency is satisfied only at the end of the execution. For instance, take Figure 14 and delete the precedence $o_4 \to o_6$. In this case it is perfectly possible that, within S_2, o_6 is executed in parallel with the sequence (o_2, o_4), but requiring a much larger time than them. In this case, even if o_4 terminated, o_5 would still unnecessarily wait for o_6.

One solution is to also enforce that every set S has a "sink", just what o_e is for the whole program. This might seem a very strict requirement, but it will suit many specifications very well. In particular, it will turn out that one often wants to partition the code in such a way that it only has one outgoing dependency at the end, since this makes it simpler to reuse it, possibly in an iterative/cyclic fashion.

Another drawback of this approach is that it can turn directed acyclic graphs into cyclic graphs.

Example 3.3 (From acyclic to cyclic specification)

Take the specification in Figure 15.

It is the same graph as in Example 3.2, with an edge added (namely, $o_1 \to o_6$). This results in a new edge in $\mathcal{P}$ as well, namely $S_1 \to S_2$.

Hence is turns out a cycle $S_1 \to S_2 \to S_1$ was created from a specification that was acyclic. $\Diamond$

For this reason it will be necessary to avoid the formation of sets that have both incoming and outgoing dependencies towards another set:

$$\forall S_1, S_2 \in \mathcal{O} \quad \exists o_1 \to o_2 \in P, o_1 \in S_1, o_2 \in S_2 \implies$$
$$\implies \nexists o_3 \to o_4 \in P, o_3 \in S_2, o_4 \in S_1.$$

3.3.4 Constraints

3.3.4.0.2 Unknowns

Scheduling is a mapping σ that assigns to each operation of the specification a clock cycle in which to start its execution:

$$\sigma : O \mapsto T : o \to \sigma(o).$$

In our approach, however, we have several differences. First of all, what needs to be scheduled are not operations anymore, but the sets of operations $S \in \mathcal{O}$. Indeed, for every S that has to be implemented one must choose which of its several implementations $\eta(S)$ to use.

Moreover, not all of the sets in $\mathcal{O}$ are to be scheduled: in fact it can well be that different sets $S_1, S_2 \in \mathcal{O}$ overlap, in the sense that they have some operations in common:

$$S_1 \cap S_2 \neq \varnothing.$$

If this is the case, one must ensure that every operation is executed only once. To accomplish this, a subset $\widetilde{\mathcal{O}} \subset \mathcal{O}$ is to be chosen, containing only the sets of operations actually scheduled to be executed.

Finally, one must decide in which reconfigurable unit every $S \in \widetilde{\mathcal{O}}$ is to be put.

For these reasons, the scheduler has to be extended like

$$\sigma : \widetilde{\mathcal{O}} \mapsto T \times 2^U \times E : S \rightarrow \sigma(S) =: (t_S, U_S, e_S),$$

where U_S is the set of reconfigurable units where S will be executed, t_S is the clock cycle when S starts running, and $e_S \in \eta(S)$ is the particular implementation of S to be used.

Also, one needs to decide when the reconfiguration takes place. Since in theory many S's can share the same execution unit, it is useful to define a separate (partial) function that tells when and where a particular execution unit is reconfigured for an operation set using it:

$$\widetilde{\sigma} : E \times \widetilde{\mathcal{O}} \mapsto T \times 2^U, e \rightarrow \widetilde{\sigma}(e) =: (\bar{t}^e_S, \overline{U}^e_S).$$

Of course, if $e \neq e_S$, $\sigma(e, S)$ is not defined.

3.3.4.0.3 Partition

All the specification is mapped, that is, no operations are left out. This can be obtained requiring that $\widetilde{\mathcal{O}}$ is a partition of O:

$$\forall S_1, S_2 \in \widetilde{\mathcal{O}}, S_1 \cap S_2 = \varnothing, \qquad \bigcup_{S \in \widetilde{\mathcal{O}}} S = O.$$

Actually, one could even want to allow some logic to be replicated in order to obtain possible speedups. However, since the primary problem tackled in this work is the shortage of available CLBs it is a wise choice to enforce this constraint.

3.3.4.0.4 Uniqueness

Another requirement is embedded in the statement that σ and $\widetilde{\sigma}$ are functions, that is, their image is only *one* point. This means that every set S that is chosen to be used in the implementation is mapped to one and only one reconfigurable unit to one and only one execution unit, and starts only once at a particular clock cycle.

Also, every chosen execution unit is reconfigured once and only once, and to a particular reconfigurable unit, for every S it serves.

3.3.4.0.5 Reconfiguration

There are several constraints that need to be taken into account.

1. If a set S gets executed, its execution unit must have been reconfigured before that happens. In particular, reconfiguration also takes some time:

$$\forall S, \quad \overline{U}_S^{e_S} = U_S \;\wedge\; \overline{t}_S^{e_S} \le t_S - \delta_{e_S}.$$

2. if a reconfiguration starts in a reconfigurable unit, nothing can use it before the reconfiguration is over: neither an execution

$$\forall S \quad \nexists S' \ne S \,|\, U_{S'} \cap \overline{U}_S^{e_S} \ne \varnothing \;\wedge\; \overline{t}_S^{e_S} \le t_{S'} < \overline{t}_S^{e_S} + \delta_{e_S},$$

nor another reconfiguration process

$$\forall S \quad \nexists S', e_{S'} \ne e_S \,|\, \overline{U}_{S'}^{e_S'} \cap \overline{U}_S^{e_S} \ne \varnothing \;\wedge\; \overline{t}_S^{e_S} \le \overline{t}_{S'}^{e_S'} < \overline{t}_S^{e_S} + \delta_{e_S}.$$

It can be noted that the latter equation *allows* the reconfiguration of another set S' to be shared with that of S, given $e_{S'} = e_S$, allowing reuse of the same execution unit for the execution of different operation sets.

3. the freedom given by the last equation must be balanced asking that there is no reconfiguration in between the reconfiguration for a set S and its execution:

$$\forall S, \quad \nexists S', e_{S'} \ne e_S \,|\, U_S' \cap U_S \ne \varnothing \;\wedge\; \overline{t}_S^{e_S} \le \overline{t}_{S'}^{e_S'} \le t_S.$$

3.3.4.0.6 Duration

Every execution unit has time to complete its task, that is, it is not interrupted by an execution

$$\forall S \quad \nexists S' \neq S \,|\, U_{S'} \cap U_S \neq \varnothing \;\wedge\; t_S \leq t_{S'} < t_S + \delta_{e_S}$$

or a reconfiguration

$$\forall S \quad \nexists S' \neq S \,|\, \overline{U}^{e_{S'}}_{S'} \cap U_S \neq \varnothing \;\wedge\; t_S \leq \overline{t}^{e_{S'}}_{S'} < t_S + \delta_{e_S}.$$

3.3.4.0.7 Precedence

The schedule must satisfy the precedence constraints in $\mathcal{P}$:

$$\forall S_i \rightarrow S_j \in \mathcal{P}, \quad t_{S_j} \geq t_{S_i} + l_{e_{S_i}}.$$

3.3.4.0.8 Size

Each execution unit must fit in the reconfigurable units assigned to it:

$$\forall S, \quad \rho(e_S) \leq \rho(U_S),$$

where $\rho(U)$ is defined to be $\rho(U) := \sum_{u \in U} \rho(u)$.

3.3.4.0.9 Adjacency

If an execution unit needs more than one reconfigurable unit, then they must be adjacent. If we assume that the set U is enumerated in the natural way according to the column position, so that u_i is adjacent to $u_{i\pm1}$, then we can require that

$$\forall S, |U_S| > 1, \quad \exists h, k \in \mathbb{N} \,|\, U_S = \{u_h, u_{h+1}, \ldots, u_{h+k}\}.$$

3.3.5 Aim

The global aim is to find the partition $\widetilde{\mathcal{O}}$ and the functions σ and $\widetilde{\sigma}$ that minimize the total latency of the specification G, namely

$$\underset{\widetilde{\mathcal{O}},\sigma,\widetilde{\sigma}}{\operatorname{argmin}} \; t_{S_e}.$$

At this point we have built a comprehensive model of the reconfigurable device, and have suggested a formalism that hints at finding a particular type of solution — *i.e.*, one that shifts part of the complexity of the problem into a somewhat disjoint subproblem to be tackled independently from the scheduling one: the partitioning of the tasks in task–sets (in our notation, going from O to $\mathcal{O}$.

In this way we can hope to maintain a larger freedom in the management of the reconfigurable areas than is often allowed in previous works (see §2.2.2), while still keeping the problem solvable in a reasonable time.

We are now going to explore the possible ways to perform this partitioning.

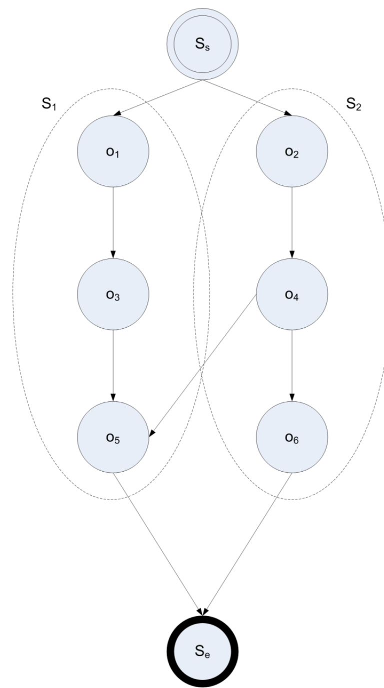

Figure 14. Example of specification

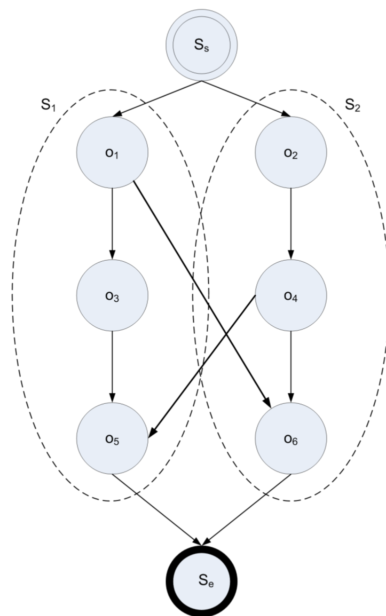

Figure 15. DAG specification becomes cyclic

CHAPTER 4

PARTITIONING

Now that the model has been built, let us investigate how to solve the problem.

The first task that has to be done is the computation of the collection $\widetilde{\mathcal{O}}$ of sets of operations that are to be mapped onto the FPGA.

As stated in §3.3.3, this can be done in three steps:

1. investigate whether or not the sets S have to satisfy certain requirements in terms of internal structure (**shape description**);
2. build a collection $\mathcal{O} \subseteq 2^{O}$ of sets S that are "reasonable" candidates as operation groupings (**candidate generation**);
3. decide what subset $\widetilde{\mathcal{O}} \subset \mathcal{O}$ contains the sets to be actually used in the final implementation (**candidate pruning**).

Let us briefly summarize here the content of §3.3.3 that deals with this phase.

The input of the partitioning step is a directed acyclic graph $G = \langle O, P \rangle$, and part of the output will be the new collection of sets $\mathcal{O}$. However, once the partitioning of the operations is performed, one still wants a schedulable specification to work on. To this end, the real output of this phase will be a new directed acyclic graph $G' = \langle \mathcal{O}, \mathcal{P} \rangle$, whose nodes are the sets $S \in \mathcal{O}$, and whose precedences are to be consistent with those of P.

Given any collection $\mathcal{O} \subseteq 2^O$ of subsets of O, the natural way to define the precedences $\mathcal{P} \subseteq \mathcal{O} \times \mathcal{O}$ induced by P is given by (Equation 3.3):

$$\mathcal{P} := \{(S_i, S_j) \mid \exists o_i \to o_j \in P, o_i \in S_i, o_j \in S_j, S_i \neq S_j\}.$$

This definition of $\mathcal{P}$ unfortunately gives some problems. Let's take a look at them, and try to solve them imposing some constraints on the structure of the sets S.

4.1 Shape description

4.1.1 The induced dependencies

All the problems arise from a simple fact:

Lemma 4.1 *In the notation of §3, consider two graphs* $G_1 = \langle O, P_1 \rangle$ *and* $G_2 = \langle O, P_2 \rangle$*. Let* $S_1, S_2 \in \mathcal{O}$ *and* $o_1 \in S_1$*,* $o_2 \in S_2$*.*

Assume $P_1 = \{o_1 \to o_2\} \cup \overline{P}$*, where* $\overline{P}$ *is a set of precedences that don't leave* S_1 *and then enter* S_2*. Set* $P_2 = \{o_i \to o_j \mid o_i \in P_1, o_j \in P_2\} \cup \overline{P}$*.*

Then $\mathcal{P}_1 = \mathcal{P}_2$*.*

Proof. Follows immediately by noticing that by (Equation 3.3) all the nodes in $P_2 \setminus \overline{P}$ end up contributing in the single $S_1 \to S_2$. That is, the same result to which the single edge $o_1 \to o_2$ amounts. $\square$

In other words, having one precedence from one node in a set to one node in another set is equivalent to having all the possible precedences among nodes (see Figure 16 and Figure 17).

From this property one gets the two issues already discussed in §3.3.3:

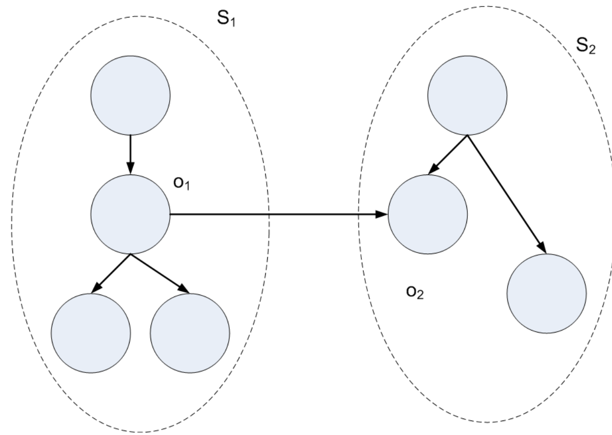

Figure 16. Specification equivalent to that in Figure 17

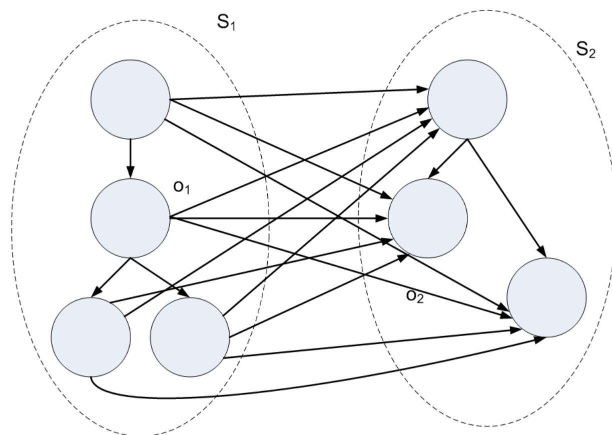

Figure 17. Specification equivalent to that in Figure 16

1. the resulting graph G' may not be acyclic (see Example 3.3). This happens when there exist two pairs of nodes (where in each pair one belongs to a set S_1 and the other to a set S_2) have a dependency in "opposite directions". Because of Lemma 4.1, the dependency causing $S_1 \rightarrow S_2$ can be seen *as if* one had all the possible dependencies from operations in S_1 to operations in S_2. Hence the dependency that causes $S_2 \rightarrow S_1$ is bound to form a cycle.

2. the resulting dependencies may force a sub–optimal schedule (see Example 3.2). This is obvious if considering the proliferation of edges seen in Lemma 4.1.

   This happens in two cases:

   (a) a precedence enters a set S at a node that has some parents inside S (see Figure 18).

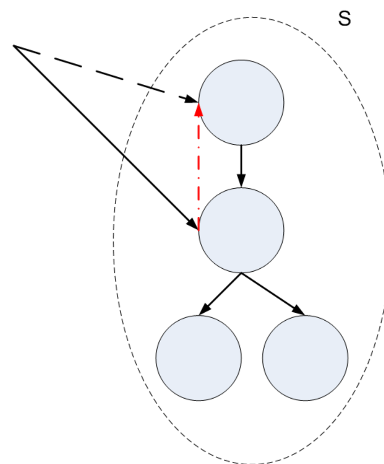

Figure 18. The precedence is actually entering at the first node being executed

In this case, the precedence must be complied to by all the nodes in S — in particular by the first one scheduled for execution.

(b) a precedence exits a set S from a node that has some children inside S (see Figure 19).

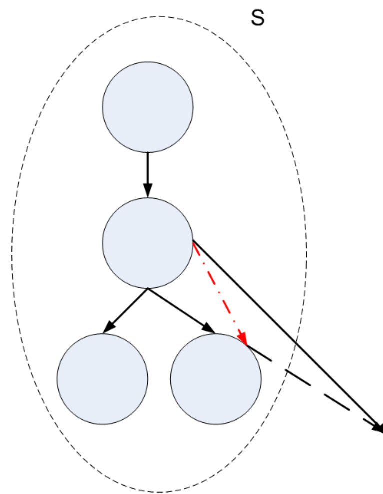

Figure 19. The precedence is actually leaving from the last node finishing execution

In this case, the precedence requires that all the nodes in S have terminated their execution — in particular, that the last one to finish running is done.

For these reasons, all the candidates will have to satisfy some requirements.

Problem 2a can be tackled as suggested in (Equation 3.4):

$$\forall S \in \mathcal{O}, \forall o \in S, \forall o' \in O \setminus S, \quad o' \rightarrow o \in P \Rightarrow \nexists \bar{o} \in S \,|\, \bar{o} \rightarrow o \in P.$$

that is, enforce that precedences enter only in nodes with no parents in their set S.

This approach does not solve all problems, though. Consider the set S in Figure 20, and suppose that operation o_2 takes the same time as operation o_1. The dependency entering at o_1 propagates to o_2, so that o_2 cannot start executing as soon as possible, but has to wait.

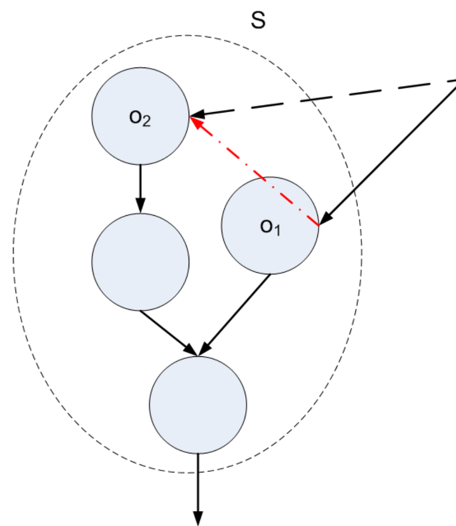

Figure 20. Counterexample to the easy solution of entering edge problem

There is no way to avoid this problem other than enforcing that every graph $\langle S, P \cap (S \times S)\rangle$ is a directed acyclic graph with a root node, and all the dependencies from outside S enter at the root.

This is a somewhat strict condition. Enforcing it means, for instance, prohibiting putting on an execution unit two independent tasks running in parallel.

Everything said for problem 2a is dually valid for problem 2b: the only solution would be to enforce that all the nodes in S and the (reverse) precedences in $P \cap (S \times S)$ form a rooted directed acyclic graph.

We can summarize this as follows:

Definition 4.1 *The sets S most suited of becoming elements of $\mathcal{O}$ are subgraphs of the specification G satisfying:*

1. $\langle S, P \cap (S \times S) \rangle$ *is a directed acyclic graph;*
2. $\langle S, \{o_1 \to o_2 \,|\, o_2 \to o_1 \in P \cap (S \times S)\} \rangle$ *is a directed acyclic graph.*

We won't enforce that all the S's satisfy Definition 4.1, but we will keep in mind that those properties are important, especially for tasks that have to be executed several times. In fact, even a small suboptimality in latency could become important if iterated.

As for problem 1, *i.e.* the possible loss of acyclicity, the matter is more serious since cyclic graphs cannot be statically scheduled. For this reason it will be necessary that

$$\forall S_1, S_2 \in \mathcal{O}, \quad \nexists o_1, o_1' \in S_1, o_2, o_2' \in S_2 \,|$$
$$|\, o_1 \to o_2 \ \wedge \ o_2' \to o_1'.$$

4.2 Candidate Generation

The question to be answered in this section is: what are the desirable properties of an execution unit from the point of view of its composing operations?

The solution has to be found in the specific domain being addressed. In this case, we are looking for a good scheduling on reconfigurable devices, where reconfiguration is used to fit a "large" specification on a "small" (but "fast") device.

The aim is to make it so that the feature (reconfiguration) that is being used to overcome the limitation (size) does not affect the good qualities (hardware execution speed) of the tool we are using (FPGA).

Moreover, let us take a look at the orders of magnitude: how much does reconfiguration affect execution?

In the reference architecture, since the number of columns is $m = 68$, there are $|U| = m = 68$ reconfigurable units. Hence one can imagine that an average execution unit e occupies 5–8 reconfigurable units. This means that it contains more or less

$$\rho(e) \approx (6.5 \times 1) \times 80 = 520$$

CLBs.

Also, experimentally in the reference architecture one finds out that the reconfiguration process can be modeled using (Equation 3.1) and (Equation 3.2) with parameters $\delta_R = 0.079\,\frac{\text{ms}}{\text{CLB}}$ and $\Delta_R = 0\,\text{ms}$, so that

$$\delta(e) \approx 520\,\text{CLB} \times 0.079\,\frac{\text{ms}}{\text{CLB}} + 0\,\text{ms} \approx 41\text{ms}.$$

Let us compare this with the execution time of an average execution unit. It is a reasonable estimate to assume that an 8–bit multiplier requires 32 CLBs and 8 clock cycles, while a 16–bit adder requires

only 8 CLBs and 3 clock cycles. If the average execution unit has 520 available CLBs, one can think of fitting on it, *e.g.*, 8 multipliers and 30 adders.

Assuming some degree of parallelism inside the execution unit, one can estimate a latency of

$$\lambda(e) \approx 0.8 \times (8 \times 8 + 30 \times 3) \approx 123$$

clock cycles.

In terms of seconds, assuming the clock runs at 100MHz, one gets

$$\lambda(e) \approx 123/100\text{MHz} \approx 1.23\mu\text{s}.$$

Comparing the two values obtained by this very quick analysis it is still apparent that reconfiguration time is big with respect to a straight–line execution of an execution unit.

Thus, it looks like the first aim of the scheduler is to *reduce the need for reconfiguration*.

The first way of doing this is a careful structuring of the sets $S \in \mathcal{O}$ enabling the system to *reuse* execution units that are already loaded on the FPGA.

4.2.1 Reuse

So our aim has become that of try and maximize the reuse of execution units.

The best way of obtaining this result is looking for parts of the specification that are "the same", *i.e.*, that perform the same computation.

Since the specification is given in terms of a directed acyclic graph, the problem reduces to that of finding similar subgraphs in G.

First of all, we will need to define the concept of “similar”.

The specification, as we said, is given in terms of a directed acyclic graph. In particular, it will be from now on assumed that the input graph G is a data flow graph.

As such, more can be said about its structure. In fact, every node is associated with an operation, and each node represents a data exchange. Suppose node o performs a division, *e.g.*, $k \leftarrow m/n$, where m and n are the inputs and k is the output. Graphically, this will be represented as shown in Figure 21.

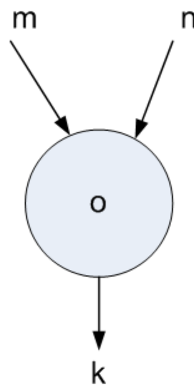

Figure 21. A simple operation represented in a data flow graph

Hence, one must first of all pay attention to the *orientation* of the edges incident to the node, since they give information about whether the data being transmitted is an input or an output.

Another important fact is apparent if one compares Figure 21 with Figure 22.

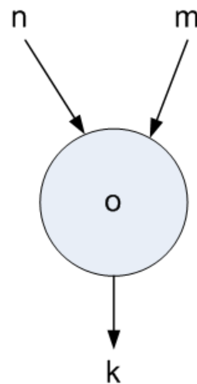

Figure 22. A different (?) simple operation represented in a data flow graph

From the usual graph–theoretic point of view, the two graphs would be "the same" (*i.e.*, they are *isomorphic*). However, one wants to distinguish the two operations $k \leftarrow m/n$ and $k \leftarrow n/m$, so that it is necessary to introduce some notation to discriminate between them. The easiest thing to do is to *enumerate* the inputs of every possible operation o, and connecting each edge to the proper input — see Figure 23.

In this manner, one can easily distinguish $k \leftarrow m/n$ from $k \leftarrow n/m$, as it is done in Figure 24a–b.

Please notice that this is not necessary for every kind of operation. In particular, *commutative* operations do not require this treatment — indeed, enforcing the numbering of the inputs also for them could prevent us from finding useful code redundancies.

One final note is the following. Let us assume (and it is indeed a reasonable assumption) that every operation o can have multiple inputs (usually up to two), but only *one* output.

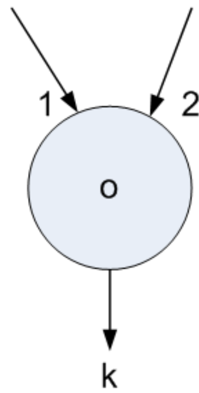

Figure 23. A data flow graph representation of an operation *with numbered inputs*

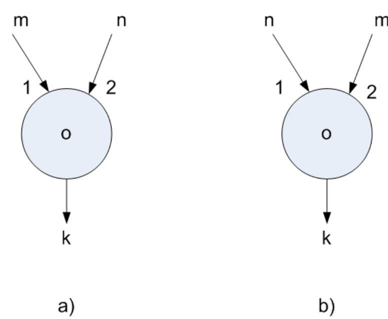

Figure 24. Different operand bindings with the same operation

Even so, it is very natural for the result of a *single* operation to be propagated to *several* different operations (see Figure 25).

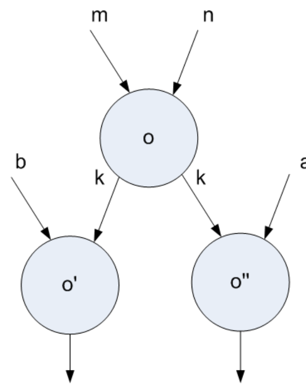

Figure 25. A single operation becomes operand for multiple operations

This, however, does not create any further problems, since all the outgoing edges from any node represent *the same data*.

In order to put all of these into formulas, we just need to add two new functions:

1. one is needed to characterize how many operands each operation $o \in O$ needs:

$$\kappa : O \mapsto \mathbb{N} : o \to \kappa(o) =: k_o;$$

2. one maps every edge entering a node to its "slot": define $P_o^{\leftarrow}$ as the set of edges entering o:

$$P_o^{\leftarrow} := P \cap (O \times \{o\}).$$

Then for every node (performing a non–commutative) one can define a function μ_o

$$\mu_o : P_o^{\leftarrow} \mapsto \{0, \dots, k_o\} : (o' \to o) \to \mu_o(o' \to o) =: m_{o' \to o}.$$

This function can be of course extended to a single (partial) one defined over P:

$$\mu : P \mapsto \mathbb{N} : (o' \to o) \to \mu(o' \to o) := m_{o' \to o}.$$

With this machinery, we are now ready to define our sub–problem.

4.2.2 Graph Isomorphism

The aim is to identify in the program to be implemented subprograms that perform the same computation. In terms of graphs, this amounts to finding similar subgraphs inside the input data flow graph.

As stated before, one needs to *define the concept of "similar"*.

From the graph–theoretical point of view the similarity of two graphs can be studied in several ways. However, we are interested in a very strong form of similarity.

In fact, it is true that if we want the subprograms to perform the same computation we might look for *different implementations* that are equivalent from with respect to the outputs — *e.g.*, we could say that $h \leftarrow a+b; k \leftarrow h-c$ is equivalent (assuming only k is given as an output) to $h \leftarrow a-c; k \leftarrow h+b$.

However, this introduces a level of complexity that is better tackled and taken into account in a previous phase of the compilation process — the code optimization phase semantically studied in Compiler Theory [43].

For this reason, we are interested in the structural *identity* of subgraphs, *i.e.* the concept of *graph isomorphism*, so that we are left with finding isomorphic subgraphs of our input specification.

4.2.2.1 Problem definition

Definition 4.2 (Isomorphic Graphs) *Two undirected [resp. directed] graphs* $G_1 = \langle O_1, P_{O_1} \rangle$ *and* $G_2 = \langle O_2, P_{O_2} \rangle$*, where* $P_O \subseteq \{\{o, o'\} \mid o, o' \in O\}$ *[resp.* $P_O \subseteq \{(o, o') \mid o, o' \in O\} = O \times O$*] are said to be* isomorphic*, written* $G_1 \cong G_2$*, if there exists a bijection* $p : O_1 \mapsto O_2$ *such that*

$$\{o, o'\} \in P_{O_1} \iff \{p(o), p(o')\} \in P_{O_2}$$

[resp. $(o, o') \in P_{O_1} \iff (p(o), p(o')) \in P_{O_2}$*].*

There are several problems dealing with graph isomorphism in Graph Theory. Only to name a few:

- GRAPH ISOMORPHISM [44]: deciding whether two given graphs are isomorphic;
- SUBGRAPH ISOMORPHISM [45][46, problem GT48]: given two graphs G_1 and G_2, decide whether a subgraph of G_1 is isomorphic to G_2;
- LARGEST COMMON SUBGRAPH (also MAXIMUM COMMON SUBGRAPH) [46, problem GT49]: given two graphs G_1 and G_2, find the largest graph $\widetilde{G}$ that is isomorphic both to a subgraph of G_1 and to a subgraph of G_2.

Of these problems, only GRAPH ISOMORPHISM is not known to be NP–complete. Also, all of them are being intensely studied for their numerous applications.

However, what we are interested in is yet a different problem:

Definition 4.3 (ISOMORPHIC SUBGRAPHS) *Given a graph G, find two disjoint isomorphic subgraphs G_1, G_2 of G.*

ISOMORPHIC SUBGRAPHS has apparently received little attention so far, as opposed to some of its related problems — *e.g.*, LARGEST COMMON SUBGRAPH is being widely studies for its applications to chemical analysis.

Unfortunately, ISOMORPHIC SUBGRAPHS cannot be reduced to any of these other problems, and one is left with the little literature available [47, 48].

First of all, one must clarify the definition. In fact, in identifying a subgraph of a graph we have at least two possibilities:

1. choosing a subset of nodes and including in the graph all the edges connecting them;
2. choosing a subset of edges and including in the graph all the nodes they are incident to.

In order to ease the notation, let us give the following auxiliary

Definition 4.4 *Let $G = \langle O, P \rangle$ be an undirected [resp. directed] graph. An edge $p = \{o_1, o_2\}$ [resp. $p = (o_1, o_2)$] is said to be* incident *to a node o, and we write $p \perp o$, if and only if $o = o_1$ or $o = o_2$.*

This leads to the following definitions (found in [47] and extended to include the case of directed graphs):

Definition 4.5 *Let $G = \langle O, P \rangle$ be an undirected [resp. directed] graph, and let $O' \subseteq O$. The subgraph H given by the nodes O' and the edges $\{p \mid \exists o \in O', p \perp o\}$ is the* node–induced subgraph $G|_{O'}$ *of G.*

Let $G = \langle O, P \rangle$ be an undirected [resp. directed] graph, and let $P' \subseteq P$. The subgraph K given by the edges P' and the nodes $\{o \mid \exists p \in P', p \perp o\}$ is the edge–induced subgraph $G|^{P'}$ *of G.*

These two definitions bring on two different versions of ISOMORPHIC SUBGRAPHS: finding two disjoint node–induced subgraphs, or finding two disjoint edge–induced subgraphs — these problems are referred to as ISOMORPHIC EDGE–INDUCED SUBGRAPHS and ISOMORPHIC NODE–INDUCED SUBGRAPHS.

Both of these problems have been shown to be NP–complete [47, §3], not only for general graphs, but also for several classes of restricted instances (*e.g.*, connected outerplanar and two–connected planar graphs).

We are interested in partitioning the operations and the communications that take place in between the nodes we are dealing with. For this reason, we will focus on ISOMORPHIC NODE–INDUCED SUBGRAPHS.

4.2.2.2 Algorithms

The only class of graphs for which a polynomial algorithm is known are trees — in this case, the algorithm is even linear in time. As for the general case, only a heuristic algorithm is available.

We are interested in subgraphs of a data flow graph, which in general is *never* a tree (because we assumed that all the operations have o_s as sink). One could think of attempting some kind of partial reduction of the problem from a general data flow graph to a tree, in order to gain some insight in the problem, even if not solving it completely.

The algorithm for isomorphic subtrees of a given tree works in three steps:

1. from the initial tree T, build its representation as a string w;
2. compute the longest identical well–nested non–overlapping substrings of w;
3. from the substrings, go back and retrieve the isomorphic subtrees.

Steps 1 and 3 are straightforward: a tree can always be represented with every node marked by a pair of parenthesis, containing all its children nodes in between the parenthesis themselves. For instance, the string (()(()())) represents a tree with the root having two children nodes. The first child is a leaf, while the second one has, in turn, two children nodes which are leaves.

All that remains is to compute the longest identical well–nested non–overlapping substrings. In order to do this, one constructs the *suffix tree* of w:

Definition 4.6 *Given a string $w\$$, its suffix tree is a rooted tree with $|w| + 1$ leaves such that:*

- *every internal node except the root has at least two sons;*
- *every edge is labeled with a nonempty substring of w such that:*
  - *the labels of every edge leaving a node begin with a different symbol;*
  - *every suffix of $w\$$ can be obtained by concatenating the labels of the edges on a path from the root to a leaf;*
- *the leaf representing the suffix of w starting at the symbol i is labeled with i.*

This tree can be build in linear time (in the length of the string) [49].

Once the tree is build, one builds the triples (v_i, s_i, P_i) for every internal node v_i, where s_i is the concatenation of the labels of the edges connecting the root to v_i, and P_i is the set of the labels of the leaves reachable from v_i, which are the indexes where the string s_i can be found in w.

This construction, again, requires linear time.

The last step is to check if the strings s_i can be found in different spots of the string w (and we know they are to be found exactly at the locations P_i) which are not overlapping and are well–nested.

Again, these checks can be done efficiently in linear time if care is taken in the implementation.

Now, it can be shown that if a graph G has two disjoint isomorphic subgraphs S_1 and S_2, and if these subgraphs satisfy some requirements (namely Definition 4.1), then also the restriction of these subgraphs to the edges of a spanning tree T of G must have some structure in common. In particular, it turns out that S_1 and S_2 have non–trivial isomorphic subgraphs.

This would mean that, assuming G has "good" isomorphic subgraphs, one can spot their location solving the restriction of the problems for trees.

Unfortunately this construction, which is detailed in Appendix A, does not prove to be useful. The problem is that the algorithm using strings finds isomorphic *subtrees* of a given tree, *i.e.*, it looks for two nodes that are roots of isomorphic trees — it checks for isomorphism all the way to the leaves, while in our case we are interested also in *local* properties, in the neighborhood of some nodes.

We need to find another approach.

Since the general case is known to be NP–complete, it is necessary to resort to some heuristic. The algorithm proposed in [48] is based on the idea of choosing a certain number of points $(o, p(o)), o \in O_1$, and checking if the neighborhoods of o and $p(o)$ are isomorphic, locally.

More specifically, the proposed algorithm is the following:

1. initialize two sets of nodes $V_1 = \{v_1\}$ and $V_2 = \{v_2\}$ which are isomorphic. Set $P = \{(v_1, v_2\}$;
2. while $P \neq \varnothing$ do
   (a) choose $(v_1, v_2) \in P$ and remove it from P;
   (b) set $V_1 = \{v_1\}, V_2 = \{v_2\}$;
   (c) choose neighborhoods N_1 and N_2 of v_1 and v_2, resp.;
   (d) compute the optimal weighted bipartite matching M over N_1 and N_2 according to a weight function;
   (e) for all the edges (u_1, u_2) of M, if $G|_{V_1 \cup \{v_1\}} \cong G|_{V_2 \cup \{v_2\}}$ then
      i. $P = P \cup \{(u_1, u_2)\}$;
      ii. $V_1 = V_1 \cup \{u_1\}, V_2 \cup \{u_2\}$.

In other words, it starts with an initial pair of nodes (that are known to be isomorphic). Then, for each of these pairs, it computes a neighborhood using a specified function[1], and created a matching between these neighborhoods maximizing a weighting function that takes into account graph–theoretic measures, trying to describe the degree of similarity of the neighborhoods. For instance, it tries to maximize the quantity $-|\text{degree}(u_1) - \text{degree}(u_2)|$ summed over for every edge (u_1, u_2) in the matching.

[1]Further details about the choice of the neighborhood are missing in the paper.

Once the matching M is completed, the algorithm checks if adding the new edges of M one by one results in enlarged subgraphs that are actually isomorphic. If so, the new nodes are added to V_1 and V_2, and the pair composed of the two nodes is inserted in P.

The initialization phase, where the first pair is chosen is critical: finding or not an isomorphism depends on the "lucky" choice of a starting pair which is *inside* it.

In the original paper [48] some approaches are suggested for the choice of the initial points, but none of them seems satisfactory, especially for large graphs.

4.2.2.3 Specialization to our case

In our case the definition of isomorphic data flow graphs requires some more thinking. First of all, we must take into account the topological properties (ordering of the inputs) that we added in §4.2.1 and that do not enter in the general Definition 4.2.

Secondly, we are not interested *only* in topology, since the nodes are not all identical, but have a property that is crucial to the execution: the action they perform, $\alpha(o)$.

We can hence restate the general definition as follows:

Definition 4.7 (Isomorphic DFGs) *Two data flow graphs* $G_1 = \langle O_1, P_1, A, \alpha_1, \mu_1 \rangle$ *and* $G_2 = \langle O_2, P_2, A, \alpha_2, \mu_2 \rangle$ *are* isomorphic*, and we write* $G_1 \doteq G_2$*, if and only if* $\langle O_1, P_1 \rangle$ *and* $\langle O_2, P_2 \rangle$ *are isomorphic with bijection* $p : O_1 \mapsto O_2$ *and if*

$$\forall o \in O_1, \alpha(o) = \alpha(p(o)), \qquad \forall o \to o' \in P_1, \mu(o \to o') = \mu(p(o) \to p(o')),$$

that is, the bijection is compatible with the actions and the ordering of the inputs.

In order to tackle the problems in the algorithm suggested in the previous section, we can exploit the fact that our specification graphs have much more structure than a normal graph. In fact not only they are colored, but they also have numbered inputs.

For instance we have to extend the weight function to take into account, for every edge of the graph:

1. the color of the node they are leaving;
2. the color of the node they are entering;
3. the number of the input they are providing.

Now we have to find a possibly small number of suitable pairs to use as initialization for different runs of the algorithm.

One approach is as follows: suppose we consider the edge $e = o_1 \to o_2 \in P$ that serves as input n for $o_2 \in O$. We create a set $I^{\alpha(o_1)}_{\alpha(o_2),n}$ and store e in it. Iterating this for all the edges takes a linear time in the number of the edges, and results in sets composed of edge–induced isomorphic subgraphs of G.

All the elements of these sets are pairwise suitable to be used as initial points for the algorithm.

After having run the algorithm for several initial pairs we will be left with a collection of subgraphs S partitioned into equivalence classes $[S]$ with respect to the equivalence relation $\cong$. All of them might be suitable for being implemented in hardware.

4.3 Pruning

We set out now to choose which of the sets S previously created can be disregarded, in order to simplify the choice of $\widetilde{\mathcal{O}}$ and subsequent the scheduling process.

There are three straightforward answers:

1. eliminate those that are "small";

2. eliminate those that are used only a "small" number of times;

3. eliminate those that do not comply with Definition 4.1.

The first "small", which is to be measured CLB–wise, has two sides.

On one hand one usually doesn't want to devote an entire reconfigurable unit to the execution of just a couple of instructions, so we will probably require that $|S|$ is not too small (say, > 5, for instance).

On the other hand one must consider that the reconfigurable units come in fixed sizes, which are multiple of a certain number of CLBs. Hence, one might want to map on the reconfigurable units sets S that fit in the area with little waste.

The former criterion is very easy to check, while the latter requires some fine–tuned metrics to evaluate the actual area needed by a set S. Such metrics have been developed in Microlab and are available for this purpose.

The second "small" refers to the size of the equivalence class of every set S. The rationale is that one certainly prefers to map in hardware an execution unit that is needed 100 times rather than one that's needed only 10 times.

This poses an issue, though: it is possible that the algorithm presented in the last section returns two equivalence classes, say $[S_1]$ and $[S_2]$, such that there are only three instances of S_1 but there are ten instances of $[S_2]$. Also, S_1 is preferable over S_2 in terms of physical size.

However, it turns out that S_1 is a subgraph of S_2. In this case it might be preferable to dismantle the equivalence class $[S_1]$ made up of the larger elements in favor of the more numerous class $[S_2]$.

To this end it could be necessary to keep track of internal relationships between the different equivalence classes.

The last criterion has already been discussed at length in the initial part of this chapter, so we won't dwell longer on it.

CHAPTER 5

ANALYSIS AND RESULTS

In this first and initial work the focus was put onto the partitioning of the tasks rather than on their scheduling.

For this reason the experimental results are limited to the algorithm used for this phase, *i.e.*, the ISOMORPHIC SUBGRAPHS problem.

Two problems arise in the evaluation of the algorithm:

- on one hand, the task it solves has not been been studied extensively. For this reasons, there is a lack of results to compare to;
- on the other hand, the few results that are available are relative to instances of the problem that have much less constraints than the ones found in our setting. In particular, in [48] the authors deal with isomorphism among general undirected graphs. In our case, however, we deal with data flow graphs, which are directed acyclic graphs with colored nodes and numbered inputs.

Unfortunately, for these reasons, a direct numerical comparison makes little sense.

We can however point out the major improvements with respect to the original algorithm proposed in [48].

5.1 Choice of initial points

As pointed out in §4.2.2.3, one of the mayor improvements is given by the selection of the starting point with which the algorithm is initialized.

In particular, we choose an initial set of isomorphic nodes made up by two nodes performing the same action.

This is actually done by searching through all the edges and cataloging them with respect to the action performed by their starting and ending node, building the sets $I^{a_1}_{a_2,n}$ (see §4.2.2.3).

The algorithm then sorts the sets according to their cardinality $|I^{a_1}_{a_2,n}|$, and iterates the algorithm for pairs taken from each of the sets, or at least for the more numerous ones. The rationale is that it is more important to analyze possible isomorphic subgraphs that have a chance of being repeated more often in the original specification (and hence can be reused more times).

This initialization process takes $\mathcal{O}(|E|)$ for the browsing of the edges and the construction of the sets $I^{a_1}_{a_2,n}$. Then, the algorithm is executed a number of times N_i equal to the number of sets $I^{a_1}_{a_2,n}$ with a cardinality bigger than a predefined threshold.

It is not easy to estimate N_i for a general graph: besides depending on the choice of the threshold and of the number of edges E, it is also a function of the number of different kinds of actions available in the specification.

If the specification is composed of O operations computing the possible actions A, there will be $2|A|^2$ sets $I^{a_1}_{a_2,n}$. As A gets larger, the number of edges in each set will be on average smaller, and hence the number of sets with less elements than the threshold (that is, the sets that will be used as starting point for the algorithm) will decrease.

On the other hand, in [48] the authors suggest to run the algorithm for all the pairs having a weight that is at least 90% of the weight of the best pair of nodes[1].

This requires $\mathcal{O}(|O|^2)$ iterations to explore all the possible pairs. On top of this, in every iteration one has to compute the weight, which in the suggested paper can be an expensive procedure.

For these reasons, the initialization process is substantially faster in our approach than in the original one.

5.2 Weighting function

In [48] the authors use a weighting function to give a "value" to each edge (in order to proceed with the matching algorithm) and also to check if a pair of nodes is a good candidate to be used as a starting pair §5.1.

They suggest to use a linear combination of different indexes w_i, $i = 1, \dots, 6$, each measuring the "likeness" of two nodes. For instance:

- w_2 is the difference of the degree od the nodes;
- w_3 is the number of common neighbours;
- w_5 is the graph–theoretical distance between the nodes.

Some of these parameters are easy to compute (like w_2 and w_3). Others require some more work, like w_5 or the more refined ones, like w_6, that checks how many isomorphic edges are present in the

---

[1] See §5.2 for a discussion about the weighting function.

graphs defined by $H_1 \cup o_1$ and $H_2 \cup o_2$ (where the H's are the isomorphic subgraphs computed at the point of evaluation of w_6).

The authors note that the best weights with respect to the size of the resulting isomorphic subgraphs are exactly those that require the biggest computation time — in particular w_6. Using faster weighting functions results in an average loss in the size of the identified subgraphs of about 10–30$\%$.

On the other hand, in our algorithm the weighting function is based on two conditions on the pair of nodes (o_1, o_2):

- penalizing the difference of the action performed by the nodes of the pair ($a_{o_1} \neq a_{o_2}$;
- for each input edge entering the node o_1 in its input port i and leaving from a node performing action a, we search for a similar edge entering o_2. If it not found, penalize the pair.

These weighting functions use only *local* structural properties and are very easy to compute if a suitable data structure is chosen for the storage of the graph.

Actually, in the current implementation they are written in a somehow inefficient way, but they can still compare in terms of execution time with the quick w's in [48].

As for the quality of the solution obtained through the weights, as we said before it is difficult to compare it with the results of the cited paper, since the problem instances are very different.

5.3 Matching algorithm

In [48] the matching problem is solved using an unoptimized version of the Hungarian algorithm [50, 51, 52].

In our implementation, instead, we use the faster LAPJV algorithm by Jonker and Volgenant [53], which is freely available as C++, Pascal and FORTRAN code [54]

5.4 Tests

Given the lack of general test suites for isomorphic subgraph problems it was necessary to create our own test suite.

The algorithm was tested against some small data flow graphs manually created (50–100 nodes), and also against directed acyclic graphs produced by Panda, a high–level tool being developed in MicroLab, which is used for code analysis, optimization and synthesis. It can take input specifications in several languages, including C, C++ and SystemC [55].

The program

CHAPTER 6

CONCLUSIONS

The aim of this work was to suggest an approach to the problem of scheduling a specification of arbitrary size on a reconfigurable architecture with a limited amount of available resources, minimizing the overall latency of the resulting system.

We have analyzed the limitations of the other approaches available in the literature — mainly due to their restricting to time–partitioning or to partial reconfiguration with little freedom in the placing of the cores.

In order to avoid these limitations without having an explosion in the problem complexity we proposed a model of the architecture and, based on that, also a model of the problem itself, suggesting to split it in two phases:

1. partitioning of the specification into sets of operations to be scheduled together as a whole;
2. scheduling of these sets.

We then suggested an algorithm for the identification of these sets, based on a little–known NP–complete problem (ISOMORPHIC SUBGRAPHS), that we adapted from the only heuristic available in the literature, and analyzed it.

There are several improvements to work on now.

First of all, we haven't dealt at all with the problem of the scheduling itself. This will require further investigation, although the formalism we have employed to model the overall problem already suggests somehow an approach.

In fact, the idea of having several implementations of the same set of tasks hints at our hope to tackle the problem as a kind of covering problem.

Moreover, having a complete flow from the data flow graph to the scheduled specification will also permit to judge the quality of the partitioning with the only metric that is actually useful, *i.e.*, how small the latency of the overall system (obtained with *that* partitioning) is.

Another approach to the scheduling problem could be that of studying the NP–complete that *geometrically* resembles it most, *i.e.*, the three–dimensional packing. In particular, there are some recent results that might give some hope to this kind of attach to the problem [56, 57, 58], even if in general 3D–packing is considered very difficult.

Another useful task will be that of developing an exact algorithm for finding the maximum isomorphic subgraphs in directed acyclic graphs, and using it to find the optimum solutions for a fixed set of graphs.

These graphs would serve as a reference for the evaluation of the heuristics, and could eventually be used as a standard reference for the different scheduling algorithms — in fact the comparison of the results with the current literature in §2 will be difficult, since every paper pubishes results based on different specifications. Also, when the program to be scheduled is the same, the data flow graph is often different because obtained with different tools, each acquiring different degrees of optimization.

Another improvement would be that of finding a new heuristic for the ISOMORPHIC SUBGRAPHS problem and defining clearly which are the parameters for the evaluation of its quality.

We still hope that, somehow, the polynomial result given in the original paper for the isomorphic subtree problem will eventually turn out to give some insight into some instances of the general problem via some kind of reduction.

APPENDIX

ISOMORPHISM FROM DATA FLOW GRAPHS TO TREES

Since, as we said in §4.2.2.2, a very good (linear time) algorithm is available for trees, one is interested in possible representations of a general graph that make use of trees.

A.1 Directed and undirected graphs

Since many results and algorithms in Graph Theory are given in terms of undirected graphs, while our specification is given in terms of an enriched directed graph, we will often need to go back and forth from directed and undirected graphs. Let us ease the work from the notational point of view, hence, before continuing.

Definition A.1 *Given a directed graph $G = \langle O, P \subseteq O \times O \rangle$, define $G^{\leftrightarrow}$ as*

$$G^{\leftrightarrow} := \langle O, P^{\leftrightarrow} \rangle, \quad P^{\leftrightarrow} := \{\{o, o'\} \mid o \to o' \in P\}.$$

Hence, the operator $\cdot^{\leftrightarrow}$ takes a directed graph to an undirected one.

Note A.1 If G is a colored graph, also $G^{\leftrightarrow}$ can be regarded as colored. ♣

Now we are faced with the opposite problem, that is, going from an undirected graph S to a directed one. This is of course impossible in absence of further data, since an undirected graph contains less information than a directed one. However, we will mostly be interested in the case where S is a subgraph of $G^{\leftrightarrow}$.

APPENDIX (Continued)

Definition A.2 *Let $G = \langle O, P \rangle$ be a directed graph and $S = \langle O', P' \rangle$ a subgraph of $G^{\leftrightarrow}$. We say that* S with the orientation induced by G*, or the* G–oriented version of S *is*

$$S^{\leftarrow G} := \langle O', \{o_1 \to o_2 \in P \mid \{o_1, o_2\} \in P'\} \rangle.$$

Please notice that $\cdot^{\leftarrow \cdot}$ is the inverse of $\cdot^{\leftrightarrow}$: in fact $(G^{\leftrightarrow})^{\leftarrow G} = G$, so that is one keeps the orientation information from G the operation doesn't result in a loss of data.

Given this definitions, it is now possible to proceed with the algorithm.

A.2 The idea

It turns out that for any given undirected connected graph G it is always possible to find a subgraph T of G that is a *spanning tree*, *i.e.*, T is a tree and all the nodes of G are connected by the edges in T [59, pg. 12].

Corollary A.1 *Given an undirected graph $G = \langle O, P \rangle$ one can decompose it in two graphs $T = \langle O, P_T \rangle$ and $\widetilde{T} = \langle O, P_{\widetilde{T}} \rangle$, where T is a spanning tree, $\widetilde{T}$ is a subgraph of G called a* co–tree*, and $P_T \cap P_{\widetilde{T}} = \varnothing$.*

Proof. It is enough to take $P_{\widetilde{T}} := P \setminus P_T$. □

We will write this decomposition as $G = T \oplus \widetilde{T}$.

Definition A.3 *Given a subgraph S of a decomposed graph $G = T \oplus \widetilde{T}$, its* restriction to the tree S_T *is the subgraph of S induced by the edges of T,* i.e.*, $S_T := S|^{P_T}$. The* restriction to the co–tree $H_{\widetilde{T}}$ *is the subgraph of H induced by the edges of $\widetilde{T}$,* i.e.*, $S_{\widetilde{T}} := S|^{P_{\widetilde{T}}}$.*

APPENDIX (Continued)

Observe now that obviously the property of being isomorphic is *not* conserved by the operations $\cdot_T$ and $\cdot_{\widetilde{T}}$.

Example A.1 (Isomorphism is not conserved under tree restriction)

Consider the graph G in Figure 26. The two encircled subgraphs S and S' are of course isomorphic, even in the stricter sense: $S \doteq S'$.

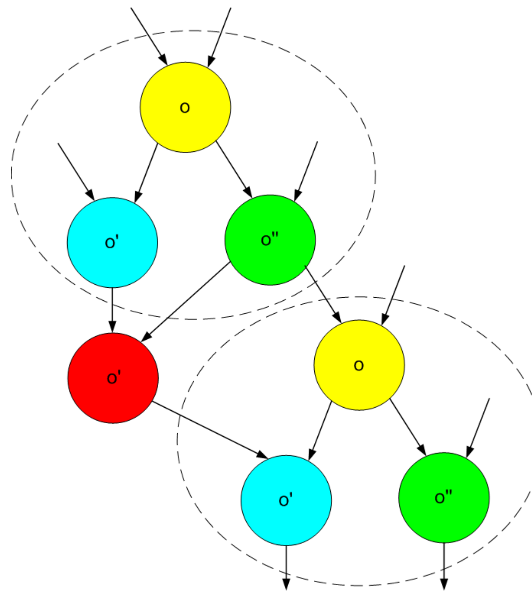

Figure 26. A specification G (colors represent the actions $\alpha(o)$)

Now let's choose a spanning tree T in the graph $G^{\leftrightarrow}$. Then compute $G_T := T^{\leftarrow G}$, and partition it into node–induced subgraphs S_T and S'_T, as shown in Figure 27. It is apparent that not only it is not true anymore that $S_T \doteq S'_T$, but even $S_T \cong S'_T$ doesn't hold anymore.

APPENDIX (Continued)

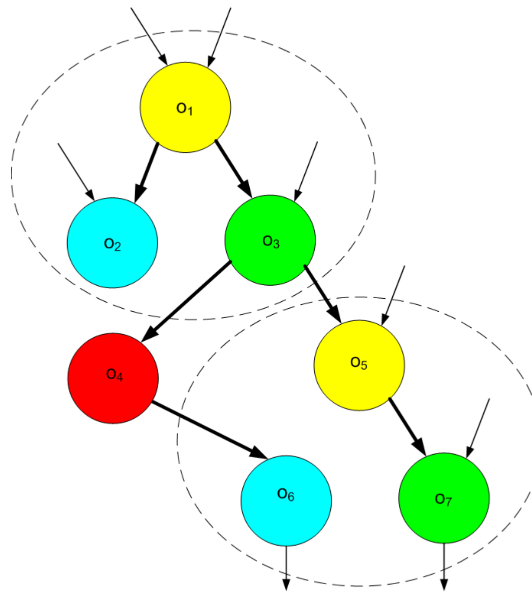

Figure 27. The restriction to a tree of the graph in Figure 26

In general, this is the case every time restricting to a tree T removes an edge from S that it doesn't remove from S'. $\Diamond$

However, one can notice that in the previous example a subgraph of S is indeed isomorphic to a subgraph of S'. In particular,

$$S|_{\{o_1,o_3\}} \doteq S|_{\{o_5,o_7\}}.$$

One might wonder if something like this always happens. It is in fact the case, although unfortunately in a not very useful way.

Example A.2 (Not always the subgraphs of isomorphic subgraphs are isomorphic)

Consider the specification in Figure 28, where the edges of a tree T of $G^{\leftrightarrow}$ are stressed.

APPENDIX (Continued)

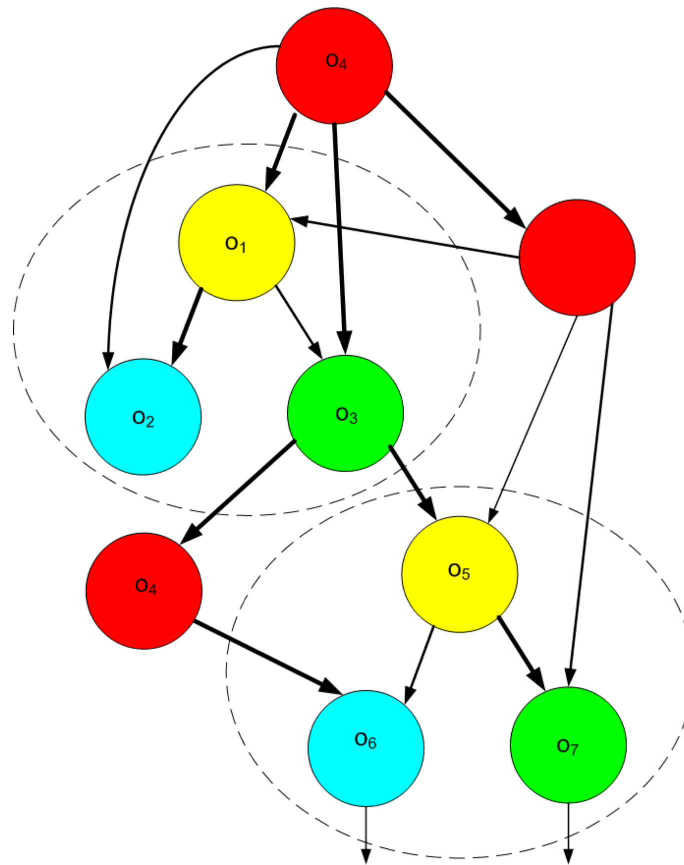

Figure 28. Not even a subgraph of the originally isomorphic subgraphs are isomorphic

It is apparent that the two subgraphs S and S' of G are isomorphic. However, the only subgraphs of S_T and S'_T that are isomorphic are those made up by only one node (o_1 with o_5, o_2 with o_6, o_3 with o_7).

Even if present, this isomorphism is not very useful because it can't be easily distinguished from the case where there are no other nodes that extend the isomorphism. $\Diamond$

As it is customary, when a property is not valid in the general case, one resorts to trying and proving it for more specific instances. In our case, since the aim is that of finding good candidates for the sets $S \in \mathcal{O}$, we can restrict our attention to subgraphs S that satisfy the properties of Definition 4.1.

In this case one can prove the following

Proposition A.1 *Let G be a data flow graph having two disjoint isomorphic subgraphs S and S' satisfying Definition 4.1. Let T be a spanning tree of $G^{\leftrightarrow}$ and $G_T := T^{\leftarrow G}$.*

Then $S_T^{\leftarrow G}$ and $S_T'^{\leftarrow G}$ have non–trivial isomorphic subsets.

For this reason, one can look for significant isomorphisms (that can be conveniently mapped to tasks meant to be iterated several times) not only in the complete graph G, but also in a spanning tree T of $G^{\leftrightarrow}$.

Of course, one doesn't get the full isomorphism (see Example A.1), but rather a *subset* of it. However, this is good enough, since it suggests reliable neighborhoods where to look for complete isomorphisms with a computationally heavier algorithm.

At this point it is time to adapt the algorithm proposed in [47], which was thought for uncolored trees with no numbering of the inputs, to our case.

The algorithm is in fact a reduction of ISOMORPHIC SUBGRAPHS to the problem of finding *longest identical well–nested non–overlapping substrings* of a given string w, for which an algorithm is known to be $O(|w|)$.

We saw in §4.2.2.2 that an alphabet $\mathcal{A} = \{(,)\}$ was defined, and that every node o of the tree is represented by a pair of parenthesis. Inside the pair representing o there is the (ordered) list of parentheses pairs representing the children nodes of o, and so on until the leaves are reached, in a depth–first search.

In order to take into account the coloring of the nodes it is enough to extend the alphabet, adding a different pair of "parentheses" for each color. For instance, the tree in Figure 27 has $A = \{1, 2, 3, 4\}$, so that we can write it as a string as w_T =12234224122131 over $\mathcal{A} = \{1, 2, 3, 4\}$. In this fashion, we still obtain a well–nested string, and can hence apply the algorithm.

We then have to consider the problem of ordering the inputs.

An approach could be that of exploiting the fact that the algorithm itself works for *ordered* trees, so that the children nodes of a node are enumerated, and hence distinguished. The idea would be to compute a spanning tree of the specification graph G starting not from the start node o_S, but from the end node o_E. In this way the children nodes of a node o would be those giving the *inputs* to o, and hence would be sorted. This would be enough to solve the problem, since for every kind of operation $\alpha(o)$ we could associate a particular operand to the first child node, another specified operand to the second one, and so on.

However this approach doesn't work. In fact, when computing the spanning tree some of the input edges of some nodes may be dropped (in order not to form cycles). If this happens the *order* of the other

APPENDIX (Continued)

nodes is maintained, but on the other hand their *position* is not. This results in misinterpretation of the input.

The proposed approach then works as follows. Suppose any node can have at most n inputs (usually $n = 2$). Extend the alphabet $\mathcal{A}$ again with n symbols $i_1, i_2, \ldots, i_n$. The build the spanning tree of the specification graph taking the end node o_E as root. However, the string associated to the child node giving the k–th input (*e.g.*, ()) of its parent node (*e.g.*, []) is not only () anymore, but rather i_k().

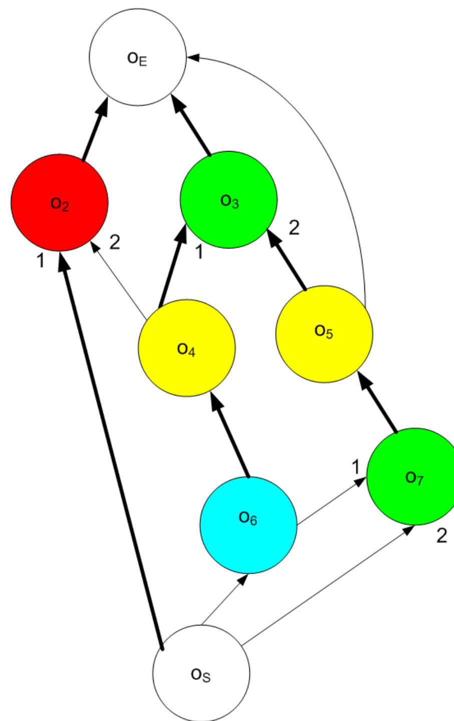

Figure 29. The tree generated by a specification

APPENDIX (Continued)

For example, let us consider the specification G in Figure 29. The specification is given by the actual directions of the arrows, and when there is more than one input the number of the input itself is written next to the arrow.

Let us build the complete string of the tree identified by the thicker arrows. The alphabet contains four colors (red, cyan, green, yellow), plus two colors for the start and end nodes (let's call them E and S). Moreover, since we have $n = 2$, we can choose to add the symbols $+$ and $-$ for the ordering of the inputs, obtaining $\mathcal{A} = \{1, 2, 3, 4, S, E+, -\}$. Then one gets ER+SSRG+YCCY-YGGYGE, which is still a well–nested string to which the algorithm can be applied.

Notice how we were able to explicitly keep the information that, *e.g.*, it is the *first* input of o_2 that is connected to a node of type S, and not the second one.

CITED LITERATURE

Bibliography

[1] *13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), 17-20 April 2003, Napa, CA, Proceedings*. IEEE Computer Society, 2005.

[2] Reiner W. Hartenstein and Manfred Glesner, editors. *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers, 6th International Workshop on Field-Programmable Logic, FPL '96, Darmstadt, Germany, September 23-25, 1996, Proceedings*, volume 1142 of *Lecture Notes in Computer Science*. Springer, 1996.

[3] L. Miguel Silveira, Srinivas Devadas, and Ricardo Augusto da Luz Reis, editors. *VLSI: Systems on a Chip, IFIP TC10/WG10.5 Tenth International Conference on Very Large Scale Integration (VLSI '99), December 1-4, 1999, Lisbon, Portugal*, volume 162 of *IFIP Conference Proceedings*. Kluwer, 2000.

[4] *16th International Conference on VLSI Design (VLSI Design 2003), 4-8 January 2003, New Delhi, India*. IEEE Computer Society, 2003.

[5] Jan Kratochvíl, editor. *Graph Drawing, 7th International Symposium, GD'99, Stirín Castle, Czech Republic, September 1999, Proceedings*, volume 1731 of *Lecture Notes in Computer Science*. Springer, 1999.

[6] Stephen G. Kobourov and Michael T. Goodrich, editors. *Graph Drawing, 10th International Symposium, GD 2002, Irvine, CA, USA, August 26-28, 2002, Revised Papers*, volume 2528 of *Lecture Notes in Computer Science*. Springer, 2002.

[7] Xiao ping Ling and Hideharu Amano. Performance evaluation of wasmii: a data driven computer on a virtual hardware. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE*, volume 694 of *Lecture Notes in Computer Science*, pages 610–621. Springer, 1993.

[8] William Fornaciari and Vincenzo Piuri. Virtual fpgas: Some steps behind the physical barriers. In *IPPS/SPDP Workshops*, pages 7–12, 1998.

[9] Karthikeya M. Gajjala Purna and Dinesh Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Trans. Comput.*, 48(6):579–590, 1999.

[10] Rhett D. Hudson, David Lehn, Jason Hess, James Atwell, David Moye, Ken Shiring, and Peter Athanas. Spatio-temporal partitioning of computational structures onto configurable computing machines. In John Schewel, editor, *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pages 62–71, Bellingham, WA, November 1998. SPIE – The International Society for Optical Engineering.

[11] Iyad Ouaiss, Sriram Govindarajan, Vinoo Srinivasan, Meenakshi Kaul, and Ranga Vemuri. An integrated partitioning and synthesis system for dynamically reconfigurable multi-fpga architectures. In *IPPS/SPDP Workshops*, pages 31–36, 1998.

[12] Meenakshi Kaul and Ranga Vemuri. Temporal partitioning combined with design space exploration for latency minimization of run-time reconfigured designs. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 43. ACM Press, 1999.

APPENDIX (Continued)

[13] Awartika Pandey and Randga Vemuri. Combined temporal partitioning and scheduling for reconfigurable architectures. In John Schewel, Peter M. Athanas, Steven A. Guccione, Stefan Ludwig, and John T. McHenry, editors, *Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844*, pages 93–103, Bellingham, WA, September 1999. SPIE – The International Society for Optical Engineering.

[14] M. Kaul and R. Vemuri. Optimal temporal partitioning and synthesis for reconfigurable architectures. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 389–397. IEEE Computer Society, 1998.

[15] Meenakshi Kaul, Ranga Vemuri, Sriram Govindarqjan, and Iyad Ouaiss. An automated temporal partitioning and loop fission approach for fpga based reconfigurable synthesis of dsp applications. In *DAC '99: Proceedings of the 36th Annual Conference on Design Automation (DAC'99)*, pages 616–622. IEEE Computer Society, 1999.

[16] Satish Ganesan and Ranga Vemuri. An integrated temporal partioning and partial reconfiguration technique for design latency improvement. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 320–325. ACM Press, 2000.

[17] Manish Handa, Rajesh Radhakrishnan, Madhubanti Mukherjee, and Ranga Vemuri. A fast macro based compilation methodology for partially reconfigurable fpga designs. In VLSID 2003 [4], pages 91–.

[18] Joco M. P. Cardoso and Horacio C. Neto. Macro-based hardware compilation of java(tm) bytecodes into a dynamic reconfigurable computing system. In *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 2. IEEE Computer Society, 1999.

[19] João M. P. Cardoso and Horácio C. Neto. Fast hardware compilation of behaviors into an fpga-based dynamic reconfigurable computing system. In Vladimir C. Alves, Marcelo Lubaszewski, and Ivan S. Silva, editors, *Proceedings of the XII Symposium on Integrated Circuits and Systems Design (SBCCI'99)*, pages 150–153, Natal-RN, Brazil, Sept. 29-Oct. 2 1999. IEEE Computer Society Press.

[20] Joco M. P. Cardoso and Horacio C. Neto. Towards an automatic path from java(tm) bytecodes to hardware through high-level synthesis. In *Proceedings of the 5th IEEE International Conference on Electronics, Circuits and Systems (ICECS-98)*, pages 85–88, September 7-10 1998.

[21] João M. P. Cardoso and Horácio C. Neto. An enhanced static-list scheduling algorithm for temporal partitioning onto rpus. In Silveira et al. [3], pages 485–496.

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

[23] João M. P. Cardoso and Horácio C. Neto. Compilation for fpga-based reconfigurable hardware. *IEEE Design & Test of Computers*, 20(2):65–75, 2003.

[24] João M. P. Cardoso. Loop dissevering: A technique for temporally partitioning loops in dynamically reconfigurable computing platforms. In *IPDPS '03: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, page 181.2. IEEE Computer Society, 2003.

[25] Rafael Maestre, Fadi J. Kurdahi, Milagros Fernandez, Roman Hermida, Nader Bagherzadeh, and Hartej Singh. A framework for reconfigurable computing: task scheduling and context management. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(6):858–873, 2001.

[26] R. Maestre, M. Fernandez, R. Hermida, and N. Bagherzadeh. A framework for scheduling and context allocation in reconfigurable computing. In *ISSS '99: Proceedings of the 12th International Symposium on System Synthesis*, page 134. IEEE Computer Society, 1999.

[27] Rafael Maestre, Fadi J. Kurdahi, Milagros Fernández, Roman Hermida, Nader Bagherzadeh, and Hartej Singh. A formal approach to context scheduling for multicontext reconfigurable architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):173–185, 2001.

[28] Marcos Sanchez-Elez, Milagros Fernandez, Roman Hermida, Rafael Maestre, Fadi Kurdahi, and Nader Bagherzadeh. A data scheduler for multi-context reconfigurable architectures. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 177–182. ACM Press, 2001.

[29] Marcos Sanchez-Elez, M. Férnandez, Rafael Maestre, Fadi J. Kurdahi, Román Hermida, and Nader Bagherzadeh. A complete data scheduler for multi-context reconfigurable architectures. In *DATE*, pages 547–552. IEEE Computer Society, 2002.

[30] M. Vasilko and D. Ait-Boudaoud. Scheduling for dynamically reconfigurable fpgas. In *Proceeding of International Workshop on Logic and Architecture Synthesis, IFIP TC10 WG10.5L*, pages 328–336, Grenoble, France, Dec. 18-19 1995. SPIE – The International Society for Optical Engineering.

[31] Milan Vasilko and Djamel Ait-Boudaoud. Architectural synthesis techniques for dynamically reconfigurable logic. In Hartenstein and Glesner [2], pages 290–296.

[32] Milan Vasilko and Graham Benyon-Tinker. Automatic temporal floorplanning with guaranteed solution feasibility. In Reiner W. Hartenstein and Herbert Grünbacher, editors, *FPL*, volume 1896 of *Lecture Notes in Computer Science*, pages 656–664. Springer, 2000.

[33] Milan Vasilko. Dynasty: A temporal floorplanning based cad framework for dynamically reconfigurable logic systems. In *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pages 124–133. Springer-Verlag, 1999.

[34] http://www.xilinx.com/products/jbits/.

[35] S. A. Guccione, D. Levi, and P. Sundararajan. Jbits: A java–based interface for reconfigurable computing. In *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*, 1999.

[36] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, editors. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.

[37] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, 1993.

[38] John R. Hauser and John Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. In *FCCM*, pages 12–21. IEEE Computer Society, 1997.

[39] Fabrizio Ferrandi, Marco D. Santambrogio, and Donatella Sciuto. The Caronte Flow: A Methodology for Partial Dynamic Reconfiguration of Embedded System. In *The 12th Reconfigurable Architectures Workshop (RAW 2005)*, 2005.

[40] Alberto Donato, Fabrizio Ferrandi, Massimo Redaelli, Marco Santambrogio, and Donatella Sciuto. Caronte: a complete methodology for the implementatio of partially dynamically self–reconfiguring systems on fpga platforms. In FCCM 2005 [1].

[41] Marco D. Santambrogio. A methodology for dynamic reconfigurability in embedded system design. Master's thesis, Politecnico di Milano, 2004. http://www.micro.elet.polimi.it/people/santa.

[42] Marco D. Santambrogio. Dynamic reconfigurability in embedded system design — a model for the dynamic reconfiguration. Master's thesis, University of Illinois at Chicago, 2004. http://www.micro.elet.polimi.it/people/santa.

[43] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison–Wesley, 1988.

[44] http://www.nist.gov/dads/html/graphisomrph.html.

[45] http://www.nist.gov/dads/html/subgraphiso.html.

[46] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[47] Sabine Bachl. Isomorphic subgraphs. In Kratochvíl [5], pages 286–296.

[48] Sabine Bachl and Franz-Josef Brandenburg. Computing and drawing isomorphic subgraphs. In Kobourov and Goodrich [6], pages 74–85.

[49] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[50] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quaterly*, 2:83–97, 1955.

[51] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice Hall, 1982. Papadimitriou.

[52] http://www.nist.gov/dads/html/munkresassignment.html.

[53] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.

[54] http://www.magiclogic.com/assignment.html.

[55] http://www.systemc.org.

[56] Sándor P. Fekete and Jörg Schepers. A combinatorial characterization of higher-dimensional orthogonal packing. *Math. Oper. Res.*, 29(2):353–368, 2004.

[57] Sándor P. Fekete and Jörg Schepers. A general framework for bounds for higher-dimensional orthogonal packing problems. *CoRR*, cs.DS/0402044, 2004.

[58] Sándor P. Fekete, Ekkehard Köhler, and Jürgen Teich. Higher-dimensional packing with order constraints. *CoRR*, cs.DS/0308006, 2003.

[59] Reinhard Diestel. *Graph Theory*. Springer, New York, 2000.