From Design Space Exploration to Code Generation

a constraint satisfaction approach for
the architectural synthesis of digital VLSI circuits

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op woensdag 3 april 1996 om 16.00 uur

door

Adwin H. Timmer

geboren te Apeldoorn

Dit proefschrift is goedgekeurd door de promotoren:

prof.Dr.–Ing. J.A.G. Jess
en
prof.dr. E.H.L. Aarts

en door de copromotor:

dr.ir. M.R.C.M. Berkelaar

CIP–gegevens Koninklijke Bibliotheek, Den Haag

Timmer, Adwin H.

From Design Space Exploration to Code Generation: a constraint satisfaction approach for the architectural synthesis of digital VLSI circuits

Proefschrift Technische Universiteit Eindhoven. – Met lit. opg. – Met samenvatting in het Nederlands.

ISBN 90–74445–24–1

Trefw.: architectural synthesis, code generation, constraint satisfaction.

voor mijn vader

Je m'appelle Zangra
Et je suis Lieutenant
Au fort de Belonzio
Qui domine la plaine
D'où l'ennemi viendra
Qui me fera héros
....

Jacques Brel (Zangra, 1962)

Summary

Ongoing technological developments result in the design and fabrication of increasingly larger and more complex electronic systems compared with the systems that are currently in use. For a short design cycle and time to market, good design methodologies for the computer–aided design (CAD) of these systems are very important. The need for productivity has initiated the development of *architectural* (or *high–level*) *synthesis* methods. This thesis deals with such methods, which have been developed at the Design Automation Section of the Eindhoven University of Technology.

Architectural synthesis tools take an algorithmic description with goals and constraints as their input, and produce a register transfer level description of an IC architecture as their output. A major part of such a description is the datapath, which consists of interconnected basic building blocks like adders, multipliers, memories etc. Three main tasks can be distinguished when synthesizing a datapath: resource selection, scheduling and binding. The organization of these interdependent subtasks can influence the quality of the result substantially, and might depend on the application domain as well. The three individual tasks are in general not solvable in polynomial time, thus leading to numerous heuristical solution methods.

Most architectural synthesis methods can only take a limit number of different kinds of constraints into account. Such constraints can either be part of the initial problem description, or can be formulated during the design flow. Furthermore, they can be related to timing, resources, specific bus architectures, etc. So, most methods concentrate on minimizing one or a few objective functions, e.g., the chip area requirements. The synthesis result is then checked against other constraints that were not yet taken into account. However, this often leads to unsatisfactory results, because the synthesis result often does not comply with the other constraints, especially when they are tight.

For that reason, *constraint satisfaction* techniques are emphasized in this thesis. In our approach, a designer can modify the initial problem description interactively by adding new constraints or by enforcing certain design decisions. In this way, a trade–off can be made between the quality of a solution and the run time efficiency.

The central theme in this thesis is a new algorithm based on a graph formulation, namely the so called 'bipartite schedule graphs' (BSGs). The algorithm advances the ability of synthesis systems to deal with combinations of timing and resource constraints, rather than being hampered by them. The BSGs are applied in two different ways.

First, the BSGs prune the search space of schedulers, which is called *domain reduction*. Classical operation execution intervals are usually obtained by a critical path analysis under the assumption of unlimited resources. The BSGs, however, take the timing, precedence and resource constraints into account, which leads to a reduction of the execution intervals. This results in a reduction of the search area as well, but does not affect the solution space at all.

Secondly, the BSGs are used to identify the 'bottlenecks' for a scheduler. Those bottlenecks typically fool heuristic methods (because they are not recognized by such methods), and they are a source of wasted search effort for exact schedulers. The BSGs identify the bottlenecks and consequently make an efficient *traversal of the search space* possible, because the bottlenecks are solved first.

The domain reduction and scheduling methods not only prove their value in the field of architectural synthesis, but also in other situations in which the constraints are very tight, such as *(retargetable) code generation*. A solution strategy is proposed in which code generation is the last step in the architectural synthesis flow. The complete datapath with timing constraints is fixed in the case of code generation, so the remaining task is to find a correct schedule and binding scheme. Industrial examples show that the methods described in this thesis lead to an efficient code generation approach, thus validating the proposed methodology.

Last but not least, the methods can be applied for lower bound analyses. These analyses can help CAD tools, they can help to make interactive design decisions, and they can help to impose extra constraints during the synthesis flow. Furthermore, such analyses give a good insight in the design space, so they can help to measure the quality of a design as well. In this thesis, lower bound analyses related to functional area and cycle budget are presented. It is shown that the analyses are more accurate than other lower bound analyses that run in polynomial time. Furthermore, the only existing method capable of performing an efficient lower bound estimation for non–trivial module libraries is presented as well.

Samenvatting

Door de voortschrijdende technologische ontwikkelingen kunnen steeds ingewikkeldere elektronische systemen ontworpen en gebouwd worden. Goede, computerondersteunde ontwerpmethoden zijn daarbij onontbeerlijk. Zij moeten ervoor zorgen dat de ontwerptijd kort is, zodat ook de tijd tussen de specificatie en de marktintroductie van een systeem kort is. Deze hoge productiviteitseis heeft geleid tot de ontwikkeling van *architectuur–* (of *hoog–niveau*) *synthese–*methoden. Dit proefschrift beschrijft nieuwe architectuur–synthese–methoden, die ontwikkeld zijn in de vakgroep Ontwerpkunde voor Elektronische Systemen van de Technische Universiteit Eindhoven.

Het belangrijkste onderdeel van de architectuur–synthese is de vorming van het datapad. Het datapad is de basis van de IC architectuur op register–niveau, het eindresultaat van de synthese. Het bestaat uit bouwblokken, zoals optellers, vermenigvuldigers en geheugenelementen, die onderling verbonden zijn. Invoer voor de synthese van het datapad is een algoritmische beschrijving waarvoor beperkingen en doelstellingen zijn geformuleerd. Bij de synthese van het datapad onderscheidt men drie taken: de selectie van de bouwblokken, de tijdstoewijzing van operaties en de afbeelding van de operaties op de bouwblokken. De volgorde en invulling van deze taken kunnen een grote invloed hebben op het uiteindelijke resultaat, en kunnen tevens afhangen van het toepassingsgebied. Deze drie taken zijn in het algemeen niet in polynomiale tijd oplosbaar. Daarom is er voor het oplossen van deze taken veel aandacht voor benaderingsmethoden.

De gangbare architectuur–synthese–methoden kunnen slechts in beperkte mate rekening houden met beperkingen die deel uitmaken van de invoer of die tijdens het ontwerptraject geformuleerd worden. Deze beperkingen kunnen te maken hebben met tijdsaspecten, bouwblokken, specifieke bus architecturen of andere aspecten van het ontwerp. De gangbare methoden gaan uit van de algoritmische beschrijving en concentreren zich op het minimaliseren van bepaalde kostenfuncties, bijvoorbeeld het benodigde chip–oppervlak. Het zoekresultaat wordt vervolgens getoetst aan de hand van de overige beperkingen. Dit levert vaak een onbevredigend resultaat op, omdat vaak niet aan de overige beperkingen wordt voldaan, zeker in situaties waarin de beperkingen zeer streng zijn.

De methoden die in dit proefschrift worden beschreven nemen juist de beperkingen als uitgangspunt voor het syntheseproces. Naast de beperkingen uit de initiële specificatie kan de gebruiker in interactie met het systeem beperkingen toevoegen of ontwerpbeslissingen afdwingen. Hierdoor kan een afweging gemaakt worden tussen de kwaliteit van de oplossing en de snelheid waarmee de oplossing wordt bereikt.

Kernpunt van de in dit proefschrift beschreven methodiek is een nieuw algoritme dat gebaseerd is op een graaf–formulering, de bipartiete tijdstoewijzings graaf BSG ('bipartite schedule graph'). De BSG brengt de synthese–opgave in kaart en heeft de volgende twee functies.

In de eerste plaats reduceert de BSG de zoekruimte van de tijdstoewijzing, hetgeen *domein reductie* genoemd wordt. Klassieke executie intervallen van operaties worden verkregen door een kritieke pad analyse, waarbij dan aangenomen wordt dat er een onbeperkt aantal bouwblokken beschikbaar is. De BSG neemt echter de beperkingen ten aanzien van de tijd, precedenties en het aantal en soort bouwblokken in de beschouwing mee, en hierdoor kan een deel van de opties buiten beschouwing gelaten worden. De zoekruimte wordt hierdoor verkleind, de oplosruimte echter niet.

De tweede functie van de BSG is de identificatie van knelpunten in de tijdstoewijzing. Gangbare methoden hebben hier moeite mee: heuristische methoden herkennen de knelpunten niet, terwijl exacte methoden er teveel tijd voor nodig hebben. De BSG identificeert de knelpunten en maakt het vervolgens mogelijk de *zoekruimte efficiënt te doorlopen*, doordat allereerst de knelpunten worden opgelost.

Behalve in de architectuur–synthese bewijzen de beschreven methoden ook hun waarde in andere situaties waarin de beperkingen zeer streng zijn, zoals bij *code–generatie*. In dit proefschrift wordt een ontwerpstrategie voorgesteld, waarbij de code–generatie een losse synthese–stap aan het eind van het architectuur–synthese–traject is. Het complete datapad met tijdsbeperkingen ligt in het geval van code–generatie vast, en de enige overgebleven taak is een correcte tijdstoewijzing van de operaties te vinden en een correcte afbeelding van de operaties op de bouwblokken. Industriële voorbeelden tonen aan dat de methoden gepresenteerd in dit proefschrift tot een efficiënte en doelmatige code–generatie leiden.

Ten slotte zijn de beschreven methoden toepasbaar voor ondergrens–analyses t.b.v. CAD–gereedschappen of interactieve ontwerpbeslissingen. Deze analyses kunnen gebruikt worden voor het bepalen van extra beperkingen die tijdens het ontwerptraject opgelegd kunnen worden.

Bovendien leveren ze een goed inzicht in de ontwerpruimte op, waardoor ze behulpzaam zijn bij het vaststellen van de kwaliteit van een ontwerp. In dit proefschrift worden ondergrens–analyses gepresenteerd voor de functionele oppervlakte en voor het tijdsbudget van een ontwerp. Er wordt aangetoond dat de beschreven methoden nauwkeuriger zijn dan andere polynomiale analyses. Verder wordt ook een methode gepresenteerd die als enige bestaande methode in staat is om een efficiënte ondergrens–analyse te doen voor niet–triviale module–bibliotheken.

Contents

Preface

Organization of this thesis

We start this thesis with a short overview of architectural synthesis and related topics in Chapter 1. The design methodology and strategy, on which the algorithms in this thesis are based, are explained in that chapter as well. In Chapter 2, the central theme of the thesis is presented: execution interval analysis based on bipartite schedule graphs. That chapter can be read in isolation. The rest of the thesis is (partly) based on that chapter and can therefore not be read in isolation.

In Chapter 3, lower bound analyses for the functional area and the cycle budget of a design are given. In Chapter 4, exact scheduling approaches are discussed. That chapter can be read without reading Chapter 3. In Chapter 5, the methods of Chapter 2 and 4 are applied on the code generation problem. That chapter can therefore also be read without reading Chapter 3. Note that the most important notations can be found at the end of this book.

Acknowledgements

First of all, I would like to express my gratitude to Prof. Jochen Jess, for the opportunity to work on my Ph.D. thesis in his research group. His, albeit often indirect, influence on my education has undoubtedly been very large. Secondly, I would like to thank the NEAT club (Wim Philipsen, Ric Hilderink, Marc Heijligers and Harm Arts) for the fruitful cooperation on architectural synthesis tools. Furthermore, I owe a lot to Leon Stok, who gave me a very good start–up on the subject.

I would like to thank my two roommates at the university, Ronald Tangelder and Koen van Eijk, for being the perfect sounding boards for me. I would also like to thank the M.Sc. students, Paul Gruijters, Henry Faber and Joost van Leeuwen, as well as the trainee students who participated in the development of the methods presented in this thesis. I am also very indebted to my copromotor Michel Berkelaar for his MILP solver, which I have been using extensively. I would like to thank all other members of the Design Automation Section of the Eindhoven University of Technology as well, for their stimulating discussions on various topics.

I am also very grateful for the cooperation with Marino Strik and Jef van Meerbergen from the Philips Nat.Lab. Without their personal involvement and their benchmarks, the research on code generation would never have taken place. I also want to thank the Philips Nat.Lab. for publishing my thesis in their series. Furthermore, I would like to thank my reading committee, and especially my second promotor Prof. Emile Aarts, for the valuable comments on the formulations in this thesis. Last but not least I want to thank Ine Waterreus and my family for their support and encouragements during the last years.

Chapter 1 Architectural Synthesis

1.1 Introduction

Ongoing technological developments result in the design and fabrication of increasingly larger and more complex electronic systems than the systems that are currently in use. For a short design cycle and a short time to market, good design methodologies and tools for the computer–aided design (CAD) of these systems are very important. Tools for the *logic* and *layout synthesis* of integrated circuits (ICs) are therefore widely used nowadays. However, the ever increasing complexity and versatility of ICs pushes CAD methods to even higher abstraction levels. This has initiated the development of methods at the *architectural level* of an IC, and nowadays *system–level synthesis* and *hardware–software codesign* are already very popular research areas.

The overall system that helps in designing very large scale integrated (VLSI) circuits is called a *silicon compiler.* In Figure 1.1, a rough sketch of the design flow of such a compiler is depicted. This thesis is concerned with the *architectural synthesis* part of a silicon compiler for synchronous (clocked) VLSI circuits.

The organization of the remainder of this chapter is as follows. Section 1.2 starts with a short overview of the design flow of silicon compilers. The architectural synthesis methods presented in this thesis are based on the design methodology explained in Section 1.3. The design methodology results in the solution strategy for the synthesis of VLSI circuits described in Section 1.4. The design methodology, its synthesis strategy and other considerations lead to general requirements for an architectural synthesis system. Section 1.5 gives a short overview of the kernel of such a system developed at the Design Automation Section of the Eindhoven University of Technology.

1.2 The design flow of silicon compilers

The design flow of a silicon compiler starts either with a *behavioral specification* of the IC, or with an *algorithmic description* in a hardware description language like VHDL, Silage or HardwareC. A behavioral specification is an

abstract description of a circuit under construction. An example of such a specification is the transfer function of a digital filter. An algorithmic description is a slightly more detailed description when compared to a behavioral specification. An example is the description of a digital filter as an FIR (finite input response) or as an IIR (infinite input response) filter.

Most compilers require an algorithmic description to start with, although some research by [Raba93] and [Potk94] has been done to automate the translation of behavioral specifications into algorithmic descriptions. Such a description can, in general, also be the result of (automated) design steps preceding the design of an IC. Examples of such design steps are system–level synthesis and hardware–software codesign.

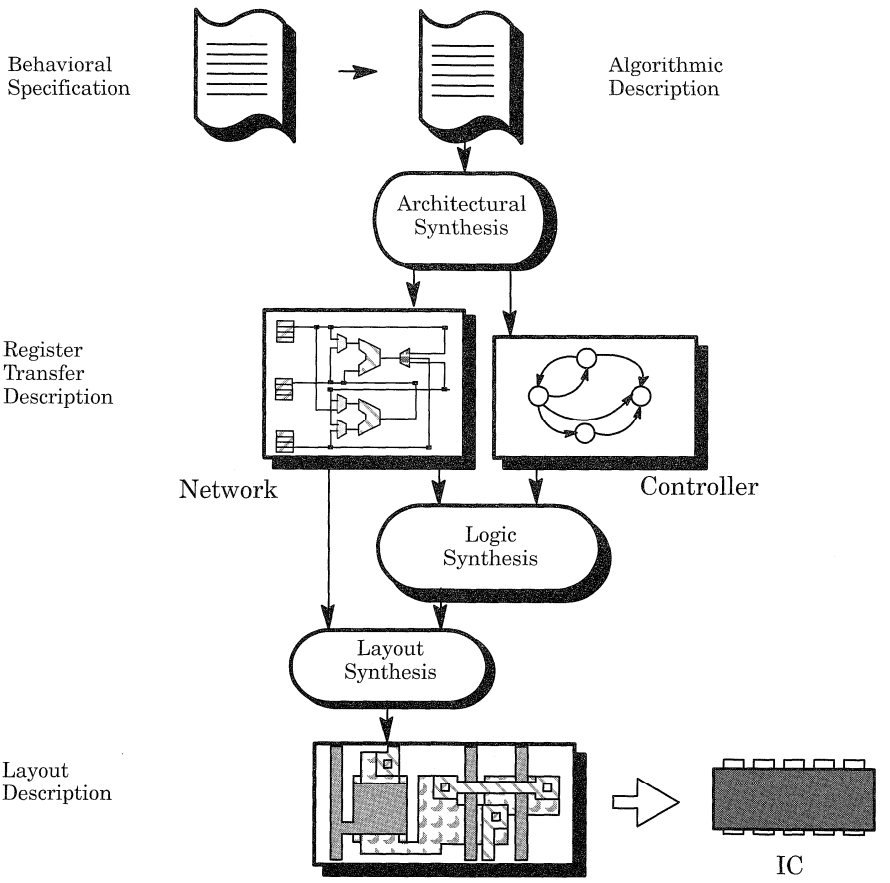


FIGURE 1.1. Computer-aided VLSI design flow, taken from [Stok91].

The *architectural synthesis*, also called *high–level synthesis*, tools take an algorithmic description together with goals and constraints as their input, and produce a so called *register transfer level (RTL)* description of a chip architecture. Such an architecture consists of a datapath, in the form of a network description, and a controller, in the form of a symbolic finite state machine. A datapath is a collection of basic building blocks like adders, multipliers, ALUs, shifters, memory elements etc., which are interconnected by buses and other interconnection units like (de–)multiplexers. The corresponding controller governs the data flow in the architecture. A comprehensive introduction on architectural synthesis and related research can be found in [McFa90].

Both the controller description and parts of the datapath are then fed to *logic synthesis* tools. These tools transform the RTL description into an implementation at the *gate level*. Logic synthesis includes the state encoding for the finite state machine, the optimization of logic blocks and the mapping onto a given technology. At this stage of the design process, modules in the datapath, like multipliers and ALUs, may be generated by module generators.

The final synthesis steps performed by the *layout synthesis* tools consist of layout design tasks such as *floor planning*, *placement* and *routing*. At this stage also the layout of basic building blocks like ROMs, RAMs, registers, basic logic cells, and regularly structured datapath components can be generated. The result is a set of geometrical descriptions of *layout masks*, which are a complete physical description of the IC under construction.

While layout and logic synthesis tools are widely used nowadays, true architectural synthesis is still working its way up from infancy. However, it can offer a lot of advantages due to its higher level of abstraction. The algorithmic description of an IC is a formal specification at a high level of the design process. The simulation of such a description allows a designer to evaluate the intended behavior of the design. The use of architectural synthesis methods can result in a shorter design cycle, it can enable a (system) designer to explore the design space more rapidly, and it can result in designs that contain fewer errors. The formal algorithmic description forces a designer to specify the design very precisely. Together with records kept by an automated design management, the result of the design process can be well documented.

Thus the objective of architectural synthesis is to support a designer at a higher level of abstraction than logic and layout synthesis do. Due to this level of abstraction and the possibility of better design space explorations, a designer should be capable of making better VLSI designs within a shorter amount of time.

1.3 A design methodology

1.3.1 The classical problem decomposition

In the previous section, the task of designing an electronic system or IC was decomposed into several subtasks. These subtasks are of course heavily interrelated but are hard to solve in their entirety. For the same reason, the architectural synthesis task is also decomposed into several subtasks. The first decomposition that is normally made is the decomposition into *datapath synthesis* and *controller synthesis*. It is very difficult to automate the trade–offs between datapath and controller costs, so most architectural synthesis systems do not design the datapath and controller of a VLSI circuit simultaneously. The following three main subtasks are generally distinguished when synthesizing a datapath from a behavioral description.

- *Selection* determines the type and number of resources needed for a datapath.
- *Scheduling* determines an assignment of the operations to be executed in the datapath to specific moments in time.
- *Binding* determines an assignment of the operations to specific resources.

Note that allocation is also a term that is heavily used in the literature. Unfortunately, the term is often used to identify different subtasks, e.g., selection only, selection and binding together, and so on. To avoid confusion, the term allocation is therefore not used in this thesis.

Many synthesis problems, like the subtasks mentioned above, are known to be NP–hard [Garey79] in general cases, see for instance [Verh95]. The organization of these interdependent subtasks influence the quality of the resulting datapath substantially. The way and order in which these subtasks are solved can depend on different application domains, e.g., micro processors versus digital signal processors or control–dominated designs versus datapath–dominated designs. These different application domains can require different optimization strategies to end up with good RTL designs.

Depending on the application, different goals and constraints, in terms of area, timing, power consumption, etc., are imposed by a designer, possibly leading to different synthesis strategies. So, one might raise the question whether an overall design methodology for all application domains can be constructed and whether it would be favorable. The answer to this question is contained in the following three elements of discussion.

1.3.2 VLSI design considerations

When choosing a VLSI design methodology, the following aspects must be considered.

1. In many cases the goal of the optimization for a design cannot be captured efficiently by means of a well defined objective function. For instance, many designs have only timing constraints at the beginning of the design process, and the optimization goal is to end up with a design with as little area as possible. However, it is almost impossible to model the interconnect costs and delays at high level. The cost of a bus or the signal delay can be very high if the modules connected by the bus are placed far away from each other by the layout synthesis tools. But they are (nearly) zero, if the modules are abutted. Counting the number of buses is therefore not a valid way to obtain a good measure for the interconnect costs and delays. On the other hand, evaluating all possible datapaths and bus configurations down to layout level to pick the best one is much too (CPU-) time consuming.

2. Many optimization goals are hard to combine or even contradicting. For instance, the optimization goal for many designs is to end up with as small as possible power consumption and area. However, minimal area can imply a lot of multiplexing in the datapath, which in turn leads to a high power consumption. At many stages of the design process a choice has to be made as to the optimization goals to be pursued at that particular stage.

3. The constraints and goals imposed on an IC design are in most cases much stricter than for instance in the case of software compilation. Hard timing constraints imposed by the environment of an IC design have to be met by all means, while every square millimeter saved in terms of chip area can lead to important economical advantages. In the field of software compilation, the completion time of an algorithm is not that important in comparison with the hard constraints imposed on the throughput of, for instance, digital signal processing (DSP) applications. There are exceptions like [Chou94], but in that approach the resulting schedule is fully serial, so no parallelism in the datapath is possible. Such parallelism is usually needed in most architectural synthesis applications in order to obtain sufficient throughput. So, because the constraints and goals are quite strict in architectural synthesis, a designer wants to have thorough control over the design process and wants to be able to enforce synthesis decisions and strategies at his will.

The ultimate architectural synthesis system is one in which no user interaction is needed. However, this is still far away. There are many cases where a designer wants to outperform computer programs and seems to have the ability to do so. This means that user interaction remains important, also in the next generation(s) of architectural synthesis systems.

1.3.3 The proposed methodology

The discussion above shows that an architectural synthesis system cannot be steered by means of an objective function only. An efficient architectural synthesis objective function does not discriminate sufficiently between good and bad architectural solutions. Another solution methodology which is not solemnly based on optimizing objective functions has therefore to be found. Consider the case in which a solution is said to be good if and only if it satisfies the constraints imposed on a design problem, without considering some objective function. In such a case a synthesis system can easily discriminate between good and bad solutions.

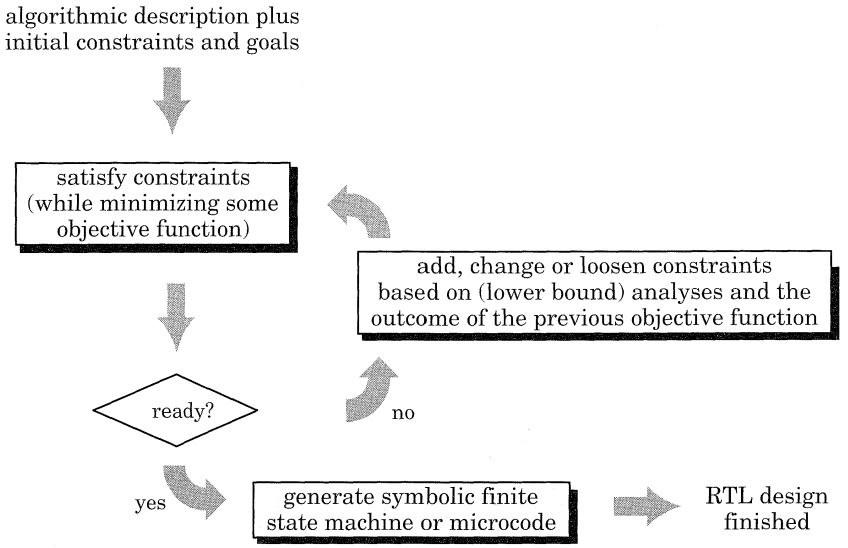


FIGURE 1.2. Design methodology for architectural synthesis.

This leads to the design methodology pictured in Figure 1.2. Starting with the initial goals and constraints imposed on a design, the design process is steered by adding more and more constraints until a design is fully constrained, i.e., specified. For instance, a design with only timing constraints at the beginning of the design process, will become more and more time *and* resource constrained along the synthesis flow by adding resource constraints. So, the initial problem description is modified during the synthesis flow. The synthesis result now relies on the constraints additionally imposed on a design and the ability of the synthesis system to satisfy all the constraints. For that reason, the ability of a synthesis system to satisfy

different combinations of constraints becomes essential in such a design methodology. The role of minimizing objective functions is merely reduced to steering the process of adding constraints; the outcome of an objective function can give an insight in the design space.

The additional constraints can be generated by CAD tools or by user interaction and can be based on (lower bound) design space analyses. It will be shown that lower bound analyses give good insight in the design space, so they can also help to measure the quality of design decisions. To enable a designer to enforce certain synthesis decisions, a synthesis system must be able to deal with different combinations of constraints. The approach of Figure 1.2 has, therefore, the additional advantage that user interaction is smoothly incorporated in the design flow.

1.3.4 Methodology considerations

Instead of concentrating on minimizing certain objective functions, the methods and algorithms in this thesis emphasize *lower bound design space analyses* and *constraint satisfaction* techniques. To obtain a conceptually sound synthesis flow supporting such a methodology, the following considerations must be kept in mind.

- For the *constraint generation*, a designer or a synthesis tool must be able to evaluate the quality and status of a design at any time, to support (interactive) changes of the added constraints. A designer should receive as much helpful information as possible, which can be used to steer certain synthesis tasks, e.g., information that can help to select resources manually. Chapter 3 deals with lower bound analyses, which help a designer or CAD tool to evaluate the quality of a design.

- After observing the current status and the quality of a design, a designer or a tool must be able to enforce certain features of a synthesis solution. Hence an architectural synthesis system should be capable of handling designs on which many constraints of different nature are imposed, i.e., such a system must have powerful *constraint satisfaction* techniques.

Many existing synthesis systems are hampered in dealing with large sets of constraints. Yet a synthesis system should exploit all the constraints that are imposed, as the number of solutions decreases with an increasing number of constraints of different nature. Chapter 2 introduces a low–order polynomial run time algorithm which shows the possibility of taking advantage of different kinds of constraints instead of being hampered by them. The algorithm prunes the search space of schedulers without limiting the solution space when both time and resource constraints are imposed on a design. It is therefore a complementary technique to heuristics.

• The *run time efficiency* of the different tasks performed by a synthesis system must be as high as possible. However, in order to give a designer maximum flexibility, a designer must also be able to control the trade-off between the solution quality and the run time efficiency of a synthesis task. The tedious task of scheduling is preferably left to a synthesis system and is not performed manually. Architectural synthesis scheduling problems are in general NP-hard, so an optimal solution cannot be guaranteed within acceptable CPU times. However, it can be profitable to spend more CPU time to obtain a (near) optimal solution for critical parts of a design. In Chapter 4, a scheduling algorithm will therefore be presented which offers the possibility of trading off the solution quality against the run time efficiency.

• Eventually, at the end of the synthesis flow, it may occur that a complete datapath consisting of interconnected resources plus timing constraints is enforced by a designer or a synthesis tool, i.e., the design is completely constrained. The remaining task for the synthesis system is then to find a correct schedule and binding scheme. This is actually the starting point of *retargetable code generation*, see [Paul94] or Chapter 5. So retargetable code generation is a natural part of the design methodology depicted above and even a touchstone of it. If it is not possible to develop a good retargetable code generator, then a design methodology built upon constraint satisfaction is not favorable.

1.4 Towards a solution strategy

The discussion in the previous section leads to the hypothesis that a good design methodology is focused on adding constraints to a design by means of analyses and design space explorations until the design space is narrowed down sufficiently to contain only (a few) satisfactory solutions. A design will be more or less completely constrained at the end of such a process. Therefore, the application domain and the *initial* goals and constraints will become less and less 'visible' and are deemphasized along the synthesis flow, because the initial problem description is modified.

This leads to a concept where the classical subtasks of architectural synthesis (selection, scheduling and binding) are not the main steps in the synthesis flow, but reflect merely tentative decisions that can be revoked in a next synthesis step. In this concept, the steps in the synthesis flow are aimed at different aspects of datapath optimizations and trade–offs. So, it must be possible to execute the different synthesis tasks and optimization strategies in an arbitrary way, in an arbitrary order and an arbitrary number of times until a satisfactory solution has been found. Two questions remain: is there a preferable order of synthesis tasks and is the run time efficiency and quality of all the synthesis tools sufficiently large to be invoked repeatedly.

Best understood subtasks

Literature on architectural synthesis shows that scheduling is one of the best understood subtasks in the field [McFa90]. As to optimality, the classical scheduling problems come in the following two clean forms.

- In time constrained scheduling one is asked to arrange the schedule within a given (maximum) number of clock cycles (the so called cycle budget) while minimizing the cost (mostly given in terms of area) of resource usage.
- In resource constrained scheduling one is asked to arrange the schedule within maximum bounds on the resource usage while minimizing the number of clock cycles.

In real life, a scheduling problem may come as a mixture of those clean forms, and many systems are known to become both time and resource constrained along the way. This is done to achieve a tight performance control, in compliance with the design methodology explained in Section 1.3. Examples are the systems HYPER [Raba90], MSSR [Ishi91], TBS [Rama91], CADDY [Gutb92], Phideo [Verh95] and NEAT [Timm93a], which are time constrained at the beginning of the design flow. The resource constraints are mostly given in terms of the number of instantiations of functional units, because there are many efficient approaches giving a lower bound estimate on the functional area, see [Jain92], [Shar93], [Timm93c] or Chapter 3. The scheduling and functional area estimation tasks can be modelled easily with a so called data flow graph model, see Section 1.5.2, and can be solved efficiently.

Flow of subtasks

Because the subtasks mentioned above are so well understood, a logical first step in a solution strategy is to determine the minimal functional area, or the minimal cycle budget in case the resources are given, without considering interconnect and memory costs. Chapter 3 and 4 show that, starting from an initial lower bound estimate of the module set with minimal functional area for a given time constraint, a module selection review can take place until a feasible module set is found. A trade–off can be made between the optimality of the solution and the run time efficiency of such an approach. A good evaluation of such a synthesis task can be obtained: both a lower bound estimate and the area of the resulting set of functional units are available. With these numbers a designer can estimate the quality of the resulting set of functional units.

The number and place of the register files or other memory elements in a datapath partly depend on the bus configuration. A logical next step in the synthesis flow is therefore to trade off functional area against interconnect

area and delay to decrease the total area, while tentatively not considering the memory area. Many synthesis systems try to optimize the interconnect after a schedule is fixed [Pang88] or during scheduling [Bala89], [Berry90]. Some systems cluster operations before a schedule is fixed based on the regularity in a DFG [Note89], [Rao93]. These clusters then reappear in the datapath as clusters of modules which are interconnected through local buses. Clustering can lead to fewer global buses at the expense of more module area.

Such a trade–off should be guided by the regularity in the DFG [Rao92] and estimates of the total functional and interconnect area / delay. These estimates must have a realistic accuracy: because of the run time efficiency it is not desirable to perform a fine–grain placement and routing at the architectural synthesis phase of a silicon compiler, if this task has to be performed many times. Estimations like the one in [Kurd91] try to be run time efficient and yet accurate enough for this purpose, but not much research has yet been done in this field. The insight in the design space can be good, if the trade–off starts with a datapath with minimal functional area and mainly global buses and evolves towards a data path which consists completely of groups ('clusters') of modules with only a few global buses. An additional advantage of such a step in the design flow is that the interconnect delay can be taken into account during later synthesis steps.

A next step in the solution strategy can possibly be memory minimization. A trade–off between memory area and functional / interconnect area can be found in a subsequent step, e.g., by enforcing an ordering between different data transfers and/or by binding some data transfers in advance. By performing the other synthesis tasks iteratively, less memory area can be allocated at the expense of functional and / or interconnect area. As has been said, in a solution strategy like the one described in this section, it should be possible to revoke the different synthesis steps an arbitrary number of times until a satisfactory solution has been reached.

1.5 Description of the NEAT design environment

1.5.1 Overview of requirements

Despite the diversity in design styles many tasks in the synthesis flow are similar. The previous sections discussed the hypothesis that a good overall design methodology will lead to a more or less uniform solution approach making it unnecessary to write a new synthesis system for each design style. A synthesis system should, nevertheless, allow as much freedom as possible with respect to the way and order in which the subtasks of the synthesis problem are solved.

To solve the architectural synthesis problem as a whole, a collection of interacting tools is needed. This leads to the insight that *software engineering* and *data management* are important aspects of the design environment requirements. Each tool has to retrieve, manipulate and store intermediate synthesis results by using a synthesis data interface. Because synthesis data is shared among different synthesis tools, the data interface should be common to all tools, which makes the maintainability of separate tools, programming efficiency and the preservation of the consistency of synthesis data much easier. On the other hand, each synthesis tool has its own requirements for the synthesis interface. It must be possible to add tool specific data to the interface which is hidden from the data interface of other tools. The interaction between different tools can be achieved by the exchange of intermediate synthesis data accompanied by specific keywords indicating the status of the data. An example is a keyword that states whether it is allowed to change the schedule of operations if the set of resources is altered.

1.5.2 NEAT and its data domains

An object oriented synthesis interface, the *New Eindhoven Architectural synthesis ToolBox (NEAT)* written in C++ has been developed in compliance with the requirements of Section 1.5.1; see [Heij94] and Figure 1.3. At the left

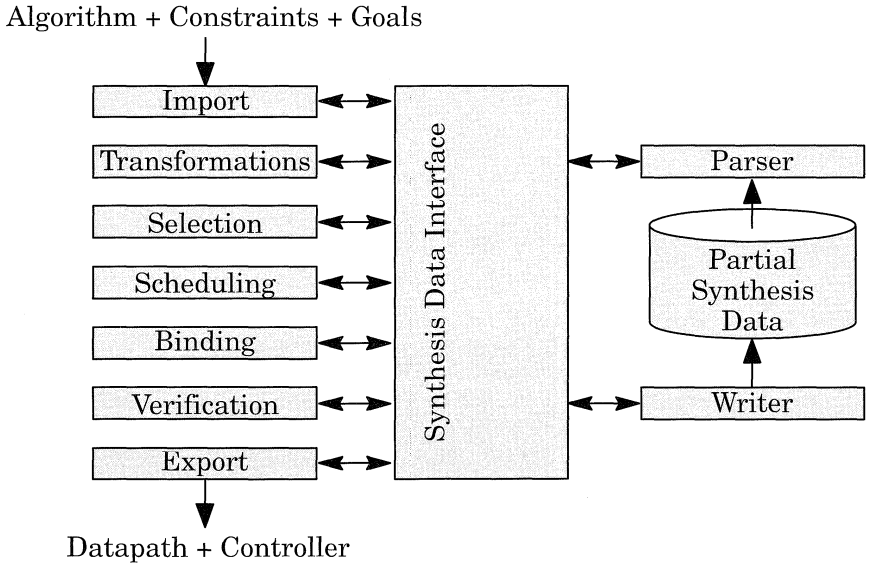


FIGURE 1.3. The architectural synthesis interface and tools.

hand side of the figure, a number of possible synthesis tools, e.g., import, selection, scheduling, export, etc., are given. A parser reads synthesis data from a file and stores it into memory in the synthesis data interface. The interface abstracts the different synthesis tools from the synthesis data by providing access routines to the data.

NEAT offers unlimited extendibility and no restrictions with respect to a synthesis flow or methodology. Three domains of data can be distinguished: a *behavioral*, a *timing* and a *structural* domain. From a theoretical point of view, these domains can be represented by a single domain by labeling timing and structure information to the behavioral data [Jong93]. A separation of the three domains has been decided for the ease of implementing the various synthesis tools and for the ease of representing the synthesis results. NEAT provides, but is not limited to, three design views corresponding to these domains. Other domains of data can be added by adding new views to the interface.

Behavioral domain

The first domain in NEAT is the behavioral domain, and in this domain ASCIS data flow graphs (DFGs) are used to describe the behavior of a design [Eijnd92]. Data flow graphs are the internal representation of the algorithmic description and can be obtained from hardware description languages by means of data flow analysis. With DFGs it is tried to give a representation of the behavior of a design which contains as much parallelism as possible. Applying synthesis directly to a DFG frees synthesis from the varying nature of input languages and facilitates possible algorithmic transformations. Data flow graphs impose no limitations with respect to architectural solutions. They are, therefore, suitable as a starting point for architectural synthesis.

Although there are a number of DFG models around, the common feature of all those models is that they map certain elementary actions on the vertices of such a graph. Those actions are arithmetic operations like additions, subtractions or multiplications, but also the generation of constants, bit operations like masking and shifting, conditionals, loop controls and memory read / write actions. In the sequel, these actions are called *operations*. The interface of a data flow graph to the outside world is defined by means of input and output vertices.

In addition to the decomposition of algorithms into elementary operations, the DFG model captures the dependencies between those operations. Mostly those dependencies are established by the fact that operations consume values, or 'tokens', produced by other operations. An operation has to be

scheduled in time after the completion of the operations producing the required input values. In this thesis, those precedence relations are called *data dependencies*, although this obscures the fact that the inputs could also be control signals as well and data independent sequencing relations can also exist.

To support the special language constructs like loops and conditionals, vertices with a special execution mechanism have been defined. An example of a data flow graph with such a special construct has been given in Figure 1.4. At the left hand side of that figure, a textual process declaration has been given. The process consists of a loop, which results in entry (EN) and exit (EX) vertices in the DFG. An entry vertex takes the input of one of its input arcs and sends it over its output arc; an exit vertex takes an input token and copies it to one of its output arcs. The choice which arcs are selected depends on a control signal provided to these vertices, see the dashed arcs in Figure 1.4. The left entry and exit vertices pass the values of the variable 'd', while the right entry and exit vertices pass the values of the variable 'a'. For a more elaborate discussion on the ASCIS data flow graph, see [Eijnd92].

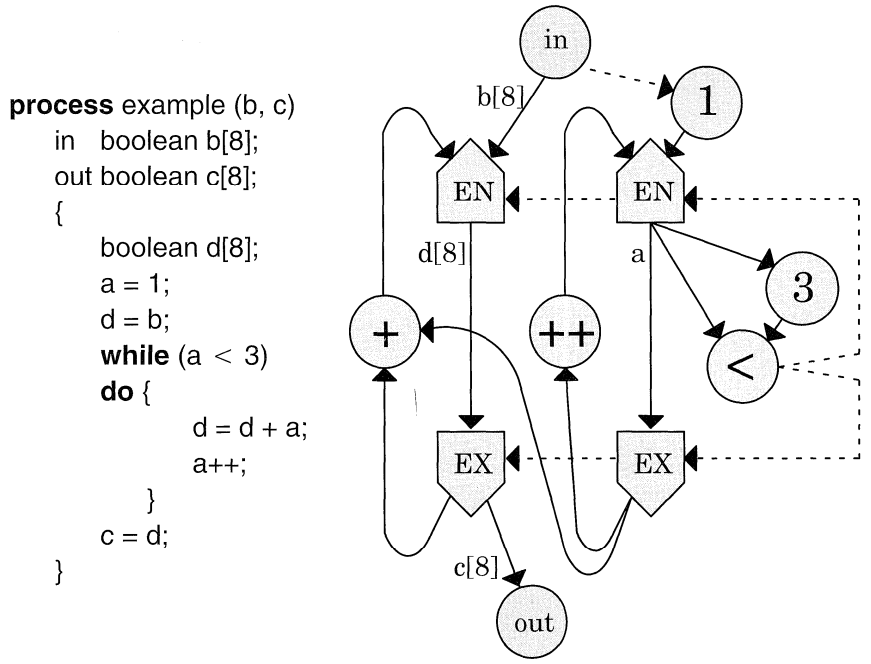


FIGURE 1.4. Example of a data flow graph.

Timing and structural domains

The second and third domain in NEAT are the timing and structural domains respectively. In the *timing domain*, the time behavior of a design is described by a control graph (CTG) which models a finite state machine, see Figure 1.5. In that figure, a DFG has been given at the left hand side, while the corresponding CTG has been given in the middle. Vertices in a control graph represent states, except for the input and output vertices, and edges represent state transitions. the dashed edges between the DFG and CTG in Figure 1.5. denote in which state a certain operation is scheduled. Furthermore, control graphs can be extended with special constructs to explicitly represent conditionals, loops, multiple active states and hierarchy to hint at the controller design.

In the *structural domain*, the datapath of a design is described by a network graph (NWG), see also Figure 1.5. There, the NWG is given at the right hand side, describing the datapath by register transfer level components and their interconnections. The dashed edges to the NWG denote in which states the modules are active and what operations are mapped to them.

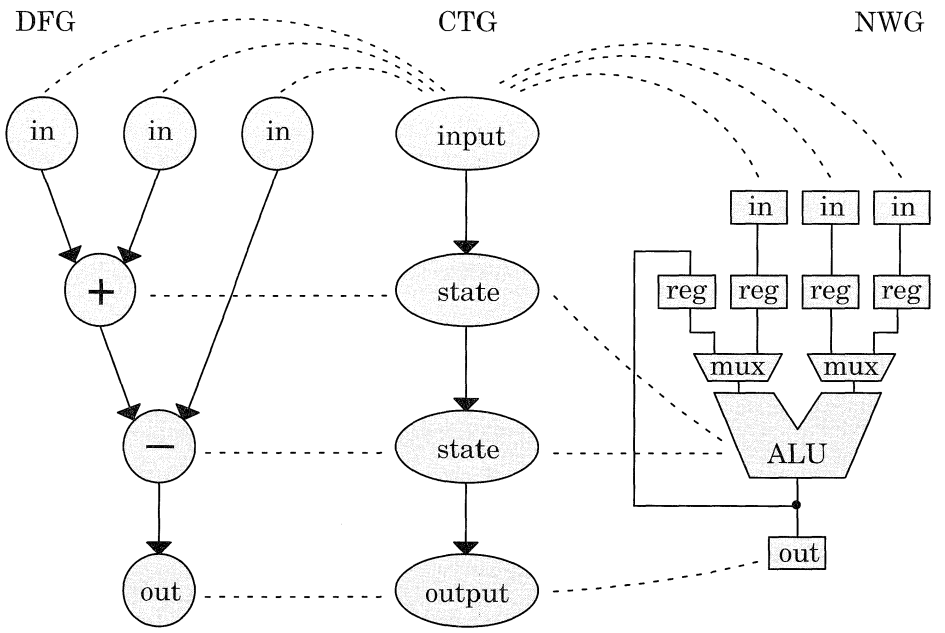


FIGURE 1.5. Simplified example of inter–domain relations.

1.5.3 Design relations

Between the three domains relationships are established to point out how the different synthesis objects are related. Two important types of relations can be distinguished: intra–domain and inter–domain relations.

The *intra–domain relations* are used to preserve consistency of data within a domain, by means of describing the behavior and interface of vertices by graphs within the same domain. An operation, state vertex or network module can be generated by referring to a graph, inheriting the interface and behavior of that graph. The interface of a graph is described by input and output vertices, the behavior by its contents (vertices and edges), documentation (for standard operations like additions and multiplications), or computer programs (for standard modules like adders, multipliers, RAMs, and so on). Hence intra–domain relations provide support for hierarchical bottom–up and top–down design methods.

The *inter–domain relations* describe the relationships among objects of different design views by means of so called graph links and vertex links. Links can be specified only partially to represent intermediate synthesis results. A *graph link* relates a data flow, control and network graph with each other. Such a link represents a relation like 'this network graph represents the datapath belonging to this data flow graph'. A *vertex link* relates a data flow vertex, control vertices and a network vertex with each other. Such a link denote the fine–grain relations among graphs, like 'this data flow vertex is bound to this network vertex', and / or 'this data flow vertex is scheduled onto these control vertices'. A graphical and simplified example of inter–domain relations has been given in Figure 1.5.

Inside links the kind and status of the relationship they represent can be defined in more detail, which makes it easier for tools to decide how particular links should be used. The links describe the complex and detailed fine–grain relationships separately from the graph descriptions in the different domains. Nevertheless, synthesis information is gently incorporated into the synthesis data. Design analysis tools like formal verifiers, simulators or graphical interactive tools can easily use the links to determine the relationships between behavior, time and structure. In [Hild94], an example of the use of graphical interface tools which are part of the ESCAPE system [Fleu96] has been given. ESCAPE reads the links from the NEAT interface and shows the information contained in the links graphically.

Chapter 2 Execution Interval Analysis

2.1 Introduction

In almost any of their formal appearances, architectural synthesis scheduling problems are not solvable in polynomial time, see [Heem90], [Verh91], [Verh95]. For that reason, many heuristical approaches have been investigated; see for typical examples [Girc85], [Park86], [Peng86], [Pang87], [Paul87], [Thom88], and [Camp90]. Under the regime of tight performance control, more and more instances of the problem appear where heuristic approaches render unsatisfactory results, see Chapter 5. A search for exact methods like integer programming (IP) techniques [Hwang91], [Gebo92] and branch–and–bound methods has therefore been initiated.

Exact methods depend on powerful branching and pruning techniques, which are called variable & value selection and domain reduction (or consistency checking) respectively in the field of constraint satisfaction, see for instance [Nuijt95]. One of the important ways to support pruning is the analysis of the *operation execution intervals* (OEIs), for which the following informal definition can be given.

Given some operation v in a DFG, the associated OEI restricts the interval of consecutive clock cycles to which v can be assigned. A classical and conservative, but pessimistic, estimate for the bounds of some OEI, which is applied in most architectural synthesis systems, is given by the 'as soon as possible' (ASAP) and 'as late as possible' (ALAP) values of an operation. These values are obtained by a critical path analysis of the DFG under the assumption of unlimited resources. Such an analysis accounts only for the data dependencies and the delays of the operations in the DFG. In the presence of constrained resources though, i.e., if there are not only timing constraints imposed on a design but also resource constraints, the pruning is still far from satisfactory.

In this chapter a new technique is proposed to enhance the pruning capability of scheduling approaches, by reducing the classical operation execution intervals mentioned above. The reduction is obtained in low–order polynomial time with a bipartite graph matching formulation that exploits both time and

resource constraints. It prunes the search space of schedulers without limiting the solution space, thus enhancing the quality of schedulers. With this technique a synthesis system makes use of all the different kind of constraints that are imposed, instead of being hampered by them. It is therefore a complementary technique to heuristics: if there is much scheduling freedom, then no or hardly any pruning will occur.

This thesis deals with the execution interval analysis first, because it is used throughout the whole NEAT synthesis trajectory and throughout the rest of this thesis as well. For instance, Chapter 4 deals with the scheduling process itself and shows that the matching formulation introduced in this chapter is not only capable of *pruning* the search space. The bipartite matching formulation is also very valuable for *traversing* the search space efficiently, i.e., for the variable & value selection part of a scheduler.

The outline of this chapter is as follows. In Section 2.2, a number of definitions and the formal problem definition of execution interval analysis are given. Section 2.3 starts with considering trivial module libraries: the section considers the case in which there is a one–to–one mapping from operation types to module types. In Section 2.4 the approach is extended to unrestricted libraries allowing for many–to–many mappings. The chapter concludes with experimental results and a discussion in Section 2.5 and 2.6 respectively.

2.2 Definitions

In the sequel of this thesis, the following formal definitions are needed.

Definition 2.1. Data flow graph DFG.
A data flow graph DFG is a 2–tuple (V, E), where V is the set of vertices (operations) and $E \subseteq V \times V$ the set of arcs representing dependencies between operations.

Definition 2.2. Operation types.
T_O is the set of operation types in a DFG. An example of an operation type is the addition.

Definition 2.3. Operation type function.
$\tau: V \rightarrow T_O$ gives for each operation its type.

Definition 2.4. Immediate predecessors.
$pred(v) = \{w \in V \mid (w, v) \in E)$ for each $v \in V$.

Definition 2.5. Immediate successors.
$succ(v) = \{w \in V \mid (v, w) \in E)$ for each $v \in V$.

Definition 2.6. Transitive closure.
The transitive closure of DFG is the graph DFG* = (V, E*), where E* = { (v, w) | v, w ∈ V and v is connected to w in DFG}.

Definition 2.7. Predecessors.
$pred^*(v) = \{w \in V \mid (w, v) \in E^*\}$ for each v ∈ V.

Definition 2.8. Successors.
$succ^*(v) = \{w \in V \mid (v, w) \in E^*\}$ for each v ∈ V.

Definition 2.9. List of cycle steps.
C is an interval from 0 to $|C| - 1$, representing the list of available cycle steps, i.e., the cycle budget.

Definition 2.10. Set of modules.
M is a set of modules. In this thesis, most of the time this set is restricted to functional units. An example of such a set is three adders and two multipliers.

Definition 2.11. Set of module types.
T_M is the set of module types, or library. An example of a module type is the carry–lookahead adder type.

It is important to notice the difference between a set of modules and a set of module types, or module library. This difference will be crucial for the execution interval analysis in this chapter.

Definition 2.12. Module type function.
$\xi: M \rightarrow T_M$ gives for each module its type.

Definition 2.13. Mapping from operation types to module types.
$\mu: P(T_O) \rightarrow P(T_M)$, where $P(X)$ denotes the power set of X, is a mapping from operation types to module types. μ(ts), ts ⊆ T_O, returns the union of module types that can implement some, but not necessarily all, operations from the set of operations whose types are in ts.

Although the definition of μ is somewhat peculiar, it is used in this chapter and the next chapter on module selection to facilitate the modelling related to many–to–many mappings of operation types to module types.

Definition 2.14. Trivial module library.
A module library T_M is trivial if there is a one–to–one mapping of operations to modules types, i.e., if $|\mu(\{\tau(v)\})| = 1$ and $|\mu^{-1}(\{m\})| = 1$ for all v ∈ V and m ∈ T_M. In case of such a library, the delay of an operation is known before scheduling.

Definition 2.15. Unrestricted module library.
A module library T_M is called unrestricted if it is not trivial. Such a library is especially interesting if $|\mu(\{\tau(v)\})| > 1$ for some $v \in V$.

Definition 2.16. Delay of a module type.
$d: T_M \rightarrow \mathbb{Q}^+$, where $\mathbb{Q}^+$ is the set of positive rational numbers, gives the minimal fractions of cycles, i.e., the delay, a module needs to execute an operation.

Definition 2.17. Minimal execution delay of an operation.
In case of a trivial module library, see Definition 2.14, the minimal execution delay of an operation $d_{min}(v)$ equals $d(m)$, with $\{m\} = \mu(\{\tau(v)\})$. In case of an unrestricted module library, see Definition 2.15, the minimal execution delay $d_{min}(v)$ of an operation v equals $\min_{m \in \mu(\{\tau(v)\})} d(m)$.

Definition 2.18. Data introduction interval of a module type.
$dii: T_M \rightarrow \mathbb{N}^+$, where $\mathbb{N}^+$ is the set of positive natural numbers, returns the data introduction interval (or restart time), which is the minimal number of cycles required between the data arrivals for two successive executions of different operations on a module.

A pipelined module can have a larger delay than data introduction interval. For a module that can be chained, the delay is smaller than the data introduction interval, which equals one clock cycle in this case.

If nothing else is said, a next instantiation of the DFG takes place only after the DFG is finished completely, i.e., the delay and dii of a DFG are equal. Furthermore, if nothing else is said, the DFG is considered to be an acyclic graph without control constructs. In that case, an operation in a DFG can be executed if at least one token, i.e., value, is available on each of its input arcs, resulting in the production of a token on each of its output arcs.

The assumptions above are made for reasons of simplicity. For instance, control constructs are handled by the implementations of the methods, but are not treated explicitly in this thesis. For most methods it is not difficult to take cyclic DFGs into account as well. In Chapter 5, for instance, a loop model is used in which the delay and dii of a DFG can differ.

Definition 2.19. Schedule interval of an operation.
The schedule interval $\phi(v) = [\phi_1(v), \phi_2(v)]$ of an operation $v \in V$ is given by the start time $\phi_1(v)$ and the completion time $\phi_2(v)$, such that the operation is being executed in that interval. Non preemptive scheduling is assumed, which implies that the delay, i.e., processing time, of a scheduled operation v equals $\phi_2(v) - \phi_1(v)$. Depending on the binding of an operation to a module, the delay of an operation can have different values.

Definition 2.20. Schedule.
A schedule ϕ assigns to each operation $v \in V$ a schedule interval $\phi(v)$. A schedule is called feasible, if all precedence, resource and time constraints for the DFG are met.

Definition 2.21. Set of feasible schedules Φ.
In the sequel only the set of *feasible* schedules Φ is considered, in which all precedence, resource and time constraints for a given DFG are met. Thus any $\phi \in \Phi$ is by definition a feasible schedule. Note that this set may be empty for certain compositions of constraints.

Definition 2.22. Classical 'as soon as possible' start time of an operation.
Let the set of constraints associated with Φ consist of precedence constraints only. The classical 'as soon as possible' start time CASAP(v) of an operation $v \in V$ is defined by: $\mathrm{CASAP}(v) = \min_{\phi \in \Phi} \phi_1(v)$.

Corollary 2.1. Polynomial calculation classical 'as soon as possible' time.
Consider an acyclic data flow graph. Then the CASAP(v) for all $v \in V$ can be determined recursively in $O(|V| + |E|)$:

$$\mathrm{CASAP}(v) = \begin{cases} 0 & \text{if } \mathrm{pred}(v) = \emptyset \\ \max\limits_{w \in \mathrm{pred}(v)} \left(\mathrm{CASAP}(w) + d_{\min}(w)\right) & \text{if } \mathrm{pred}(v) \neq \emptyset \end{cases}$$

Definition 2.23. Classical 'as late as possible' completion time.
Let the set of constraints associated with Φ consist of precedence constraints and a timing constraint only. The classical 'as late as possible' completion time CALAP(v) of an operation $v \in V$ is defined by: $\mathrm{CALAP}(v) = \max_{\phi \in \Phi} \phi_2(v)$.

Corollary 2.2. Polynomial calculation classical 'as late as possible' time.
Consider an acyclic data flow graph. Let the time constraint be defined by the list of clock cycles C. Then the CALAP(v) for all $v \in V$ can be determined recursively in $O(|V| + |E|)$:

$$\mathrm{CALAP}(v) = \begin{cases} |C| & \text{if } \mathrm{succ}(v) = \emptyset \\ \min\limits_{w \in \mathrm{succ}(v)} \left(\mathrm{CALAP}(w) - d_{\min}(w)\right) & \text{if } \mathrm{succ}(v) \neq \emptyset \end{cases}$$

Definition 2.24. Classical operation execution interval.
The CASAP and CALAP values are rational numbers to allow the recursive calculation of these values in case the operations can be chained: in such a case not all of the CASAP and CALAP values are on clock cycle boundaries. Figure 2.1 points out the difference between clock cycles and clock cycle boundaries. In contrast to those values the classical operation execution interval COEI(v) of an operation $v \in V$ is defined by an interval of clock cycles: $\mathrm{COEI}(v) = [\lfloor \mathrm{CASAP}(v) \rfloor, \lceil \mathrm{CALAP}(v) - 1 \rceil]$.

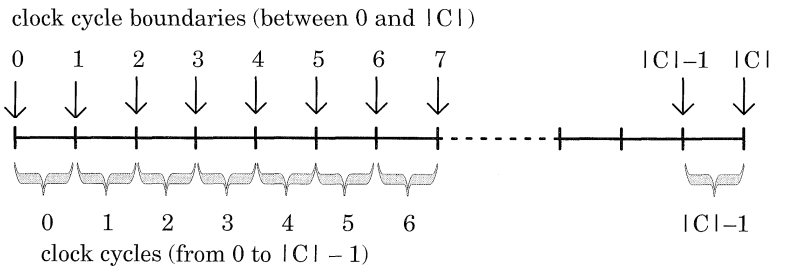


FIGURE 2.1. Difference between clock cycles and cycle boundaries.

In the previous definitions, no resource constraints were part of the set of constraints for Φ. In the following three definitions, resource constraints are part of the set of constraints, which leads to subtle differences in the definitions.

DEFINITION 2.25. 'As soon as possible' start time of an operation.
Let the set of constraints for Φ consist of precedence constraints *and* resource constraints. Then the 'as soon as possible' start time ASAP(v) of an operation $v \in V$ is then defined by:
$ASAP(v) = \min_{\phi \in \Phi} \phi_1(v)$.

DEFINITION 2.26. 'As late as possible' completion time of an operation.
Let the set of constraints for Φ consist of precedence constraints, timing constraints *and* resource constraints. Then the 'as late as possible' completion time ALAP(v) of an operation $v \in V$ is defined by:
$ALAP(v) = \max_{\phi \in \Phi} \phi_2(v)$.

DEFINITION 2.27. Operation execution interval.
The operation execution interval OEI(v) of an operation $v \in V$ is defined by the following interval of clock cycles: $OEI(v) = [\lfloor ASAP(v) \rfloor, \lceil ALAP(v) - 1 \rceil]$.

An important difference between a COEI and OEI is the fact that the CASAP and CALAP values can be calculated in polynomial time, while the ASAP and ALAP values can not be calculated in polynomial time in the general case. We will return to this issue later on.

Because the ASAP, ALAP and OEI can, in general, not be calculated in polynomial time, the determination of conservative estimates will be discussed in this chapter. For that reason we introduce the following definitions for the conservative estimates of these values.

Definition 2.28. Conservative estimate of the 'as soon as possible' time. The conservative estimate $\overline{ASAP}(v)$ of an operation $v \in V$ is an estimate of ASAP(v), satisfying $CASAP(v) \leq \overline{ASAP}(v) \leq ASAP(v)$.

Definition 2.29. Conservative estimate of the 'as late as possible' time. The conservative estimate $\overline{ALAP}(v)$ of an operation $v \in V$ is an estimate of ALAP(v), satisfying $ALAP(v) \leq \overline{ALAP}(v) \leq CALAP(v)$.

Definition 2.30. Conservative estimate of operation execution interval. The conservative estimate $\overline{OEI}(v)$ of an operation $v \in V$ is an estimate of OEI(v), defined by the following interval of clock cycles:
$\overline{OEI}(v) = [\lfloor \overline{ASAP}(v) \rfloor, \lceil \overline{ALAP}(v) - 1 \rceil]$.

Definition 2.31. Execution interval analysis problem.
Consider an acyclic DFG, a set of modules M on which the operations in the DFG must be mapped, and a list of cycles C, forming the time constraint of the DFG. Find the OEI(v) for each operation $v \in V$.

In general, the problem of Definition 2.31 is not solvable in polynomial time, otherwise the accompanying scheduling problem would be solvable in polynomial time as well. Informally, the accompanying scheduling problem is to find a feasible schedule from the set Φ. The formal definition is given in Definition 4.1 in Chapter 4.

The proof for the fact that the problem of Definition 2.31 is not solvable in polynomial time is as follows. After solving the execution interval analysis problem of Definition 2.31 for the first time, one operation can be selected and scheduled within its OEI. According to the execution interval analysis, some schedule must exist with that schedule interval for the selected operation. The execution interval analysis can be run a second time, and a second operation can be selected and scheduled within its renewed OEI. This iteration can be continued until all operations are scheduled. Thus, if the analysis problem of Definition 2.31 could be solved in polynomial time, the scheduling problem of Definition 4.1 could be solved in polynomial time as well.

Because the OEIs cannot be calculated in polynomial time, the estimates $\overline{OEI}(v)$ are calculated for all operations $v \in V$, such that the properties in Definition 2.28 and 2.29 are fulfilled and $|\overline{OEI}(v)|$ is as small as possible. The size of the $\overline{OEI}$s in comparison to the COEIs determines the obtained pruning of the search space, i.e., the obtained domain reduction.

2.3 Trivial module sets

2.3.1 Overview

The basic execution interval analysis algorithm explained in the following subsections is meant for *trivial* module sets, see Definition 2.14. In Figure 2.2, the flow of the execution interval analysis is depicted. In this section, the important steps in the analysis will be mentioned, without the exact details. The purpose of this overview is to show where the different algorithms explained in the remainder of this chapter fit in the analysis flow. The purpose is not to explain what kind of actions the different algorithms perform in detail.

In case of a *trivial* module set, resource conflicts can occur for each separate module type $m \in T_M$, for a given DFG with time and resource constraints. To take these resource conflicts into account, two (related) sets of intervals for each module type are calculated.

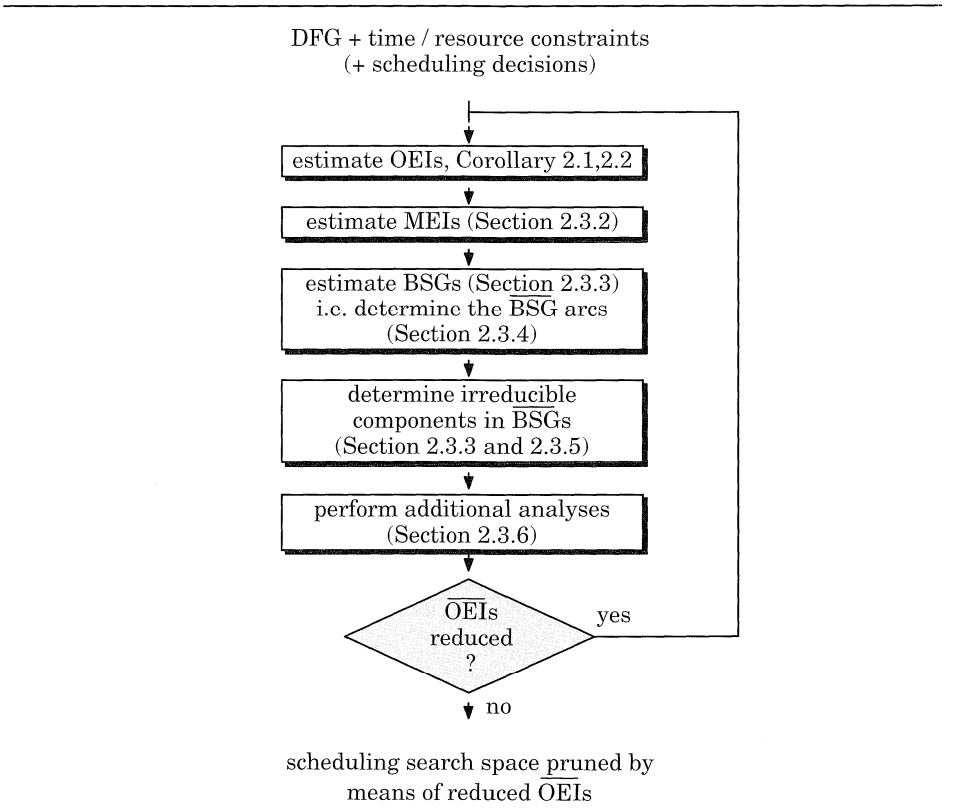


FIGURE 2.2. Flow of the execution interval analysis of Section 2.3.

First of all, for each module type $m \in T_M$, the operations $W_m \subseteq V$ are identified that can be executed on type m. So, the set W_m consists of operations that may have resource conflicts with each other. For any such operation an $\overline{OEI}$ is determined. In the first iteration of the algorithm of Figure 2.2, the $\overline{OEI}$s are all equal to the corresponding COEIs.

If a next iteration takes place, not all $\overline{OEI}$s are equal to the corresponding COEIs anymore, i.e., one or more $\overline{OEI}$s will be reduced. So, in the second and subsequent runs of the algorithm of Figure 2.2, the critical path analysis uses the latest determined $\overline{ASAP}$ and $\overline{ALAP}$ values and not the values determined by the Corollaries 2.1 and 2.2.

Secondly, in order to account for the resource constraints, a set with cardinality $|W_m|$ of so called *module execution intervals* (MEIs) is calculated. To be more precise, a set of conservative estimates $\overline{MEI}$s of those MEIs is determined. Within each MEI, some module of type m has to execute some operation $v \in W_m$. The exact definition of an MEI will be presented in the next section in Definition 2.33.

A necessary condition for an operation $v \in W_m$ to be executed on a module of type m is that its $\overline{OEI}$ must have sufficient overlap with some $\overline{MEI}$, i.e., there must be an overlap which is at least as long as the execution delay d(m). Consequently, if the number of cycles of a $\overline{MEI}$ is equal to d(m), then some module must start the execution of some operation $v \in W_m$ in the first cycle of that $\overline{MEI}$. In computing the $\overline{MEI}$s, not only the $\overline{OEI}$s are used, but also the bound on the number of modules of type m and the data introduction interval dii(m). It will be shown how additional pruning of the $\overline{OEI}$s can be derived from information contained in the $\overline{MEI}$s.

The OEIs and MEIs are used to construct a *bipartite schedule graph* (BSG), which is a bipartite graph with the sets of OEIs and MEIs establishing the required pair of vertex sets. The relation associated with the arcs of a BSG is defined by sufficient overlap of OEIs and MEIs. A necessary condition for the existence of a schedule is the existence of a *complete matching* in the BSG.

Bipartite graphs posses a unique canonical decomposition of their arc set in terms of so called *irreducible components* [Dulm63]. It is shown that arcs outside these components can be removed from the BSG without removing any feasible schedule from the solution space. As a consequence $\overline{OEI}$s can be narrowed, thus reducing the search space without loss of completeness.

2.3.2 Module execution intervals

Because resource conflicts occur for each module type separately in case of trivial module sets, recall Definition 2.14, we consider an arbitrary module type $m \in T_M$. Assume that $K_m \subseteq M$ is the set of modules of type m which can execute simultaneously, i.e., $K_m = \{k \in M \mid \xi(k) = m\}$. The set of operations $W_m = \{v \in V \mid \{\tau(v)\} = \mu^{-1}(\{m\})\}$ identifies those operations executable by modules of type m. In the sequel the index 'm' is dropped wherever possible. Thus, if nothing else is said, W is the set of operations to be executed by the set K.

Formal definition

For the definition of module execution intervals, assume a schedule $\phi \in \Phi$ is given. The schedule ϕ induces a notion of a *partial* order on the set W by ordering the set according to the value $\phi_1(v)$ for any $v \in W$. An ordering based on the values $\phi_1(v)$ for any $v \in W$ is not linear, because some operations might have equal start times.

However, as will be seen shortly, we need a linear ordering based on the start times of operations, to be able to define the MEIs. A linear ordering can be achieved by breaking ties on the start times arbitrarily, which is expressed by Definition 2.32. In that definition, an arbitrary linear ordering denoted by $\lll$ is introduced, which can be interpreted as an assignment of a unique number to every operation $v \in W$.

Definition 2.32. Linear ordering of operations induced by a schedule.
Let $\lll$ represent an arbitrary linear ordering on W. Given a schedule $\phi \in \Phi$, $<_\phi$ is a linear ordering relation defined by:

$$\forall_{\phi \in \Phi} \; \forall_{v,w \in W} : v <_\phi w \Leftrightarrow \big(\phi_1(v) < \phi_1(w)\big) \vee \big(\phi_1(v) = \phi_1(w) \wedge v \lll w\big).$$

If $\phi_1(v) = \phi_1(w)$ then $\phi_2(v) = \phi_2(w)$ as well, because we are considering trivial module sets for which the delay of all operations $v \in W$ are the same. As $<_\phi$ is a linear ordering it can be used to assign an integer value $i \in [1, |W|]$, to any operation $v \in W$. This is captured by defining a bijective function π_ϕ: $[1, |W|] \rightarrow W$. Thus $\pi_\phi(i)$ is the i^{th} operation in the linear order induced by the schedule ϕ, and i can be interpreted as the 'operation number' denoting the 'position' of operation $\pi_\phi(i)$ in the schedule ϕ. It is now possible to formally introduce the notion of a module execution interval.

Definition 2.33. Module execution interval MEI.
Consider the set of operations from W assigned to the value $i \in [1, |W|]$ over the set of all schedules Φ. Furthermore, let $\phi_1(i)$ and $\phi_2(i)$ be the short hand notation for $\phi_1(\pi_\phi(i))$ and $\phi_2(\pi_\phi(i))$ respectively, i.e., $\phi_1(i)$ and $\phi_2(i)$ are the start

and completion times of the i^{th} operation $v \in W$ in the schedule $\phi \in \Phi$. Then MEI(i) is defined by the following interval of clock cycles:

$$MEI(i) = \left[M_1(i),\ M_2(i)\right] = \left[\ \left\lfloor \min_{\phi \in \Phi} \phi_1(i) \right\rfloor ,\ \left\lceil \max_{\phi \in \Phi} \phi_2(i) - 1 \right\rceil \ \right].$$

Note that Definition 2.33 is equivalent to Definition 2.27. For any schedule $\phi \in \Phi$, the schedule interval of the i^{th} operation must be within the interval of clock cycles of MEI(i). Note that for different schedules, different operations can be the i^{th} operation. So, an MEI does not have to be equal to some OEI. Note also that the arbitrary ordering in Definition 2.32 does not have an impact on the definition of any MEI.

Furthermore, the number of MEIs is equal to the number of operations. The length of a MEI indicates a kind of 'freedom' for the set of modules K: within the interval, some module $k \in K$ must execute some operation $v \in W$ which can be the i^{th} operation in a schedule. If the number of cycles in some MEI(i), $i \in [1, |W|]$, equals the execution delay, i.e. $|MEI(i)| = \lceil d(m) \rceil$, then some module from K must start an execution in the first cycle of MEI(i). This notion of freedom is similar to the freedom expressed by the length of an OEI: if $|OEI(v)| = \lceil d(m) \rceil$ for any $v \in W$, then operation v lies on a 'critical path', i.e., operation v cannot be scheduled in more than one way.

Note also that a MEI is not related to a specific module if $|K| > 1$. Nor is it related to some specific operation $v \in W$ like the OEIs are. Furthermore, the MEIs are strictly ordered: the first time some module must execute an operation lies in MEI(1), the second time this must happen lies in MEI(2), etc. Consequently we can identify the MEIs and their ordering by their indexes.

Conservative estimates

Unfortunately, it is not possible to calculate the exact bounds of the MEIs in polynomial time, otherwise the existence of a feasible schedule under given time and resource constraints could be decided in polynomial time and the scheduling problem would be solvable in polynomial time. The reason is simple: if the set Φ is empty, then the MEIs are undefined and cannot be determined. If one can determine the MEIs, then the set Φ is not empty. So, one has to resort to estimates of the MEIs which do not limit the solution space. For this reason the estimates have to satisfy the following definition.

Definition 2.34. Conservative estimate of a module execution interval. The conservative estimate $\overline{MEI}(i) = [\overline{M_1}(i),\ \overline{M_2}(i)]$, $i \in [1, |W|]$, is an estimate of MEI(i), satisfying $\overline{M_1}(i) \leq M_1(i) \wedge \overline{M_2}(i) \geq M_2(i)$.

Note that Definition 2.34 is equivalent to Definition 2.30. It will be shown that the more accurate $\overline{MEI}$s are, the more accurate the $\overline{OEI}$s will be, i.e., the

better the pruning of the search space will be. For $\overline{MEI}(i)$, we have to determine, as accurately as possible, the earliest possible start and the latest possible completion time of the i^{th} operation of any schedule.

Let for a moment the set of operations W be transformed into a list of operations ordered by increasing $\overline{ASAP}$. If two operations have the same $\overline{ASAP}$, the tie is broken in an arbitrary way. In the first iteration of the algorithm, recall Figure 2.2, these $\overline{ASAP}$ values are equal to the CASAP values. Let W(i), $i \in [1, |W|]$, be the i^{th} operation in the ordering.

We can now derive the following two properties. Property 2.1 is a consequence of the fact that operations cannot start before their $\overline{ASAP}$ value. Property 2.2 states, that the maximum number of MEIs starting within dii(m) cycles is equal to the number of available modules of type $m \in T_M$. Recall that the number of modules is $|K|$, as K is the set of modules of type m.

Property 2.1. Start of MEI(i) cannot be smaller than the i^{th} $\overline{ASAP}$.

$$\forall_{i \in [1, |W|]} : \overline{M_1}(i)) \geq \lfloor \overline{ASAP}(W(i)) \rfloor .$$

Property 2.2. In any interval of dii(m) cycles, maximal $|K|$ MEIs can start.

$$\forall_{i \in [1, |W|] \mid i > |K|} : \overline{M_1}(i)) \geq \overline{M_1}(i - |K|) + dii(m).$$

Lemma 2.1.
Algorithm 2.1 determines $\overline{M_1}(i)$, $i \in [1, |W|]$, while satisfying the properties in Definition 2.34. The proof follows directly from Property 2.1 and 2.2.

The last cycles of the $\overline{MEI}$s, i.e. $\overline{M_2}(i)$, $i \in [1, |W|]$, can be determined similarly and is given in Algorithm 2.2. For that algorithm, the list of operations W must be ordered by decreasing $\overline{ALAP}$.

Algorithm 2.1. Calculation of $\overline{M_1}(i)$, $i \in [1, |W|]$.

for (i := 1 to min {|K|, |W|}) →
  $\overline{M_1}(i) := \lfloor \overline{ASAP}(W(i)) \rfloor$;
for (i := |K| + 1 to |W|) →
  $\overline{M_1}(i) := \max \{\lfloor \overline{ASAP}(W(i)) \rfloor, \overline{M_1}(i - |K|) + dii(m)\}$;

Algorithm 2.2. Calculation of $\overline{M_2}(i)$, $i \in [1, |W|]$.

for (i := |W| to max {1, |W| – |K| + 1)} →
  $\overline{M_2}(i) := \lceil \overline{ALAP}(W(i)) - 1 \rceil$;
for (i := |W| – |K| to 1) →
  $\overline{M_2}(i) := \min \{\lceil \overline{ALAP}(W(i)) - 1 \rceil, \overline{M_2}(i + |K|) - dii(m)\}$;

Example

An example of a DFG, some of its COEIs and the corresponding $\overline{\text{MEI}}$s are given in Figure 2.3 and 2.4. The number of cycles available for the DFG is ten, i.e., $|C| = 10$. Due to the precedence relations, and because the additions have a delay of one clock cycle, the multiplications cannot be executed in cycle zero or cycle nine. Figure 2.4 shows, therefore, that the COEIs of the multiplications are in the cycles one to eight. Recall that these COEIs are based on a critical path analysis under the assumption of unlimited resources (Corollary 2.1 and 2.2).

Figure 2.4 shows the $\overline{\text{MEI}}$s for the multipliers as well, which have been calculated by the Algorithms 2.1 and 2.2. So, the COEIs of the multiplications in Figure 2.3, the number of modules ($|K| = 5$), and the data introduction interval of the multipliers, dii(multiplier) = 2, have been taken into account when determining the $\overline{\text{MEI}}$s.

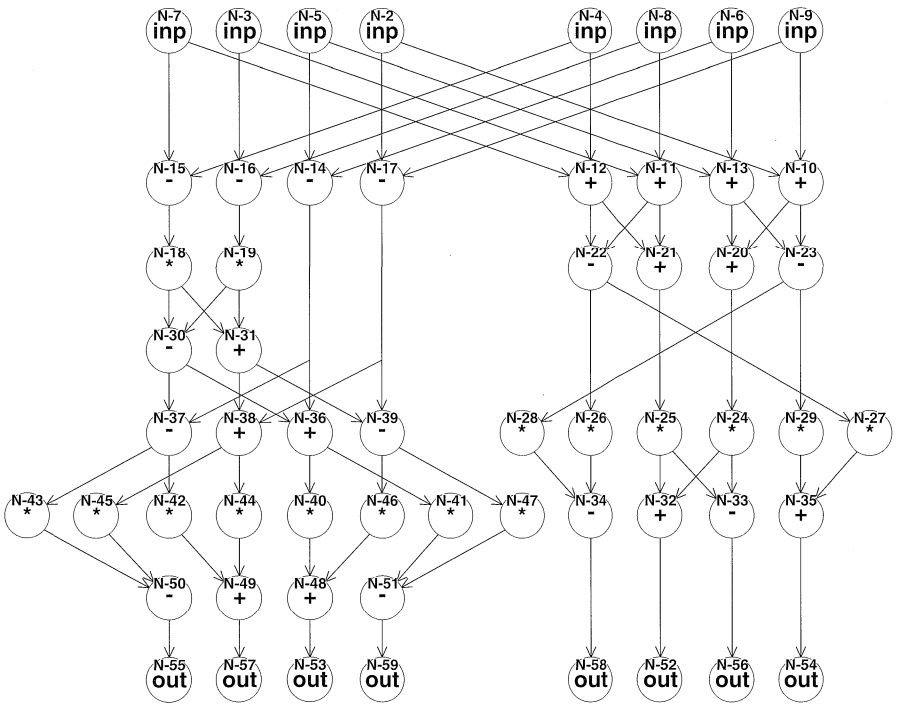


FIGURE 2.3. Acyclic data flow graph of the fast discrete cosine transform (FDCT) from [Mall90].

Because only two multiplications have cycle one in their COEI, only two $\overline{\text{MEI}}$s start in cycle 1. In cycle two, three other $\overline{\text{MEI}}$s can start, because the number of modules equals five. Because the dii equals two, cycle three is the first cycle in which the multipliers can start an operation for the sixth and seventh time, so the sixth and seventh $\overline{\text{MEI}}$s start in cycle three.

Only the eighth $\overline{\text{MEI}}$, and not the ninth and tenth $\overline{\text{MEI}}$, starts in cycle four. The reason is that there are only eight operations that can start between cycle one and four, see the COEIs in Figure 2.4. In this way, all the first cycles of the $\overline{\text{MEI}}$s can be derived.

The last five $\overline{\text{MEI}}$s end in cycle eight, because there are enough multiplications with a CALAP equal to nine. No more MEIs can end in cycle eight, because there are only five multipliers available. For similar reasons, the seventh to eleventh $\overline{\text{MEI}}$s end two cycles earlier, i.e., in cycle six, etc.

From Figure 2.4, some conclusions can already be made. For instance, the figure shows that at least three multipliers must *start* an execution in cycle five (in the ninth, tenth and eleventh $\overline{\text{MEI}}$), while a fourth multiplier must at least be busy executing some operation in that cycle (in the eighth $\overline{\text{MEI}}$).

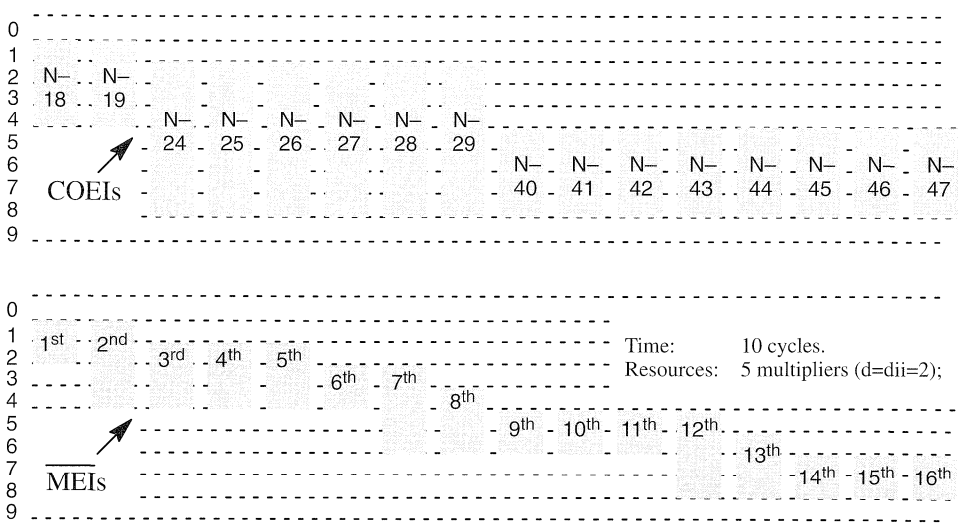


Figure 2.4. COEIs & $\overline{\text{MEI}}$s for the sixteen multiplications of FDCT given in Figure 2.3.

Improved accuracy of the $\overline{\text{MEI}}$s

We conclude this section with a few remarks on the accuracy of the $\overline{\text{MEI}}$s. Additional analyses may improve the determination of the $\overline{\text{MEI}}$s from Algorithm 2.1 and 2.2. In Figure 2.5a, the acyclic DFG representation of the fifth order wave digital filter (WDELF) from [DeWi85] is presented. If the fifth addition (operation 14) is executed, no other operation can be executed at the same time.

Such operations can be identified as follows. Consider the smallest number of cycles possible for a DFG. If under that condition the COEI of an operation $v \in V$ does not overlap with the COEI of any other operation, then operation v can never be executed simultaneously with any other operation.

In Section 2.3.4 it will be shown that the mentioned fifth addition must always be matched with MEI(5) of the additions. Assume that it takes one cycle to execute an addition and two cycles to execute a multiplication. Because a sixth addition can only start after multiplication 15 or 25, which succeed the fifth addition, has been executed, the difference between $M_1(5)$ and $M_1(6)$ of the additions, as well as the difference between $M_2(5)$ and $M_2(6)$, must at least be 3 cycles. Such analyses can improve the accuracy of the $\overline{\text{MEI}}$s and consequently the overall execution interval analysis.

2.3.3 Bipartite graph matching formulation

The execution interval analysis explained in this chapter is based on a bipartite graph matching formulation that incorporates both the OEIs and MEIs. In Figure 2.5, the DFG of the fifth order wave digital filter from [DeWi85] is presented with a time constraint of 21 cycles and a set of resources consisting of one multiplier and two adders. First the COEIs, i.e., the CASAP and CALAP values under the assumption of unlimited resources, are determined. For the multiplications these COEIs are given in Figure 2.5c.

Applying Algorithm 2.1 results in the following $\overline{\text{MEI}}$s. There are eight multiplications, so there are exactly eight $\overline{\text{MEI}}$s in which the applicable module must perform a multiplication. The multiplications have to be executed within sixteen cycles (between cycle four and cycle nineteen). As the multiplier has a dii and a delay of two cycles and the eight multiplications have to be scheduled within sixteen cycles, there are eight (in this case non–overlapping) $\overline{\text{MEI}}$s of two cycles to perform the multiplications. Figure 2.5c shows these $\overline{\text{MEI}}$s, which start every two cycles from cycle step four onward.

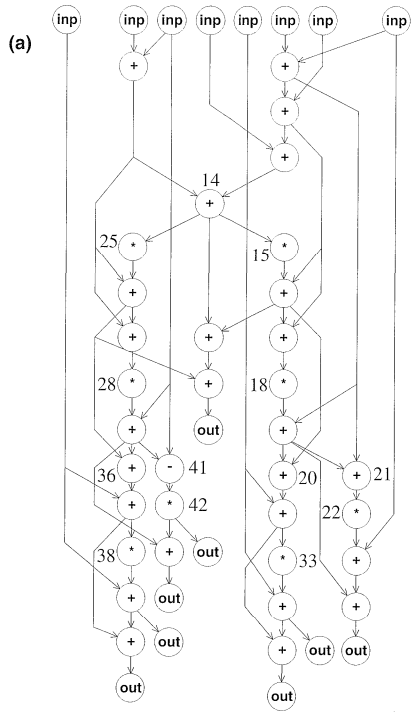
Formal definition

Based on the operation and module execution intervals it is possible to define the following bipartite graph, see also Figure 2.5c:

Definition 2.35. Bipartite schedule graph.
Consider the module type $m \in T_M$, the DFG represented by (V, E), and time & resource constraints. If $W = \{v \in V \mid \tau(v) \in \mu^{-1}(\{m\})\}$, then the bipartite schedule graph BSG(m) is a bipartite graph represented by a 2–tuple (N, A), where:

- $N = W \cup R$ is the set of vertices with $R = \{ i \mid i \in [1, |W|]\}$, $W \cap R = \emptyset$, and $|W| = |R|$; an MEI(i) is defined for each $i \in [1, |W|]$.
- $A \subseteq W \times R$ is a set of arcs; there is an arc $(v, i) \in A$, if and only if there is a schedule $\phi \in \Phi$ for which $v = \pi_\phi(i)$.

For convenience, operations in a BSG are sometimes said to be adjacent to MEIs instead of being adjacent to the index numbers.



(b) Time constraint: 21 cycles.
Resources: 1 multiplier (d=dii=2);
2 adders (d=dii=1).

(c) Initial $\overline{\text{BSG}}$ for the multiplications (direction of all arcs: left to right):

opera –tion	COEI	$\overline{\text{MEI}}$	#
15	[4,9]	[4,5]	1
25	[4,9]	[6,7]	2
18	[8,13]	[8,9]	3
28	[8,13]	[10,11]	4
22	[12,18]	[12,13]	5
33	[13,18]	[14,15]	6
38	[13,18]	[16,17]	7
42	[12,19]	[18,19]	8

(d)

	average freedom	fixed operations
before analysis	4.82	0 %
after analysis	1.53	55.9%

Figure 2.5. Fifth order wave digital filter, WDELF, from [DeWi85].

LEMMA 2.2.
For an arc $(v, i) \in A$ the 'overlap' between OEI(v) and MEI(i) is at least as large as the execution delay d(m). This means that $ALAP(v) - M_1(i) \geq d(m) \wedge M_2(i) - ASAP(v) + 1 \geq d(m)$. The proof follows directly from Definition 2.33.

THEOREM 2.1.
A correct determination of $\overline{OEI}$ is the following.

$$\forall_{v \in W} : \overline{OEI}(v) = \left[\min_{(v,i) \in A} M_1(i), \max_{(v,i) \in A} M_2(i) \right].$$

PROOF.
The proof follows directly from Definitions 2.27 and 2.35. ■

DEFINITION 2.36. Complete matching.
For a BSG represented by (N, A), with $N = W \cup R$, a matching is a subset of arcs in A that do not share any vertices, so every vertex in N is connected to at most one arc in a matching. A *complete* matching is a matching for which every vertex in N is connected to exactly one arc in the matching, i.e., a complete matching has cardinality $|W|$ and results in a bijection between W and R.

THEOREM 2.2.
For each feasible schedule $\phi \in \Phi$, a corresponding *complete* matching exists in each BSG(m), $m \in T_M$. If no complete matching exists for some BSG(m), $m \in T_M$, then the combination of time and resource constraints is infeasible and the set of feasible schedules Φ is empty.

PROOF.
Assume some feasible schedule $\phi \in \Phi$. From Definition 2.35 follows directly that the arcs $(\pi_\phi(i), i)$ must be part of BSG(m). Furthermore, these arcs constitute a complete matching. From that definition follows as well, that if there is no complete matching, then there is no feasible schedule. ■

Conservative estimates

Unfortunately, it is in general not possible to construct a BSG in polynomial time, e.g., it is not possible in general to calculate the MEIs in polynomial time. Because deriving the set of schedules is in general NP–hard, the set of arcs in a BSG cannot be determined in polynomial time as well. As in the case of the OEIs and MEIs, we can determine a conservative estimate of a BSG. Until now, the $\overline{OEI}$s were equal to the COEIs. It will be shown that with the conservative estimates of BSGs, we can possibly reduce the $\overline{OEI}$s without limiting the solution space, i.e., we can make the $\overline{OEI}$s more accurate, thus reducing the search space.

Definition 2.37. Conservative estimate of a bipartite schedule graph.
The conservative estimate $\overline{BSG}(m)$, $m \in T_M$, is represented by the 2–tuple $(N, \overline{A})$ and is an estimate of $BSG(m) = (N, A)$, where:

- $N = W \cup R$ is the set of vertices, see Definition 2.35;
- $\overline{A} \subseteq W \times R$ is a set of arcs satisfying $\overline{A} \supseteq A$.

Lemma 2.3.
Let for all $v \in W$ and $i \in R$, an arc (v, i) be an element of $\overline{A}$ if and only if $|COEI(v) \cap \overline{MEI}(i)| \geq d(m)$. This determination of the set $\overline{A}$ satisfies the property $\overline{A} \supseteq A$ mentioned in Definition 2.37, which follows directly from Lemma 2.2 and Definition 2.34.

Lemma 2.3 gives an initial and correct determination of the set of arcs in a $\overline{BSG}$. Together with the $\overline{MEI}$s and $\overline{OEI}$s, a conservative estimate of a BSG is obtained. Note that the $\overline{OEI}$s are equal to the corresponding COEIs in the first run of the execution interval analysis, recall the discussion in Section 2.3.1. This initial $\overline{BSG}$ is the starting point for the analysis explained in the remainder of this section.

Not all complete matchings in a $\overline{BSG}$ have to represent feasible schedules, because the calculated $\overline{BSG}$s are just conservative estimates of the BSGs. The exact difference lies in the fact that the dependency relations of the DFG are not fully incorporated in the arcs and the $\overline{MEI}$s of the $\overline{BSG}$s. However, a possible $\overline{OEI}$ reduction can be achieved by identifying those arcs that do *not* belong to any complete matching. For this purpose, the so called irreducible components [Dulm63] are identified; see Theorem 2.3 and Definition 2.38.

Theorem 2.3.
The arcs in a $\overline{BSG}$ that do not belong to any complete matching can be removed from $\overline{A}$ without violating the property $\overline{A} \supseteq A$ mentioned in Definition 2.37. The proof follows directly from Theorem 2.2: none of the feasible schedules contains any of these arcs in their corresponding complete matching.

Definition 2.38. Irreducible components in a BSG.
Consider the set of complete matchings of a bipartite graph BSG, and let A' be the union of arcs from that set. The *irreducible components* of a BSG are the connected subgraphs induced by the set A'.

The irreducible components, and therefore the arcs that can be removed from a $\overline{BSG}$, can in the general case be derived in $O(|N|^{1/2} \cdot |\overline{A}|)$ [Sang76]. Note that Section 2.3.5 will present a problem specific algorithm that determines the irreducible components in $O(|N| \cdot \log |N|)$. The removal of an BSG arc can reduce the $\overline{OEI}$ of the operation connected to that arc, as is shown in Theorem 2.4.

THEOREM 2.4.
Let $\overline{A}$ be the set of arcs of $\overline{BSG}(m)$, $m \in T_M$, after removing the arcs that do not belong to any irreducible component. Then the following determinations of $\overline{ASAP}(v)$ and $\overline{ALAP}(v)$ for an operation $v \in W$ are correct, i.e., do not limit the set of feasible schedules. The proof follows directly from Theorem 2.1 and 2.3 and the properties in the Definitions 2.28 and 2.29: the schedule interval of an operation has to be within the interval of clock cycles of its adjacent $\overline{MEI}$s.

$$\overline{ASAP}(v) = \max\left\{CASAP(v), \min_{(v,i) \in \overline{A}} \overline{M_1}(i)\right\}.$$

$$\overline{ALAP}(v) = \min\left\{CALAP(v), \max_{(v,i) \in \overline{A}} \overline{M_2}(i) + 1\right\}.$$

The calculations above show, that the number of clock cycles within an $\overline{OEI}(v)$ of any operation v depends on the clock cycle intervals covered by the adjacent $\overline{MEI}$s. Therefore, the more accurate the adjacent $\overline{MEI}$s in a $\overline{BSG}$ are determined, i.e., the smaller the adjacent $\overline{MEI}$s are, the more accurate the estimated $\overline{OEI}$s can be.

If an $\overline{OEI}$ is reduced, a new run of the execution interval analysis can be started for a further reduction of the $\overline{OEI}$s, recall Figure 2.2. A second or subsequent run of the analysis starts with a critical path analysis, using the latest determined $\overline{ASAP}$ and $\overline{ALAP}$ values and not the values determined by the Corollaries 2.1 and 2.2. Because in a new run the dependency relations of the DFG are reconsidered, a reduction of an $\overline{OEI}$ in the $\overline{BSG}$ of one module type can also lead to the reduction of $\overline{OEI}$s in the $\overline{BSG}$s of other module types.

Example

Figure 2.5c shows that only multiplication 42 can be scheduled in $\overline{MEI}(8)$, because it's the only operation adjacent to $\overline{MEI}(8)$. This is another way to see that the arcs from multiplication 42 incident with $\overline{MEI}(5)$, $\overline{MEI}(6)$ and $\overline{MEI}(7)$ do not belong to any complete matching and can be removed. Figure 2.5c also shows that only the operations 15 and 25 can be scheduled in the first two $\overline{MEI}$s, that consequently only the operations 18 and 28 can be scheduled in $\overline{MEI}(3)$ and $\overline{MEI}(4)$, and that operations 33 and 38 must be scheduled in $\overline{MEI}(6)$ and $\overline{MEI}(7)$. So the arcs (15, 3), (25, 3), (18, 5), (28, 5), (22, 6) and (22, 7) do not belong to any irreducible component and can be removed as well.

Multiplication 22 remains connected to $\overline{\text{MEI}}(5)$ only, and consequently it must be scheduled in the first two cycles of its COEI, i.e., for multiplication 22 the schedule must be ϕ_1 = CASAP and ϕ_2 = CASAP + 2. As the dependency relations of the DFG must be preserved, the operations 15 and 18, which are predecessors in the critical path of multiplication 22, become 'fixed' to the first two cycles of their COEIs as well in the next run of the algorithm.

After three runs of the analysis only the bold arcs in Figure 2.5c remain in the $\overline{\text{BSG}}$, resulting in maximally reduced $\overline{\text{OEI}}$s: all the multiplications can be scheduled in only one way. This example shows the capabilities of the approach to achieve a considerable domain reduction, i.e., to prune the scheduling search space effectively.

2.3.4 $\overline{\text{BSG}}$ arcs

The dependency relations in a DFG must be satisfied in any feasible schedule. Consider a $\overline{\text{BSG}}$, an operation in that $\overline{\text{BSG}}$, and its pre– and successors in the DFG which are also elements of the same $\overline{\text{BSG}}$. Let a predecessor of a MEI be a MEI with a lower index number, and a successor of a MEI one with a higher index number. From Definition 2.33, it follows that in any schedule an operation can only be matched with a MEI if its predecessors in the BSG can be matched with preceding MEIs and its successors with succeeding MEIs. So, the following proposition must hold; see also Figure 2.6:

$$\forall_{\phi \in \Phi} \; \forall_{\{v,w \in \text{BSG}(m) \mid (v,w) \in E^*\}} : \overset{-1}{\pi}_{\phi}(v) < \overset{-1}{\pi}_{\phi}(w).$$

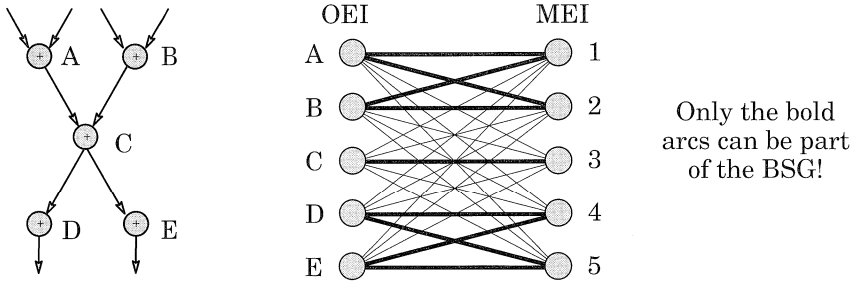


FIGURE 2.6. Example DFG and corresponding BSG.

This condition was neither considered in Lemma 2.3 nor in the execution interval analysis until now, i.e., the condition was not considered in the determination of the arcs of $\overline{BSG}(m)$. $\overline{\pi}^{-1}_{\phi}(v)$ also identifies the number of the MEI to which v is matched in a schedule ϕ, i.e., $(v, \overline{\pi}^{-1}_{\phi}(v)) \in A$ is an element of the complete matching representing $\phi \in \Phi$. Let $preds^*(v)$ be the list of predecessors of v in BSG(m), and let $preds^*(v)$ be ordered by increasing $\overline{ASAP}$. If two operations have the same $\overline{ASAP}$, the tie is broken in an arbitrary way.

Let $num(v) \in [1, |W|]$ be a conservative estimate of the first MEI that can be adjacent to operation v in the BSG. Then Algorithm 2.3 can lead to a smaller set of arcs than the set constructed in Lemma 2.3, thereby still satisfying the requirements of Definition 2.37. The Algorithm 2.3 runs in linear time and can lead to a smaller set of arcs, because it takes the proposition above into account.

Note that such kind of analysis can also be applied to determine the first cycle in which all the predecessors of an operation v can be completed, independent of the fact whether these predecessors are elements of the same BSG as operation v.

Similar algorithms based on the successors of an operation can be applied, for instance to determine the last $\overline{MEI}$ to which an operation is adjacent, see Algorithm 2.4. In that algorithm, $succs^*(v)$ is the list of successors of v in BSG(m), and $succs^*(v)$ has to be ordered by decreasing $\overline{ALAP}$. The number of arcs in a $\overline{BSG}$ can decrease due to these calculations, which in turn can lead to more accurate, i.e., smaller, $\overline{OEI}$s.

ALGORITHM 2.3. Calculation of the first $\overline{MEI}$ for operation v.

```
num(v) := 1;
for_all (predecessor ∈ preds*(v)) →
    num(v) := max {num(v), num(predecessor)} + 1;
while (⌊ASAP(v)⌋ > M2(num(v)) – d(v) + 1)
    num(v) := num(v) + 1;
```

ALGORITHM 2.4. Calculation of the last $\overline{MEI}$ for operation v.

```
num(v) := |W|;
for_all (successor ∈ succs*(v)) →
    num(v) := min {num(v), num(successor)} – 1;
while (⌈ALAP(v)⌉ < M1(num(v)) + d(v))
    num(v) := num(v) – 1;
```

An additional analysis for the determination of the $\overline{BSG}$ arcs can also help to improve the overall result of the execution interval analysis as follows. Consider an operation $v \in V$ that is fixed to a schedule interval, i.e., for which $\lceil d_{min}(v) \rceil = |\overline{OEI}(v)|$. In that case, if there is an arc $(v, i) \in A$ for which $\overline{MEI}(i) = \overline{OEI}(v)$, then operation v can be matched to $\overline{MEI}(i)$ without limiting the solution space of any operation. All arcs connected to these two vertices in the $\overline{BSG}$ except for the arc between them can be removed. This leads, in some cases, to new and smaller irreducible components, thus inducing a more accurate execution interval analysis.

2.3.5 Run time complexity

In the early approach of [Timm93b], the total complexity of determining the $\overline{BSG}$s and their irreducible components is $O(|V|^2 + |V|^{1/2} \cdot |\overline{A}|)$, with $|\overline{A}| \leq |V|^2$. It takes $O(|V|+|E|)$ with $|E| \leq |V|^2$ to calculate the $\overline{OEI}$s. The $\overline{MEI}$ calculation runs in $O(|V| \cdot \log |V|)$ due to the sorting of the operations for Algorithm 2.1. The determination of the arcs of a $\overline{BSG}$ runs in $O(|V|^2)$: the number of left and right vertices is $O(|V|)$, so in the worst case, $O(|V|^2)$ tests have to be performed to check whether an operation can be scheduled within a $\overline{MEI}$. The additional analysis at the end of the previous subsection also has a worst case complexity of $O(|V|^2)$.

The term $O(|V|^{1/2} \cdot |\overline{A}|)$ is due to the derivation of the irreducible components which consists of two steps [Sang76]. In the first step, an arbitrary complete matching is determined which takes $O(|V|^{1/2} \cdot |\overline{A}|)$ in the general case [Hopc73]. In Figure 2.7a, the bold arcs represent such a complete matching for the initial bipartite schedule graph of Figure 2.5c.

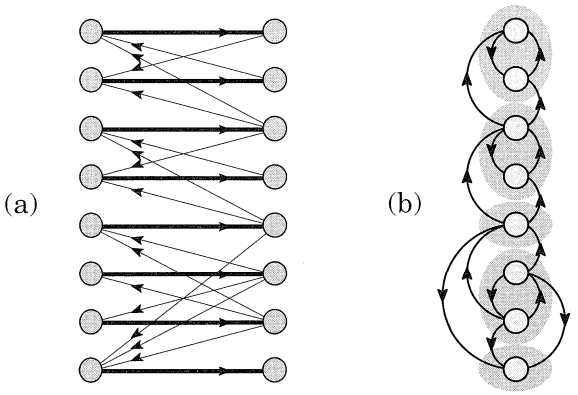


FIGURE 2.7. Determination of irreducible components.

In the second step the direction of the arcs in the $\overline{BSG}$ is changed as follows: the arcs in the matching are given a direction opposite to the direction of the arcs not in the matching. In Figure 2.7a, the arcs of the selected complete matching are directed from left to right, and the other arcs are directed from right to left.

In the third step, we substitute each subgraph induced by an arc in the matching by a new vertex, see Figure 2.7b. The arcs that were previously connected to such a subgraph remain connected to the new vertex and remain their direction as well. The arcs that are inside the strongly connected components of this new graph form the arcs of the corresponding irreducible components of the original $\overline{BSG}$, together with the arcs of the previously determined complete matching [Sang76]. In Figure 2.7b, the strongly connected components are denoted by the gray ellipses. Strongly connected components can be determined in $O(|V| + |\overline{A}|)$ using depth first search.

Run time complexity improvements

Although $O(|V|^{1/2} \cdot |\overline{A}|)$ is, in general, the lowest run time complexity to find a maximal matching in a bipartite graph [Hopc73], the step of finding an initial complete matching in a BSG can be done in $O(|V| \cdot \log |V|)$. There are two reasons why we can find a complete matching more efficiently than the algorithm of [Hopc73]. First of all, we are only interested in a complete matching, not in a maximal matching. Secondly, we use problem specific information to find the complete matching efficiently, as is shown below.

Consider a bipartite graph $\overline{BSG}(m) = (N, \overline{A})$, $m \in T_M$. Let W be the list of operations in $\overline{BSG}(m)$, let W be ordered by increasing $\overline{ASAP}$, and let W(i) be the i^{th} operation in that order. If two operations have the same $\overline{ASAP}$, the tie is broken in an arbitrary way. Let L be an initially empty list of operations which is kept ordered by increasing $\overline{ALAP}$. Again, if two operations have the same $\overline{ALAP}$, the tie is broken in an arbitrary way. Let BM be the (initially empty) bipartite matching.

THEOREM 2.5.
Algorithm 2.5 finds a complete matching if one exists.

PROOF.
The first operation $v_1 \in L$ to be selected for $\overline{MEI}(1)$ is the operation with the smallest $\overline{ALAP}$ that can possibly be scheduled within $\overline{MEI}(1)$. Suppose a complete matching exists in which $\overline{MEI}(1)$ is not matched with v_1 but with $v_j \in W$, and in which v_1 is matched with $\overline{MEI}(j)$, $j \in \{2, 3, ..., |W|\}$. Then there also exists a complete matching in which v_1 is matched with $\overline{MEI}(1)$ and v_j with $\overline{MEI}(j)$, as is shown below.

Because v_1 can start in $\overline{MEI}(1)$ and $\overline{MEI}(j)$, and because $\overline{ALAP}(v_1) \leq \overline{ALAP}(v_j)$, i.e., v_j is not a predecessor of v_1 in the DFG, $\overline{M}_1(j) \geq \overline{M}_1(1)$, and v_j can start in $\overline{MEI}(1)$, there is also an arc between v_j and $\overline{MEI}(j)$ in the $\overline{BSG}$. So if there exists a complete matching, then there also exists a complete matching which starts with a greedy choice. Once the greedy choice of the first matching has been made, the problem reduces to finding a complete matching for the remaining vertices. By induction the greedy choice at every step of the algorithm produces a complete matching if one exists. ■

ALGORITHM 2.5. Determining a complete matching.

```
j := 1; L := ∅;
— note that W is ordered by increasing ASAP —
for (i := 1 to |W|) →
    while (j ≤ |W| ∧ (W(j), i) ∈ Ā) →
        insert W(j) in L, while keeping L ordered by increasing ALAP;
        j := j + 1;
    if ((L(1), i) ∈ Ā) →
        add (L(1), i) to BM;
        delete L(1) from L;
    else
        no complete matching possible;
```

The $O(|V| \cdot \log |V|)$ run time complexity of the algorithm above is due to the ordering of the lists of operations. With this algorithm, the total run time complexity of determining the bipartite schedule graphs and their irreducible components becomes $O(|V|^2)$ instead of $O(|V|^2 + |V|^{1/2} \cdot |\bar{A}|)$. In practice, the average run time complexity is about linear in the size of the DFG, i.e., linear in the number of vertices and arcs in a DFG.

A new run of the execution interval analysis is started only if a reduction of some $\overline{OEI}$ has taken place, recall Figure 2.2. In the worst case just one cycle is removed from the $\overline{OEI}$ of just one operation. The union of all $\overline{OEI}$s contains $O(|C| \cdot |V|)$ cycles, so the number of runs is $O(|C| \cdot |V|)$. However, in practice the algorithm already stops after a few runs.

2.3.6 Additional analyses

So far different steps involving the bipartite graph matching formulation have been discussed. There are some other types of analyses which can also improve the estimation of the OEIs, and which are *not* based on $\overline{BSG}$s.

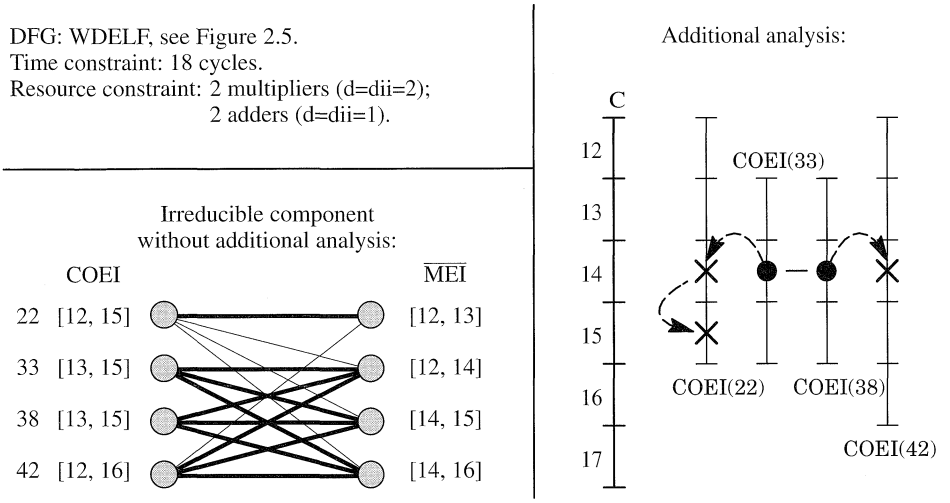


FIGURE 2.8. Additional analysis.

An example of such an additional analysis is based on the examination of individual clock cycles, and is given in Figure 2.8. The multiplications 33 and 38 from WDELF, see Figure 2.5, must occupy both multipliers in cycle fourteen. This is denoted by the black dots. The multiplications 22 and 42 can therefore not be active in that cycle, which is denoted by the bold Xs in cycle fourteen. This analysis is somewhat comparable to other consistency checks in the field of constraint satisfaction, see [Nuijt94].

Because multiplication 22 cannot be active in cycle fourteen, it can also not be active in cycle fifteen, as a multiplication takes two cycles and non preemptive scheduling is assumed. Therefore, the $\overline{\text{OEI}}$ of multiplication 22 equals [12, 13], i.e., multiplication 22 and all its predecessors in the critical path must start their execution in the first cycle of their COEIs. Figure 2.8 shows that this is not detected by the BSG analysis described so far; only the bold arcs can be present in the BSG.

In general, if an $\overline{\text{OEI}}$ is small enough, then independent of the chosen schedule, the corresponding operation will always occupy a module in one and the same specific set of cycles. If in some cycle the number of such operations is equal to the number of modules, then for any schedule $\phi \in \Phi$ the complementary set of operations is prevented from being scheduled in that cycle. Such a cycle can consequently be excluded from the $\overline{\text{OEI}}$s of the operations of this complementary set.

2.4 Unrestricted module sets

2.4.1 Bipartite schedule graph definition

In case of an *unrestricted* module set, many–to–many mappings between operation types and module types are allowed; see Definition 2.15. Such a module set can contain modules with a large variety in delay, area and so on for one operation type. If an unrestricted module set is used, several aspects of the bipartite graph matching formulation are affected. In this section, the main differences with *trivial* module sets are highlighted.

First of all, $\overline{\text{BSG}}$s can be constructed for each set of operations that can have resource conflicts. The previous section showed that, in case of a trivial module library, $\overline{\text{BSG}}$s are therefore constructed for each module type separately. For an unrestricted module set, $\overline{\text{BSG}}$s can be constructed for each set of operation types that can be mapped on the same module type. For instance, if a module set consists of an adder, a subtracter and an adder/subtracter, then three $\overline{\text{BSG}}$s can be constructed: one containing the additions, one containing the subtractions and one containing both types of operations.

2.4.2 Module execution intervals

The data introduction intervals and execution delays of the modules for a certain (set of) operation type(s) can differ. If so, some adaptations can be made in the calculation of the $\overline{\text{MEI}}$s to determine them more accurately than Algorithm 2.1 does. To take the different data introduction intervals (diis) into account when estimating the MEIs, the specific modules are considered during the calculation, as is shown below.

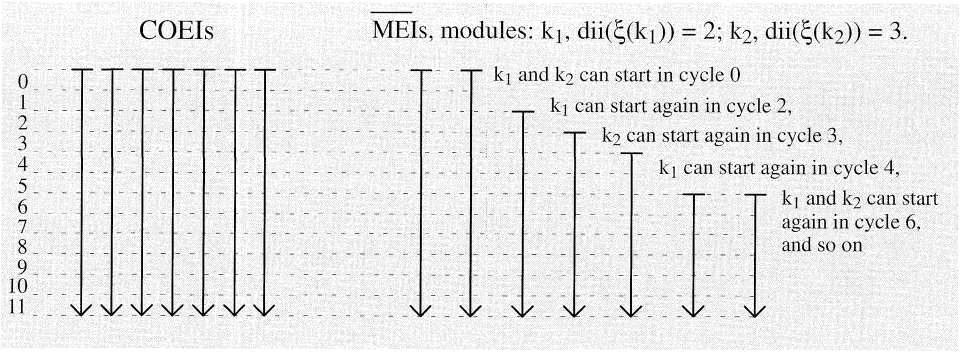


FIGURE 2.9. 1st example of $\overline{\text{MEI}}$s with an unrestricted module set.

Examples

In Figure 2.9, a module set with two modules k_1 and k_2, $dii(\xi(k_1)) = 2$ and $dii(\xi(k_2)) = 3$, and seven $\overline{OEI}$s of operations that can be executed by k_1 and k_2 are given. If these two modules can be active all the time, then k_1 can start every two cycles and k_2 can start every three cycles. So within six cycles five starts of $\overline{MEI}$s can take place. Applying Algorithm 2.1 with the smallest dii of two cycles would lead to six starts in six cycles, which is less accurate.

In Figure 2.10, the same module set is used as in Figure 2.9, but now the modules cannot be active all the time. In case k_1 would start in cycle zero, the $\overline{M_1}$s are equal to the cycles zero, one, two and four respectively. In case k_2 would start in cycle zero, the $\overline{M_1}$s are equal to the cycles zero, one, three and three respectively. The earliest possible $\overline{M_1}(3)$ occurs when k_1 starts in cycle zero, while the earliest possible $\overline{M_1}(4)$ occurs when k_2 starts in cycle zero. Because of the possible swaps between slower and faster modules, a greedy choice of a module for a certain $\overline{MEI}$ is not correct when deriving the earliest possible first cycle for each individual $\overline{MEI}$.

Determination of $\overline{MEI}$s revisited

To ensure that the approach determines the $\overline{MEI}$s in case of different data introduction intervals as accurately as possible, but without limiting the solution space, Algorithm 2.1 is changed into Algorithm 2.6 as follows.

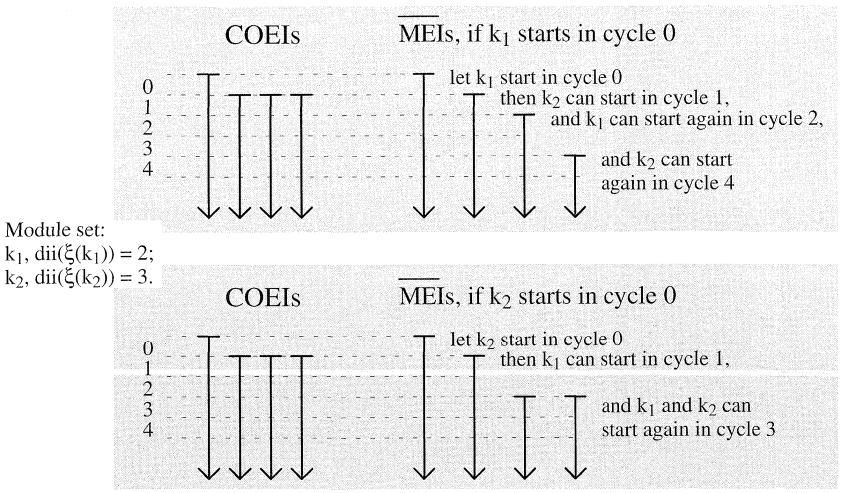


FIGURE 2.10. 2nd example of $\overline{MEI}$s with an unrestricted module set.

First, the elements of the module list K are sorted by increasing dii, with K(i), $1 \leq i \leq |K|$, the i^{th} module in that order. Let the list of operations that can be executed by modules from K be ordered by increasing $\overline{ASAP}$. If two operations have the same $\overline{ASAP}$, the tie is broken in an arbitrary way. Let W(i), $i \in [1, |W|]$, be the i^{th} operation in that order. Let newstart(k) of a module $k \in K$ denote the first cycle in which the module k can start to execute a new operation. Then the function 'select' returns the module with the smallest dii among the earliest available modules.

Furthermore, newstart(k) is calculated incorporating possible swaps between modules with different diis, as is shown below. In the function 'update' the modules are traversed with increasing dii to check whether a swap against the selected module k is possible and leads to earlier $\overline{M_1}$s. If during the traversal the module k is encountered, then there is no swap possible with a module which has a smaller dii that leads to earlier $\overline{M_1}$s.

Until that moment it is checked whether the modules could be swapped, which is true if module k is idle since the previous start of the other module $l \in K$, and whether this swap leads to earlier $\overline{M_1}$s. If this is true, the selected module k is said to be ready for a new execution after dii(l) instead of dii(k) cycles, with dii(l) < dii(k). In this way all possible swaps are incorporated, and the earliest possible $\overline{M_1}$s are calculated correctly, i.e., without limiting the solution space of a scheduler. So, applying Algorithm 2.6 to the examples of Figure 2.9 and 2.10 will lead to a correct determination of the $\overline{M_1}$s.

The $\overline{M_2}$s of the $\overline{MEI}$s whose $\overline{M_1}$s are calculated by Algorithm 2.6 *cannot* be calculated similarly. This is due to the linear ordering of scheduled operations, recall Definition 2.32. In case of a *trivial* module set, the ordering is the same if it is based on the ϕ_2s instead of ϕ_1s. In case of an *unrestricted* module set, the two orderings can be different, because of the possibly different execution delays of the operations within a $\overline{BSG}$.

Therefore, if the $\overline{M_1}$s of the $\overline{MEI}$s are calculated by Algorithm 2.6, then the $\overline{M_2}$s must be calculated as follows, in order to obtain correct $\overline{MEI}$s. Let $W_i \subseteq W$ be the set of operations that can possibly be matched with $\overline{MEI}(i)$. Then the following estimate is correct, i.e., conservative:

$$\overline{M_2}(i) = \max_{v \in W_i} \lceil \overline{ALAP}(v) - 1 \rceil .$$

In order to improve the accuracy of the overall execution interval analysis, it is of course possible to construct a second set of $\overline{BSG}$s in which the $\overline{M_2}$s are calculated by an algorithm similar to Algorithm 2.6 and the $\overline{M_1}$s are determined by:

$$\overline{M_1}(i) = \min_{v \in W_i} \lfloor \overline{ASAP}(v) \rfloor .$$

ALGORITHM 2.6. calculating $\overline{M_1}$s in the case of differing diis.

```
for (i := 1 to |K|) →
    newstart(K(i)) := 0;
for (i := 1 to |W|) →
    k := select(⌊ASAP(W[i])⌋);
    M1(i) := max {⌊ASAP(W(i))⌋, newstart(k)};
    newstart(k) := update(k, M1(i));


select (real asap)
{
    choice := K(1);
    for (i := 1 to |K|) →
        if (newstart(K(i)) ≤ asap) →
            return K(i);
        if (newstart(K(i)) < newstart(choice)) →
            choice := K(i);
    return choice;
}


update (module k, int first)
{
    for (i := 1 to |K|) →
        if (k = K(i)) →
            return first + dii(ξ(k));
        prev := previous time module K(i) started;
        if (newstart(k) ≤ prev < first ∧
           prev + dii(ξ(k)) ≤ first + dii(ξ(K(i)))) →
            return first + dii(ξ(K(i)));
}
```

Furthermore, in case of a *trivial* module set, the earliest possible $\phi_2(v)$ of an operation v is the smallest $\overline{M_1}$ of the adjacent $\overline{MEI}$s plus the execution delay of the corresponding module type. [Timm94] shows that, in case of an *unrestricted* module set, the earliest possible ϕ_2s can possibly be calculated more accurately than just using the smallest execution delay within the set of applicable modules. This may also have an impact on the accuracy of the $\overline{OEI}$s, because all successors of an operation v cannot start earlier than $\phi_2(v)$.

2.5 Experiments and results

For the example of Figure 2.5, the execution interval analysis deletes all arcs in the $\overline{BSG}$ except for the bold ones. All the multiplications can only be scheduled in one way, i.e., they are *fixed*, as are a number of additions: 55.9% of all the operations are already fixed before scheduling. A measure for the search space of a scheduler is expressed by the following definitions of freedom.

Definition 2.39. freedom, i.e., slack, of an operation.
The freedom F(v) of an operation $v \in V$ is defined as the number of cycles within its $\overline{OEI}$ minus the minimal delay of the operation within the set of modules, i.e. $F(v) = \overline{ALAP}(v) - \overline{ASAP}(v) - d_{min}(v)$.

Definition 2.40. average freedom AF(V) of the set of operations V in a DFG.

$$AF(V) = \sum_{v \in V} \frac{F(v)}{|V|} = \sum_{v \in V} \frac{\overline{ALAP}(v) - \overline{ASAP}(v) - d_{min}(v)}{|V|}.$$

The average freedom decreases from 4.82 to 1.53 in the example of Figure 2.5, by using the algorithms depicted in Figure 2.2.

Some results of the execution interval analysis implemented in the NEAT system are given in Table 2.1, in which the optimal functional area versus time (AT) points have been given for WDELF, see Figure 2.5a. The way these AT points are obtained is explained in the next chapter. These execution interval analyses before scheduling were run within a few tenths of seconds on an HP9000/755 workstation.

Table 2.1. Execution interval analysis results for WDELF.

$\|C\|$	# *multi-pliers (d=dii=2)*	# *fast adders (d=1)*	# *slow adders (d=dii=2)*	*average freedom**	*fixed operations**
17	3	3		0.38 (0.82)	88.24%(70.59%)
18	2	2		1.38 (1.82)	32.35% (0%)
21	1	1	1	1.29 (4.82)	64.71% (0%)
28	1	1		9.29 (11.82)	2.94% (0%)
54	1		1	21.06 (37.82)	2.94% (0%)

* The values in parentheses denote the results for the classical CASAP / CALAP analysis under the assumption of unlimited resources.

The table shows that, for all these optimal AT points, the execution interval analysis results in a reduction of the estimated OEIs. As the number of clock cycles gets larger, the reductions of the $\overline{\text{OEI}}$s become less significant in comparison with the COEIs. Figure 2.5a shows that this is due to the fact that WDELF has a lot of freedom, in case there are only a few modules but many clock cycles.

2.6 Discussion

It is clear from the previous section that the execution interval analysis can decrease the freedom of operations, and it can therefore improve the results of schedulers considerably by pruning their search space. However, the run time efficiency of schedulers can be improved as well: IP schedulers for instance need fewer variables and constraints. The number of variables and constraints is directly related to the amount of freedom operations have, so if the average freedom decreases, the number of variables and constraints will also decrease. Any IP scheduling model, e.g., the node packing model of [Gebo92], is still valid after the execution interval analysis, so a decrease in the number of variables and constraints will lead to run time efficiency improvements.

The importance of the execution interval analysis can also be pointed out by a greedy list scheduler that schedules a set of operations from the ready list in some cycle step if and only if:

1. the predecessors of all the selected operations are scheduled;
2. the current cycle is within the current $\overline{\text{OEI}}$ of all the selected operations;
3. the execution interval analysis, after scheduling the selected operations, does not detect infeasibility.

Such a greedy list scheduler can never fail on the examples of Table 2.1, while normal list schedulers without the analysis can fail, depending on their priority function. Any priority function will always lead to a feasible schedule within the cycle budget if the analysis is applied. So the quality of the results of heuristic schedulers can be improved considerably due to the presented analysis.

The average values of freedom and the percentages of fixed operations in Table 2.1 are values *before* scheduling. If the execution interval analysis is iteratively continued during scheduling, these values can change considerably from step to step. For instance, in the case of eighteen cycle steps, two multipliers and two fast adders, the additions 20, 21, 36 and 41 in Figure 2.5a are in the ready list in cycle step eleven. At this point, the average freedom is 0.65 and 56% of the operations are fixed.

The analysis at this point shows that addition 21 must be scheduled in cycle step eleven, while any of the three remaining additions can also be selected. After the selection of any of them, all but one operation, i.e., 97% of the operations, are fixed by the execution interval analysis, while the average freedom decreases to 0.03. This example shows that the iterative application of the execution interval analysis between the various steps of a scheduling procedure can have a large influence on the quality of the result as well.

Chapter 3 Lower Bound Analyses

3.1 Introduction

In this chapter we present a unified approach of lower bound functional area and cycle budget estimations to predict the area vs. timing characteristics of designs. The ability to predict these characteristics without actually implementing them is important in producing high quality designs in a reasonable time, and is therefore an important part of an (interactive) system design environment [Fleu93]. If a design is part of a larger system, then it is important to select one or more 'good' points from the design space without synthesizing all possibilities. So an accurate lower bound area vs. timing (AT) curve can speed up the design space exploration and can also be used to evaluate the quality of a design; see also Section 1.3.

Most architectural synthesis schedulers are only capable of mapping an operation to one specific module type. However, to ensure a full design space exploration, a synthesis system should select freely from an unrestricted library containing module types with a large variety in delay, area and so on. The lower bound functional area estimation presented in this chapter is very useful as a starting point for time constrained scheduling approaches with such a capability, as will be shown in Chapter 4.

Both the area and cycle budget estimations are mainly based on relaxing the precedence constraints in a DFG. Section 3.2 shows that this is also the case for related work on this subject. Section 3.3 deals with a lower bound cycle budget estimation approach based on the execution interval analysis of Chapter 2. Section 3.4 deals with the lower bound functional area estimation. In case of unrestricted module libraries, this lower bound area estimation is modelled as a mixed integer linear programming (MILP) problem.

The lower bound area estimation selects a module set with minimal area. If a module set turns out to be infeasible, or a scheduler cannot find a correct schedule within short run times, then a new module set must be selected. Section 3.5 explains how the next module set with respect to functional area can be selected, i.e., how to select the smallest possible, but different, module set with an equal or larger functional area. In that section we also show that the cycle budget estimation of Section 3.3 can be used as a partial check for the feasibility of a module set.

Previous approaches, like [Chen91], [Jain92], [Shar93], [Timm93a], [Dalk94], [Chau94], [Holm94], determine a lower bound estimate of the minimal functional area for each possible number of cycle steps separately. For a complete AT curve the number of these estimations and hence the CPU time can become rather large. Unrestricted module libraries with a large range of delays for each operation type can increase the design space and hence the total number of area estimations dramatically. To avoid a large number of estimations, Section 3.6 presents an improved approach to calculate a complete lower bound AT curve efficiently.

3.2 Related work

There are only a few other lower bound *cycle budget*, i.e., timing, estimation approaches; see [Rim92] or [Shar93]. These approaches are less accurate than the approach described in Section 3.3, as will be shown in that section.

Most other lower bound *functional area* estimation approaches, like those of [Chen91], [Jain92], [Shar93] and [Dalk94], handle only trivial libraries and are based on relaxing the precedence constraints. Section 3.4.2 deals with trivial module libraries and shows that these four approaches are less accurate. This is due to the fact that the approach of Section 3.4.2 guarantees that, with the selected module set, all operations can be scheduled within their COEIs, or within their $\overline{\text{OEI}}$s after the execution interval analysis is done. The four approaches above do not give such a guarantee. Furthermore, the (average) run time complexity of [Dalk94] is also larger than the (average) complexity of the approach in Section 3.4.2.

Another approach covering trivial libraries can be found in [Chau94]. That approach guarantees that operations can be scheduled within their COEIs, but only for operations with a delay of one clock cycle, i.e., not for operations with delays larger than one clock cycle. The approach of Section 3.4.2 gives this guarantee for both kinds of operations and has a better run time complexity and efficiency as well, because the solution method of [Chau94] is based on linear programming.

Like the approach in this chapter, [Chau94] can detect that some combination of modules for different operation types constitute an infeasible module set. However, unlike the approach of Section 3.5.2, it does not try to determine a subsequent module set with the smallest area, but enumerates all possible subsequent module sets. This is due to the fact that the approach of [Chau94] cannot discriminate between the costs of different module sets: it only considers the number of modules of each type.

Another approach that produces AT curves, like the approach in Section 3.6, is [Holm94]. In that approach, memory access times are also taken into account. However, the approach of [Holm94] calculates each point of the AT curve separately and the resulting curve is not guaranteed to be a lower bound curve. Furthermore, a first approach that claims to determine an accurate lower bound on memory area can be found in [Shar94]. In this chapter, memory area as well as interconnect area are not considered.

3.3 Lower bound cycle budget estimation

In this section, it is shown how the approach of Chapter 2 can be used to determine a very accurate lower bound estimate on the number of clock cycles needed for a design with resource constraints. This estimate can be derived using a small adaptation of the execution interval analysis, and is merely the determination of the smallest possible cycle budget according to that analysis.

Definition 3.1. Cycle budget problem.
Given an acyclic DFG and a set of modules M on which the operations in the DFG must be mapped. Find the minimum cycle budget $|C|$ needed to execute the DFG for the given set of modules.

In general, the problem of Definition 3.1 is NP–hard [Heem90]. Some lower bound C_{low} is therefore calculated, satisfying $C_{low} \leq |C|$. Of course, it is tried to determine an accurate, i.e., an as large as possible, value for C_{low}.

Definition 3.2. Lower bound cycle budget problem based on COEIs.
Given are the prerequisites of Definition 3.1. Calculate the minimum number of cycles C_{COEI}, $C_{COEI} \leq |C|$, such that all operations can be scheduled within their COEI for the given set of modules.

An *initial* lower bound on the cycle budget can be derived by Algorithm 2.1 or 2.6, which determine $\overline{M_1}$s. From these cycles, an initial lower bound on the earliest possible completion time of a DFG can be derived. For instance, if a DFG consists of ten multiplications only, and the multiplications have a delay of two, then an initial and correct C_{low} equals $\overline{M_1}(10) + 2$. This bound on the cycle budget is in fact equivalent to the lower bound calculated in [Rim92].

The approach of [Rim92] stops with this initial bound, but the estimation process can be continued by determining the $\overline{M_2}$s and the COEIs using the initial bound C_{low}. After this process, it may occur that the number of cycles of some $\overline{MEI}$ turns out to be smaller than the smallest execution delay of the corresponding modules. In that case there is no feasible schedule within the

'current' bound on the number of cycles, C_{low}, so this lower bound has to be increased. The lower bound C_{low}, the $\overline{M_2}$s and the CALAPs of the operations have to be increased such that all $\overline{MEI}$s are at least as large as the smallest execution delay of the corresponding modules.

After the $\overline{MEI}$s and $\overline{OEI}$s have been determined, it can be checked whether all the $\overline{BSG}$s contain complete matchings. At this point it is still possible that the lower bound estimate on the number of cycles is detected as too small. This happens as soon as a complete matching within some initial $\overline{BSG}$ cannot be found. In that case the estimate of the number of cycles must be incremented and the check is performed a second time. The additional steps in this approach clearly make the lower bound estimate more accurate than the approach of [Rim92], which stops after the initial bound based on the $\overline{M_1}$s.

In case of a *trivial* module set, recall Definition 2.14, if complete matchings are found within all the initial $\overline{BSG}$s, all operations can be scheduled within their COEI, i.e., then C_{low} has become equal to C_{COEI}. Note that in case of *unrestricted* module sets, recall Definition 2.15, it is not guaranteed that C_{low} equals C_{COEI}; see the determination of the $\overline{MEI}$s in Section 2.4.2.

Determining $\overline{MEI}$s, COEIs and a complete matching in each $\overline{BSG}$ has a complexity of $O(|V| \cdot \log |V| + |E|)$, recall Section 2.3.5. Let $|C_{up}|$ be an upper bound on the number of cycles, for instance determined by a heuristic scheduler, e.g., a list scheduler. A binary search on the number of cycles has a complexity of $O(\log |C_{up}|)$, so determining the C_{COEI} of Definition 3.2 for a trivial module set takes $O(|V| \cdot \log |V| \cdot \log |C_{up}| + |E| \cdot \log |C_{up}|)$.

However, the lower bound estimation does not have to stop as soon as complete matchings are found within all the initial $\overline{BSG}$s. The rest of the execution interval analysis of Chapter 2 can still be performed, which can lead to the detection that the current lower bound estimate cannot be equal to $|C|$. This leads to the lower bound cycle budget problem of Definition 3.3.

Definition 3.3. lower bound cycle budget problem based on $\overline{OEI}$s.
Considering the prerequisites of Definition 3.1, calculate the minimum number of cycles C_{OEI}, $C_{OEI} \leq |C|$, such that all operations can be scheduled within their $\overline{OEI}$ with the given set of modules. Of course, the mentioned $\overline{OEI}$ values are the values after the interval analysis of Chapter 2 is done.

A full execution interval analysis for one cycle budget takes $O(|C_{up}| \cdot |V|^3)$, recall Section 2.3.5, and a binary search on the number of cycles has a complexity of $O(\log |C_{up}|)$. Consequently, the total run time complexity of solving the problem of Definition 3.3 for a trivial module set has a run time complexity of $O(|C_{up}| \cdot \log |C_{up}| \cdot |V|^3)$. Of course, in practice, the average run time complexity is much lower.

3.4 Lower bound functional area estimation

3.4.1 Introduction

The objective of the lower bound functional area estimation, or 'module selection', is to select a module set with minimal cost (area) for a given time constraint. If the precedence constraints of a DFG are taken into account, then the problem is NP–hard [Verh91]; see Definition 3.4.

Definition 3.4. Functional area problem.
Consider an acyclic DFG, a set of module types T_M, and a list of cycles C. Find a set of functional units with minimal area from T_M, for which the DFG can be executed within the time budget $|C|$.

After relaxing the precedence relations the module selection problem is 'easier' to solve. It can then be stated as finding the minimal module set needed to schedule the operations within either their COEIs, see Definition 3.5, or their $\overline{\text{OEI}}$s after the execution interval analysis is done, see Definition 3.6. The next sections will discuss in which cases the relaxed module selection problems of Definition 3.5 and 3.6 can be solved in polynomial time.

Definition 3.5. Lower bound functional area problem based on COEIs.
Given are the prerequisites of Definition 3.4. Find a set of functional units with minimal area from T_M, such that all operations can be scheduled within their COEI with the selected set of modules.

The difference between Definition 3.4 and 3.5 is that in Definition 3.5 the precedence relations are not taken into account, except for the COEIs.

Definition 3.6. Lower bound functional area problem based on $\overline{\text{OEI}}$s.
Given are the prerequisites of Definition 3.4. Find a set of functional units with minimal area from T_M, such that all operations can be scheduled within their $\overline{\text{OEI}}$, after the execution interval analysis of Chapter 2 is done, with the selected set of modules.

3.4.2 Trivial module libraries

For trivial module libraries, for which there is a one–to–one mapping from operation types to module types, the relaxed module selection problem of Definition 3.5 can be solved in $O(|V| \cdot (\log |V|)^2 + |E| \cdot \log |V|)$, using an adaptation of the analysis of Section 2.3.

Starting with one module of each module type, the number of modules is increased until a complete matching is found for each initial $\overline{\text{BSG}}$. Determining $\overline{\text{MEI}}$s, COEIs and a complete matching in each $\overline{\text{BSG}}$ has a complexity of $O(|V| \cdot \log |V| + |E|)$; recall Section 2.3.5. Because at most $O(\log |V|)$

module sets must be evaluated in a binary search, the total worst case complexity of this lower bound estimation is $O(|V| \cdot (\log |V|)^2 + |E| \cdot \log |V|)$. Because in practice the run time complexity of determining the initial $\overline{BSG}$s is about linear in the size of the DFG and the number of modules is limited, the average run time complexity of this lower bound estimation is also about linear in the size of the DFG, i.e., linear in the number of vertices and arcs in the DFG.

The approach pictured above can be made even more accurate. For each module type, the minimum number of modules required can be determined *separately*, by performing the execution interval analysis iteratively until the corresponding operations can be scheduled within their $\overline{OEI}$s. To do this correctly, one must assume that the number of modules of all other module types is unlimited. Because Section 2.3.5 showed that a full execution interval analysis for one cycle budget takes $O(|C| \cdot |V|^3)$ and a binary search on the number of modules has a complexity of $O(\log |V|)$, the total run time complexity of solving this problem is $O(|C| \cdot \log |V| \cdot |V|^3)$. Again, in practice, the average run time complexity is much lower.

Because the modules of each module type are determined independently of the modules of other types, the problem pictured above is not yet the lower bound estimation problem of Definition 3.6. It is possible that the execution interval analysis detects that the resulting combination of modules of different types is still infeasible. If this is the case, the non–polynomial approach that will be explained in Section 3.5.2 must be followed to obtain a better, i.e., more accurate, lower bound.

3.4.3 Unrestricted module libraries

For unrestricted libraries, the relaxed module selection problem of Definition 3.5 is not solvable in polynomial time. The problem of finding the optimal module set for operations with fixed intervals on two types of modules can be identified as a special case of the relaxed module selection problem, and is proven NP–hard [Naka82].

However, for unrestricted libraries, a lower bound module selection problem can be formulated in the form of a run time efficient mixed integer linear programming (MILP) problem [Nemh88]. The MILP approach tries to find bottlenecks in a DFG with respect to the module area needed and formulates constraints based on these bottlenecks. The formal definition of the MILP problem is given in Definition 3.7.

Definition 3.7. Lower bound functional area MILP problem.
Given are the prerequisites of Definition 3.4. Find a set of functional units with minimal area from T_M, such that the constraint sets 1A, 1B, 2 and 3, which will be presented in the following sections, are fulfilled.

It must be noted that the MILP problem of Definition 3.7 is a relaxation of the problem of Definition 3.5. So, the resulting area of the MILP problem is equal or less than the resulting module area of the problem of Definition 3.5.

The objective function of the MILP module selection problem is to minimize the function representing the total functional area. Let area(m) be the area of a module with type $m \in T_M$ and let n(m) be the number of selected modules with type $m \in T_M$. Then the objective is to minimize

$$\sum_{m \in T_M} (\text{area}(m) \times n(m)).$$

Only the variables n(m), $m \in T_M$, and no other variables in the constraint sets, have to be integral in the MILP problem formulation. The number of constraints is small as well, as will be shown in the next sections. So, the run time efficiency of solving such an MILP problem depends mainly upon the size of the module library and not on the size of the DFG, and is therefore high.

Furthermore, an MILP solver can always come up with a feasible, but not necessarily lower bound, solution, just by ceiling the integer variables n(m), $m \in T_M$ after solving the LP relaxation. So, a feasible solution can be found in polynomial time for the MILP problem of Definition 3.7. This means that, in contrast to ILP scheduling, see Chapter 4, an MILP solver can always find a solution within acceptable run times. In the following sections the different constraint sets to determine a lower bound on the functional area are discussed.

3.4.4 Distribution constraints

Consider the example in Figure 3.1 with a cycle budget $|C|$ of four cycles. [Jain92] and [Chen91] estimate the number of modules needed from a trivial library, while assuming that all cycles are available to perform any operation. There are four additions and an adder can perform four additions within the time constraint, so the result of their estimation is one adder. Those approaches guarantee that the operations can be scheduled within the cycle budget $|C|$, but not necessarily within their COEIs. So, their relaxations of the problem of Definition 3.4 result in less accurate lower bound estimations than the relaxations described in this chapter.

Distribution intervals

The execution intervals of v_1, v_2 and v_3 in Figure 3.1 show that three additions must be executed within two cycles, so at least two adders are needed. These three intervals form a so called 'distribution interval' of the operation type '+'.

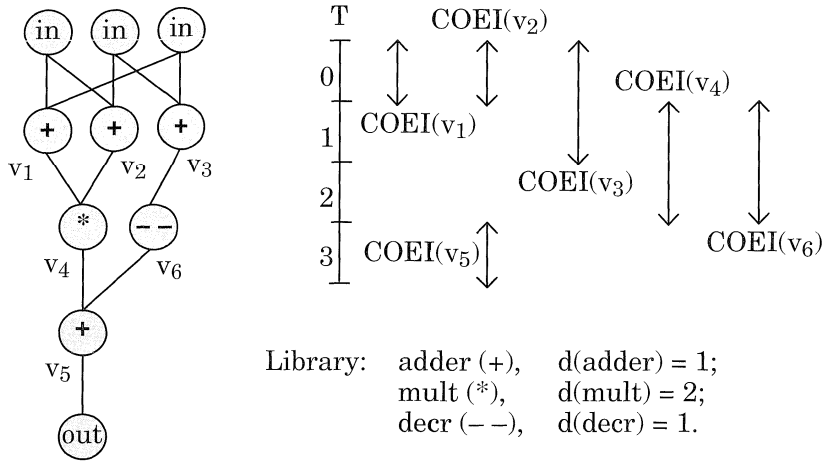


FIGURE 3.1. Example DFG, module library and COEIs.

Distribution intervals can be derived from COEIs as follows. If two COEIs have an overlap in their clock cycles, they are joined into a new interval being the union of the former two. The example shows that the notion of distribution intervals leads to a more accurate estimation than the approaches of [Jain92] and [Chen91].

In Figure 3.2, an example of COEIs and distribution intervals is given; note that the example is not related to a DFG in the rest of this thesis. COEI(a), COEI(b) and COEI(c) are in the same distribution interval, because there is a 'path' from COEI(b) to COEI(c) through COEI(a); this is denoted by the dashes. Furthermore, COEI(e) is in the same distribution interval as COEI(d). The formal definition of distribution intervals is given below.

DEFINITION 3.8. Distribution intervals.
Let $ts \subseteq T_O$ and $W' \subseteq V$ with $\tau(v) \in ts$ for all $v \in W'$. The distribution graph DG(ts) is an undirected graph represented by the 2–tuple (W', A'), where A' is the set of arcs. There is an arc $(v, w) \in A'$ between $v \in W'$ and $w \in W'$, if and only if there is an overlap between the COEIs of v and w, i.e., if $CASAP(v) < CALAP(w) \wedge CALAP(v) > CASAP(w)$, or if there is a $z \in W'$ for which $(v, z) \in A'$ and $(w, z) \in A'$. It is easy to see that DG(ts) consists of a set of complete subgraphs, DGS(ts), with no arcs between those subgraphs. So, the set DGS(ts) denotes a partition of W'. Based on these complete subgraphs, the following function is defined. $\varepsilon: P(T_O) \rightarrow P(C \times C)$ is a function returning a set of disjoint distribution intervals:

$$\varepsilon(ts) = \left\{ \left[\left\lfloor \min_{v \in S} CASAP(v) \right\rfloor , \left\lceil \max_{v \in S} CALAP(v)-1 \right\rceil \right] \,\middle|\, S \in DGS(ts) \right\}.$$

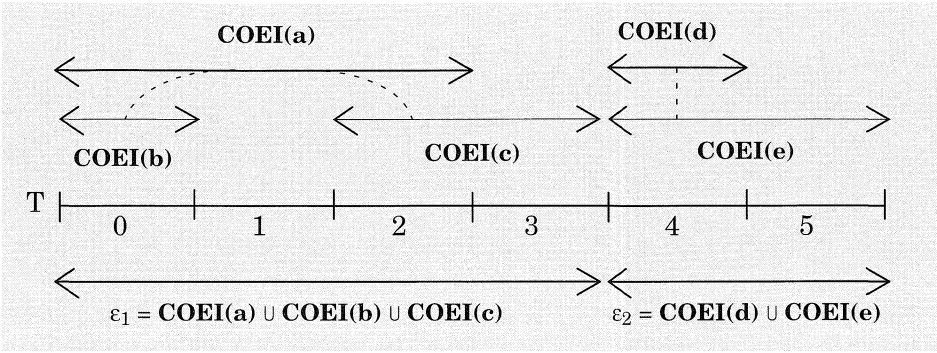


FIGURE 3.2. Execution and distribution intervals.

Had–intervals, preliminary mappings and module capacity

The MILP constraints related to the distribution intervals try to enforce the selection of sufficient module 'capacity' to perform the operations in all the distribution intervals. For each (set of) operation type(s) only the interval with the highest average number of operations per cycle step is interesting. This interval is called the *had–interval* (highest average distribution interval). All other intervals normally need equal or less module capacity and are therefore not considered in the MILP formulation.

To formulate the MILP constraints, we will now define the 'capacity' and the 'number of preliminary mappings' of a module type. The capacity denotes the maximum number of operations a module can execute within some clock cycle interval; see Definition 3.10. Because several module types can be selected for operations in such an interval, the number of operations mapped on each type must be modelled; see Definition 3.9. These numbers are merely preliminary help variables. The exact mappings are determined later on in the synthesis flow by a scheduler or a binder: the MILP module selection problem formulation does not model the actual scheduling problem.

DEFINITION 3.9. number of preliminary mappings.
$m: T_M \times P(T_O) \times T_O \rightarrow \mathbf{Q}$ is a function describing the number of preliminary mappings to a module type in a had–interval. Example: m(alu, {+, *, –}, +) is the number of additions mapped to the module type alu in the had–interval of the operation type set {+, *, –}.

Definition 3.10. capacity of a module type.
Let $m \in T_M$ and $e \in (C \times C)$. cap: $T_M \times (C \times C) \rightarrow \mathbb{N}$ is the function describing the number of operations a module of type m can execute in a had–interval e:

$$cap(m, e) = \left\lfloor \frac{|e| - d(m) + 1}{dii(m)} \right\rfloor .$$

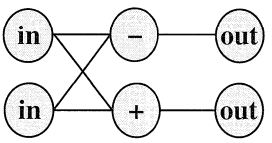
Sometimes the capacity of a module type can be determined more accurately. Figure 2.5a shows the fifth order wave digital filter from [DeWi85]. During the execution of the fifth addition, operation 14, no other operations can be executed as has already been pointed out at the end of Section 2.3.2.

As soon as the fifth addition is in the same distribution interval as any of its succeeding additions, there will be a 'hole' in that distribution interval in which no addition can take place. This 'hole' is as least as large as the minimal delay of a multiplication, and occurs, in any feasible schedule, after the fifth addition is executed. Such 'holes' can be incorporated in the module selection by changing the capacity of a module type. The calculation of the capacity is thus changed by decreasing the available number of cycles $|e|$ in an interval $e \in (C \times C)$ with the minimal number of cycles of the 'hole'.

Example, introducing the first MILP (distribution) constraint set

Consider the example DFG with the library given in Figure 3.3. The optimal module set for time constraints of two or more cycle steps consists of one alu. For the constraint of one cycle step it consists of one add and one sub module. The constraints for the latter case can be formulated as shown in Figure 3.4.

The values in italic in Figure 3.4 represent numbers of operations in a had–interval. The bold **1**s denote module capacities. If the time constraint would be 2 cycle steps, the bold **1**s have to be replaced by **2**s. If several operation types can be mapped on the same module type, then extra constraints are required for selecting enough modules of that type. For that reason, the constraints on both operation types {+, –} together are very useful: without them 1 alu would be selected for the time constraint of one cycle step, instead of 1 add and 1 sub module.



Library:
add (+), d(add) = 1, area(add) = 1;
sub (–), d(sub) = 1, area(sub) = 1;
alu (+, –), d(alu) = 1, area(alu) = 1.5.

Figure 3.3. Example DFG with library.

Formal definition of distribution constraints

The first set of constraints related to the distribution of operations can now be formalized as follows.

CONSTRAINT SET 1A.
Let $ts \subseteq T_O$, $t \in ts$ and no(ts, t) be the number of operations $v \in V$ with $\tau(v) = t$ within the had–interval e of the operation type set ts. Then the following constraints must hold:

$$\forall_{ts \subseteq T_O} \; \forall_{t \in ts} : \sum_{m \in \mu(\{t\})} m(m, ts, t) = no(ts, t).$$

$$\forall_{ts \subseteq T_O} \; \forall_{m \in \mu(ts)} : \sum_{t \in ts \mid t \in \mu^{-1}(\{m\})} m(m, ts, t) \leq cap(m, e) \times n(m).$$

REMARKS.
The variables n(m) are the only integer variables in the above and following constraint sets. If for a set of operation types not all types are present in the had–interval, or if there is no module type which can perform all these operation types, then the corresponding constraints have no impact at all and can be omitted.

The COEI of each operation is of course an upperbound for its execution time, a condition not yet taken care of in the constraint set 1A. Consider for instance the had–interval of the {+}–operation type set of Figure 3.1 with

add1 (+), d(add1) = 1, area(add1) = 5;
add2 (+), d(add2) = dii(add2) = 2, area(add2) = 2;
mult (*), d(mult) = dii(mult) = 2;
decr(– –), d(decr) = 1.

m(add, {+}, +) + m(alu, {+}, +) = *1*; m(add, {+}, +) ≤ **1** x n(add); m(alu, {+}, +) ≤ **1** x n(alu);	constraints related to the additions
m(sub, {–}, –) + m(alu, {–}, –) = *1*; m(sub, {–}, –) ≤ **1** x n(sub); m(alu, {–}, –) ≤ **1** x n(alu);	constraints related to the subtractions
m(add, {+, –}, +) + m(alu, {+, –}, +) = *1*; m(sub, {+, –}, –) + m(alu, {+, –}, –) = *1*; m(add, {+, –}, +) ≤ **1** x n(add); m(sub, {+, –}, –) ≤ **1** x n(sub); m(alu, {+, –}, +) + m(alu, {+, –}, –) ≤ **1** x n(alu).	constraints related to both operation types

FIGURE 3.4. Constraints for Figure 3.3, cycle budget |C| = 1.

The constraints of constraint set 1A related to the {+}–operation type are

$$m(add1, \{+\}, +) + m(add2, \{+\}, +) = 3;$$
$$m(add1, \{+\}, +) \leq \mathbf{2} \times n(add1);$$
$$m(add2, \{+\}, +) \leq \mathbf{1} \times n(add2).$$

Consequently three add2 modules would be chosen. Figure 3.1, however, shows $|COEI(v_1)| = |COEI(v_2)| = 1$, requiring mappings to add1 modules. The respective constraint is: $m(add1, \{+\}, +) \geq 2$. In general terms we thus obtain the additional constraint set 1B.

CONSTRAINT SET 1B.
Let $ts \subseteq T_O$, $t \in ts$, $i \in \mathbb{N}^+$ and let noo(ts, t, i) be the number of operations $v \in V$, $\tau(v) = t$, $|COEI(v)| \leq i$ within the had–interval e of the operation type set ts. Then the following constraints must always be valid:

$$\forall_{ts \subseteq T_O} \quad \forall_{t \in ts} \quad \forall_{i \in [1, 2, \ldots, |e|]} : \sum_{m \in \mu(\{t\}) \mid d(m) \leq i} m(m, ts, t) \geq noo(ts, t, i).$$

REMARK.
Many of the constraints in set 1B are superfluous and can be omitted, which can be seen as follows. When the left hand side is equal to the left hand side of another constraint, while the right hand side of the other constraint is equal or larger, then the first constraint can be omitted. Or, when the right hand sides are equal, while the left hand side of the other constraint is a subset, then the first constraint can be omitted as well.

3.4.5 Fixed operation constraints

The constraint sets 1A and 1B do not identify that some operations might always occupy a module in a certain cycle step. Again consider the example of Figure 3.1 with the module library described above. Applying constraint set 1A/B, one add1 module and one add2 module are selected. As can be seen in Figure 3.1, the operations v_1 and v_2 will always be scheduled in cycle zero. We already have seen that they must be mapped on add1 modules, so this leads to a new constraint: $n(add1) \geq 2$. Now two add1 modules, one decr module and one multiplier are selected, which constitute the optimal module set. This constraint set, related to 'fixed' operations which must be executed within a certain amount of time, can be formalized as follows.

CONSTRAINT SET 2.
Let $ts \subseteq T_O$, $i \in \mathbb{N}^+$ and nof(ts, i) be the maximal number of operations $v \in V$, with $\tau(v) \in ts$ and $|COEI(v)| \leq i$, that always occupy a module in the same cycle step. Then the following constraints must hold:

$$\forall_{ts \subseteq T_O} \quad \forall_{i \in [1, |T|]} : \sum_{m \in \mu(ts) \mid d(m) \leq i} n(m) \geq nof(ts, i).$$

Remark.
Many constraints can be omitted in the same way as constraints of set 1B.

3.4.6 Path delay constraints

The previous constraints did not consider whether the minimal delay of each path in a DFG can be within the time constraint. The critical paths are those which can have the largest delay. If two paths have the same number of operations of each type, then only one of these paths is taken into account. Only such paths are considered in the respective constraints, and in practice they can be determined efficiently by breadth first search. The corresponding constraints are as follows.

Constraint Set 3.
Let $h: T_M \times T_O \rightarrow Q_{0,1}$ be an auxiliary variable, where $Q_{0,1}$ is the set of rational numbers in the range of 0 to 1. This variable states whether the delay of a module type may be used for the determination of the minimal delay of a critical path. Let CP be the set of critical paths to be considered, and nop(c, t) the number of operations $v \in V$, $\tau(v) = t$ within the path $c \in CP$. Then the following constraints must hold:

$$\forall_{t \in T_O} : \sum_{m \in \mu(\{t\})} h(m,t) = 1.$$

$$\forall_{t \in T_O} \; \forall_{m \in \mu(\{t\})} : h(m,t) \leq n(m).$$

$$\forall_{c \in CP} : \sum_{t \in T_O} \sum_{m \in \mu(\{t\})} (h(m,t) \times d(m) \times nop(c,t)) \leq |C|.$$

The first constraint set states, that for each operation type $t \in T_O$, the sum over all module types $m \in T_M$ of h(m, t) must be equal to one. If h(m, t) would be allowed to be either 0 or 1, then this constraint set would say that for one module type, h(m, t) must be equal to one. However, a close examination of the constraints shows, that h(m, t) may be rational instead of integral.

The second constraint set states, that a h(m, t) may only be larger than 0, if at least one module of the corresponding module type $m \in T_M$ is selected. The variables h(m, t) are used in the third constraint set to point out the fastest modules in the set of selected modules. The fastest modules in the set of selected modules are used in the third constraint set to assure that every path in the DFG can be executed within the time constraint. Of course, the constraint set 3 does not guarantee that the complete DFG can be executed within the time constraint, because the data dependencies between the paths are not taken into account.

3.5 Reviewing infeasible module sets

The cycle budget estimation of Section 3.3 is a partial check for the correctness of the module sets derived by the functional area estimation. If the lower bound estimate of the cycle budget exceeds the number of cycles for which such a module set is derived, then the module set is not correct. It can then be tried to detect why the module set is not correct. Subsequently, a better lower bound estimate of the module set with minimal area can be determined.

3.5.1 Detection in case of COEIs

Consider an *initial* $\overline{\text{BSG}}$ with initial $\overline{\text{OEI}}$s equal to the corresponding COEIs, i.e., no execution interval reduction has yet taken place, and consider a cycle budget equal to the time constraint for which the module set was derived. If in such a case no complete matching can be found in the $\overline{\text{BSG}}$, then there was not enough module capacity selected to schedule all the operations in the $\overline{\text{BSG}}$ within their COEIs.

In case of a trivial library, extra modules of the applicable module type are added until a complete matching can be found, recall Section 3.4.2. In case of an unrestricted library, one cannot just add some arbitrary applicable module and guarantee that the result is still a lower bound module set. Instead, the reason that the module set is not correct is interpreted as follows.

The distribution intervals of Section 3.4.4 try to detect local 'concentrations' of operations. If a module set is not correct, then there exists a local concentration within a distribution interval that has not been detected and for which there is not enough module capacity selected. During Algorithm 2.5 for finding a complete matching, some $\overline{\text{MEI}}$ cannot be matched and the algorithm stops. All operations that could have been matched with that $\overline{\text{MEI}}$ are matched with previous $\overline{\text{MEI}}$s, and the $\overline{\text{MEI}}$ denotes a 'low' in the distribution of operations.

If all operations that could be scheduled within that $\overline{\text{MEI}}$ are not considered in the distribution interval construction, the had–interval will be split into two (or more) new intervals and the local concentration is expected to be 'isolated' in one of them. That interval will then become the new had–interval, and a renewed module selection can lead to an improved lower bound estimate of the optimal module set. If the renewed selection does not lead to an improved module set, then the approach of Section 3.5.2 has to be applied.

3.5.2 Detection in case of reduced $\overline{\text{OEIs}}$

If the execution interval analysis detects that a module set is not correct during a second or later run of the analysis, then the module set is not feasible because of the violation of a combination of resource and precedence constraints. A new lower bound module set can then be derived by adding new constraints, thus forcing the selection of a different module set with a total area equal to or larger than the previous selection.

Let M be a previous module set that is detected as infeasible, let $n_M(m)$ be the corresponding number of selected modules of type $m \in T_M$, and let $diff_M(m) \in \mathbb{N}$ denote an increase in the number of modules of type m with respect to the module set M. Note that only increases and no decreases have to be considered in this formulation. For these prerequisites, the following constraints must be valid.

$$\sum_{m \in T_M} (\text{area}(m) \times n(m)) \geq \sum_{m \in T_M} \left(\text{area}(m) \times n_M(m)\right).$$

$$\sum_{m \in T_M} \text{diff}_M(m) \geq 1.$$

$$\forall_{m \in T_M} : \left(n_M(m) + 1\right) \times \text{diff}_M(m) - n(m) \leq 0.$$

The first constraint states that the total area of the new module set must at least be as large as the total area of the previous module set. The second constraint states that for, at least, one module type the number of selected modules must increase. The remaining constraints are also needed to force the increase of the number of modules of at least one type. Note that the increase for one module type can result in the decrease for another module type; for instance, two slow modules might be replaced by one fast, but more expensive, module.

The constraints can be added to the constraint sets of Section 3.4 until a module set is selected that can be feasible according to the lower bound cycle budget estimation. These additional constraints due to the infeasibility of some previous module set can be discarded as soon as their total module area becomes less than the lower bound area of the latest module selection. In [Timm93c] another constraint set has been given that has the same result. However, the following constraints result in a better MILP model, thus leading to a solution method with a better run time efficiency.

3.6 Efficient lower bound AT curve prediction

Area estimations will often lead to the same module set for different time constraints. Based on this observation a unified approach of lower bound area and cycle budget estimations to predict the area vs. timing characteristics of a design can be derived; see Figure 3.5. The approach starts with the derivation of the module set with the smallest area and the lower bound on the number of cycles for that module set. The constraints to derive this module set are

$$\forall_{t \in T_O} : \sum_{m \in \mu(\{t\})} n(m) = 1.$$

The next step consists of decrementing the number of cycles and determining the corresponding lower bound module set. For this new module set the lower bound on the number of cycles is derived. The process is repeated until the smallest possible number of cycles has been reached. In this approach each module set is derived only once which leads to a run time efficient prediction of the AT curve.

Figure 3.5 shows that, for each cycle budget, the lower bound area will be larger than or equal to the lower bound area of the cycle budget plus one cycle. If $\overline{area}(i)$, $i \in \mathbb{N}^+$, is the lower bound functional area for a cycle budget of i cycles, then the following constraint must be valid when calculating the lower bound area for a cycle budget of $i - 1$ cycles:

$$\sum_{m \in T_M} (area(m) \times n(m)) \geq \overline{area}(i), \; i \in \mathbb{N}^+.$$

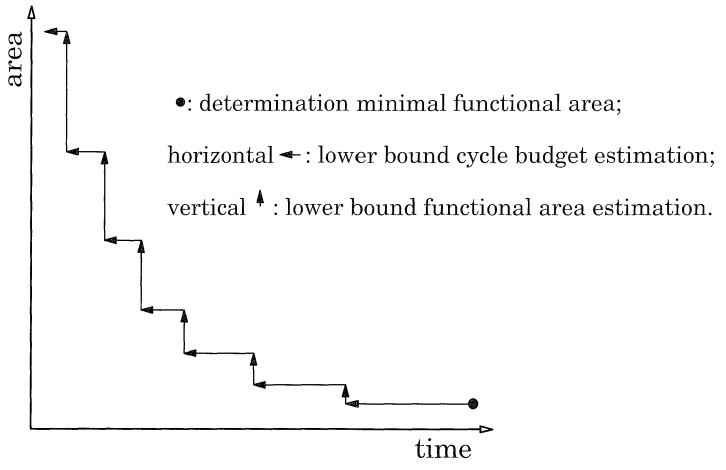


FIGURE 3.5. Fast AT curve prediction.

Although the constraint does not change the module selection result, it can decrease the search space of an MILP solver. During a branch–and–bound process, the solver does not have to investigate module sets which violate this constraint, which can lead to considerable run time efficiency improvements.

3.7 Experiments and results

The analyses presented in this chapter have been implemented in the NEAT system using a public domain mixed integer linear programming (MILP) solver [Berk94]. The solver uses the simplex algorithm with sparse matrix methods for the linear programming part and branch–and–bound techniques for the integer part.

In this section results of the approach on two benchmarks are given. The benchmarks are: WDELF, the fifth order wave digital filter from [DeWi85], see Figure 2.5, and FDCT, the fast discrete cosine transform which originates from [Mall90], see Figure 2.3. FDCT contains in contrast to WDELF a lot of parallelism which can be reduced significantly when the number of available cycle steps increases.

In Table 3.1, 3.2 and 3.3 different module libraries have been given. Table 3.1 gives a trivial library which is used in most papers. Table 3.2 gives an extended module library which has already been used in [Timm93a], while the module library of Table 3.3 with the largest range of delays originates from [Ishi91]. The last three columns of the tables state which operation types can be executed by some module type. In Table 3.5 and 3.6 the results for the Libraries 1 and 2 have been given, while in Figure 3.6 and 3.7 the results for the Libraries 2 and 3 are depicted. An arrow, $\leftarrow$, in Table 3.5 and 3.6 indicates, that the optimal solution is equal to the lower bound estimation, i.e., that the lower bound estimation is optimal. The tests were run on a HP9000/755 workstation. The optimal results in these tables have been obtained by either exhaustive branch–and–bound or by hand.

The figures and tables show that there is hardly any difference between the lower bound estimates and the optimal AT curve. Table 3.4 presents an overview of results and run times. That table also shows that the estimates are really accurate and that the average CPU times are small. However, keep in mind that the approach for unrestricted libraries uses a MILP formulation which can lead to unacceptable run times in some cases. This can be avoided by accepting intermediate, but not necessarily optimal lower bound, solutions to a MILP problem by ceiling the integer variables.

In [Ishi91] module sets are determined for only twelve different time constraints of WDELF with the library of Table 3.3. For those module sets the smallest possible cycle budgets were determined as well. Six of the derived module sets of [Ishi91] are equal to the lower bound estimates and are therefore optimal. One other module set is optimal but not equal to the lower bound estimate. The remaining five module sets derived by the scheduling approach of [Ishi91] are not exact. Only five of the twelve cycle budgets of [Ishi91] are equal to the corresponding lower bound estimates, so these five are optimal. The other cycle budget estimates are the estimates for the seven smallest module sets and these seven estimates are not optimal.

TABLE 3.1. Library 1: trivial library.

module type	*area*	*delay in cycle steps*	*operations*		
			*	+	−
mult	144	2	x		
alu1	16	1		x	x

TABLE 3.2. Library 2: extended library.

module type	*area*	*delay in cycle steps*	*operations*		
			*	+	−
mult	144	2	x		
alu1	16	1		x	x
sub1	15	1			x
add1	15	1		x	
alu2	9	2		x	x
sub2	8.5	2			x
add2	8.5	2		x	

TABLE 3.3. Library 3: library from [Ishi91].

module type	*area*	*delay in cycle steps*	*operations*		
			*	+	−
mpy1	256	1	x		
mpy2	32	16	x		
mpy3	2	256	x		
add1	16	1		x	x
add2	5	4		x	x
add3	2	16		x	x

TABLE 3.4. Overview of results.

	range of time constraints (in cycles)	*range of functional area*	*% wrong estimations*	*largest absolute difference*	*largest relative difference*	*average CPU time per time constraint (in sec)*
WDELF, Library 1	17 – 28	160 – 480	0%	0	0%	0.19
WDELF, Library 2	17 – 54	152 – 480	2.63%	7.5	2.34%	0.21
WDELF, Library 3	14 – 2144	4 – 560	2.30%	11	16.67%	0.07
FDCT, Library 1	8 – 34	160 – 1216	7.41%	16	2.04%	0.17
FDCT, Library 2	8 – 52	153 – 1212	6.67%	14	1.79%	0.30
FDCT, Library 3	6 – 4128	4 – 2128	3.47%	17	16.67%	0.11

TABLE 3.5. WDELF results, Library 1 and 2.

Library 1			Library 2		
/C/	*estimated area*	*optimal area*	*/C/*	*estimated area*	*optimal area*
17	480	←	17	480	←
18 to 20	320	←	18 to 19	320	←
21 to 27	176	←	20	312.5	320
28 to	160	←	21 to 27	168.5	←
			28 to 53	160	←
			54 to	152.5	←

TABLE 3.6. FDCT results, Library 1 and 2.

Library 1			Library 2		
/C/	*estimated area*	*optimal area*	*/C/*	*estimated area*	*optimal area*
8	1216	←	8	1212	←
9	1200	1216	9	1198	1212
10	768	784	10	766	780
11 to 12	624	←	11	615	622
13	608	←	12	615	←
14 to 17	464	←	13	606	←
18 to 25	320	←	14 to 17	462	←
26 to 33	304	←	18	320	←
34 to	160	←	19 to 25	312.5	←
			26 to 33	304	←
			34 to 51	160	←
			52 to	153	←

FIGURE 3.6. WDELF results, Library 2 and 3.

[Rim92] and [Shar93] report lower bound cycle budget estimates for the module sets of WDELF with Library 1; see also Table 3.5. One of the four estimates reported by [Rim92] and [Shar93] is not exact. Their estimations derive a lower bound of 27 cycles for the smallest module set, while the optimal bound is 28 cycles. This bound is derived by the approach of Section 3.3. Furthermore, the approaches of [Rim92] and [Shar93] do not generate the module sets of Table 3.5 by means of a lower bound estimation.

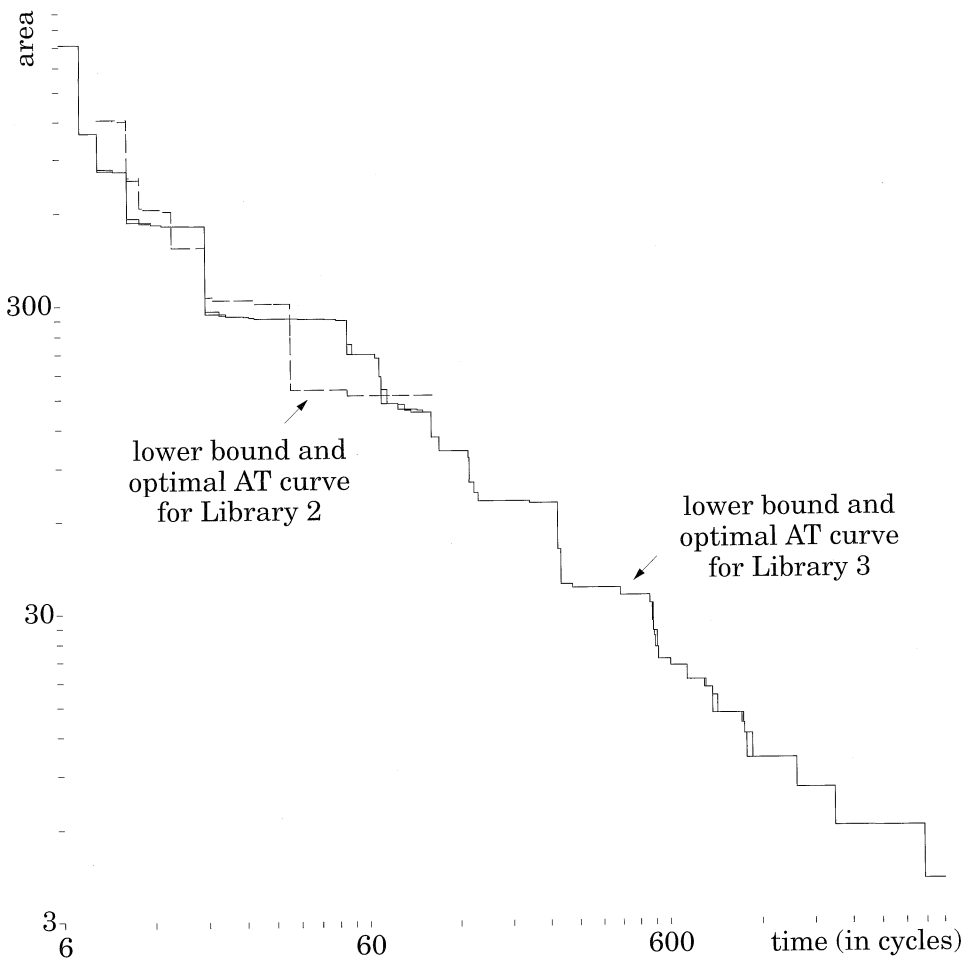


FIGURE 3.7. FDCT results, Library 2 and 3.

Chapter 4 Scheduling

4.1 Introduction

As has been explained in Section 1.4, a time or resource constrained scheduling problem can be transformed into a time *and* resource constrained scheduling problem. This can be achieved by deriving either a lower cycle bound, which is shown in Section 3.3, or by deriving a module set with lower bound module area, which is shown in Section 3.4. The transformation is also depicted in Figure 4.1. If the lower bound turns out to be infeasible, or a schedule cannot be found, then the cycle bound must be incremented or a new module set must be derived; recall Section 3.5. This scheme can be repeated until a schedule is found.

In Chapter 2, an approach has been presented that *prunes* the scheduling search space with the help of bipartite schedule graphs (BSGs). This chapter deals with exact scheduling methods that try to find a feasible schedule for given time and resource constraints, i.e., methods that *traverse* the search space in order to find a feasible schedule. The reason this chapter focuses on exact scheduling methods, and not on heuristic methods, is as follows.

If both time and resource constraints are imposed on a design, the 'only' goal of a scheduler is to find a feasible schedule as fast as possible. Because in practice the combination of these constraints is very strict, more and more instances appear where heuristic approaches render unsatisfactory results; see for instance the code generation examples in Chapter 5. So techniques like backtracking or branch–and–bound are needed if a scheduler does not succeed immediately and has to revoke certain decisions.

Furthermore, exact methods have gained a lot of interest within the architectural synthesis community, especially since the introduction of the integer linear programming (IP) formulation of the scheduling problem based on node packing; see for instance [Hwang91] and [Gebo92]. This chapter will show that IP scheduling is generally not the most run time efficient exact method. Another exact method is based on zero–suppressed binary decision diagrams (ZBDDs), see [Radi95]. However, that approach has not (yet) generated appealing results and is therefore not discussed in this thesis.

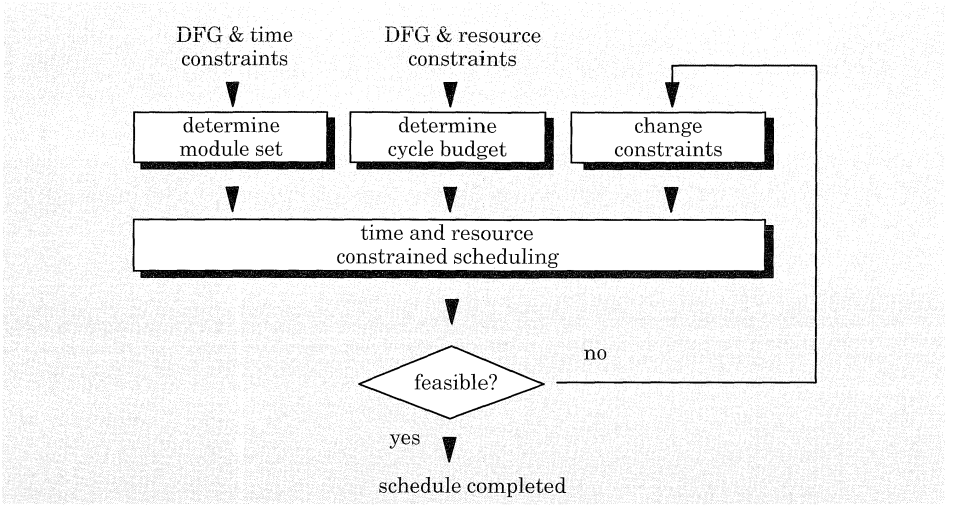


FIGURE 4.1. Towards a time and resource constrained scheduler.

In Section 4.2, an exact branch–and–bound approach is presented based on the BSGs introduced in Chapter 2; see also [Timm95a]. The following two aspects are dealt with: *problem formulation* and *bottleneck identification*.

Section 4.2.1 proves that, in case of a trivial module set, the question of the existence of a feasible schedule for a given DFG with time and resource constraints can be decided more efficiently by finding a correct ordering of the operations rather than by generating an exact schedule directly.

In Section 4.2.2 it is shown that the topology of the graph matching formulation yields a wealth of possibilities to identify bottlenecks that normally hamper scheduling algorithms. These bottlenecks typically fool heuristic methods and are also a source of wasted search effort for exact schedulers.

In Section 4.3, integer linear programming (IP) scheduling is discussed. In Section 4.4, well–known benchmarks demonstrate the cut in run time and the reliability of the scheduling approach based on BSGs in comparison with IP scheduling.

Again, for reasons of simplicity, only acyclic DFGs are considered in this chapter. Chapter 5 on retargetable code generation extends the scheduling method of Section 4.2, i.e., the scheduling method based on BSGs, with a loop model. Furthermore, due to some theoretical limitations of the method, only trivial module sets are considered in this chapter. However, in [Faber94], an

exact method can be found which does handle unrestricted module sets, and which is also based on the execution interval analysis of Chapter 2. That approach applies a strategy based on list scheduling and backtracking to schedule and bind operations.

4.2 Scheduling based on bipartite schedule graphs

4.2.1 Problem formulation

In Definition 4.1, the time and resource constraint scheduling problem considered in this chapter has been given. In general, the problem is not solvable in polynomial time. Furthermore, many of the common architectural synthesis systems schedule operations by deriving their schedule intervals immediately, i.e., the operations are directly assigned to specific cycle steps. In this section, it will be proven for trivial module sets that the existence of a schedule for a problem instance of Definition 4.1 can be decided more efficiently by finding a *correct ordering* of the operations. With a correct ordering we mean a linear ordering that corresponds to a feasible schedule, see Definition 4.2.

Definition 4.1. Time and resource constrained scheduling problem.
Given an acyclic DFG, a set of modules M on which the operations in the DFG must be mapped, and a list of cycles C. Find a feasible schedule $\phi \in \Phi$ if one exists, i.e., if Φ is not empty.

Definition 4.2. Correct ordering of operations.
A linear ordering $\lessdot$ of the operations in a BSG is correct if there exists a feasible schedule that induces that ordering, i.e., if $\exists_{\phi \in \Phi} : v \lessdot w \Leftrightarrow \pi_\phi^{-1}(v) < \pi_\phi^{-1}(w)$.
See Definition 2.32 for the meaning of $\pi_\phi^{-1}(v)$.

If a linear ordering on the operations in a $\overline{\text{BSG}}$ is imposed, then each operation is adjacent to *at most* one $\overline{\text{MEI}}$; see Section 2.3.4. A correct ordering implies a bijection between $\overline{\text{MEI}}$s and operations and, consequently, defines a complete matching in the $\overline{\text{BSG}}$. Furthermore, in case of a correct ordering, all adjacent $\overline{\text{OEI}}$s and $\overline{\text{MEI}}$s will have equal clock cycle intervals after the execution interval analysis is done. Otherwise a $\overline{\text{OEI}}$ or $\overline{\text{MEI}}$ could be further reduced and the analysis would continue with a new run. This leads to the following theorem.

Theorem 4.1.
Consider a *trivial* module set and the case in which, for each module type in a design problem, a linear ordering is imposed on the operations that have to be executed by modules of that type. If the number of modules equals one,

then imposing such a linear ordering is equivalent to sequencing the operations. Consider the case in which, for the set of these orderings for each module type, no infeasibility is detected by the execution interval analysis of Section 2.3. Then these orderings are correct and a feasible schedule can be derived from the result of the execution interval analysis in linear time.

PROOF.
If the schedule intervals of all operations $v \in V$ begin in the first cycle of the corresponding $\overline{OEI}(v)$, i.e., the schedule intervals are left justified and equal $[\overline{ASAP}(v), \overline{ASAP}(v) + d_{min}(v)]$ for all $v \in V$, then no precedence constraints are violated. If for each $\overline{MEI}$ some module starts the execution of an operation in the $\overline{M_1}$ of that $\overline{MEI}$, then no resource constraints are violated, see Property 2.2 and Algorithm 2.1.

The result of constructing the schedule $[\overline{ASAP}(v), \overline{ASAP}(v) + d_{min}(v)]$ for all $v \in V$, is that some module will start to execute an operation in the $\overline{M_1}$ of each $\overline{MEI}$. This is due to the fact that all adjacent $\overline{OEI}$s and $\overline{MEI}$s have equal clock cycle intervals after the execution interval analysis leaves its iteration. Thus a feasible schedule can be derived from such a correct ordering of the operations by scheduling every operation $v \in V$ in $[\overline{ASAP}(v), \overline{ASAP}(v) + d_{min}(v)]$, obtained after the execution interval analysis is done. ■

It follows from Theorem 4.1 that, in case of a trivial module set, the correctness of an ordering can be checked in polynomial time. A correct ordering can represent more than one schedule, so the number of different orderings is equal to or less than the number of different schedules. Because checking the correctness does not become more accurate when the schedule intervals of the operations are directly assigned to specific cycle steps, it is more efficient to search only for a correct ordering of the operations. So, the problem Definition 4.1 can be substituted by problem Definition 4.3.

DEFINITION 4.3. Scheduling problem revisited.
Consider the prerequisites of Definition 4.1. Find a *correct* ordering of the operations, if one exists.

If there is a correct ordering, Φ is not empty. From the correct ordering, the schedule intervals for all $v \in V$ can be constructed in linear time. Note that Theorem 4.1 cannot be applied directly to unrestricted module sets. It is unknown whether no resource constraints are violated in case the schedule intervals of all operations are left justified, after an ordering of the operations is imposed and the execution interval analysis is done; recall Section 2.4.2. In case of unrestricted module sets, additional steps must therefore be performed to check whether a certain ordering of operations corresponds to a feasible schedule. In [Faber94], this is solved by a strategy based on list scheduling and backtracking to schedule and bind operations.

Theorem 4.1 leads to the following scheduling approach. Starting from the $\overline{\text{BSG}}$s resulting from the execution interval analysis, an ordering is imposed by matching the operations one–by–one to specific $\overline{\text{MEI}}$s, while removing the arcs that can no longer be part of a complete matching. These arcs are the ones previously connected to the operation and the $\overline{\text{MEI}}$ that have just been matched, except for the one between them, together with other arcs that are no longer part of any irreducible component.

Each time an operation is matched to a $\overline{\text{MEI}}$, the whole execution interval analysis of Section 2.3 can be rerun. Matching operations and $\overline{\text{MEI}}$s is a process in which the $\overline{\text{BSG}}$s become more and more sparse: once an arc is removed, it is not involved again in a $\overline{\text{BSG}}$ as long as a matching is not revoked. The search space is also becoming smaller because the $\overline{\text{OEI}}$s and $\overline{\text{MEI}}$s are reduced more and more by the execution interval analysis as the scheduling proceeds.

4.2.2 Bottleneck identification

The previous section showed that, in case of a trivial module set, an exact scheduler 'only' needs to find a correct ordering of the operations. If an approach based on list scheduling is pursued, then the $\overline{\text{MEI}}$ with the smallest $\overline{M_2}$ is selected and matched with the operation with the smallest $\overline{\text{ALAP}}$ value. If a matching between two vertices in a $\overline{\text{BSG}}$ turns out to be infeasible, i.e., leading to an incorrect ordering, then the $\overline{\text{MEI}}$ is matched with the next operation, and so on. Such an approach is not really capable of identifying bottlenecks of the scheduling process. In this section it is shown, that the topology of the $\overline{\text{BSG}}$s yields many possibilities to identify such bottlenecks, which may thus be used to guide the scheduling process effectively. This is clarified by the following example.

Example

Consider the fast discrete cosine transform (FDCT) from [Mall90] in Figure 2.3, with the following time and resource constraints: $|C|$ is nine cycles, the number of multipliers (d = dii = 2 cycles) is eight and the number of adder/subtracters (d = 1 cycle) is three. This set of constraints leads to *infeasibility* as can be seen as follows. An extra adder/subtracter is needed for a feasible schedule of FDCT in nine cycles. Because the set of constraints is infeasible, i.e., the set of schedules Φ is empty, any kind of exact scheduler has to traverse the complete search space before it knows for sure that there is no feasible schedule. This example is therefore a good test to check the run time efficiency of an exact scheduler.

Lower bound area or cycle budget estimation approaches, e.g., [Rim92], [Jain92] or Chapter 3, cannot detect the infeasibility of the constraints above. Also the LP relaxation in case of IP scheduling leads to a solution, so an IP

scheduler like the one in [Gebo92] performs an exhaustive branch–and–bound process before the infeasibility is detected. Therefore, solving this problem instance with an IP scheduling approach leads to unacceptable run times.

The same conclusion is valid for an approach based on list scheduling and backtracking. Such an approach starts with matching operations to the $\overline{\text{MEI}}$s containing cycle zero. In Figure 4.2, the $\overline{\text{BSG}}$ for the additions/subtractions of this problem instance is given. There are three $\overline{\text{MEI}}$s with interval [0, 0] and eight operations are adjacent to these $\overline{\text{MEI}}$s, so three out of eight operations have to be matched with the first three $\overline{\text{MEI}}$s, i.e., have to be scheduled in cycle zero. For the same reason, three out of nine operations have to be matched with $\overline{\text{MEI}}(4)$, $\overline{\text{MEI}}(5)$ and $\overline{\text{MEI}}(6)$, etc. This leads to a very dense and wide search tree and again to unacceptable run times before the infeasibility is detected.

Instead of using an approach based on list scheduling, a careful analysis of the topology of the $\overline{\text{BSG}}$s is performed to obtain a sparse search tree for the branch–and–bound. Figure 4.2 shows that only three out of four operations have to be matched with $\overline{\text{MEI}}(16)$, $\overline{\text{MEI}}(17)$ and $\overline{\text{MEI}}(18)$, instead of three out of eight operations in cycle zero. So the top of the search tree will be less wide if an exact branch–and–bound scheduler first matches these $\overline{\text{MEI}}$s.

Every scheduling decision in terms of matching an operation to a $\overline{\text{MEI}}$ removes arcs from $\overline{\text{BSG}}$s, so at the time the scheduler has to decide which operations to match to $\overline{\text{MEI}}(1)$, $\overline{\text{MEI}}(2)$ and $\overline{\text{MEI}}(3)$, the number of operations to choose from may be much smaller. Also the possibility of a wrong decision will be smaller, if a scheduler favours the choice of three out of four operations above the choice of three out of eight operations. In this way the search tree becomes much sparser than in an approach based on list scheduling and backtracking. In fact, after trying each of the four possibilities of matching three operations with the $\overline{\text{MEI}}$s containing interval [5, 5], the scheduler has already detected that the problem instance is infeasible.

Comparison with force directed scheduling

The approach pictured above is somewhat counterintuitive though, as will be shown by a comparison with force directed scheduling. In the force directed scheduling approach [Paul87], the notion of the 'probability distribution' function is introduced to characterize the way operations are distributed over the different cycle steps. In the calculation of the distribution function, the probabilities of the start times of an operation are considered to be uniformly distributed over its COEI; see Definition 4.4 or [Stok91]. Then the sum of the probabilities of operations being executed in a cycle step is calculated for each (set of) operation type(s) and each cycle step; see Definition 4.5.

Definition 4.4. Probability P(v, c) of $v \in V$ being scheduled in cycle $c \in C$.

$$P(v, c) = \frac{N(v, c)}{F(v) + 1},$$

where F(v) is the freedom of v, see Definition 2.39, and N(v, c) gives the number of different schedules in which operation v is being executed in cycle c if unlimited resources are available, i.e:

$$N(v, c) = \begin{cases} \min \begin{cases} c - CASAP(v) + 1 \\ CALAP(v) - c \\ d_{min}(v) \\ F(v) + 1 \end{cases} & \text{if } c \in COEI(v) \\ 0 & \text{if } c \notin COEI(v) \end{cases}$$

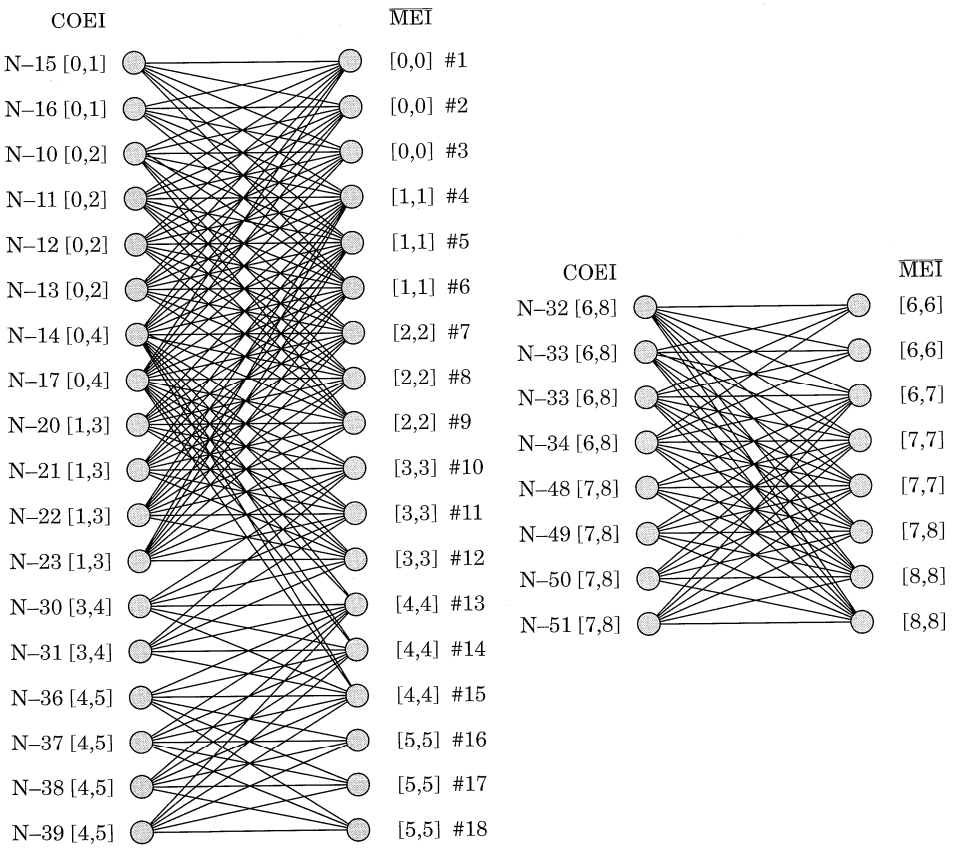


Figure 4.2. $\overline{BSG}$ for additions/subtractions of FDCT in Figure 2.3.

Definition 4.5. probability distribution function DF(ts, c) for ts $\subseteq T_O$.

$$DF(ts, c) = \sum_{v \in V \mid \tau(v) \in ts} P(v, c).$$

In Figure 4.3, the probability distribution of the additions / subtractions has been given according to the Definitions 4.4 and 4.5. Force directed scheduling starts with decreasing the probability function in cycle one, i.e., it tries to 'move' operations away from cycle one, because the maximum of the probability distribution lies in that cycle.

The discussion on the example above showed, that the scheduling approach based on finding correct orderings in the $\overline{BSG}$s starts in an opposite direction: it starts with deciding which operations will be scheduled in cycle five, recall the topology of the $\overline{BSG}$ in Figure 4.2. The decision which operations to schedule in cycle one is postponed as long as possible. Instead of starting the search tree at a high point in the probability distribution, i.e., trying to *decrease* the probability in cycle one, the scheduler starts with a low point in the probability distribution, i.e., trying to *increase* the probability in cycle five.

Resume of the proposed scheduling heuristic

The discussions above showed that the first step in the variable & value selection part of our scheduling approach is the variable selection, i.e., the selection of a set of $\overline{MEI}$s to match operations to. Note that there was little discussion about the value selection part. Experiments showed that the quality of the value selection is not very sensitive to the applied heuristic, if a careful variable selection is performed like the one described above.

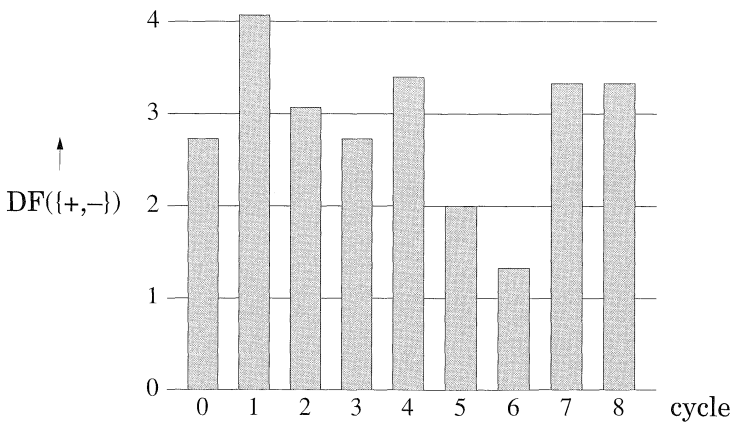


Figure 4.3. Probability distribution of FDCT in Figure 2.3.

The way the set of MEIs is selected, is as follows. If the number of cycles within a MEI is equal to the corresponding execution delay, then some module must start an execution in the M_1 of that MEI, and the MEI is said to be *fixed*. Such a fixed MEI is said to be *critical* if it also has only a small number of adjacent operations in the BSG.

From the *critical* MEIs the one with the smallest number of adjacent operations is selected. A *set* of critical MEIs can be selected, if they belong to the same BSG and if they have the same start cycle. MEI(16), MEI(17) and MEI(18) in Figure 4.2 form such a set of critical MEIs. If there are no critical MEIs, then the MEI with the smallest M_2 is selected, i.e., then the scheduling process continues in a list scheduling manner.

After a MEI or set of MEIs is chosen, it is tried to find an operation or set of operations that are suitable to be matched with the selected set of MEIs. The priority of matching an operation with a MEI is chosen to be inversely proportional to the difference between the ALAPs of the operation and the initially matched operation in Algorithm 2.5. So matching the operations according to the initial complete matching of Algorithm 2.5 has the highest priority. If a matching turns out to be incorrect, then a next operation is selected until a correct matching is found, or infeasibility is detected.

4.3 Scheduling based on integer linear programming

4.3.1 Polyhedral theory, node packing and scheduling

Before discussing integer linear programming (IP) formulations of the scheduling problem in general and those based on node packing in particular, the basics of the *polyhedral theory* are discussed in this section; see [Nemh88] for a very elaborate discussion on this topic. The background and motivation for the polyhedral theory is the notion that formulating a good model is of crucial importance to solve an problem instance of the model.

Basics of the polyhedral theory

The *polyhedron* of an IP problem instance is the region of integer and non–integer solutions that form all feasible solutions of the linear programming (LP) relaxation of the problem instance [Nemh88]. The LP relaxation of an IP problem is obtained by dropping the requirement that the variables of the problem must be integral in the solution.

The *convex hull* of a polyhedron is the smallest convex polygon for which each integer solution of the polyhedron is either on the boundary of the polygon or in its interior. A bounded polyhedron is called a *polytope*. It has been proven that for every polytope, there exists an *integral* polyhedron whose corresponding problem instance can be solved as a LP problem in polynomial time and yet always produces integral solutions for the variables.

The constraints that model the so called *facets* are the necessary constraints required to define the integral polyhedron. In fact, these facets together define the convex hull of a polytope. General IP problems are NP–complete, so finding all facets is in general NP–hard and in many cases their number cannot be bounded by a polynomial. However, if one can find the facets over the region of the minimum objective function, then one can solve an IP problem as a linear programming problem.

Another, less 'ambitious', possibility is to improve an IP model by making it as tight as possible, meaning that so called *valid inequalities* for the model are derived. These valid inequalities reduce the polyhedron without affecting the convex hull, but do not necessarily represent facets. Because less branch–and–bound effort may be needed to obtain an optimal integral solution if a polyhedron is smaller, the derivation of valid inequalities could lead to improved run time efficiency for solving IP problem instances.

Node packing and IP scheduling

Nowadays there is much interest in modelling scheduling and other architectural synthesis problems as integer linear programming problems [Hwang91]. It can be useful to have an exact model of some optimization problem, but a model itself does not necessarily provide an efficient solution method. So the question is whether this is the case for architectural synthesis scheduling.

Recent research [Gebo92] showed that the *resource* constrained scheduling problem can almost, but not completely, be modelled as a *node packing* problem. A feature of the node packing problem is that the characteristics of the facets are *partially* known. The exact definition of this problem is as follows.

Definition 4.6. node packing problem.
Let NPG be an undirected graph represented by the 2–tuple (N′, A′), where N′ is the set of vertices (binary variables) and A′ the set of arcs between the vertices. The objective is to find the maximum independent set of vertices in the graph, i.e., the largest set of vertices that do not induce a single arc.

Maximum clique constraints, i.e., constraints expressing that the sum of a set of binary variables may not exceed the value one, can possibly represent facets of a node packing polytope, as well as some other constraints [Nemh88].

The most run time efficient way to obtain an integer solution for a node packing problem instance is in general as follows; see for a more elaborate discussion [Nemh92]. First, it is tried to identify violated clique constraints. Then it is tried to identify other constraints that can possibly represent facets and to perform a process called 'lifting of variables' [Nemh88]. As a last step a branch–and–bound process is performed. Section 4.2 already showed, that a well structured branch–and–bound scheme can help to obtain a solution efficiently.

The *resource* constrained scheduling problem cannot always be modelled as a pure node packing problem, because the objectives in the resource constrained scheduling problem, i.e., minimize the number of clock cycles needed to execute a DFG, and the node packing problem can differ. However, the constraints of the resource constrained scheduling problem can be modelled in terms of (possibly maximum) clique constraints, which are in accordance with the constraints in the node packing problem. Modelling the constraints in this way leads to a better, i.e., 'tighter', IP model than previously published models [Gebo92]. So, the application of this model can in some cases lead to short run times; see [Gebo92] and the results in Section 4.4.

4.3.2 Evaluation of time versus resource constrained scheduling

The constraints of time constrained scheduling can, in contrast to resource constrained scheduling, not be modelled as pure node packing constraints [Gebo91]. The following time constrained IP scheduling formulation is moulded to be as close as possible to the node packing model.

Definition 4.7. Objective function for time constrained scheduling.
The objective of time constrained scheduling is to minimize the total module area as is the case in the functional area estimation of Section 3.4:

$$\sum_{m \in T_M} (\text{area}(m) \times n(m)).$$

Given Definition 4.8 and the case in which no chaining of operations is allowed, the corresponding constraints for a trivial module library are given in Definition 4.9, 4.10 and 4.11.

Definition 4.8. IP decision variables for time constrained scheduling.
Let $x(v, c) \in \{0, 1\}$, and let $x(v, c) = 1$ represent that $\lfloor \phi_1(v) \rfloor = c$, with $v \in V$ and $c \in C$, i.e., operation v starts its execution in cycle c.

Definition 4.9. Operation assignment constraints.
The *operation assignment constraints* ensure that the start time $\phi_1(v)$ of each operation $v \in V$ is assigned to exactly one cycle step:

$$\forall_{v \in V} : \sum_{c \in COEI(v)} x(v,c) = 1.$$

Definition 4.10. Module constraints.
The *module constraints* prevent that too many operations are assigned to the same module type:

$$\forall_{m \in T_M} \quad \forall_{c \in C} : \sum_{v \in V \mid m \in \mu(\tau(v))} \sum_{s=c}^{c+dii(m)-1} x(v,s) \leq n(m).$$

Definition 4.11. Precedence constraints.
The *precedence constraints* ensure that the precedence relations in the DFG are maintained; see for an explanation of these constraints [Gebo92]:

$$\forall_{(v,w) \in E} \quad \forall_{c \in COEI(v) \cap COEI(w)} : \sum_{s \leq c + d_{min}(v)-1} x(w,s) + \sum_{c \leq s} x(v,s) \leq 1.$$

The time constrained scheduling constraints cannot be modelled as strict node packing constraints, because the module constraints above cannot be interpreted as node packing constraints. So, the module constraint cannot represent facets of a node packing problem.

Another modelling of this time constrained scheduling problem is also possible. By specifying an upperbound on the number of modules of each type, binary variables can be used to identify whether a module is used or not. The objective function and constraints can be changed accordingly, but the scheduling problem is still not transformed into a node packing problem. So, time constrained scheduling can be modelled as an IP problem, but the question is how 'tight' such a problem formulation is.

It is therefore important to see what the quality of the linear programming (LP) relaxation of the time constrained IP scheduling problem is, because it indicates the 'tightness' of the IP model. In Figure 4.4 and 4.5, the results of this relaxation for the module library of Table 3.1 have been given, together with the estimates and optimal results that have been presented in Table 3.5 and Table 3.6. The LP relaxations of Figure 4.4 and Figure 4.5 take into account that at least one module of each type is needed.

As can be seen in the figures, the LP relaxations are in no cases more accurate than the lower bound estimates of Chapter 3. In many cases the difference between an LP relaxation and the lower bound estimate is a multiple of the

area of the smallest module. This means that a branch–and–bound process on top of such a LP relaxation is unavoidable to get 'even' with the lower bound estimation results presented in the Tables 3.5 and 3.6.

Furthermore, also in the case of *resource* constrained scheduling, the LP relaxation of the IP scheduling problem is less accurate than the lower bound cycle budget estimates for these examples. For instance, according to the LP

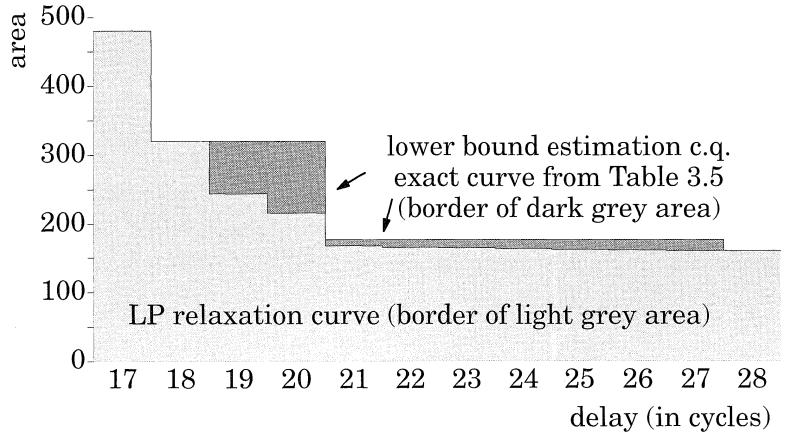


FIGURE 4.4. LP relaxation results for WDELF.

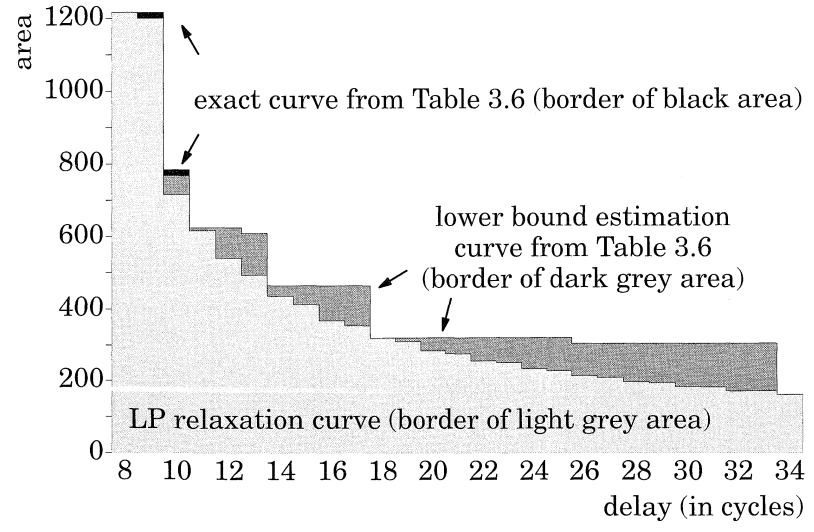


FIGURE 4.5. LP relaxation results for FDCT.

relaxation, the lower bound for the schedule of WDELF with the smallest possible module set is 27 cycles. However, the lower bound estimate from Table 3.5 shows that at least 28 cycles are needed.

The lower bound analyses of Chapter 3 are in general more accurate, and can be obtained more efficiently than the LP relaxation of the IP scheduling model based on node packing. This means that, even in the case of IP based scheduling, it is always more efficient to use the approach of Figure 4.1, i.e., to transform an initial scheduling problem into a time and resource constrained scheduling problem. So, if one wants to use IP scheduling, one should use the same *solution strategy*, see also the discussions in the next section and in Chapter 6.

4.3.3 Enhancements of the IP model

In this section improvements of the IP model for time and resource constrained scheduling are given, which are the result of the exploitation of the node packing model. However, the exercise will show that even more accurate forms of the IP scheduling problem do *not* provide run time advantages.

This section starts with the exact formulation of the time *and* resource constrained scheduling problem. Note that in the previous section, the formulation for time constrained scheduling was given. It will be shown that time and resource constrained scheduling problem can easily be modelled as a pure node packing problem, in contrast to time *or* resource constrained scheduling.

IP problem formulation

In case of time and resource constrained scheduling there is no objective function, but the objective function of the node packing problem, maximizing the sum of all variables, can be taken. Given Definition 4.12, the constraints for a trivial module library are formulated in Definition 4.13, 4.14 and 4.15.

Definition 4.12. IP variables for time and resource constrained scheduling. Recall Definition 2.19, in which it was stated that $\phi_1(v)$ is the start time of operation $v \in V$ in the schedule $\phi \in \Phi$. Let $x(v, c, k) \in \{0, 1\}$ and let $x(v, c, k) = 1$ represent that $\lfloor \phi_1(v) \rfloor = c$, with $c \in C$, and that operation $v \in V$ is bound to module $k \in M$.

Definition 4.13. Operation assignment constraints.
The *operation assignment constraints* ensure that the start time of each operation is assigned to one cycle step only. If the node packing objective function is taken, the equal sign of Definition 4.9 can be replaced by a $\leq$:

$$\forall_{v \in V} : \sum_{c \in \overline{OEI}(v)} \sum_{\xi(k) \in \mu(\{\tau(v)\})} x(v, c, k) \leq 1.$$

Definition 4.14. Module constraints.
The *module constraints* prevent that too many operations are assigned to the same module:

$$\forall_{k \in M} \; \forall_{c \in C} : \sum_{v \in V \mid \xi(k) \in \mu(\{\tau(v)\})} \; \sum_{s=c}^{c+dii(\xi(k))-1} x(v,s,k) \leq 1.$$

Definition 4.15. Precedence constraints.
The *precedence constraints* ensure that the precedence relations in the DFG are maintained; see for an extensive discussion of these constraints [Gebo92]. The difference with Definition 4.11 is related to the IP variables in the formulation.

$$\forall_{(v,w) \in E} \; \forall_{c \in \overline{OEI}(v) \cap \overline{OEI}(w)} :$$
$$\sum_{\xi(k) \in \mu(\{\tau(w)\})} \; \sum_{s \leq c + d_{min}(v)-1} x(w,s,k) + \sum_{\xi(k) \in \mu(\{\tau(v)\})} \; \sum_{c \leq s} x(v,s,k) \leq 1.$$

In almost all cases, the constraints above lead to a fractional solution of the LP relaxation. So the question remains whether modelling the scheduling problem as an IP problem can result in run time efficiency improvements.

IP model improvements

In practice, solving an IP scheduling problem depends heavily on the branch–and–bound process. Applying an IP model for scheduling can therefore only be useful if the LP relaxation detects a certain scheduling decision leading to infeasibility at an earlier stage than the execution interval analysis of Chapter 2.

It was already shown in Section 4.3.2 that the LP relaxation leads to less accurate results than the lower bound analyses of Chapter 3. This means that, in most cases, the LP relaxation of the IP formulation above does not achieve the objective of detecting a certain scheduling decision leading to infeasibility at an earlier stage. The only way the LP relaxation can achieve an early detection is when the IP constraints formulated above can be extended with *violated constraints*. This means that constraints have to be found which reduce the polyhedron, but do not affect the convex hull of it, and which are violated by the current LP solution.

In [Chau93], two new sets of constraints are proposed to tighten the IP model. The first set of constraints deals with the fact that all the predecessors of an operation must be scheduled on the set of available modules before the

operation itself can be scheduled. The proposed method is a proper subset of, i.e., similar to but less accurate than, the execution interval analysis of Chapter 2. So this set of constraints from [Chau93] is obsolete if the execution interval analysis is applied.

The second set of constraints from [Chau93], which will be dealt with in the remainder of this section, is more interesting, because it combines precedence and resource constraints. Here lies an advantage of the IP scheduling model. Most scheduling techniques, like the execution interval analysis of Chapter 2, list scheduling, etc., consider the precedence and resource constraints alternately.

The IP scheduling constraints mentioned until now also treat the resource and precedence constraints separately. This is the reason that such an IP model does not detect an infeasibility at an earlier stage than the execution interval analysis. By combining precedence and resource constraints into a unified constraint set, an IP model can possibly add to the solution method by means of an early detection of infeasibility.

Below, such a formulation, i.e., the second constraint set originating from [Chau93], is presented. Let $V_{m,c}$ be the set of operations that must be mapped to modules of type $m \in T_M$, in cycle $c \in C$, i.e. $V_{m,c} = \{v \in V \mid m \in \mu(\{\tau(v)\}) \wedge c \in \overline{OEI}(v)\}$.

The set $V_{m,c}$ results in a module constraint, see the beginning of this section, indicating that not more than one operation from $V_{m,c}$ can be mapped to a module $k \in K$ in cycle $c \in C$, where K is the set of modules of type m. If there are also precedence constraints between operations from $V_{m,c}$, then these precedence constraints can be incorporated in the resource constraints to tighten the IP model.

In [Chau93] these precedence constraints in $V_{m,c}$ are used to construct a minimal, but not unique, clique cover $V_{m,c} = \bigcup_{l=1}^{p_{m,c}} V_{m,c,l}$, where each $V_{m,c,l}$ represents a clique based on the precedence arcs in the DFG. See [Chau93] for an extensive discussion on this topic. Let $p_{m,c,v}$ give the corresponding number of cliques that contain $v \in V_{m,c}$. Then it can be proven that the following constraints are also valid [Chau93]. It is assumed that $p_{m,c} > |K|$, because otherwise the corresponding resource constraints can be omitted.

$$\forall_{m \in T_M} \; \forall_{c \in C} : \sum_{v \in V_{m,c}} \sum_{s=c}^{c+dii(m)-1} \sum_{k \in K} c_{m,c,v} \times x(v,s,k) \leq |K|,$$

$$\text{where } c_{m,c,v} = \max\{1, |K| + p_{m,c,v} - p_{m,c}\}.$$

If $|K| + p_{m,c,v} - p_{m,c} \geq 1$ for all $v \in V_{m,c}$, then there is a (linear) combination of clique constraints that is tighter than the constraint set above. Such constraints are more in accordance with the fact that clique constraints can possibly represent facets of the polytope of a node packing problem. These clique constraints are given in theorem 4.2.

THEOREM 4.2.
If $|K| + p_{m,c,v} - p_{m,c} \geq 1$ for all $v \in V_{m,c}$, then the linear combination of the following valid clique constraints is tighter than the constraints above, i.e., then the following constraints lead to a tighter model.

$$\forall_{m \in T_M} \; \forall_{c \in C} \; \forall_{V_{m,c,l} \subseteq V_{m,c}} : \sum_{v \in V_{m,c,l}} \sum_{s=c}^{c+dii(m)-1} \sum_{k \in K} x(v,s,k) \leq 1.$$

PROOF.
The summation over all cliques $V_{m,c,l} \subseteq V_{m,c}$ leads to:

$$\forall_{m \in T_M} \; \forall_{c \in C} : \sum_{v \in V_{m,c}} \sum_{s=c}^{c+dii(m)-1} \sum_{k \in K} \left(p_{m,c,v} \times x(v,s,k)\right) \leq p_{m,c}.$$

Multiplying the left and right hand sides with $(|K| / p_{m,c})$ yields:

$$\forall_{m \in T_M} \; \forall_{c \in C} : \sum_{v \in V_{m,c}} \sum_{s=c}^{c+dii(m)-1} \sum_{k \in K} \left(\frac{|K| \times p_{m,c,v}}{p_{m,c}} \times x(v,s,k)\right) \leq |K|.$$

This means that Theorem 4.2 is true if and only if:

$$\frac{|K| \times p_{m,c,v}}{p_{m,c}} \geq |K| + p_{m,c,v} - p_{m,c}.$$

The inequality is equivalent to:

$$(p_{m,c})^2 - (p_{m,c} \times p_{m,c,v}) + |K| \times (p_{m,c,v} - p_{m,c}) \geq 0.$$

Both $|K|$ and $p_{m,c,v}$ are smaller than or equal to $p_{m,c}$, so they can be substituted by $|K| = p_{m,c} - \alpha$ and $p_{m,c,v} = p_{m,c} - \beta_{m,c,v}$, with α and $\beta_{m,c,v}$ non–negative. Then the inequality above is true if and only if:

$$(p_{m,c})^2 - \left((p_{m,c})^2 - (\beta_{m,c,v} \times p_{m,c,v})\right) + (p_{m,c} - \alpha) \times (-\beta_{m,c,v}) \geq 0.$$

This results in: $\alpha \times \beta_{m,c,v} \geq 0$, which is always true because α and $\beta_{m,c,v}$ are non–negative. ■

The constraint set of Theorem 4.2 denotes a set of clique constraints that were not captured by the three sets of constraints established before. However, in general, a large number of other (clique) constraints can be found that are still violated by the LP relaxation, even if the constraints of Theorem 4.2 are added to the problem formulation [Leeu95]. In many cases, these violated constraints are also based on combinations of both resource and precedence constraints.

In [Leeu95], a comprehensive survey can be found of violated constraints. However, even by adding more violated constraints, in most cases the IP model still does not detect infeasibility any earlier than the execution interval analysis of Chapter 2. Thus, in the experiments we conducted, the addition of extra constraints lead to a loss of run time efficiency of the IP problem formulation. Due to this, only the first three constraint sets at the beginning of this section are considered in the following section.

4.4 Experiments and results

The two time and resource constrained scheduling approaches presented in this chapter, i.e., the approach based on BSGs and the IP approach, have been implemented in C++ using the interface of the NEAT system. Both approaches use the execution interval analysis before scheduling, thus reducing the number of variables and constraints for the IP scheduler.

The number of IP variables is directly related to the amount of freedom of the operations, which is expressed in the size of the $\overline{\text{OEI}}$s. So, if the average freedom of the operations decreases, the number of variables will also decrease. The solution space of any IP scheduling model does not change due to the execution interval analysis, so a decrease in the number of variables and constraints leads in most cases to run time efficiency improvements. The public domain MILP solver which was used for the module selection problem, see Section 3.7, also has been used to solve the IP problems.

In Table 4.1 and 4.2, results for FDCT and the fifth order wave digital filter (WDELF) are given. The tests were run on an HP9000/735 workstation. The last column of each table shows two numbers. The first one indicates the number of infeasible branches in the BSG scheduling approach, i.e., the number of times an operation was matched to some $\overline{\text{MEI}}$ without detecting immediately that the matching leads to an incorrect ordering. The second number indicates the number of times a matching was immediately detected to be incorrect, so the branch–and–bound process does not follow such a branch at all.

TABLE 4.1. Scheduling results for FDCT.

|C|	#multipliers (d = dii = 2)	# adder / subtracters (d = 1)	CPU times IP scheduler (sec)	CPU times BSG scheduler (sec)	# infeasible branches BSG
8	8	4	1	0.5	0 / 0
9	8	4	3	1.3	0 / 0
9	8	3 (not feas.)	13,988	0.5	0 / 3
10	5	4	20	1.2	0 / 0
10	5	3 (not feas.)	> 2 hrs	> 2 hrs	–
11	4	3	128	1.4	0 / 6
12	4	3	102	1.2	0 / 2
13	4	2	501	1.0	0 / 1
14	3	2	230	1.1	0 / 2
15	3	2	243	1.3	0 / 3
16	3	2	590	1.2	0 / 1
17	3	2	853	1.1	0 / 1
18	2	2	240	1.0	0 / 0
19	2	2	449	1.3	0 / 3
20	2	2	496	1.1	0 / 0
21	2	2	434	1.2	0 / 0
22	2	2	2,496	1.2	0 / 0
23	2	2	854	1.2	0 / 0
24	2	2	1,590	1.2	0 / 0
25	2	2	453	1.2	0 / 0
26	2	1	3,093	1.7	0 / 7
27	2	1	> 2 hrs	1.9	0 / 7
28	2	1	2,331	1.2	0 / 0
29	2	1	4,765	1.2	0 / 0
30	2	1	4,757	1.2	0 / 1
31	2	1	> 2 hrs	1.3	0 / 0
32	2	1	5,954	1.2	0 / 0
33	2	1	4,467	1.2	0 / 0
34	1	1	> 2 hrs	1.2	0 / 0

The tables show that WDELF is a 'simple' example in comparison with FDCT: for WDELF there was not a single case in which the scheduler tried a wrong assignment of an operation to an $\overline{\text{MEI}}$. Both schedulers have about the same run times for WDELF, although the IP scheduler tends to become less efficient as the number of cycles increases. In general, the number of IP variables and constraints increases as the cycle bound increases.

TABLE 4.2. Scheduling results for WDELF.

|C|	#multipliers (d = dii = 2)	# adder / subtracters (d = 1)	CPU times IP scheduler (sec)	CPU times BSG scheduler (sec)	# infeasible branches BSG
17	3	3	1.6	0.5	0 / 0
18	2	2	1.8	0.4	0 / 0
19	2	2	2.2	0.5	0 / 0
20	2	2	2.9	0.8	0 / 0
21	1	2	1.8	0.9	0 / 0
22	1	2	3.8	1.0	0 / 0
23	1	2	5.3	1.1	0 / 0
24	1	2	6.0	1.1	0 / 0
25	1	2	7.2	1.2	0 / 0
26	1	2	8.0	1.1	0 / 0
27	1	2	10.6	1.1	0 / 0
28	1	1	11.1	1.0	0 / 0

The results for FDCT clearly show the limits of an IP scheduler, which becomes very inefficient for that example. The inefficiency is due to the parallelism and the symmetry of FDCT, which makes it very difficult to obtain an integral solution for the IP formulation of the scheduling problem. The scheduler based on the BSGs remains very efficient for FDCT. The only exception is the infeasible module set for 10 cycles; the search space for this example is large for any kind of scheduler and there is no method (yet) which solves this problem instance efficiently.

Chapter 5 Retargetable Code Generation

5.1 Introduction

The previous chapters of this thesis are targeted towards the synthesis and optimization of datapaths of hard–wired VLSI circuits, i.e., application specific ICs (ASICs). Another class of VLSI circuits that is gaining a lot of interest are the so called application *domain* specific processors or application specific *instruction–set* processors (ASIPs), especially in the field of digital signal processing (DSP). These DSP cores, which are tailored towards specific application domains, are becoming increasingly popular due to their advantageous trade–off between *flexibility* and *cost*; see Figure 5.1.

Such a core is relatively flexible in comparison to an ASIC: different algorithms can be mapped on it, while an ASIC is a tailored solution for only one algorithm. On the other hand, domain specific DSP cores are more targeted towards a specific application domain, making them more suitable for such a domain than general processors: dedicated hardware is available for time critical tasks, e.g., a module performing a FFT butterfly in a single cycle. So, domain specific cores are applied to allow the *fine tuning* of an application. Therefore a new research topic is emerging: 'retargetable' code generation for domain specific DSP cores and other application specific instruction–set processors.

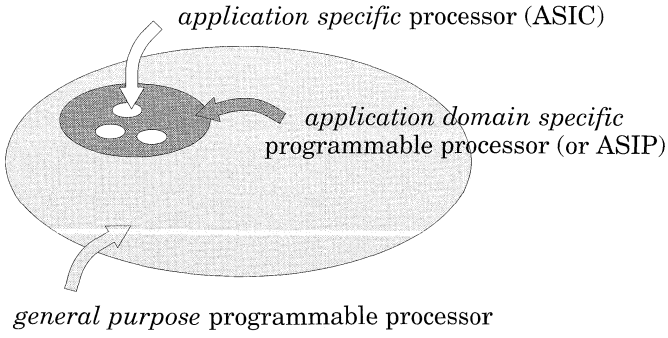


FIGURE 5.1. Processor spectrum indicating application domain sizes

Tight feasibility constraints

The size of the application domain of a core is inversely proportional to the required efficiency. Experiments show that in some cases the utilization of the functional units, in this chapter called operation processing units (OPUs), in an application domain specific core exceeds 90% of the total cycle budget [Strik95]. So, there is a need for a code generator capable of generating very efficient, i.e., compact, microcode under *tight feasibility constraints*. These feasibility constraints originate from the algorithm, i.e., the timing and precedence constraints, and from the DSP core and instruction set, i.e., the resource constraints. The combination of these constraints results in high OPU utilization rates, while the objective is to find a feasible, i.e., correct, mapping from algorithm to domain specific DSP core.

Because of the relatively high efficiency required, the use of domain specific DSP cores leads to new design tools and / or methods [Paul92]. Despite the fact that many researchers consider (retargetable) code generation as a separate research area, it can be seen as a natural part of, and the last step in, an architectural synthesis system; recall Section 1.3 and 1.4. Retargetable code generation is therefore a good test case for the validity of the synthesis flow depicted in Section 1.4.

Code generation can roughly be divided into three interdependent subtasks: code selection, instruction scheduling and register binding. Previous approaches concentrate on the *code selection* problem [Marw93], [Liem94], [Praet94] or the *register binding* problem [Cheng94], [Lann94]. However, under the regime of tight feasibility constraints, heuristic approaches for the *instruction scheduling* problem render unsatisfactory results for many instances, i.e., they often do not find a feasible schedule within the throughput constraints although such schedules do exist.

The existing scheduling methods do not produce satisfactory results because they are hampered by the combination of tight timing and resource constraints, instead of exploiting them. On one hand, in the field of software compilation, the completion time of an algorithm is often not that important in comparison with the hard constraints on the throughput of DSP algorithms. An exception is [Chou94], but in that approach the resulting schedule is fully serial, so no parallelism in the datapath is allowed. However, Section 5.3 shows that a lot of parallelism in the datapath is needed in DSP applications. On the other hand, in the field of hardware compilation, most architectural synthesis systems do not handle hard resource constraints correctly, i.e., they often just add resources in order to find a solution.

This chapter therefore concentrates on *modelling* the resource and instruction set conflicts and *exploiting* the combination of all possible constraints, by applying the methods developed in Chapter 2 and 4. Because

of the large number and the tightness of the different resource constraints, these methods are highly suitable for the code generation problem. The target cores considered are in–house DSP cores for which the application domains are relatively small and the microcode efficiency must be high. As a consequence of the use of in–house DSP cores, design rules can be developed and enforced for the core architectures and the instruction set definitions, to facilitate the code generation approach presented in this chapter. The design rules can be found in [Strik94] and will only be mentioned in this chapter if they are a prerequisite to understand the code generation approach.

Chapter overview

The outline of this chapter is as follows. In Section 5.2, it is shown how different resource constraints, with respect to OPUs, memory accesses, buses and multiplexers can be modelled uniformly. In our case, the instruction set cannot activate all modules in the datapath simultaneously, in order to limit the instruction word width. So, the instruction set definition imposes additional restrictions on the amount of parallelism in the datapath. A method has been developed, such that these restrictions can be handled as normal resource conflicts. This means that the instruction set conflicts are modelled statically before scheduling, making a compaction pass, which is used in other code generation systems like CodeSyn [Paul94], superfluous. Note that register file size constraints are not yet dealt with automatically in the approach presented here.

In Section 5.3, the different resource conflicts are cast into the bipartite graph matching formulation introduced in Chapter 2. The method is based on that formulation, but is completely tailored to the code generation flow. Two methods to construct $\overline{\text{BSG}}$s are proposed and it is explained which of the two leads to the best formulation. Furthermore, the scheduling method based on finding a correct complete matching in the $\overline{\text{BSG}}$s, see Chapter 4, is also applied to the code generation problem. Section 5.4 presents results for real life examples demonstrating the efficiency of the approach.

5.2 Modelling resource and instruction set conflicts

5.2.1 Register transfer generation

For the code generation flow of the application domain specific DSP cores, tools and therefore parts of the synthesis flow from the Mistral2^{TM} compiler [Nieu94] have been reused. Preceding the instruction scheduling step, register transfers (RTs) and their dependencies are generated from the input

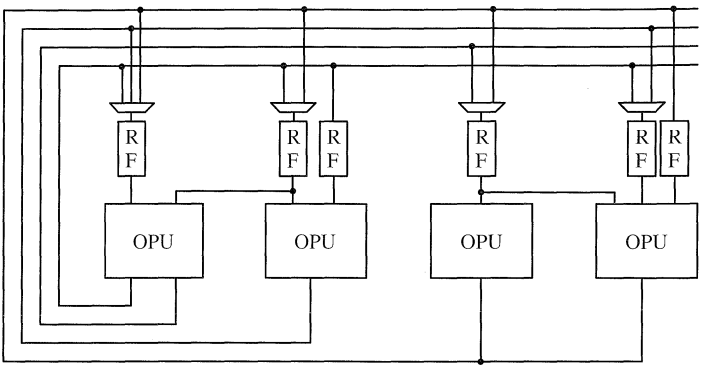


FIGURE 5.2. Generic datapath architecture.

description using a generic architectural model. Figure 5.2 shows the architectural model in which a number of, possibly pipelined, OPUs appears. Each OPU input is connected to a register file (RF). The outputs of the OPUs are connected to RFs via buffers, buses and (optionally) multiplexers.

RTs correspond to a complete, in this case single clock cycle, path from source register files to a destination register file. So the RTs already contain the binding information regarding the resources on which actions from the input description are mapped. RTs are fully characterized by the resources that are used and the mode in which these resources have to operate.

An example is given in Figure 5.3, in which the destination RF is reg_2_ram_1 and the origin RFs are reg_1_acu_1 and reg_2_acu_1. The resources of the RT are found at the left hand side of the '=' sign and the mode, or 'usage', is positioned at the right hand side. Figure 5.3 shows an addition on an OPU called 'acu_1' and the storage of the result in a register of the OPU called 'ram_1' via the second of the two available multiplexer inputs.

```
Dest_1:   reg_2_ram_1 <-  Source_1: reg_1_acu_1,
                          Source_2: reg_2_acu_1,
          acu_1           =  add,
          buf_1_acu_1     =  write,
          bus_1_acu_1     =  'add(Source_1, Source_2)',
          mux_2_ram_1     =  pass[0, 1].
```

FIGURE 5.3. Example register transfer.

The RT generation step has similarities to the instruction set matching and selection techniques of other approaches like [Liem94]. However, in this case the step is performed by the existing RT generation tool from the Mistral2™ compiler. The tool uses the architectural model of Figure 5.2 as a starting point. Register files and buses that are merged in the actual core are taken into account by modifying the generated RTs [Strik94]. Restrictions due to the instruction set are taken into account as well by modifying the RTs, as will be shown in Section 5.2.3.

5.2.2 Resource conflicts

RTs can be combined into a single instruction by a scheduler if they do not have any resource conflicts. If RTs do not use the same resources, then they can be combined. Otherwise it depends on the usage of these resources. In Figure 5.4a, an RT is given that can be combined with the RT of Figure 5.3, because the usage of the shared resources is the same. The only difference between the two RTs is the destination RF, see the resources in bold in Figure 5.4a. In Figure 5.4b, an RT has been given that *cannot* be packed into the same instruction as the RT of Figure 5.3. The OPU is used differently, see the usage in bold in Figure 5.4b, which leads to a conflict.

Let M be the set of resources in the datapath, i.e., not only the OPUs, but also buses, multiplexers and so on. For each resource $k \in M$, the following conflict graph CG(k) can be constructed. The CGs will be used in the discussions and methods that are presented in the sequel of this chapter.

Definition 5.1. Conflict graph.
CG(k) for resource $k \in M$ is an undirected graph represented by a tuple (W', A'), where:

- W' is the set of vertices representing the RTs that use resource k;
- $A' \subseteq W' \times W'$ is the set of arcs, such that there is an arc $(v_i, v_j) \in A'$ if and only if $v_i \in W'$ and $v_j \in W'$ have a different usage of resource k.

(a):

```
Dest_1: reg_2_acu_1 <- Source_1: reg_1_acu_1,
                       Source_2: reg_2_acu_1,
        acu_1          = add,
        buf_1_acu_1    = write,
        bus_1_acu_1    = 'add(Source_1, Source_2)',
        mux_1_acu_1    = pass[0, 1].
```

(b):

```
Dest_1: reg_2_ram_1 <- Source_1: reg_1_acu_1,
                       Source_2: reg_2_acu_1,
        acu_1          = addmod,
        buf_1_acu_1    = write,
        bus_1_acu_1    = 'add(Source_1, Source_2)',
        mux_2_ram_1    = pass[0, 1].
```

Figure 5.4. RTs without *(a)* or with *(b)* a conflict with RT of fig. 5.3.

The CGs point out that the resource conflicts are modelled statically before scheduling. Two RTs can be packed into one instruction if they are not adjacent to each other in the CG of any resource (and, of course, if dependency relations between RTs are not violated). For all possible cliques of RTs only one RT at the time can be packed into one instruction. Solving the resource conflicts of a design problem can therefore be interpreted as finding independent sets of RTs for each cycle step, such that the dependencies in the DFG are not violated.

5.2.3 Instruction set conflicts

A given DSP core is not only specified by its datapath but also by its instruction set. In our case, the instruction set cannot steer all modules in the datapath simultaneously, in order to limit the instruction word width. So, the instruction set definition imposes additional restrictions on the amount of parallelism in the datapath.

These restrictions are modelled by adding artificial 'resources' to the RTs. As an example, *load immediate (LDI)* is often a separate instruction class, or 'optype', during which no other RTs can take place. In Figure 5.5, the artificial resource LDI with the mode false has been added to model this instruction set conflict. The advantage of modelling the instruction set conflicts as artificial resource conflicts is twofold. First of all, a uniform modelling of all conflicts is obtained. Secondly, and more importantly, the domain reduction techniques of Chapter 2 can be applied for the instruction set conflicts, if they can be modelled as artificial resource conflicts.

There is however a catch in this approach. The CGs in Section 5.2.2 showed that the resource conflicts from the datapath are modelled statically before scheduling. The bipartite graph matching formulation introduced in Chapter 2 uses a similar static modelling of conflicts as well. The question arises

---

```
Dest_1:  reg_2_ram_1 <-  Source_1: reg_1_acu_1,
         acu_1           =  pass,
         buf_1_acu_1     =  write,
         bus_1_acu_1     =  'pass(Source_1)',
         mux_2_ram_1     =  pass[0, 1],
         LDI             =  false.
```

FIGURE 5.5. Example RT with artificial resource.

whether such a static modelling of the instruction set conflicts imposes any restrictions or demands on the instruction set definition itself. Recall that in-house DSP cores are considered, so the definition of the instruction sets can be controlled to make them suitable for the code generation method presented in this chapter. To model the instruction set conflicts statically before scheduling, the instruction set conflict graph of Definition 5.2 is used.

DEFINITION 5.2. Instruction set conflict graph.
Let an RT class be a set of RTs which use the same resources in the same mode. We can then define the following instruction set conflict graph (ICG), which has similarities to the CGs for the different resources in the datapath. ICG is an undirected graph represented by a tuple (C', A'), where:

- C' is the set of vertices representing the RT classes;
- $A' \subseteq C' \times C'$ is the set of arcs such that there is an arc $(cl_i, cl_j) \in A'$ if and only if two RTs from the respective RT classes $cl_i \in C'$ and $cl_j \in C'$ *cannot* be combined into one instruction because of an instruction set conflict.

Such an ICG is a valid model for the instruction set conflicts, if and only if the following condition is satisfied. For *every* arbitrary set of operations, for which the set of their resource classes form an independent set within the ICG, there must be a legal instruction. This means, amongst others, that NOP (no operation) must be a possible instruction, as well as an RT from each individual RT class on its own. So the use of an ICG puts some demands on the definition of the instruction sets. However, these demands are very well acceptable in real life situations and have no influence on the efficiency of the implementation.

Artificial resource conflicts

The conflicts modelled by the ICG can be transformed into artificial RT resource conflicts in two ways. First of all, an approach can be followed which is implied by Figure 5.5. For each RT class $cl \in$ ICG, a resource is added to the corresponding RTs with the mode 'true'. To all RTs whose classes are adjacent to cl in ICG the same resource is added, but with the mode 'false'. In Figure 5.6, as an example, a resource 'S' with the mode 'true' can be added to RTs belonging to the class S. RTs belonging to the classes X and Y obtain a resource 'S' as well, but with the mode 'false'. With this model, the number of artificial resources equals the number of RT classes, which is six in the example.

Secondly, we can find a minimal set of cliques such that all arcs in the ICG are covered at least once. For each clique, all corresponding RTs obtain a resource identifying that clique, with a mode corresponding to their individ-

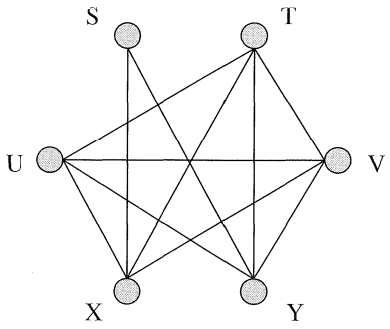


FIGURE 5.6. Example instruction set conflict graph (ICG).

ual RT class. Such a clique cover for Figure 5.6 is given by the set {T, U, V, X}, {T, U, V, Y}, {S, X} and {S, Y}. As an example, a resource 'TUVX' is added to RTs belonging to the classes T, U, V and X, with a mode 'T', 'U', 'V' or 'X' depending on their class. With this model, the number of artificial resources equals the number of cliques, which is four in the example.

The run time efficiency of the instruction scheduling step can depend on the total number of resources, see the following sections. Therefore, the choice how to model the instruction set conflicts is based on the number of artificial resources each model introduces.

5.3 Instruction scheduling based on BSGs

5.3.1 Background

Because of the large number and tightness of the different resource and instruction set constraints, the execution interval analysis and scheduling approach based on BSGs is highly suitable for the retargetable code generation problem. A bipartite graph matching formulation is therefore used to map algorithms to the application domain specific DSP cores. However, an adaptation of the approaches of Chapter 2 and 4 is needed to make them suitable for the code generation flow, due to the following three reasons.

- In general, the number of times a certain resource is occupied is not known beforehand in the RT model used. Consider the case in which two RTs do not have any resource conflicts, although they do use the same resources, i.e., they use resources in the same mode. In that case it depends on the final schedule whether these resources are used once or twice for the two RTs.

- In the approaches of Chapter 2 and 4, only resource constraints with respect to functional units, i.e., OPUs, were taken into account. The methods are extended for all other resource types that are part of an RT, i.e., constraints with respect to memory accesses, buses, multiplexers and the instruction set are now considered as well.

- In many cases, the loops in a data flow graph have to be 'folded' to satisfy the throughput constraints, see Definition 5.3. This was not considered in the previous chapters.

DEFINITION 5.3. Loop folding.
Consider a data flow graph with possibly cyclic paths. Let the throughput constraint of the DFG be defined by a list of clock cycles D and the execution delay constraint by a list of clock cycles C. Furthermore, let the difference in clock cycles between two executions (instantiations) of an RT in the DFG be equal to $|D|$, i.e., the period of the RTs is $|D|$. If it is not possible to construct a schedule for the DFG, such that $|C|$ equals $|D|$, then the DFG has to be 'folded'. If $|C|$ equals $|D|$, then the DFG is said to be unfolded. If $|C|$ equals $2*|D|$, then the graph is folded once, if $|C|$ equals $3*|D|$, then the graph is folded twice, etc.

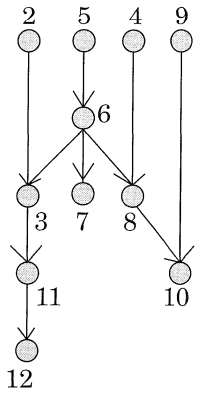
5.3.2 Time potentials

Let the execution delay of a DFG be equal to $|C|$ clock cycles. Then, in case of loop folding, an RT is not repeated every $|C|$ cycles, but every $|D|$ cycles, with $|D| < |C|$. For detecting resource constraints, it is not sufficient anymore to check whether two RTs occupy a resource in the same clock cycles, as the following example will show.

Consider two RTs, v and w, with $\phi(v) = [1, 2]$ and $\phi(w) = [9, 10]$. If $|D|$ is equal to eight, then the two RTs may not have resource conflicts: the second instantiation of v occurs in the same clock cycle as the first instantiation of w. Therefore, the resource conflicts have to be considered with the period $|D|$ of the RTs in mind. In Figure 5.7, a DFG has been given with $|D| = 4$ cycles and $|C| = 8$ cycles. As a next example, RT 5 is executed in cycle zero for the first time, while RT 3 is scheduled in cycle four for the first time. However, because $|D| = 4$, the second instantiation of RT 5 takes place in cycle four, so RT 5 and RT 3 may not have resource conflicts.

The schedule of an RT is fully defined by the first clock cycle in which it is executed together with its period $|D|$. If for two RTs, v and w, $\phi_1(v) \bmod |D|$ and $\phi_2(w) \bmod |D|$ are equal, then their start times are said to be in the same *time potential*, or *phase*. So, RT 5 and RT 3 of Figure 5.7 may not have resource conflicts, because they are both scheduled in the same time potential, i.e., in time potential zero.

data flow graph:



schedule (I D I = 4 cycles, I C I = 8 cycles):

time potential	cycle	RT	
	0	5	
	1	6	pre–amble
	2		
	3	2	
0	4	5 3	
1	5	6 4 11	loop body
2	6	8 9 12	
3	7	2 7 10	
		3	
		4 11	post–amble
		8 9 12	
		7 10	

FIGURE 5.7. Example of a folded DFG with schedule.

Figure 5.7 shows that after I C I – I D I cycles the *loop body* of the schedule begins. In the loop body, the full concurrency of the RTs is present. The *pre–amble* precedes the loop body, so RTs are scheduled for the first time in either the pre–amble or the loop body. If the loop body is not executed an infinitely number of times, then there is also a *post–amble*. In the post–amble, all RTs, that were not executed in the pre–amble, are executed for the last time.

Because the full concurrency of the RTs is present in the loop body, the list of time potentials D can be interpreted as the clock cycles of the loop body. In the application domain specific cores considered, the data introduction intervals (diis) of all OPUs are equal to one. Due to that reason, the schedule of each RT corresponds to the occupation of resources during one time potential in the loop body. So, resource conflicts occur when two RTs are scheduled at the same time potential in the loop body, while they use the same resource in a different mode.

5.3.3 Definitions

In Definition 2.27, the operation execution interval OEI, based on clock cycles, has been defined. A similar execution interval can now be determined based on the time potentials, see Definition 5.4.

Definition 5.4. Operation time potential interval.
Consider a data flow graph, and let the set of constraints consist of precedence, resource and timing constraints. Let D be the list of time potentials denoting the throughput constraint and C the list of cycles denoting the execution delay constraint. The operation time potential interval OTI(v) of $v \in V$ is then defined by the following set of time potentials:

$$OTI(v) = \bigcup_{\phi \in \Phi} \left(\phi_1(v) \bmod |D|\right).$$

In Definition 5.4, only the start times of RTs are considered. The reason is that in our case the diis of all OPUs are equal to one, i.e., every RT 'occupies' an OPU for only one time potential. However, the execution delay of an RT can be more than one clock cycle.

Note also that, in case there is no execution delay constraint C and the register file sizes are not taken into account, all RTs have all time potentials in their OTI. In case of an execution delay constraint C, the OTIs cannot be calculated in polynomial time anymore. This is due to the same reasons the OEIs in Chapter 2 can, in general, not be determined in polynomial time. Therefore, the conservative estimate of an OTI is defined, see Definition 5.5.

Definition 5.5. Conservative estimate of time potential interval.
The conservative estimate $\overline{OTI}(v)$ of an operation $v \in V$ is an estimate of OTI(v), satisfying $\overline{OTI}(v) \supseteq OTI(v)$.

Furthermore, loop folding can result in extra timing constraints for the RTs as well. In case each instantiation of a value is stored in the same memory location, extra timing constraints are required. These constraints have to assure that the consumption of a value occurs before a new version of the value is produced: $\forall_{w \in succ(v)} : \phi_1(w) \leq \phi_2(v) + |D| - 1$.

It is now possible to formulate the exact problem definition related to the instruction scheduling problem, see Definition 5.6. Recall that the only conflicts not considered in this problem formulation are the conflicts that can result from limited register file sizes.

Definition 5.6. Instruction scheduling problem.
Given a DFG and a set of, possibly artificial, resources M to which the RTs in the DFG have been mapped. Let D be the list of time potentials denoting the throughput constraint and C the list of clock cycles denoting the execution delay constraint. Let |D| be the period of all RTs, and let dii(m) = 1 for all $m \in T_M$. Furthermore, let each instantiation of a value be stored in the same memory location, see the discussion above. Determine whether the set of schedules Φ is empty or not. If Φ is not empty, calculate a correct $\phi(v)$ for all operations $v \in V$.

5.3.4 Module execution intervals

All the considerations mentioned above have a huge impact on the definition and calculation of the MEIs, which account for the resource conflicts. Resource conflicts occur when two RTs are scheduled at the same time potential in the loop body, while they use the same resource in a different mode. Therefore, the MEIs have to be defined based on the *time potentials*, and not on the clock cycles. So, a redefinition of Definition 2.33 is needed.

Consider an arbitrary, possibly artificial, resource k from the datapath or instruction set. This is because the $\overline{\text{MEI}}$s can now be calculated for each resource separately, recall that the binding to resources has already taken place. Let W_k be the set of RTs that use resource k, the index 'k' is dropped whenever possible. In the instruction scheduling approach of this chapter, a schedule $\phi \in \Phi$ imposes a notion of order on the set W by assigning *time potentials*, and not start times, to the elements $v \in W$. Any time some RTs have equal time potentials this tie is broken arbitrarily, in analogy to Definition 2.32.

Definition 5.7. Linear ordering of scheduled RTs.
Let $\lll$ represent an arbitrary linear ordering on the set W. Furthermore, let $\phi_D(v)$ be the short hand notation for $\lfloor \phi_1(v) \rfloor \bmod |D|$. Given a schedule $\phi \in \Phi$, $\prec_\phi$ is a linear ordering relation defined by:

$$\forall_{\phi \in \Phi} \; \forall_{v,w \in W} : v \prec_\phi w \Leftrightarrow \left(\phi_D(v) < \phi_D(w)\right) \vee \left(\phi_D(v) = \phi_D(w) \wedge v \lll w\right).$$

Again, $\pi_\phi(i)$ is defined as the i^{th} RT under the linear order induced by the schedule ϕ according to Definition 5.7. Module execution intervals can now be defined as follows, which is analogous to Definition 2.33.

Definition 5.8. Module execution interval MEI.
Consider the set of RTs from W assigned to the value $i \in [1, |W|]$ over the set of all schedules Φ. Furthermore, let $\phi_D(i)$ be the short hand notation for $\lfloor \phi_1(\pi_\phi(i)) \rfloor \bmod |D|$, i.e., $\phi_D(i)$ is the time potential of the i^{th} RT $\in W$ in the schedule $\phi \in \Phi$. Then MEI(i) is defined by the following interval of time potentials. Note that, in this case, the MEIs are defined by time potentials and not by clock cycles:

$$\text{MEI}(i) = \left[M_1(i),\ M_2(i)\right] = \left[\min_{\phi \in \Phi} \phi_D(i),\ \max_{\phi \in \Phi} \phi_D(i)\right].$$

Thus, for any schedule $\phi \in \Phi$, the time potential of the i^{th} RT must be within the interval of time potentials of MEI(i). Again, we have to be content with estimates of those bounds which do not limit the solution space. For this reason the estimates have to satisfy Definition 2.34.

5.3.5 Construction of BSGs per resource

In this section, the exact details of the $\overline{\text{MEI}}$ calculation are given when $\overline{\text{BSG}}$s are constructed for each, possibly artificial, resource separately. In the next section, another formulation will be introduced that constructs the $\overline{\text{BSG}}$s differently and in a *better* way, see also [Timm95b]. This means that one can skip reading this section. The reason to give both approaches is to show what the difference between a good and a less good formulation is. Furthermore, the formulation in this section is *more general* than the formulation of Section 5.3.6, and might therefore be more appropriate for other scheduling problems that could arise in the future. However, we restrict ourselves in this section to a more or less constructive explanation of the method.

Definition of main auxiliary variables

Because, in general, it is not known beforehand how many times a resource will be occupied, the $\overline{\text{MEI}}$s cannot be calculated directly if $\overline{\text{BSG}}$s are constructed per resource. For that reason, one must start with determining the maximum number of RTs that can be executed within the first c time potentials of the loop body, $c \in [0, |D|-1]$. Let 'usages' be the set of different usages (modes) of resource m, and let $W_u = \{v \in W \mid v \text{ uses resource k with mode u}\}$. For each mode $u \in$ usages, the following values are determined.

- $\Psi_u(c)$: a lower bound estimate of the minimum number of cycles within the first c time potentials that resource k must be occupied by RTs from W_u.
- $\Xi_u(c)$: an upper bound estimate of the maximum number of RTs from the set W_u that can possibly be executed if the number of occupations with usage u equals $\Psi_u(c)$.

With the values $\Psi_u(c)$, $u \in$ usages, it is possible to determine the minimum total number of time potentials in which resource k must be occupied within the first c potentials. This number is given by:

$$\text{occ}(c) = \sum_{u \in \text{usages}} \Psi_u(c).$$

The corresponding maximum total number of RTs that can possibly be executed if the number of occupations of resource k equals occ(c) is given by:

$$\text{rts}(c) = \sum_{u \in \text{usages}} \Xi_u(c).$$

Let holes(c) be the number of time potentials within the first c time potentials in which resource k will never be occupied by an RT, because there isn't any RT that can be executed in such a time potential.

If $c - occ(c) - holes(c) < 0$ for any resource or range of time potentials, then the combination of timing, resource and instruction set constraints is not feasible and the set of schedules Φ is empty. If $c - occ(c) - holes(c) > 0$, then the number of time potentials in which resource k is occupied during the first c time potentials may exceed occ(c).

Calculation of auxiliary variables

Let the list of time potentials D be ordered from 0 to $|D|-1$. Each subset W_u can then be split into the following two disjoint lists: $W_{u,a} = \{v \in W_u \mid v$ can possibly be scheduled in time potential 0 *and* v can possibly be scheduled in time potential $|D|-1\}$ and $W_{u,b} = W_u \setminus W_{u,a}$. $W_{u,a}$ is the list of RTs that can either be scheduled at all time potentials, or have a discontinuous interval of time potentials in which they can possibly be scheduled. $W_{u,b}$ is the list of RTs with a continuous interval of time potentials. In Figure 5.8, $W_{u,a} = \{g, h, i\}$ and $W_{u,b} = \{a, b, c, d, e, f\}$: g and h have a discontinuous interval of time potentials and i can be scheduled at all time potentials.

For all $v \in W_{u,b}$, $\overline{OTI}(v)$ consists of a continuous interval of time potentials and can therefore be represented by a 2–tuple. Let $\overline{MASAP}(v)$ and $\overline{MALAP}(v)$ be the short hand notations for $\overline{ASAP}(v) \bmod |D|$ and $\overline{ALAP}(v) \bmod |D|$ respectively. Then $\overline{OTI}(v) = [\overline{MASAP}(v), \overline{MALAP}(v) - 1]$ for all $v \in W_{u,b}$. Let $W_{u,b}$ be ordered by increasing $\overline{MALAP}$, and let $W_{u,b}(i)$ be the i^{th} entry in that order. If two RTs have the same $\overline{MALAP}$, then the tie is broken in an arbitrary way. Algorithm 5.1 then constructs a list L_u of RTs from $W_{u,b}$ that do not have any overlap in their $\overline{OTI}$. The list is kept ordered by increasing $\overline{MALAP}$, and $L_u(i)$ is the i^{th} entry in that order. With the use of this list, the lower bound estimate of the minimum number of times resource k must be occupied, $\Psi_u(c)$, can be calculated.

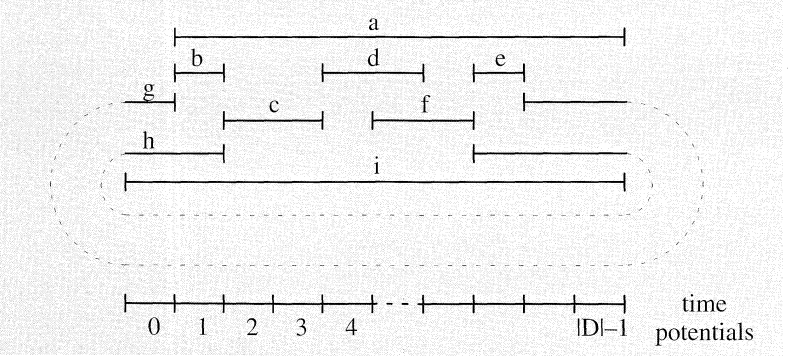


FIGURE 5.8. Example $\overline{OTI}$s.

ALGORITHM 5.1. Construct list L_u of RTs without overlap from $W_{u,b}$.

```
j := 1;
Lu(j) := Wu,b(1);
for (i := 2 to |Wu,b|) →
    if (MASAP(Wu,b(i)) ≥ MALAP(Lu(j))) →
        j := j + 1;
        Lu(j) := Wu,b(i);
```

For the example of Figure 5.8, $L_u = \{b, c, d, e\}$.

$\Psi_u(c)$ can now be calculated with the following formula.
$\Psi_u(c) = \Psi_u(\overline{MALAP}(L_u(n))) = n$, where n has to satisfy:
$\overline{MALAP}(L_u(n)) \leq c < \overline{MALAP}(L_u(n+1))$, $1 \leq n < |L_u|$ and $1 \leq c < |D|$.

For the example of Figure 5.8, we thus obtain the following values.
No RT has to be finished after the first cycle, so $\Psi_u(1) = 0$.
$\overline{MALAP}(L_u(1)) = \overline{MALAP}(b) = 2$, so $\Psi_u(2) = 1$.
$\overline{MALAP}(L_u(2)) = \overline{MALAP}(c) = 4$, so $\Psi_u(3) = \Psi_u(2) = 1$ and $\Psi_u(4) = 2$, etc.

Based on L_u, W_u can be partitioned into the following new disjoint subsets.

$$W_u = \bigcup_{n=1}^{|L_u|+1} W_{u,n}\ ,\ \text{where } W_{u,|Lu|+1} = W_u \setminus \{W_{u,1} \cup \ldots \cup W_{u,|Lu|}\}, \text{ and}$$

$$\forall_{1 \leq n \leq |L_u|} : W_{u,n} = \left\{ v \in \{ W_u \setminus \bigcup_{l=1}^{n-1} W_{u,l} \} \mid \overline{OTI}(v) \cap \overline{OTI}(L_u(n)) \neq \emptyset \right\}.$$

For the example of Figure 5.8, the following subsets are obtained:
$W_{u,1} = \{a, b, h, i\}$, $W_{u,2} = \{c\}$, $W_{u,3} = \{d, f\}$, $W_{u,4} = \{e\}$, $W_{u,5} = \{g\}$.

Depending on whether the set $W_{u,|Lu|+1}$ is empty or not, the estimate of the total minimum number of times resource k must be occupied equals $|L_u|$ or $|L_u|+1$. So if $W_{u,|Lu|+1} = \emptyset$, then $\Psi_u(|D|) = |L_u|$, otherwise it is $|L_u|+1$. For the example of Figure 5.8, the set $W_{u,|Lu|+1} = W_{u,5} \neq \emptyset$, so $\Psi_u(|D|) = |L_u|+1 = 5$, i.e., the resource must be occupied during five time potentials or more.

Consider the case in which the number of occupations in the first c potentials equals $\Psi_u(c) = n$. Then an upperbound for the maximum number of RTs that can be scheduled with these occupations is: $\Xi_u(c) = |\{W_{u,1} \cup W_{u,2} \cup \ldots \cup W_{u,n}\}|$.
For the example of Figure 5.8, if the number of occupations after the first five potentials equals two, then $\Xi_u(5) = |\{W_{u,1}, W_{u,2}\}| = |\{\{a, b, h, i\}, \{c\}\}| = 5$.

Sometimes more RTs can be scheduled within c time potentials than the RTs determined above. For instance, if $c > \overline{MALAP}(L_u(n))$, then extra RTs can possibly be scheduled from the set $W_{u,n+1}$ at the cost of, at least, one extra occupation of resource k. This set of extra RTs is given by:
$W_{u,n+1,c} = \{v \in W_{u,n+1} \mid \overline{MASAP}(v) < c\}$.

Continuing the example of Figure 5.8 after the first five time potentials, we get: $W_{u,3,5} = \{v \in W_{u,3} \mid \overline{MASAP}(v) < 5\} = \{v \in \{d, f\} \mid \overline{MASAP}(v) < 5\} = \{d\}$.

Also RTs from the set $W_{u,a}$ that are not part of the set $\{W_{u,1} \cup W_{u,2} \cup ... \cup W_{u,k}\}$ can possibly be scheduled at the cost of, at least, one extra occupation of resource k. This set of extra RTs is given by:
$W_{u,a,n} = W_{u,a} / \{\{W_{u,1} \cap W_{u,a}\} \cup \{W_{u,2} \cap W_{u,a}\} \cup ... \cup \{W_{u,n} \cap W_{u,a}\}\}$.

Continuing the example of Figure 5.8, we obtain:
$W_{u,a,2} = \{g, h, i\} / \{h, i\} = \{g\}$.

The sets $W_{u,n+1,c}$ and $W_{u,a,n}$ contain all RTs that can additionally be executed if the number of occupations of resource k exceeds the value $\Psi_u(c) = n$. Because $W_{u,n+1,c} \cap W_{u,a,n}$ can be non–empty, the maximum additional number of RTs that can be scheduled with one extra occupation is given by:
$\Xi_{u,1}(c) = \max\{|W_{u,n+1,c}|, |W_{u,a,n}|\}$.

The maximum additional number of RTs from W_u that can be scheduled with a second extra occupation of resource m is given by:
$\Xi_{u,2}(c) = |W_{u,n+1,c}| + |W_{u,a,n}| - |\{W_{u,n+1,c} \cap W_{u,a,n}\}| - \Xi_{u,1}(c)$.

So, the maximum number of RTs from W_u that can possibly be scheduled after c time potentials equals: $\Xi_u(c) + \Xi_{u,1}(c) + \Xi_{u,2}(c)$.

Let Ξ_c be the list of $\Xi_{u,1}(c)$ and $\Xi_{u,2}(c)$ for all $u \in$ usages, ordered by decreasing cardinality, and let $\Xi_c(i)$ be the i^{th} entry in that order. Let mn(c) be the maximum number of RTs that can possibly be scheduled in the first c potentials on resource k. Then it is possible to formulate the following theorems.

THEOREM 5.1.

$$\forall_{0 \le c < |D|} : mn(c) \le rts(c) + \sum_{i=0}^{c - occ(c) - holes(c)} \Xi_c(i).$$

Proof.
The term rts(c) gives an upper bound on the number of RTs that can be scheduled for the minimum number of occupations occ(c). The number of remaining time potentials still available to occupy the resource equals c – occ(c) – holes(c). Thus the second term of the right hand side gives the maximum number of RTs that can be scheduled in the remaining cycles. ■

THEOREM 5.2.
Let $n_u(c)$ be the maximum number of RTs with usage u that can be scheduled at time potential c on resource k. Then one can postulate:

$$\forall_{0 < c < |D|} : mn(c) \leq \left(mn(c-1) + \max_{u \in usages} n_u(c) \right).$$

Proof.
The proof follows from the fact that the maximum difference between mn(c–1) and mn(c) cannot exceed $\max_{u \in usages} n_u(c)$. ■

Estimation of the module execution intervals

With the theorems above, we can now estimate the module execution intervals when they are based on time potentials.

THEOREM 5.3.
Let mn(c) be the minimum of the right hand sides in the Theorems 5.1 and 5.2. Then the following calculation of $\overline{M_1}(i)$, $i \in [1, |W|]$, satisfies the properties of Definition 2.34. The values for $\overline{M_2}(i)$ can be calculated similarly.

$\overline{M_1}(i) = a$, where a has to satisfy: $\sum_{c=0}^{a-1} mn(c) < i \leq \sum_{c=0}^{a} mn(c)$.

Proof.
The value for mn(c), $0 \leq c < |D|$, gives a correct upper bound for the number of RTs that can be scheduled in the first c potentials. With Theorem 5.3, the cardinality of $\{\overline{M_1}(i) \mid i \in [1, |W|] \wedge \overline{M_1}(i) = 1\}$ equals mn(1), the cardinality of $\{\overline{M_1}(i) \mid i \in [1, |W|] \wedge \overline{M_1}(i) = 2\}$ equals mn(2) – mn(1), and so on. The total number of MEIs started after c time potentials equals mn(c), which is correct because no more than mn(c) RTs can have been scheduled after c time potentials. ■

With the calculated $\overline{MEI}$s, the resource conflicts can be cast to $\overline{BSG}$s, similar to the formulation in Section 2.3.3. The only difference is that predecessors of an RT do not necessarily have to be matched with preceding $\overline{MEI}$s in case a DFG is folded. This is different to the approach described in Section 2.3.4. So, initially there is an arc $(v, \overline{MEI}(i))$, $i \in [1, |W|]$, in a $\overline{BSG}$ if and only if register transfer $v \in W$ can be scheduled in $\overline{MEI}(i) \in \overline{R}$.

To schedule a DFG, the approach based on finding a correct ordering of operations can be applied again; recall Chapter 4. The only difference is that the following Theorem 5.4 must be applied as well, in order to guarantee that an ordering is indeed correct, after the execution interval analysis introduced in Chapter 2 is done.

THEOREM 5.4.
Let the arcs (v, $\overline{\text{MEI}}$(i)) and (w, $\overline{\text{MEI}}$(i+1)), $1 \leq i < |W|$, be member of a $\overline{\text{BSG}}$ with a bijection between RTs and $\overline{\text{MEI}}$s. If there is a resource that both v and w use in a different mode, then $M_1(i+1) \geq M_1(i)+1$, and $M_2(i+1) \geq M_2(i)+1$.

Proof.
The RTs v and w cannot be scheduled at the same time potential, and therefore their adjacent MEIs must differ in their first and last cycles. ■

Because an RT can be an element of several $\overline{\text{BSG}}$s, the priority functions in the instruction scheduling process are as follows. First the $\overline{\text{MEI}}$ with the smallest $\overline{M_2}$ is selected. This $\overline{\text{MEI}}$ is matched with an RT, and the selected RT is also matched with the first $\overline{\text{MEI}}$s in the other $\overline{\text{BSG}}$s of which the RT is an element. Then the next $\overline{\text{MEI}}$ with the smallest $\overline{M_2}$, which is not yet matched, is selected, then matched with an RT and so on. If a matching turns out to be incorrect, the matching is revoked and another RT is matched to the $\overline{\text{MEI}}$. So, again a backtracking approach is applied to obtain an exact scheduler.

5.3.6 Construction of BSGs per clique of RTs

$\overline{\text{BSG}}$s try to model the combination of resource and timing conflicts between different RTs as accurately as possible. A drawback of the $\overline{\text{BSG}}$ construction per resource in Section 5.3.5 is, that two RTs can be part of the same $\overline{\text{BSG}}$ while they can not have any conflict with each other. So, such RTs can be scheduled in the same time potential although they use the same resource, recall Section 5.2.2. This leads to a very cumbersome estimation of the MEIs, which is inevitably less accurate than the original approach of Chapter 2 as well. Therefore, in case the $\overline{\text{BSG}}$s are constructed per resource, the $\overline{\text{BSG}}$ formulation is not as powerful as the original formulation of Chapter 2 and 4.

Luckily it is possible to overcome these problems, namely by modelling the resource conflicts differently, i.e., not per separate resource. Consider the following definition of an overall conflict graph (OCG).

DEFINITION 5.9. Overall conflict graph.
The overall conflict graph OCG is an undirected graph represented by a tuple (V, A′), where:

- V is the set of vertices representing *all* the RTs that are part of a DFG;
- $A' \subseteq V \times V$ is the set of arcs; there is an arc $(v_i, v_j) \in A'$ if and only if there is a CG(k), $k \in M$, with $(v_i, v_j) \in CG(k)$ or if there is an arc between the RT classes of v_i and v_j in the ICG.

Every clique of RTs from the OCG represents RTs that have resource conflicts with each other, and for all possible cliques only one RT at the time can be packed into one instruction. Similar to the ICG in Subsection 5.2.3, it is possible to construct a clique cover such that all arcs in the OCG are induced, at least, once by a clique of RTs from the clique cover.

$\overline{\text{BSG}}$s can now be constructed for each clique from such a clique cover, thus modelling all possible resource and timing constraints correctly and more accurately than the approach of Subsection 5.3.5. Note that a clique from the OCG can incorporate resource conflicts from different resources; the $\overline{\text{BSG}}$s are therefore not related to individual resources anymore.

Consider a clique W of RTs from the OCG. Let FTP(v) be the first time potential in which register transfer $v \in W$ can possibly be scheduled, i.e., $\text{FTP}(v) = \min \{c \mid c \in \overline{\text{OTI}}(v)\}$. Let the list of register transfers W be ordered by increasing FTP. If two RTs have the same FTP then the tie is broken in an arbitrary way. Let W(i), $1 \leq i \leq |W|$, be the i^{th} RT in that order. If a $\overline{\text{BSG}}$ is constructed for each clique of RTs from the clique cover mentioned above, then the following two properties hold. The two properties are analogous to the Properties 2.1 and 2.2. Note that Property 5.2 does not hold in case the $\overline{\text{BSG}}$s are constructed per resource.

Property 5.1. Start of MEI(i) cannot be smaller than the i^{th} FTP.

$$\forall_{1 \leq i \leq |W|} : \overline{M_1}(i) \geq \text{FTP}(W(i)).$$

Property 5.2. At each time potential, at most one MEI can start.

$$\forall_{2 \leq i \leq |W|} : \overline{M_1}(i) \geq \overline{M_1}(i-1) + 1.$$

Lemma 5.1.
Analogous to Lemma 2.1 and Algorithm 2.1, Algorithm 5.2 determines $\overline{M_1}(i)$, $1 \leq i \leq |W|$, while satisfying the properties in Definition 2.34. The proof follows directly from Property 5.1 and Property 5.2.

Algorithm 5.2. Calculation of $\overline{M_1}(i)$, $i \in [1, |W|]$.

$\overline{M_1}(1) := \text{FTP}(W(1))$;
for $(i := 2$ to $|W|) \rightarrow$
    $\overline{M_1}(i) := \max \{\text{FTP}(W(i)), \overline{M_1}(i-1) + 1\}$;

Values for the $\overline{M_2}$s can, again, be determined similarly. With this simple determination of $\overline{MEI}$s and corresponding $\overline{BSG}$s, a much more powerful formulation of the scheduling problem is achieved than the formulation in the previous Section 5.3.5. This will be shown by the results in Section 5.4 as well. Aspects, like the determination of the arcs of the $\overline{BSG}$s and the scheduling priority functions, remain as described at the end of the previous Section 5.3.5.

5.4 Experiments and results

In [Strik95], a DSP core together with an instruction set containing 13 RT classes has been given. The examples of Table 5.1 have been mapped onto this core, and it is tried to obtain the highest throughput possible within acceptable run times. The examples range from a simple delay line to a portable audio application, which is a real life industry example. The instruction scheduler based on BSGs has been implemented in C++ using the NEAT system.

In Table 5.2, the lower bound throughput estimates, recall Section 3.3, of the two BSG formulations are given. In one case, Example 4b, the formulation based on BSGs per OCG clique is already more accurate.

In Table 5.2, the dominant factors that lead to the lower bound throughputs are given as well. These factors are obtained by comparing the lower bound results with the sizes of the largest cliques and the largest critical paths in Table 5.1. If a lower bound throughput is totally determined by the number of RTs in the largest critical path, then the precedence relations in the DFG are dominant. This is denoted in the table by a 'P'.

If a lower bound throughput is equal to the size of the largest OCG clique, then the resource conflicts are totally dominant. This is denoted in the table by an 'R'. In three cases, either the precedence relations or the resource conflicts determine almost completely, but not totally, the lower bound throughput: this is denoted by 'P + 1', 'P + 3' and 'R + 4'. The values added denote the extra cycles in comparison to a totally precedence or resource dominated throughput. In two cases, the combination of precedence and resource conflicts leads to the lower bound throughput. This is denoted by a 'C' in the table.

In Table 5.3, the two BSG approaches depicted in this chapter have been compared with an industrial high–level synthesis (HLS) list scheduler. The table shows that the HLS list scheduler only finds the guaranteed optimal throughput, if the precedence relations are dominant, i.e., if there is a 'P' or a 'P+1' in the last column of Table 5.2.

A comparison between the two BSG approaches shows that the outcome and the run times of the approach based on BSGs per OCG clique are better. That approach finds the guaranteed optimal throughput within acceptable run times for all but two cases. These two cases are the ones in which the combination of precedence and resource conflicts leads to the lower bound throughput, i.e., the cases in which there is a 'C' in the last column of Table 5.2.

TABLE 5.1. Characteristics of the various examples.

Example	*#RTs*	*#OCG cliques*	*size largest clique*	*largest critical path*
1a: RAM delay line (unfolded)	12	4	4	4
1b: RAM delay line (folded once)	12	4	4	4
2a: FIR filter (unfolded)	37	11	7	17
2b: FIR filter (folded once)	37	11	7	17
3a: FIR & Bass Boost (unfolded)	114	12	16	27
3b: FIR & Bass Boost (folded once)	114	12	16	27
4a: Sym. FIR & Bass B. (unfolded)	288	22	29	24
4b: Sym. FIR & Bass B. (folded once)	288	22	29	24
5a: Portable audio appl. (unfolded)	358	21	58	14
5b: Portable audio appl. (folded once)	358	21	58	14

TABLE 5.2. Lower bound throughput results of the BSG approaches.

Example	*BSG per resource*	*BSG per OCG clique*	*dominant factor**
1a: RAM delay line (unfolded)	5 cycles	5 cycles	P + 1
1b: RAM delay line (folded once)	4 cycles	4 cycles	R
2a: FIR filter (unfolded)	17 cycles	17 cycles	P
2b: FIR filter (folded once)	9 cycles	9 cycles	P
3a: FIR & Bass Boost (unfolded)	30 cycles	30 cycles	P + 3
3b: FIR & Bass Boost (folded once)	25 cycles	25 cycles	C
4a: Sym. FIR & Bass B. (unfolded)	36 cycles	36 cycles	C
4b: Sym. FIR & Bass B. (folded once)	28 cycles	29 cycles	R
5a: Portable audio appl. (unfolded)	62 cycles	62 cycles	R + 4
5b: Portable audio appl. (folded once)	58 cycles	58 cycles	R

* P = precedence relations are dominant, R = resource conflicts are dominant,
C = combination of precedence and resource conflicts leads to lower bound throughput.

TABLE 5.3. Instruction scheduling results.

Example	*HLS list scheduler*	*BSG per resource*	*CPU*** *(sec)*	*BSG per OCG clique*	*CPU*** *(sec)*
1a	5* cycles	5* cycles	0.3	5* cycles	0.3
1b	5 cycles	4* cycles	0.3	4* cycles	0.3
2a	17* cycles	17* cycles	1.8	17* cycles	1.2
2b	9* cycles	9* cycles	1.9	9* cycles	1.2
3a	31 cycles	30* cycles	15.9	30* cycles	7.9
3b	26 cycles	26 cycles	17.2	26 cycles	8.0
4a	43 cycles	43 cycles	259.6	38 cycles	56.2
4b	36 cycles	36 cycles	328.1	29* cycles	56.3
5a	67 cycles	62* cycles	381.4	62* cycles	138.3
5b	61 cycles	58* cycles	373.0	58* cycles	139.1

* the throughput equals the lower bound estimate, i.e., is guaranteed to be optimal.

** measured on a HP 9000/735 workstation.

The most interesting example is the largest Example 5a, consisting of 58 multiplications, 58 additions, clip actions and delays. The throughput constraint of this real life application is 64 cycles. The results on Example 5a show, that a dedicated instruction–set scheduler exploiting the combination of resource and timing constraints is needed to meet this throughput constraint without folding. Furthermore, the result on Example 5b is also interesting. It shows that the BSG approaches succeed in generating a schedule in which the multiplier, ALU and RAM in the DSP core have a 100% utilization during all clock cycles.

Although the number of examples is too small to make a well–founded comment on the results, they could be explained by considering the dominant factors of Table 5.2. The list scheduler is only capable of generating good results if the precedence relations are dominant. In fact every scheduler should be capable of handling these relations correctly, so schedulers of any type should be able to obtain results that are at least as good.

The results of the scheduler based on BSGs per OCG clique can be explained in two ways. First of all, if the combination of precedence and resource conflicts leads to the lower bound throughput, then the approach could have trouble finding the optimal solution because the precedence relations are modelled in one graph, the DFG, while the resource conflicts are modelled in other graphs, the BSGs.

A second explanation is as follows. On one hand, if the precedence relations are dominant, many OTIs consist of one time potential. On the other hand, if the resource conflicts are dominant, many MEIs consist of one time potential. However, if the combination of precedence and resource conflicts leads to the lower bound throughput, then neither many OTIs nor many MEIs will consist of one time potential. This will lead to relatively many arcs in the BSGs, thus resulting in a possibly less effective modelling of the search space.

Chapter 6 Conclusions and Discussion

Conclusions

In this thesis, we presented an architectural synthesis approach that emphasizes *constraint satisfaction* techniques and *lower bound design analyses*. Although all architectural synthesis systems try to optimize a design for a given set of constraints and goals, our solution strategy is different than most of the existing approaches. Instead of focusing on the optimization of objective functions, constraint satisfaction is emphasized in our approach. This difference in focus leads to a number of advantages.

It was shown that a very accurate lower bound functional area vs. time (AT) curve can be calculated for a design. It was shown that, even for unrestricted module libraries, such a calculation can be performed efficiently as well. AT curves are of considerable importance in an (interactive) system design environment. They can be used to rapidly explore the design space and to evaluate the quality of an implementation. Furthermore, an accurate AT curve is a good starting point for a concise trade–off between functional, memory and interconnect costs. So the ability to calculate accurate lower bound AT curves efficiently is very important in an architectural, and system level, design environment.

A designer or a synthesis system must be able to enforce certain sets of constraints in order to have control over the design process. As an example, a design specification could start with a throughput constraint, and in a subsequent step a module set could be added as a resource constraint. In such cases constraint satisfaction techniques are important, because it is generally difficult to comply to a set of different constraints. An important part of constraint satisfaction is domain reduction, i.e., the pruning of the search space of a synthesis tool. In this thesis, a new, efficient, domain reduction algorithm algorithm is introduced, namely the execution interval analysis.

Another aspect of constraint satisfaction is variable and value selection. A new scheduling technique is presented, which is based on analyzing the topology of bipartite schedule graphs (BSGs). Based on this graph analysis, an explicit variable selection is performed. This is, together with an efficient formulation of the scheduling problem, the reason why the BSG scheduling approach achieves an optimal solution in many cases and in short run times.

A touchstone of the solution strategy introduced in this thesis is when a design is fully constrained and 'only' a feasible schedule and mapping scheme has to be found. This is in fact the last step of the proposed synthesis flow, which is equivalent to the code generation problem. Benchmark results on real life examples show that it is possible to achieve optimal code generation results run time efficiently. These results thus show the value of an architectural synthesis approach based on constraint satisfaction techniques and lower bound design analyses.

Discussion

Although we presented a promising first approach towards an architectural synthesis system based on design analyses and constraint satisfaction techniques, still a lot of research needs to be done to make such an approach mature. In the following, a number of topics that have not been addressed in the thesis are listed.

This thesis has not dealt with memories, e.g., register files. For instance, the code generation approach of Chapter 5 does not consider register files with limited sizes. Furthermore, it does also not account for register binding. However, memories can have a huge impact on the total cost of a design, especially in high throughput applications as occur in the video domain. Some work has been done on lower bound analyses for memory area [Shar94], and at IMEC work has been done on constraint satisfaction techniques for limited register file sizes [Depu94]. The proposed methods could possibly fit in the synthesis approach of this thesis.

Another topic related to constraint satisfaction is domain reduction which may result in a reduction of the solution space. The execution interval analysis presented in Chapter 2 reduces the search space, but does *not* reduce the solution space. However, consider the case in which only a feasible solution has to be found, while no objective function has to be optimized. Then it is perfectly legitimate to reduce the solution space, as long as it does not become empty. If such a step results in a faster, more efficient, traversal of the search space, a solution can possibly be found more efficiently as well. The following two paragraphs give examples of such domain reductions.

A first 'feasibility preserving' domain reduction can possibly be achieved by exploiting symmetries in a DFG [Gruij92]. If one can detect that some partial ordering of operations does not influence the decision problem described above, then an order can be enforced on these operations without influencing the corresponding decision problem. An example is the case in which two operations must be mapped on the same module, have the same delay, and have the same predecessors and successors as well.

Another 'feasibility preserving' domain reduction is the application of a property of the node packing problem; see Definition 4.4. It was shown in Section 4.3.3. that time and resource constrained scheduling can be formulated as a node packing problem. A unique property of the node packing problem is the following: if not all variables are integral in a solution of the LP relaxation, then there is an optimum solution for which the variables that are integral in the LP relaxation remain integral [Nemh88]. This property can be used for a further reduction of the search space which may result in a reduction of the solution space. However, this is still the topic of future research.

References

[Arts91] H.M.A.M. Arts, J.T.J. van Eijndhoven and L. Stok, "Flexible Block–Multiplier Generation", Proc. ICCAD–1991, pp. 106–109, 1991.

[Bala89] M. Balakrishnan and P. Marwedel, "Integrated Scheduling and Binding: A Synthesis Approach for Design Space Exploration," Proc. 26th DAC, pp. 68–74, 1989.

[Berk94] M.R.C.M. Berkelaar, "lp_solve", a public domain MILP solver available by anonymous ftp from ftp.es.ele.tue.nl.

[Berry90] N. Berry and B.M. Pangrle, "Schalloc: An Algorithm for Simultaneous Scheduling and Connectivity Binding in a Datapath Synthesis System", Proc. EDAC '90, pp. 78–82, 1990.

[Chau93] S. Chaudhuri, R.A. Walker and J. Mitchell, "The Structure of Assignment, Precedence, and Resource Constraints in the ILP Approach to the Scheduling Problem", Proc. ICCD'93, pp. 25–29, 1993.

[Chau94] S. Chaudhuri and R.A. Walker. "Computing Lower Bounds on Functional Units before Scheduling", Proc. Int. Symp. on HLS, pp. 36–41, Niagara–on–the–Lake (Canada), May 1994.

[Chen91] L.–G. Chen and L.–G. Jeng, "Optimal Module Set and Clock Cycle Selection for DSP Synthesis", Proc. ISCAS–91, pp. 2200–2203, 1991.

[Cheng94] W.–K. Cheng and Y.–L. Lin, "Code Generation for a DSP Processor", Proc. Int. Symp. on HLS, pp. 82–87, Niagara–on–the–Lake (Canada), May 1994.

[Chou94] P. Chou and G. Borriello, "Software Scheduling in the Co–Synthesis of Reactive Real–Time Systems", Proc. of the 31st DAC, pp. 1–4, San Diego (CA), June 1994.

[Dalk94] M.E. Dalkiliç and V. Pitchumani, "Optimal Operation Scheduling using Resource Lower Bound Estimations", Proc. ED&TC (EDAC–ETC–EuroASIC) '94, pp. 319–324, Paris (France), March 1994.

[Deny90] P. Denyer, "SAGE – A User Directed Synthesis Tool", Proc. of the ASCIS Open Workshop on Synthesis Techniques for (lowly) Multiplexed Datapaths, Leuven, Belgium, August 1990.

[Depu94] F. Depuydt, G. Goossens and H. De Man, "Scheduling with Register Constraints for DSP Architectures", to be published in "Integration – the VLSI Journal", Elsevier Science B.V., Amsterdam (Netherlands), 1994.

[DeWi85] P. DeWilde, E. Deprettere and R. Nouta, "Parallel and Pipelined VLSI Implementations of Signal Processing Algorithms", in S.Y. Kung, H.J. Whitehouse and T. Kailath, VLSI and Modern Signal Processing, Prentice Hall, pp. 258–264, 1985.

[Dulm63] A.L. Dulmage and N.S. Mendelsohn, "Two Algorithms for Bipartite Graphs", J. Soc. Indust. Appl. Math., Vol. 11, No. 1, pp. 183–194, 1963.

[Eijnd92] J.T.J. van Eijndhoven and L. Stok, "A Data Flow Graph Exchange Standard", Proc. EDAC '92, pp. 193–199, 1992.

[Faber94] H. Faber, "Branch–and–Bound Scheduling using Execution Interval Analysis", Master Thesis, Eindhoven University of Technology, 1994.

[Fleu93] H. Fleurkens, "Interactive System Design in ESCAPE", Proc. IEEE Int. Workshop on Rapid System Prototyping, pp. 108–113, 1993.

[Fleu96] H. Fleurkens, "Interactive Modelling and Simulation of heterogeneous Systems", Ph.D. Thesis, Eindhoven University of Technology, March 1996.

[Fran86] J. Frankle and R.M. Karp, "Circuit Placement and Cost Bounds by Eigenvector Decomposition, " in Proc. ICCAD–86, pp. 414–417, 1986.

[Garey79] M.R. Garey and D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP–Completeness", W.H. Freeman and Company, San Francisco, 1979.

[Gebo92] C.H. Gebotys, M.I. Elmasry, "Optimal VLSI Architectural Synthesis", Kluwer, 1992.

[Girc84] E.F. Girczyc and J.P. Knight, "An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling," Proc. ICCD, pp. 726–731, 1984.

[Gruij92] P.W.F. Gruijters, "Resource Constrained List Scheduling Using Unrestricted Libraries", Master Thesis, Eindhoven University of Technology, The Netherlands, 1992.

[Gutb91] P. Gutberlet, H. Krämer and W. Rosenstiel, "CASCH – A Scheduling Algorithm for "High Level"– Synthesis", Proc. EDAC–91, pp. 311–315, 1991.

[Gutb92] P. Gutberlet, J. Müller, H. Krämer and W. Rosenstiel, "Automatic Module Allocation in High Level Synthesis", Proc. EuroDAC '92, pp. 328–333, 1992.

[Heem90] S.M. Heemstra de Groot, "Scheduling Techniques for Iterative Data–Flow Graphs; An approach based on the range chart", Ph.D. Thesis, University Twente, 1990.

[Heij94] M.J.M. Heijligers, H.A. Hilderink, A.H. Timmer and J.A.G. Jess, "NEAT: An Object Oriented High–Level Synthesis Interface", Proc. ISCAS–94, pp. 1.233–1.236, London (UK), May 1994.

[Hild94] H.A. Hilderink, "NESCIO: An Interactive High Level Synthesis Framework", Proc. of the IEEE Benelux / ProRISC Workshop on Circuits, Systems and Signal Processing, pp. 119–123, Papendal, the Netherlands, March 1994.

[Holm94] N.D. Holmes and D.D. Gajski, "An Algorithm for Generation of Behavioral Shape Functions", Proc. ED&TC (EDAC–ETC–EuroASIC), pp. 314–318, Paris (F), 1994.

[Hopc73] J.E. Hopcroft and R.M. Karp, "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs", SIAM J. Comput., Vol. 2, No. 4, 1973.

[Hwang91] C.–T. Hwang, J.–H. Lee and Y.–C. Hsu, "A Formal Approach to the Scheduling Problem in High level Synthesis", IEEE Trans. on CAD, vol. 10, no. 4, pp. 464–475, 1991.

[Ishi91] M. Ishikawa and G. DeMicheli, "A Module Selection Algorithm for High–Level Synthesis", Proc. ISCAS–91, pp. 1777–1780, 1991.

[Jain92] R. Jain, A.C. Parker and N. Park, "Predicting System–Level Area and Delay for Pipelined and Nonpipelined Designs", IEEE Trans. on CAD, vol. 11, no. 8, pp. 955–965, august 1992.

[Jong93] G.G. de Jong, "Generalized data flow graphs, theory and applications", Ph.D. Thesis, Eindhoven University of technology, 1993.

[Küçü90] K. Küçükçakar and A.C. Parker, "BAD: Behavioral Area–Delay Predictor", Tech. Report 90–31, University of Southern California, November 1990.

[Kurd91] F.J. Kurdahi and C. Ramachandran, "LAST, A Layout Area and Shape function esTimator for High Level Applications," in Proc. EDAC–91, pp. 351–355, 1991.

[Lann94] D. Lanneer, M. Cornero, G. Goossens and H. De Man, "Data Routing: a Paradigm for Efficient Data–Path Synthesis and Code Generation", Proc. Int. Symp. on HLS, pp. 17–22, Niagara–on–the–Lake (Canada), May 1994.

[Leeu95] J.C. van Leeuwen, "Applying Integer Programming in High–Level Synthesis Scheduling", Master Thesis, Eindhoven University of Technology, 1995.

[Liem94] C. Liem, T. May and P. Paulin, "Instruction–Set Matching and Selection for DSP and ASIP Code Generation", Proceedings ED&TC (EDAC–ETC–EuroASIC) '94, pp. 31–37, Paris (France), March 1994.

[Mall90] D.J. Mallon and P.B. Denyer, "A New Approach to Pipeline Optimization", Proc. EDAC '90, pp. 83–88, 1990.

[Marw93] P. Marwedel, "Tree–Based Mapping of Algorithms to Predefined Structures", Digest of Technical Papers of ICCAD–93, pp. 586–593, Santa Clara (CA), Nov. 1993.

[McFa90] M.C. McFarland, A.C. Parker and R. Camposano, "The High–Level Synthesis of Digital Systems", Proc. of the IEEE, vol. 78, no. 2, pp. 301–318, 1990.

[Naka82] K. Nakajima, S.L. Hakimi and J.K. Lenstra, "Complexity results for scheduling tasks in fixed intervals on two types of machines", SIAM J. Comput., no. 11, pp. 512–520, 1982.

[Nemh88] G.L. Nemhauser and L.A. Wolsey, "Integer and Combinatorial Optimization", Wiley, 1988.

[Nemh92] G.L. Nemhauser and G. Sigismondi, "A Strong Cutting Plane / Branch–and–Bound Algorithm for Node Packing", J. Opl. Res. Soc, Vol. 43, No. 5, pp. 443–457, 1992.

[Nieu94] K. van Nieuwenhoven, J. de Moortel, D. Genin and S. Note, "Mistral2 a True Architectural Synthesis™ Tool: from a Behavioral Specification down to a Register Transfer Level Description", DSP Applications and Multimedia, Oct. 1994.

[Note89] S. Note, F. Catthoor, J. van Meerbergen and H. de Man, "Definition and Assignment of Complex Data–Paths suited for High Throughput Applications," in Proc. ICCAD–89, pp. 108–111, 1989.

[Nuijt94] W.P.M. Nuijten, "Time and Resource constrained Scheduling", Ph.D. Thesis, Eindhoven University of Technology, 1994.

[Pang88] B.M. Pangrle, "Splicer: A Heuristic Approach to Connectivity Binding," Proc. 25th DAC, pp. 536–541, 1988.

[Paul87] P.G. Paulin and J.P. Knight "Force Directed Scheduling in Automatic Data Path Synthesis", Proc. 24th DAC, pp. 195–201, 1987.

[Paul92] P.G. Paulin, "DSP Design Tool Requirements for the Nineties: An Industrial Perspective", 6th Int. HLS Workshop, Laguna Niguel (CA), Nov. 1992.

[Paul94] P.G. Paulin, C. Liem, T.C. May and S. Sutarwala, "CodeSyn: A Retargetable Code Synthesis System", Int. Symp. on HLS, Niagara–on–the–Lake (Canada), May 1994.

[Potk94] M. Potkonjak and J. Rabaey, "Algorithm Selection: A Quantitative Computation–Intensive Optimization Approach", Digest of technical papers of ICCAD–94, pp. 90–95, Nov. 1994.

[Praet94] J. Van Praet, G. Goossens, D. Lanneer and H. De Man, "Instruction Set Definition and Instruction Selection for ASIPs", Proc. Int. Symp. on HLS, pp. 11–16, Niagara–on–the–Lake (Canada), May 1994.

[Radi95] I. Radivojevic and F. Brewer, "On Applicability of Symbolic Techniques to Larger Scheduling Problems", Proc. of ED&TC (EDAC–ETC–EuroASIC), pp. 48–53, 1995.

[Rao92] D.S. Rao and F.J. Kurdahi, "Partitioning by Regularity Extraction", Proc. 29th DAC pp. 235–238, 1992.

[Rao93] D.S. Rao and F.J. Kurdahi, "An Approach to Scheduling and Allocation Using Regularity Extraction," in Proc. EDAC/EuroASIC '93, pp. 557–561, 1993.

[Raba90] J. Rabaey and M. Potkonjak, "Resource Driven Synthesis in the HYPER System", Proc. ISCAS–90, pp. 2592–2595, 1990.

[Rama91] L. Ramachandran and D.D. Gajski, "An Algorithm for Component Selection in Performance Optimized Scheduling", Proc. ICCAD–91, pp. 92–95, 1991.

[Rim92] M. Rim and R. Jain, "Estimating Lower–Bound Performance of Schedules Using a Relaxation Technique", Proc. ICCD '92, pp. 290–294, 1992.

[Sang76] A. Sangiovanni–Vincentelli, "A Note on Bipartite Graphs and Pivot Selection in Sparse Matrices", IEEE Trans. Circuits & Systems, vol. CAS.23, no. 12, pp. 817–821, 1976.

[Shar93] A. Sharma and R. Jain, "Estimating Architectural Resources and Performance for High–Level Synthesis Applications", Proc. of the 30th DAC, pp. 355–360, 1993.

[Shar94] A. Sharma and R. Jain, "Register Estimation from Behavioral Specifications", Proc. ICCD '94, pp. 576–580, 1994.

[Stok91] L. Stok, "Architectural Synthesis and Optimization of Digital Systems", Ph.D. Thesis, Eindhoven University of Technology, 1991.

[Strik94] M. Strik, "Efficient Code Generation for Application Domain Specific Processors", Eindhoven University of Technology IVO–report, ISBN 90–5282–390–1, 1994.

[Strik95] M. Strik, J. Van Meerbergen, A. Timmer, J. Jess and S. Note, "Efficient Code Generation for In–House DSP–Cores", Proc. ED&TC (EDAC–ETC–EuroASIC), pp. 244–249, 1995.

[Timm93a] A.H. Timmer, M.J.M. Heijligers, L. Stok and J.A.G. Jess, "Module Selection and Scheduling using Unrestricted Libraries", Proc. EDAC/EuroASIC '93, pp. 547–551, 1993.

[Timm93b] A.H. Timmer and J.A.G. Jess, "Execution Interval Analysis under Resource Constraints", Digest of technical papers of ICCAD–93, pp. 454–459, 1993.

[Timm93c] A.H. Timmer, M.J.M. Heijligers and J.A.G. Jess, "Fast System–Level Area–Delay Curve Prediction", Proc. APCHDLSA '93, pp. 198–207, 1993.

[Timm94] A.H. Timmer, "Improved Execution Interval and Binding Analysis", Proc. IEEE Benelux & ProRISC Workshop on CSSP, pp. 247–251, Papendal, Netherlands, 1994.

[Timm95a] A.H. Timmer and Jochen A.G. Jess, "Exact Scheduling Strategies based on Bipartite Graph Matching", Proc. ED&TC–95 (EDAC–ETC–EuroASIC), pp. 42–47, 1995.

[Timm95b] A.H. Timmer, M.T.J. Strik, J.L. van Meerbergen and J.A.G. Jess, "Conflict Modelling and Instruction Scheduling in Code Generation for In–House DSP Cores", Proc. of the 32nd DAC, pp. 593–598, 1995.

[Verh91] W.F.J. Verhaegh, E.H.L. Aarts, J.H.M. Korst and P.E.R. Lippens, "Improved Force–Directed Scheduling", Proc. EDAC–91, pp. 430–435, 1991.

[Verh95] W.F.J. Verhaegh, "Multidimensional Periodic Scheduling", Ph.D. Thesis, Eindhoven University of Technology, December 1995.

Notation

General notations

$X = \{x_1, x_2, x_3, ...\}$	set of elements $x_1, x_2, x_3, ...$
$\lvert X \rvert$	cardinality of X
$\#X$	cardinality of X
$X \cup Y$	set union
$X \cap Y$	set intersection
$X \setminus Y$	set difference
$X \subseteq Y$	X is a proper subset of Y
$X \supseteq Y$	Y is a proper subset of X
$x \in X$	x is an element of X
$\varnothing$	empty set
$P(X)$	power set of X
$\mathbf{Q}$	set of non–negative rational numbers
$\mathbf{Q}^+$	set of positive rational numbers
$\mathbb{N}$	set of non–negative natural numbers
$\mathbb{N}^+$	set of positive natural numbers
$\lfloor x \rfloor$	floor of x
$\lceil x \rceil$	ceiling of x
[x, y]	2–tuple denoting the interval between x and y
x mod y	modulo y of x

Objects

		page:
$DFG = (V, E)$	data flow graph	18
V	set of DFG vertices (operations)	18
E	set of DFG arcs	18
$DFG^* = (V, E^*)$	transitive closure of DFG	19
E^*	set of DFG^* arcs	19
T_O	set of operation types	18
T_M	set of module types	19
M	set of modules	19
C	list of cycle steps	19
D	list of time potentials	99
Φ	set of feasible schedules	21
$\phi \in \Phi$	feasible schedule	21
$K, K_m \subseteq M$	set of modules with type m	26
$BSG = (N, A)$	bipartite schedule graph	32
$N = W \cup R$	set of BSG vertices	32
$W, W_m \subseteq V$	set of operations implemented by type m	26
R	set of indices related to MEIs	32
A	set of BSG arcs	32
$\overline{BSG} = (N, \overline{A})$	conservative estimate of BSG	34
$\overline{A}$	conservative estimate of A	34
C_{low}	lower bound estimate of $\lvert C \rvert$	51
C_{coei}	same as C_{low}, all $v \in V$ in $COEI(v)$	51
C_{oei}	same as C_{low}, all $v \in V$ in $\overline{OEI}(v)$	52
$\lvert C_{up} \rvert$	upper bound estimate of $\lvert C \rvert$	52

$V_{m,c}$	operations to be mapped to type m in cycle c	86
CG	conflict graph	95
ICG	instruction set conflict graph	97
OCG	overall conflict graph	108

Functions on operations and their types ($v \in V$ and $ts \subseteq T_O$)

$\tau(v)$	operation type of v	18
pred(v)	immediate predecessors of v	18
$pred^*(v)$	predecessors of v	19
$preds^*(v)$	predecessors of v, with v in same BSG	37
succ(v)	immediate successors of v	18
$succ^*(v)$	successors of v	19
$succs^*(v)$	successors of v, with v in same BSG	37
$d_{min}(v)$	minimal delay of v	20
$\phi(v) = [\phi_1(v), \phi_2(v)]$	schedule interval of v	20
$\phi_1(v)$	start time of v	20
$\phi_2(v)$	completion time of v	20
$\phi_D(v)$	time potential of v	102
CASAP(v)	classical 'as soon as possible' time of v	21
ASAP(v)	'as soon as possible' time of v	22
$\overline{ASAP}(v)$	conservative estimate of ASAP(v)	23
$\overline{MASAP}(v)$	conservative estimate of ASAP(v) mod |D|	104
CALAP(v)	classical 'as late as possible' time of v	21
ALAP(v)	'as late as possible' time of v	22

$\overline{ALAP}(v)$	conservative estimate of ALAP(v)	23
$\overline{MALAP}(v)$	conservative estimate of $\overline{ALAP}(v)$ mod $\lvert D \rvert$	104
COEI(v)	classical operation execution interval	21
OEI(v)	operation execution interval	22
$\overline{OEI}(v)$	conservative estimate of OEI(v)	23
OTI(v)	operation time potential interval	101
$\overline{OTI}(v)$	conservative estimate of OTI(v)	101
$\mu(ts)$	set of module types implementing ts	19
$\varepsilon(ts)$	set of disjoint distribution intervals	56
$\prec_\phi$	linear ordering induced by schedule ϕ	26
$\pi_\phi(i)$	'i^{th} operation' in schedule ϕ	26
$\phi_1(i)$	start time of i^{th} operation in schedule ϕ	26
$\phi_2(i)$	completion time of i^{th} operation in schedule ϕ	26
$\phi_D(i)$	time potential of i^{th} operation in schedule ϕ	102
F(v)	freedom, i.e., slack of v	46
AF(V)	average freedom of the set V	46
DF(ts,c)	probability distribution function for cycle c	78
x(v,c)	binary variable expressing $\lfloor \phi_1(v) \rfloor = c$	81
x(v,c,k)	same as x(v,c), v executed by module k	84

Functions on modules and their types ($k \in M$ and $m \in T_M$)

$\xi(k)$	module type of k	19
d(m)	delay of m	20
dii(m)	data introduction interval of m	20
area(m)	area of m	55

$n(m)$	number of (selected) modules of type m	55
$MEI(i) = [M_1(i), M_2(i)]$	i^{th} module execution interval	26
$M_1(i)$	first clock cycle of MEI(i)	27
$M_2(i)$	last clock cycle of MEI(i)	27
$\overline{MEI}(i)$	conservative estimate of MEI(i)	27
$\overline{M_1}(i)$	conservative estimate of $M_1(i)$	27
$\overline{M_2}(i)$	conservative estimate of $M_2(i)$	27
cap(m,e)	capacity of a module type in a had–interval	58
m(m,ts,t)	number of preliminary mappings	57

Biography

Adwin Timmer was born on June 21st, 1966 in Apeldoorn, the Netherlands. After attending the "Stedelijk Gymnasium" in Breda, he started his study Electrical Engineering at the Eindhoven University of Technology. In May 1990, he graduated on the design of telecommunication ICs.

In January 1991, Adwin Timmer started working towards a doctorate under the supervision of Prof. Jochen Jess in the Design Automation Section of the Department of Electrical Engineering of the Eindhoven University of Technology. He expects to receive his Ph.D. based on the work presented in this thesis on April 3rd, 1996. He has published several papers on various aspects of architectural synthesis and code generation for digital VLSI circuits.

Since February 1995, Adwin Timmer has been working as a research scientist in the IC Design Centre of Philips Nat.Lab. in Eindhoven, the Netherlands.

Stellingen

behorende bij het proefschrift

From Design Space Exploration to Code Generation

van Adwin H. Timmer

1. Een architectuur-synthese-systeem met een goede oplosstrategie kan ook herprogrammeerbaarheid aan [dit proefschrift].

2. Met een nauwkeuriger onderverdeling in klassen kan een niet-triviale klasse van module bibliotheken gedefinieerd worden, waarvoor het functionele oppervlakte-probleem van definitie 3.5 toch in polynomiale tijd oplosbaar is [dit proefschrift].

3. De kans dat een digitale signaal-processor goed te programmeren is met een hoog-niveau taal neemt sterk toe, indien zowel de architectuur als de compiler ervan gelijktijdig en in samenhang ontworpen zijn.

4. De verklaring dat architectuur-synthese een uitgegroeid vakgebied is en dat men andere onderwerpen zoekt, heeft meer te maken met het gebrek aan vooruitgang op dat gebied dan met het feit dat er geen belangrijke vooruitgang meer geboekt zou kunnen worden.

5. Wetenschappelijke kringen waarin men een goede positie kan verwerven met een artikel waarvan de samenvatting eindigt met: "This research breaks new ground by (...) providing a polynomial run time algorithm for solving this NP-complete problem" zijn niet erg hoog te achten [C.H. Gebotys and M.I. Elmasry, "*A Global Optimization Approach for Architectural Synthesis*", Proc. of the ICCAD, 1990].

6. Omdat ontwikkelaars van architectuur-synthese-methoden vertrouwd zijn met software, hebben zij een betere uitgangspositie voor onderzoek naar hardware-software codesign dan ontwikkelaars van software-methoden.

7. Het minutieus bijhouden van vakliteratuur is contra-produktief.

8. Het schrijven van een proefschrift duurt minder lang dan de

evaluatie-cyclus van menig tijdschriftartikel; dientengevolge kan men van een promovendus niet verwachten dat er referenties naar dergelijke artikelen van eigen hand in zijn proefschrift staan.

9. Het bestaansrecht van een universiteit is omgekeerd evenredig met de frequentie waarmee zij haar missiedocument verandert en de mate waarin zij zich bezig houdt met businessplannen in plaats van onderzoek.

10. Gezien de maatschappelijke status van de wetenschap is het dragen van een toga door hoogleraren een anachronisme.

11. Vooral wiskundigen en informatici wijzen op syntactische onjuistheden in een proefschrift; net als compilers testen zij bij het parsen blijkbaar eerst of de grammatica wel context-vrij is.

12. Dat in Eindhoven nooit iemand met lof promoveert en in Delft wel is een rechtsongelijkheid onder promovendi, welke te wijten is aan de vrees van de Technische Universiteit Eindhoven om de nek uit te steken.

13. Gezien de vroegere connotatie van 'traveling salesman' in de Verenigde Staten, is het handelsreizigersprobleem in het huidige AIDS-tijdperk pas echt een lastig probleem geworden [H.L. Mencken, "*The American Language: An Inquiry into the Development of English in the United States*", 4th edn., pp. 360-I, New York: Alfred A. Knopf, 1937].

14. De aard van de boeren-protesten tegen het mestbeleid laat zien, hoe snel de betekenis van woorden als 'boerenslimheid' kunnen veranderen.

15. Uitzonderingen daargelaten hebben niet-bergbeklimmers geen enkele *nut*.