

Storage constraint satisfaction for embedded processor compilers

Citation for published version (APA):

Alba Pinto, C. A. (2002). *Storage constraint satisfaction for embedded processor compilers*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven.
<https://doi.org/10.6100/IR556103>

DOI:

[10.6100/IR556103](https://doi.org/10.6100/IR556103)

Document status and date:

Published: 01/01/2002

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Storage Constraint Satisfaction for Embedded Processor Compilers

Carlos A. Alba Pinto

Storage Constraint Satisfaction for Embedded Processor Compilers

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 20 juni 2002 om 16.00 uur

door

Carlos Antonio Alba Pinto

geboren te Lima, Peru

Dit proefschrift is goedgekeurd door de promotoren:

prof.Dr.-Ing. J.A.G. Jess
en
prof.dr. J.L. van Meerbergen

Copromotor:
dr. B. Mesman

Druk: Universiteitsdrukkerij, Technische Universiteit Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Alba Pinto, Carlos A.

Storage constraint satisfaction for embedded processor compilers / by
Carlos A. Alba Pinto. - Eindhoven : Technische Universiteit Eindhoven,
2002.

Proefschrift. - ISBN 90-386-1910-3

NUR 958

Trefw.: ingebette systemen / scheduling / digitale systemen ; CAD /
digitale signaalverwerking ; software / compilers.

Subject headings: embedded systems / processor scheduling / storage
allocation / digital signal processing chips / resource allocation.

To my lovely Renata.

Acknowledgements

I would like to thank Prof. Jochen Jess for gave me the chance to do my Ph.D. at the Eindhoven University of Technology, and for his support during the four years of work.

I am grateful to Bart Mesman for his coaching, the scientific discussions, his cooperation that resulted in a number of publications, and his support with the review of this thesis. I am also grateful to Koen van Eijk for his advises and for the creation of the valuable framework FACTS.

Furthermore, special thanks to professors Jef van Meerbergen, Ralph Otten, and Henk Corporaal for their guidance to improve this thesis.

My thanks to my colleagues from the *Information and Communication Systems Group* (ICS) at Eindhoven University of Technology. In particular, to Qin Zhao and Srinath Naidu for their friendship, and the nice conversations during the coffee breaks. Also, to the ICS staff, especially Rian van Gaalen for her care and support, Lia de Jong, Oege Koopmans, Marja de Mol-Regels, and Herman Rohenkohl.

Last but not least I am grateful to my family that, even being far away, is always supporting me in all aspects of my life, and to Renata Trunci for her love, care, encouraging and support during the writing of this thesis.

Summary

Increasing interest in the high-volume high-performance embedded processor market motivates the stand-alone processor world to consider issues like design flexibility (synthesizable processor core), energy consumption, and silicon efficiency. Implications for embedded processor architectures and compilers are the exploitation of hardware acceleration, instruction-level parallelism (ILP), and distributed storage files.

In that scope, VLIW architectures have been acclaimed for their parallelism in the architecture while orthogonality of the associated instruction sets is maintained. Code generation methods for such processors will be pressured towards an efficient use of scarce resources while satisfying tight real-time constraints imposed by DSP and multimedia applications.

Limited storage (e.g. registers) availability poses a problem for traditional methods that perform code generation in separate stages, e.g. operation scheduling followed by register allocation. This is because the objectives of scheduling and register allocation cause conflicts in code generation in several ways. Firstly, register reuse can create dependencies that did not exist in the original code, but can also save spilling values to memory. Secondly, while a particular ordering of instructions may increase the potential for ILP, the reordering due to instruction scheduling may also extend the lifetime of certain values, which can increase the register requirement. Furthermore, the instruction scheduler requires an adequate number of local registers to avoid register reuse (since reuse limits the opportunity for ILP), while the register allocator would prefer sufficient global registers in order to avoid spills. Finally, an effective scheduler can lose its achieved degree of instruction-level parallelism when spill code is inserted afterwards.

Without any communication of information and cooperation between scheduling and storage allocation phases, the compiler writer faces the problem of determining which of these phases should run first to generate the most efficient final code. The lack of communication and cooperation between the instruction scheduling and storage allocation can result in code that contains excess of register spills and/or lower degree of ILP than actually achievable. This problem called *phase coupling* cannot be ignored when constraints are tight and efficient solutions

are desired. Traditional methods that perform code generation in separate stages are often not able to find an efficient or even a feasible solution. Therefore, those methods need an increasing amount of help from the programmer (or designer) to arrive at a feasible solution. Because this requires an excessive amount of design time and extensive knowledge of the processor architecture, there is a need for automated techniques that can cope with the different kinds of constraints *during* scheduling.

This thesis proposes an approach for instruction scheduling and storage allocation that makes an extensive use of timing, resource and storage constraints to prune the search space for scheduling. The method in this approach supports VLIW architectures with (distributed) storage files containing random-access registers, rotating registers to exploit the available ILP in loops, stacks or fifos to exploit larger storage capacities with lower addressing costs.

Potential access conflicts between values are analyzed before and during scheduling, according to the type of storage they are assigned to. Using constraint analysis techniques and properties of colored conflict graphs essential information is obtained to identify the bottlenecks for satisfying the storage file constraints. To reduce the identified bottlenecks, this method performs partial scheduling by ordering value accesses such that to allow a better reuse of storage. Without enforcing any specific storage assignment of values, the method continues until it can guarantee that any completion of the partial schedule will also result in a feasible storage allocation. Therefore, the scheduling freedom is exploited for satisfaction of storage, resource, and timing constraints in one phase.

Samenvatting

Door een toenemende interesse in de markt voor zeer krachtige ingebedde processoren is de stand-alone processor wereld meer gericht op zaken als ontwerp flexibiliteit (synthetiseerbare processoren), energie verbruik, en de efficiëntie van het silicium. De implicatie voor architecturen en compilers voor ingebedde processoren is de noodzaak tot het benutten van hardware versnelling, instructie-niveau parallelisme, en gedistribueerde register files.

VLIW architecturen worden geroemd vanwege het beschikbare parallelisme in de architectuur met behoud van de orthogonaliteit van het overeenkomstige instructie repertoire. Methodes voor het genereren van programma code voor zulke processoren staan onder grote druk om de schaarse beschikbare middelen efficient te gebruiken, en ondertussen te voldoen aan de strakke tijdsbeperkingen opgelegd door de DSP en multi-media applicaties.

De beperkte beschikbaarheid van opslagmiddelen (bv registers) vormt een probleem voor methodes die programma code genereren in afzonderlijke fases, bijvoorbeeld het schedulen van instructies gevolgd door de allocatie van registers. Dit probleem wordt op verschillende manieren veroorzaakt door de conflicterende belangen van de twee genoemde fases: Het hergebruik van registers voorkomt het overlopen van de register files, maar creeert ook afhankelijkheden die niet in de originele code aanwezig zijn. Hoewel een bepaalde ordening van instructies de mogelijkheden kan doen toenemen tot het uitbuiten van ILP, kan deze ordening het tijds-interval vergroten waarin tussentijdse data beschikbaar dient te zijn. Dit vergroot op zijn beurt weer de druk op de register files. Aan de ene kant heeft de instructie scheduler afdoende lokale registers nodig om het ILP te benutten. Aan de andere kant worden tijdens register allocatie meer globale registers verkozen om het overlopen van de registerfiles te voorkomen. Een goede scheduler kan zijn bereikte niveau van ILP verliezen door het invoegen van overloop-code.

Zonder interactie tussen de fases voor scheduling en register allocatie zal de compiler schrijver moeten beslissen welk van de fases als eerste uitgevoerd dient te worden teneinde de meest efficiënte code te genereren. In afwezigheid van enige interactie tussen de fases van scheduling en register allocatie wordt code gegenereerd met een te hoog gehalte aan register overloop en/of een lager

niveau van ILP dan bereikbaar. Dit probleem wordt het fase-koppelingsprobleem genoemd en kan niet worden genegeerd in geval van stringente beperkingen en indien efficiënte oplossingen gewenst zijn. Traditionele methodes die programma code genereren in afzonderlijke fases, zijn vaak niet in staat om een efficiënte of zelfs geldige oplossing te vinden. Daarom hebben traditionele methoden in toenemende mate hulp nodig van de programmeur (of ontwerper) om een geldige oplossing te vinden. Omdat dit een buitensporige ontwerpinspanning vereist en een verregaande bekendheid met de processor architectuur, is er behoefte aan geautomatiseerde methoden die efficient om kunnen gaan met de verschillende beperkingen tijdens scheduling.

Dit proefschrift stelt een benadering voor instructie scheduling en register allocatie voor, waarin veel gebruik gemaakt wordt van de beperkingen mbt tijd en de beschikbaarheid van reken- en opslagmiddelen, teneinde de zoekruimte voor scheduling in te perken. De methode in deze benadering ondersteunt VLIW architecturen met gedistribueerde opslageenheden bestaande uit registers met willekeurige toegankelijkheid, roterende registers om ILP in loops te benutten, en stacks of fifos om een grotere opslagcapaciteit te benutten met lagere addresseringskosten. Mogelijke toegangsconflicten tussen data worden voor en tijdens scheduling geanalyseerd naargelang het type opslag toegekend aan deze data. Door het gebruik van technieken die de beperkingen analyseren en van eigenschappen van een gekleurde conflict graaf, wordt essentiële informatie verworven om knelpunten te identificeren mbt het voldoen aan de beperkingen opgelegd door de opslageenheid. Teneinde de geïdentificeerde knelpunten te verkleinen brengt deze methode een ordening aan op de lees- en schrijfacties van en naar de opslageenheid om meer hergebruik mogelijk te maken van die opslageenheid. Zonder een specifieke toekenning van data aan opslageenheden af te dwingen gaat de methode door totdat de garantie gegeven kan worden dat de opgelegde ordeningsrelaties overeenkomen met een geldige toewijzing van data aan opslageenheden. Dientengevolge wordt in een fase de schedule vrijheid benut ten behoeve van de beperkingen omtrent tijd en beschikbaarheid van reken- en opslagmiddeleen.

Contents

Acknowledgements	v
Summary	vii
Samenvatting	ix
1 Introduction	1
1.1 Design of Embedded Processors	1
1.1.1 Code Size	6
1.2 Compilers for Embedded Processors	7
1.2.1 VLIW architectures	8
1.2.2 Code generation phases	8
1.3 Silicon Compilers for ASICs	10
1.4 The Aim of this Work	12
1.5 Related Work	15
1.6 Limitations of our Approach	17
1.7 Thesis Overview	17
2 Definitions, Assumptions and Tools	19
2.1 Introduction	19
2.2 Architecture Template	19
2.3 Data Flow Graph	21
2.3.1 Vertices and operations	22
2.3.2 Data edges and values	23
2.3.3 Distances between operations and timing constraints	24
2.3.4 Resource and storage constraints	26
2.4 Representation of the Schedule Search Space	27
2.4.1 Apparent schedule freedom	27
2.4.2 Distance matrix	28
2.5 Basic Constraint Analysis Techniques	30
2.5.1 Execution interval analysis	30

2.5.2	Storage constraint analysis	33
2.5.3	Symmetry detection	33
2.5.4	Infeasibility Analysis	34
2.6	Conflict Graphs and Coloring	35
2.6.1	Basic definitions	35
2.6.2	Conflict graph	36
2.7	Description of the FACTS Research Tool	38
3	Storage Constraint Satisfaction	41
3.1	Introduction	41
3.2	Motivation	41
3.3	Problem Statement and Solution Approach	44
3.4	Storage File Selection	46
3.5	Conflict Graphs and Coloring	48
3.5.1	Constructing conflict graphs	48
3.5.2	Inaccuracies in conflict modeling	50
3.5.3	Conflict graph characteristics	51
3.5.4	Exact coloring algorithm	53
3.6	Bottlenecks for Storage Satisfaction	54
3.6.1	Bottleneck identification	56
3.7	Performing Access Ordering	58
3.8	Branch-and-Bound Approach	59
3.9	Run-time Complexity	61
4	Register Files	63
4.1	Introduction	63
4.2	Constructing Conflict Graphs	65
4.2.1	Conflict Rules	65
4.3	Lifetime Serialization	67
4.3.1	Heuristic of the minimum sacrificed distance	67
4.3.2	Serialization method	69
4.4	A Folded-Case Example	72
4.5	Value Merging	73
4.6	Lower Bound Register Estimation	76
4.6.1	The importance of a good lower bound	77
4.6.2	Lower bound register estimation over time intervals	78
4.6.3	Register estimation using probabilities	80
4.7	Experimental Results Using FACTS	81
4.7.1	Experiments with register file selection criteria	82
4.7.2	Comparison between lower bound algorithms	84
4.7.3	Experiments for register file capacity satisfaction	85

4.7.4	Experiments with value merging and symmetry detection	89
4.8	Rotating Register Files	90
4.8.1	Storage model	91
4.8.2	Constructing conflict graphs	92
4.8.3	Lifetime serialization	95
4.8.4	Experiments with rotating register files	96
5	Sequential-Access Storage	99
5.1	Introduction	99
5.2	Storage Model	100
5.2.1	Multiple consumption of values	102
5.3	Satisfaction Approach for Stacks and Fifos	103
5.4	Conflict Graphs for Capacity Satisfaction	104
5.4.1	Stack access conflict rules	105
5.4.2	Fifo access conflict rules	108
5.5	Access Ordering	111
5.5.1	Stack access ordering	111
5.5.2	Fifo access ordering	114
5.6	Conflict Graphs for Unit Size Satisfaction	116
5.6.1	Sequential access conflict rules for stacks	116
5.6.2	Multiple value instances in conflict graphs for fifos	118
5.6.3	Sequential access conflict rules for fifos	119
5.6.4	Bottleneck identification and reduction	120
5.7	Experimental Results Using FACTS	122
6	Storage Files of Different Access Types	127
6.1	Introduction	127
6.2	Storage Model	128
6.3	Proposed Approach	129
6.4	Experimental Results Using FACTS	132
7	Conclusions and Further Research	135
Bibliography		139
Biography		147

Chapter 1

Introduction

1.1 Design of Embedded Processors

Embedded systems [25] are electronic systems characterized by being reactive on events coming from its environment, very often in real-time, and efficient in terms of performance, small area, low power, short time-to-market and cost (for the high volume market). Typical application domains of embedded systems include telecommunications, automotive and consumer electronics products.

Systems-on-chip (SoCs) are embedded systems that embed (processor) cores, *field-programmable gate arrays* (FPGAs), or *application specific integrated circuits* (ASICs). Processor cores or *embedded processors* are designed specifically for integration on a chip together with other processors, communication structures, and data/program memory. Due to their flexibility, programmable embedded processors are increasingly used as components of SoCs [61]. Integrating electronic systems on a single chip rather than on a printed circuit board offers significant advantages in terms of inter processor communication, packaging cost, time delay, and energy consumption. If possible, part of the required functionality is implemented in dedicated hardware, but frequently the requirements impose a degree of flexibility on the design that can only be realized with a programmable component.

Different classes of embedded processors exist [43]: microcontrollers, RISC processors, *digital signal* processors, multimedia processors and *application specific instruction set* processors known as ASIPs.

- Microcontrollers and RISC processors are employed mostly for control-type tasks that are usually not very computationally intensive.
- Digital signal processors (DSPs) [41, 42] have been designed for arithmetic-intensive signal processing applications like channel/source (de-)coding,

signal transformations (e.g. FFT), noise reduction, etc. They can have special hardware components and a certain degree of *instruction level parallelism* (ILP), i.e. the number of operations that can execute in parallel. Also, DSPs contain *special-purpose* registers. Due to the irregularity of their architecture (see Figure 1.1) the compiler construction for DSPs is more difficult compared with the other processor classes. Texas Instruments leads the DSPs market followed by Motorola, Analog Devices, and NEC.

- Multimedia processors are based on the *very long instruction word* (VLIW) philosophy, in which different functional units can operate in parallel and are controlled by separate fields in the instruction word. The architecture is more general-purpose oriented than DSPs and has general-purpose registers grouped in files. These processors are developed with the purpose of allowing efficient compiler support. Examples of multimedia processors are the Philips TriMedia TM1000 [30] and the Texas Instruments C62xx processor family [70].
- ASIPs are domain-specific processors. ASIPs serve only a narrow range of applications and are characterized by a small, well-defined instruction-set tuned to the critical parts of the application code. The basic architecture of an ASIP is fixed (template-based architecture), but it can be customized for a given application by setting a number of different parameters. With different sets of parameters, different configurations of an ASIP may be available. Therefore, *retargetable* compilers can be used instead of using a large number of different compilers for each configuration. This implies that the compiler has to be machine-independent and be adapted to a certain machine by writing machine-specific compiler components or by providing a model or machine description (see diagram of Figure 1.2). However, retargetability compromises code efficiency, since the compiler can make fewer assumptions about the target machine and no aggressive code optimizations can be performed.

An embedded processor has to fulfill some practical requirements to be able to communicate and interact with the environment. Those include I/O peripherals, a bus interface, hardware for receiving interrupts, and possibly a real-time kernel. The following are considered important criteria in the design of embedded processors for DSP and multimedia applications: low power consumption, silicon cost, performance, short hardware/software development time, and porting of legacy code.

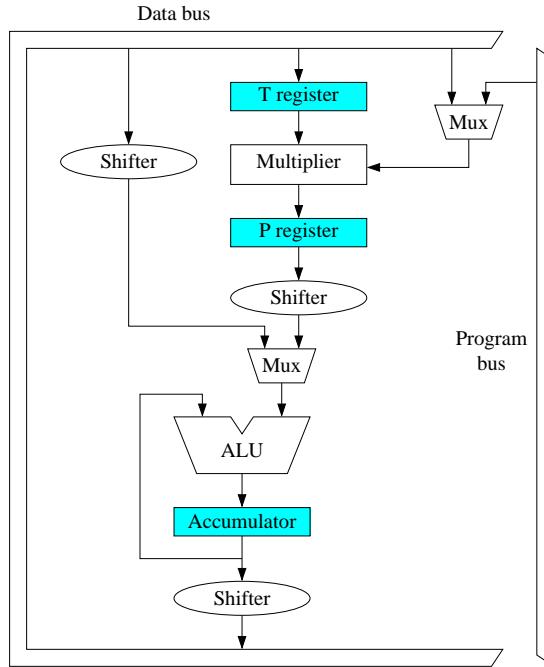


Figure 1.1: Rough representation of the TMS320C25 DSP processor.

Low power consumption

Low power consumption is important not only to save battery energy in case of portable devices, but also to have low cooling requirements. With the expected increase in transistor density, it can be expected that cooling requirements will increase as well.

One of the major contributors to power consumption in SoCs is memory access. A large research effort has been spent at IMEC to reduce power inefficient data memory accesses [13]. Other research efforts to limit the power consumed by accesses to program memory have mainly focussed on the use of a program cache [32]. A caching strategy may be effective for stand-alone processors with off-chip program memory because instruction fetches take a long time and are more efficient in bursts than when individual instructions are fetched. For processors with embedded program memories an instruction fetch from cache can hardly be more efficient than from the program memory, and the efficiency of bursts is less than in the case of off-chip memories. The alternative strategy is to minimize the program code size, which is discussed in more detail later.

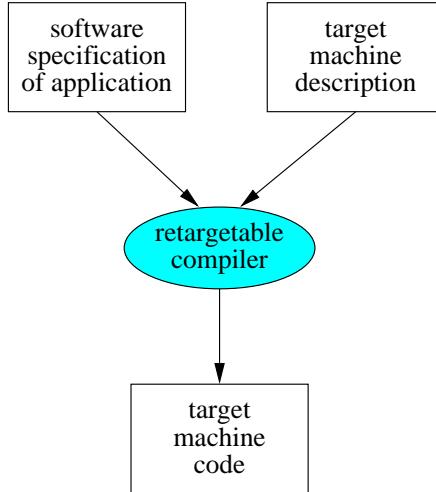


Figure 1.2: Retargetable compiler.

Silicon cost

The importance of the silicon cost depends on the application area. Since SoCs are mostly targeted at the consumer electronics market, they are inherently concerned with high volumes. For high volumes the total cost per product depends largely on the silicon cost of individual chips.

The silicon cost of an embedded processor is partly determined by the functional (e.g. arithmetic) units, the register files for storing intermediate values, and the communication structure, all of which depend on the required data bit width. It is therefore advisable to design or buy a processor core that does not offer a larger data bit width than necessary.

The size (cost) of such an embedded processor however tends to be dominated by the data and program memories. This is largely due to the different technologies used for fabricating memory and logic circuits (the largest part of a processor core). It is difficult to combine these different technologies on a single chip. As a result, usually the choice is made to apply logic technology for implementing both the logic devices and the memory modules, thereby using more area compared to stand-alone memory implemented in memory technology.

Furthermore, the dominance of memory will grow in the future because the increasing amount of data processing in practical applications, the increasing need for flexibility often resulting in more program code, and technology advances in VLSI integration which provide a larger benefit for logic circuits than for (already density packed) memory circuits. Therefore, minimizing program code size is also a sensible strategy for minimizing costs.

Performance issue

Certainly, the performance of the processor has to satisfy the application requirements. Performance overkill should be avoided because other criteria are easily compromised.

One characteristic of processors for DSP and multimedia applications is that performance is not obtained from a high clock speed, which is the case with general-purpose processors like the Intel family, but from the parallel execution of multiple operations. A parallel processor is easier to design than a sequential processor with the same performance, but can only be used for inherently parallel applications like DSP. Increasing the performance of an embedded processor comes at the cost of spending more hardware development effort and time, larger silicon area, and tolerating more energy consumption.

An efficient way to increase processor performance is to identify performance critical (sub-) functions in the specification (often program code) with highly regular behavior that requires little flexibility, and implement those functions in dedicated hardware. If functions are relatively large, the implementation may take the form of a (non-programmable) coprocessor that communicates with the main processor via a system bus. Otherwise, the function is implemented as a functional unit within the processor (like an ALU), which is controlled from the instruction set.

HW/SW development time

The design of SoCs is considered a HW/SW codesign problem [39], where from a specification the system is partitioned into hardware and software components following some criteria and respecting constraints according to the application. The hardware components are then synthesized, or taken from a library, or brought in as IP (*intellectual property*) blocks, and the software components are to be executed on a processor core.

HW/SW development time is a significant factor in the time-to-market. It is important to keep it short in order to obtain a reasonable market share on a short term. Hardware development time is only a valid criterion if the processor core is developed in-house.

This development time can be limited by designing at a high abstraction level. Programming at assembly level simply requires more work than a high-level language like C. C programming requires the use of a compiler, which introduces a certain overhead in both the schedule length and code size [61]. Designing the processor hardware at a high abstraction level essentially requires the availability and use of standard hardware component libraries and automated synthesis tools (also called *silicon compilers*). Similar to software compilers, silicon compilers

introduce a certain overhead in both area and timing, thereby limiting the clock speed and thus the performance of the processor.

Porting the legacy code

With the accumulation of software and the reuse policy that is currently popular among the consumer electronics vendors, it is becoming increasingly important to be able to re-implement (port) existing software on the new processors. This inherited software is called *legacy code*.

Coping with legacy code is related to the need for a compiler. This observation results from the fact that the legacy code is usually written in C because it is too difficult to port code efficiently at the assembly level (unless there exists a clear correspondence between instructions of the previous processor and the new one).

As a consequence, three issues were identified which are considered important in the design of embedded processors for DSP and multimedia applications: code size, compilers, and application specific hardware (ASIC). In the next subsection the code size issue is briefly described, while for the compiler and the application specific hardware issues more extensive descriptions are provided in Sections 1.2 and 1.3 respectively.

1.1.1 Code Size

Limiting code size has been considered a good engineering practice rather than a topic of scientific research. The strategy taken so far is to design the instruction set in such a way that individual instructions have a high “expressive power”. That is, as many operations (RISC instructions) as possible are packed in a single instruction word. This limits the flexibility for executing individual operations.

As an example, there may be an instruction for performing a load and a multiplication simultaneously, but the load can only be done to registers $r0$ or $r1$, and the left operand for the multiplication can be only fetched from registers $r3$ or $r4$, and the right operand from $r7$ or $r8$.

Another possibility is to use small instruction words (e.g. 16 bits) for frequently occurring combinations of operations, and larger instruction words (e.g. 32 bits) for less frequently occurring combinations.

Another example is the use of indirect addressing. When data are fetched from memory, a memory address of e.g. 16 bits has to be specified. A more efficient addressing mechanism is to have a base address of 16 bits which is stored inside the processor, and an offset of e.g. four bits which is specified in the instruction word. In this way, highly efficient instruction sets have been designed such as the one for the EPICS processor [80].

1.2 Compilers for Embedded Processors

Nowadays, the importance of software compilers is increasing in HW/SW code-sign processes, since large parts of the application functionality become implemented in software.

The use of a compiler introduces a certain overhead in both the schedule length and the code size, and therefore conflicts with cost, power, and performance criteria. Much effort has been spent at improving DSP compilation techniques, unfortunately with disappointing results [61]. From that research it was learned that the efficiency of the compiler has a very strong interaction with the processor architecture and the corresponding *instruction set architecture* (ISA). Unfortunately, an ISA designed for small code size is just about the worst possible compiler target.

The constraints that result from an irregular ISA are hard to model efficiently. To deal with them in a systematic and structural way is very difficult. If it is possible to make an efficient compiler for one such ISA, it would still require a huge software engineering effort. Furthermore, the compilation times still would be unacceptable, and the compiler would be extremely hard to retarget for another processor.

Therefore, embedded processors need:

- High-performance compilers for low-cost irregular architectures with heterogeneous register structures.
- This implies that the compiler development process must offer:
 - Rich data structures, to support the complex instruction sets and algorithmic transformations required to exploit these effectively.
 - Extensive search to explore the numerous register allocation, scheduling, and code selection permutations.
 - Methods for capturing architecture specific compiler optimizations easily.
- An environment that supports the rapid development of these compilers, due to the variety of processors to support.

In this context, the main goal for software designers is to develop a compiler that can efficiently transform high-level language descriptions of the applications and map them onto a cost- and power efficient architecture.

The numerous architectural constraints of an embedded processor have to be taken into consideration by the compiler. This implies that all the phases of compilation need knowledge of the architectural features of the target. A compiler would benefit from the incorporation of an architectural model to describe hardware constraints.

1.2.1 VLIW architectures

In previous sections, the fundamental contradiction between small code size and “compiler friendliness” was discussed. This makes it extremely difficult to create an embedded processor that is cheap and energy efficient on one hand, and allows for high-level programmability and easy porting of legacy code on the other.

“Compiler friendliness” is associated with *orthogonality* of the instruction set. An instruction set exhibiting operation orthogonality is the one in which the presence, elements (operands), and execution of one operation in the data-path are independent of the presence, elements, and execution of any other operation in the instruction.

But, which processors and ISAs are good compiler targets? Over the last five years the *very long instruction word* (VLIW) architecture [65, 24] has gained enormously in popularity among compiler builders.

Characteristics of the VLIW architecture are the high levels of ILP, more registers and a regular interconnection, which all provide a large flexibility of the instructions (see Figure 1.3). For example, in the ideal situation where the processor contains a single register file, each instruction may address the complete range of registers. This is very convenient for a compiler because the decision which functional resource would execute a certain operation can be made completely independent of the decision which registers would store the operands.

This type of independence of mapping decisions allows for a modular approach of the compiler software, which is a must for the transparency, maintainability, extendibility, retargetability, and efficiency of the compiler. Such flexible instructions are in contrast with processors designed for a small code size, where flexibility is sacrificed to limit the number of control bits necessary for addressing the corresponding instruction.

1.2.2 Code generation phases

Generally, a compiler uses an *intermediate representation* (IR) of the source program, which is generated by the source language front-end at the beginning of the process (see Figure 1.4). This representation is machine-independent and a number of transformations are applied on the IR to obtain a more efficient machine code. Those transformations include: common sub-expression elimination, dead code elimination, loop-invariant code motion, constant folding, and so on [1].

Code generation maps machine-independent IR statements to machine-specific assembly instructions, thereby generating an assembly program whose functionality is equivalent to that of the source. Code generation is roughly divided in three main processes:

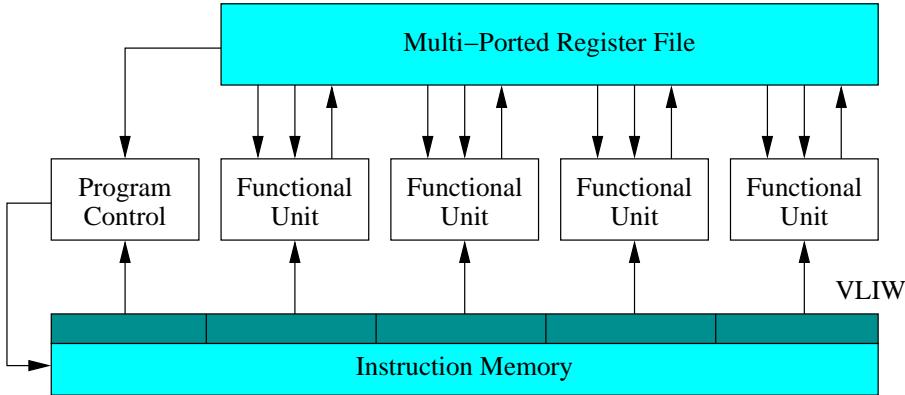


Figure 1.3: Data-Path of a typical VLIW architecture.

- *Code selection.* This is the process that decides which assembly instructions will be used to implement the input statements. The optimization of this process consists of selecting a set of assembly instructions of which the total cost is minimal according to some criteria. Code selection is highly related to *tree pattern matching* and solved with techniques like *tree parsing* [1]. Work concentrating on code selection problem includes [46, 45, 7] and [44] for media processors with special instruction sets.
- *Scheduling.* Decides when instructions will be executed during run time in terms of clock cycles. In this process the dependencies between instructions, delays, limited number of resources, latency, and pipeline effects are taken into account. Also, it has to efficiently exploit the amount of parallelism available in the processor. The corresponding optimization should be to minimize the execution time for the schedule. Optimal instruction scheduling is an NP-hard problem, so heuristics are applied. The most popular of these heuristics is list scheduling [19], which schedules instructions one after another following a *priority function*.
- *Register allocation.* This process determines where the results of computations are stored, either in memory or in registers. Since the number of available physical registers can be limited, this process decides which program values can share a register, such that the number of registers required at any clock cycle does not exceed the physical limit. During register allocation additional instructions can be added that *spill* and *reload* values to/from memory. A popular approach to deal with the register allocation problem is based on graph coloring [14, 12]. The goal of the optimization is to store many values to registers thus avoiding more expensive memory accesses.

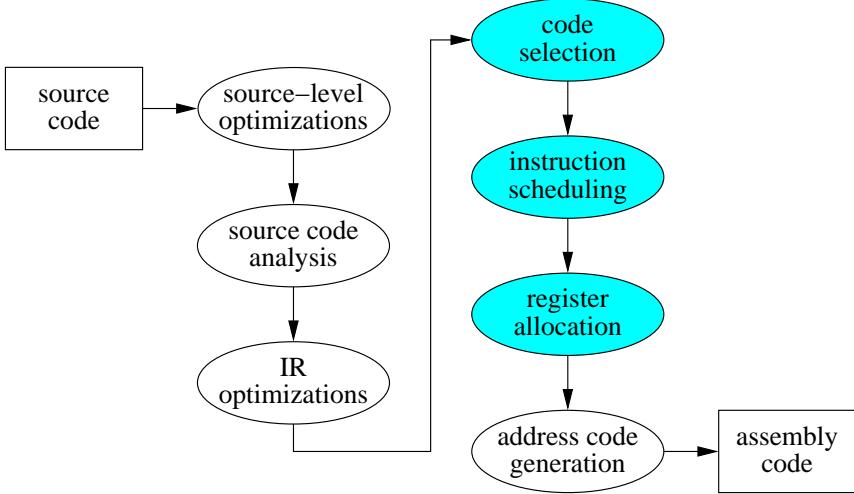


Figure 1.4: Coarse compilation flow.

1.3 Silicon Compilers for ASICs

An *application specific integrated circuit* (ASIC) is a chip designed and dedicated to a single application. The process of translating an ASIC specification to a chip layout description is called *silicon compilation*.

The design flow of a silicon compiler, see Figure 1.5, starts either with a *behavioral specification* of the integrated circuit, or with an *algorithmic description* in a hardware description language like VHDL, Verilog or a C-based language.

Architectural synthesis (also called *high-level synthesis* [49]) tools take the behavioral specification together with goals and constraints as their input, and produce a so called *register transfer level* (RTL) description of the chip architecture. Such architecture consists of a *data-path* in the form of a network description, and a *controller* in the form of a symbolic finite state machine. A data-path is a collection of basic building blocks like adders, multipliers, ALUs, shifters, memory elements, etc., which are interconnected by buses and (de-)multiplexers. The corresponding controller governs the data flow in the architecture.

Both the controller description and parts of the data-path are then transferred to *logic synthesis* tools, that transform the RTL description into an implementation at the *gate level*. The final synthesis steps performed by the *layout synthesis* tools consist of layout design subtasks such as *floor planning*, *placement* and *routing*. The final result is a set of geometrical descriptions of *layout masks*, which are a complete physical description of the ASIC under construction.

The objective of architectural synthesis is to support design at a higher level

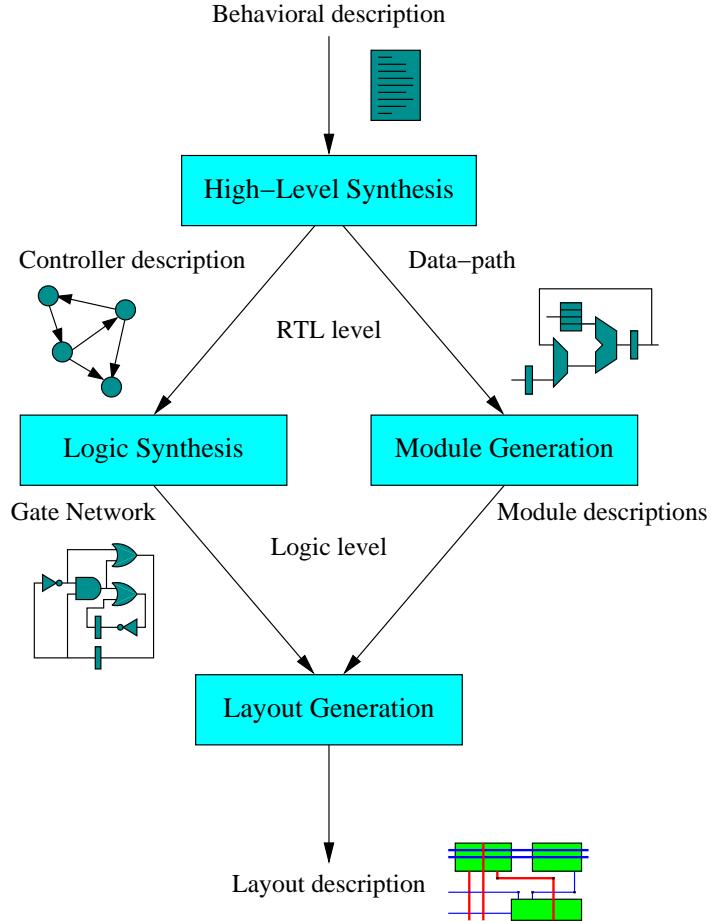


Figure 1.5: Silicon compiler overview taken from [31].

of abstraction than logic and layout synthesis. That level of abstraction enables better design space explorations, so a designer should be capable of making better VLSI designs within a shorter amount of time.

The architectural synthesis task is decomposed into several subtasks [54]. The first decomposition that is normally made is the decomposition into *data-path synthesis* and *controller synthesis*. The following three main subtasks are generally distinguished when synthesizing a data-path from a behavioral description:

- *Selection* or *allocation* determines the type and number of resources needed for a data-path. The storage type and the number of storage files are also considered.
- *Scheduling* determines an assignment of the operations to be executed in the

data-path to specific moments in time (clock cycles).

- *Binding* determines an assignment of the operations to specific resources, and an assignment of values to specific storage, e.g. registers.

Depending on the application, different goals and constraints in terms of area, timing, power consumption, etc. are imposed, possibly leading to different search strategies for selection, scheduling and binding. Scheduling and binding have a first-order impact on the area and performance of the final design and hence, are considered to be the most important steps in the synthesis process.

The presence of large compute-intensive basic blocks in DSP and multimedia applications lends itself to a wide variety of optimizations, which can result in a large number of RTL designs. In order to quickly explore the large design space for good solutions, a careful definition of the target architecture prior to synthesis task is essential [64].

1.4 The Aim of this Work

The focus of this work is on scheduling and storage allocation in *code generation* and *architectural synthesis* of embedded processors. The targeted application domains are DSP and multimedia.

DSP and multimedia algorithms are characterized by intensive computation (as compared to I/O), that can be represented by data flow graphs. Examples of such application kernels are convolution, correlation, filtering, transformations, and modulations, which require simple arithmetic operations of multiplication, addition/subtraction, and shifts to carry out.

The general VLIW architecture (template) assumed is shown in Figure 1.6.

Code generation and synthesis methods are hampered by the combination of tight timing constraints imposed by signal processing applications, and resource constraints implied by the processor architecture or template. Additionally, because of the distributed storage file architecture, the possibly limited connectivity, and ILP exploitation for innermost pipelined DSP loops, the amount of storage resources required per storage file (*storage pressure*) increases, so the compiler or synthesis tool is pushed to efficient usage of storage.

The storage requirements of a schedule are of extreme importance for a compiler since any valid schedule must fit in the available number of storage units of the target machine. *Spilling* values to background memory should be avoided because the additional load and store operations jeopardize the timing constraints and memory bandwidth availability, which is already critical in inner DSP loops.

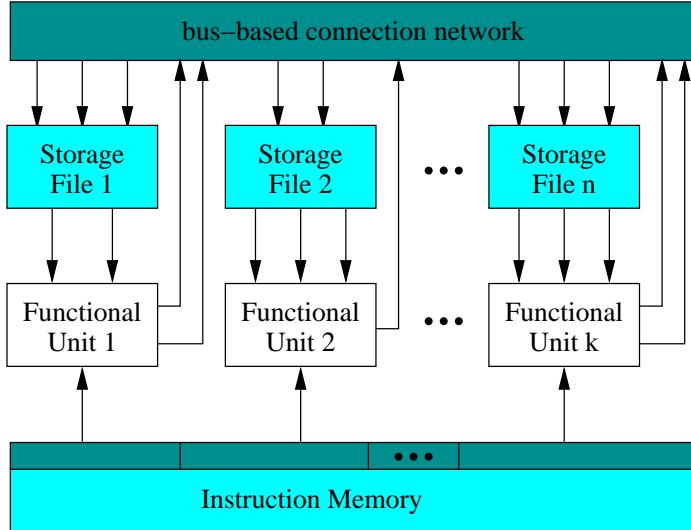


Figure 1.6: General VLIW architecture (template).

Specially limited resource availability (e.g. registers) presents a problem for traditional methods that perform code generation in separate stages (e.g. scheduling followed by register allocation). This separation often results in sub-optimality or even infeasibility of the generated solutions because it ignores the problem of *phase coupling*. I.e. since value lifetimes are a result of scheduling, scheduling affects the solution space for register allocation.

However, even a throughput-optimal schedule with minimum storage requirements is useless if it requires more storage than the target machine has. When there is a limited amount of storage and the storage allocator fails to find a solution, some additional action must be taken. One alternative is timing constraint relaxation, e.g. for loop folded schedules is to reschedule the loop with an increased initiation interval, like in the Cydra 5 compiler [20]. Rescheduling the loop with a larger initiation interval usually leads to schedules with less iteration overlapping, and therefore with less register requirements. Unfortunately, the register reduction is at the expense of reduction in performance (less parallelism is exploited).

Therefore, although the separation of scheduling and storage allocation offers better chances to get methods that are run-time efficient, it makes it much more difficult to cope with the interaction of timing, precedence, resource, and storage constraints.

Traditional methods need an increasing amount of help from the programmer (or designer) to arrive at a feasible solution. Because this requires an excessive

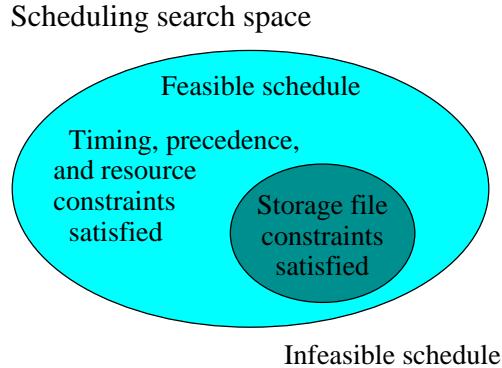


Figure 1.7: The scheduling search scope.

amount of design time and extensive knowledge of the processor architecture, there is a need for automated techniques that can cope with different kinds of constraints *during* scheduling. By exploiting these constraints to prune the schedule search space (as performed in [50]) the scheduler is often prevented from making a decision that inevitably violates one or more constraints.

Given the application and the target architecture, the goal therefore is to find a scheduling solution that will respect timing, precedence, resource, and the storage constraints, during code generation or synthesis of DSP and multimedia algorithms. In order to do that, constraint analysis techniques, described in Section 2.5, are used in the core of our approach to drive the scheduler to find a solution.

The combination of storage allocation and scheduling overcomes the problem of phase coupling and relaxes the insertion of spill code. The method “alternates” between scheduling and storage allocation tasks by making a decision for storage allocation and subsequently analyzes how that prunes the search space for scheduling (see Figure 1.7).

Furthermore, not only random-access (distributed) register files are considered. The proposed method also supports storage files with other types of units like rotating registers, stacks, and fifos. This work shows that the main storage constraint satisfaction method is independent on the type of storage units used.

The constraint satisfaction process can be summarized as follows:

- Potential access conflicts between values are analyzed before and during scheduling according to the type of storage they are assigned to. Using constraint analysis techniques and properties of colored conflict graphs essential information is obtained to identify the bottlenecks for satisfying the storage file constraints.

- To reduce the identified bottlenecks, this method performs partial scheduling by ordering value accesses such that to allow a better reuse of storage.
- Without enforcing any specific storage assignment of values, the method continues until it can guarantee that any completion of the partial schedule will also result in a feasible storage allocation. Therefore, the scheduling freedom is exploited to satisfy storage, resource and timing constraints in one phase.

1.5 Related Work

Most methods for global register allocation and assignment are based on graph coloring [14, 12]. However, they assume that operations are already scheduled and perform spilling when the number of registers is not enough.

Integration of the scheduler and the register allocator has been considered as complicated. Another reason not to integrate the two tasks is the lack of a global view for the register allocator. In spite of these disadvantages, integration of the two tasks is important for ILP processors [57].

Bradlee [10] presents a method where operation scheduling and register allocation are mixed. Given a certain number of registers, the initial scheduling gives estimates of the scheduling cost in terms of machine clock cycles. The scheduling is performed per basic block with a limited number of registers first, and with the maximum number of registers. These cost results are used to allocate a certain number of registers for each basic block.

An integer programming model which simultaneously schedules and allocates functional units, registers, and buses, is presented by Gebotys and Elmasry [27] for synthesizing architectures. However, as a synthesis approach it tries to find the minimum resource usage and best performance, without any specific goal for constraint satisfaction.

Some work is being done on the interaction between operation scheduling and register allocation. Papers [56, 63] describe a framework where register allocation is done before scheduling. In those approaches, the allocator is made aware of the code motions the scheduler wants to perform, i.e. loads and stores inserted as spill code can be moved outside of loops. With this information the allocator tries to avoid register assignments that prevent code motions.

A different approach integrating scheduling and allocation is presented in [9]. Extra dependences and spill code are added to the program dependence graph such that excessive resource (both functional unit and register) requirements are reduced before scheduling.

Sharma and Jain in [72] present a resource and performance constrained versions of a synthesis algorithm developed for bus architectures that integrates allocation and scheduling of functional, storage, and interconnection units into a single phase. The proposed synthesis algorithm is driven by estimation tools [71] that perform hardware allocation. Their approach tries to minimize the number of functional units, registers, and interconnections during scheduling, however they consider only the latency (performance) constraint. A fixed number of resources (e.g. registers) is not considered. A similar approach can be derived from the work of Ohm et al. in [60].

Janssen and Corporaal [35] present a method that integrates instruction scheduling and register allocation called “registers on demand”. Their approach schedules sequentially, i.e. following a precedence order, basic blocks from a region. Then, a register is assigned to a variable as soon as an operation that access the variable (read or write) is scheduled. To ensure that variables with overlapping lifetimes are not assigned to the same register some bookkeeping is done, which allows also to find free registers whenever a variable needs one. When there are no more available registers spill code is inserted. Spill code is not allowed in already scheduled code, therefore rescheduling is avoided in Janssen’s approach. Moreover, when register assignment and scheduling of operations become critical, there is a certain flexibility to relax timing constraints.

Kolson et al. describe a special technique for allocation of special-purpose registers in loops [36] which minimizes the number of spills between registers and memory. Their technique is based on incorporating loop-unrolling techniques into the algorithm, assigning registers to values, and checking if the register assignment at the beginning of the loop matches the assignment at the end. However, this approach does not work with timing constraints, but tries to find some optimum initiation interval with a budget of registers and spill code added.

Rau et al. present a fine-grained approach of phase coupling designed for compilers of VLIW processors [67]. Their approach consists of gradual operation and register binding according to constraints like timing and number of resources. The process goes through code selection, partitioning, scheduling, register allocation, and final code emission. However, scheduling of operations is performed before register allocation with spill code insertion. A second pass of scheduling (post-pass scheduling) is necessary to schedule the spill code introduced by the register allocator.

Constraint analysis techniques introduced by Timmer [73] are the foundation of the work presented in this thesis. They provide a method to prune the scheduling search space that exploits the constraints imposed by the application algorithm or by the targeted architecture. Additionally, the work of Mesman [50] uses constraint analysis to drive the scheduling process to make decisions that are feasible, i.e. respecting precedence, timing and resource constraints. Mesman’s

work introduces the idea of alternated scheduling and register allocation by inserting precedences between operations in order to satisfy register constraints during scheduling, avoiding the insertion of spill code. This thesis describes with details the scheduling and register allocation approach introduced by Mesman, and shows that the approach can be applied not only to random-access register files but also to files with rotating registers, stacks or fifos.

1.6 Limitations of our Approach

The storage satisfaction approach presented in this work does not deal with the following issues:

- Operation assignment is not included in the process. A given assignment of values to storage files is assumed often implied by the assignment of operations to functional units. Bekooij [8] presents an approach that uses constraint analysis for the assignment of operations to functional units (data routing through a limited connection network between functional units and register files).
- No conditional or nested loop constructions (with several basic blocks) are supported. However, *if-conversion* can be applied previously storage constraint satisfaction and resulting predicates are included in the code [82].
- Instruction constraints are not considered. When instruction sets are highly encoded to minimize code size there exists a phase coupling problem between instruction selection and scheduling. Zhao et al. in [81] present an approach that reduces the need for explicit instruction selection by translating the constraints implied by the instruction set into resource constraints.
- The compiler retargetability is limited when tuning the processor to an application (domain) because no complete machine description is defined (refer to Section 2.7, the description of FACTS research tool).

Despite these limitations our approach is in many cases able to satisfy tight precedence, timing, resource, and storage constraints.

1.7 Thesis Overview

This thesis is organized as follows. Chapter 2 defines the basic concepts necessary to understand the storage satisfaction approach. The architecture template, data flow and conflict graphs are introduced, and the basic constraint analysis

techniques are described. The FACTS tool developed for scheduling and storage allocation which makes intensive use of constraint analysis, is presented at the end of the chapter.

In Chapter 3 the problem statement is given, and the general solution strategy is proposed, which proves to be effective for different types of storage. Also, it describes the construction, properties, and the way conflict graphs are used in the satisfaction approach.

The satisfaction approach applied to (random-access) register files is described in Chapter 4. The number of registers available called the *capacity* of the register file is the constraint to satisfy through serialization of value lifetimes. That chapter also presents some techniques to improve the accuracy and performance of the approach for random-access registers. Moreover, in order to better exploit the available parallelism of resources in the target processor by means of reducing the loop period, rotating register files are introduced, and the satisfaction approach for this storage type is also described.

Chapter 5 extends the satisfaction approach to the case where stacks or fifos are used in the storage files. Constraint satisfaction has not only to deal with the number of storage units available but also with the number of registers per unit, and their particular access behavior.

In Chapter 6 all the rules and methods presented in previous chapters are combined to obtain a constraint satisfaction method for storage files of different access types.

Finally, Chapter 7 provides the conclusions and further research.

Chapter 2

Definitions, Assumptions and Tools

2.1 Introduction

As an input of the constraint satisfaction and scheduling approach, each application is described by a *data flow graph*, which is the most widely used RTL-level specification model [38]. In addition, the timing constraints of the application and the characteristics of the target architecture, e.g. functional units and storage files, are given to complete the set of requirements.

This chapter presents the basic definitions, assumptions, and the tools used in this work. The main characteristics of the targeted architectures are described first. Data flow graphs, the operation distance matrix, and graphs to model the potential access conflicts between values, are introduced next. Finally, the basic analysis tools implemented in the framework FACTS are presented.

2.2 Architecture Template

As introduced in the previous chapter, the general processor architecture shown in Figure 1.6 (repeated again for convenience in Figure 2.1) is assumed.

It is generally a VLIW architecture. The processor issues a fixed number of instructions formatted as one large instruction. Each instruction addresses a functional unit which executes the operation, and a number of source or target operands in storage files. The functional units execute in parallel, each fetching its operands from the (distributed) storage files at the beginning of a clock cycle, and writing the result back to one or more storage files at the end of a later clock cycle. It is assumed that the delay for each data transfer is fixed. Access to external memory is assumed by specific load and store operations (*load/store* architecture).

A *storage file* SF denotes a group of storage units that share read/write ports and addresses, e.g. a file with random-access registers as storage units (register

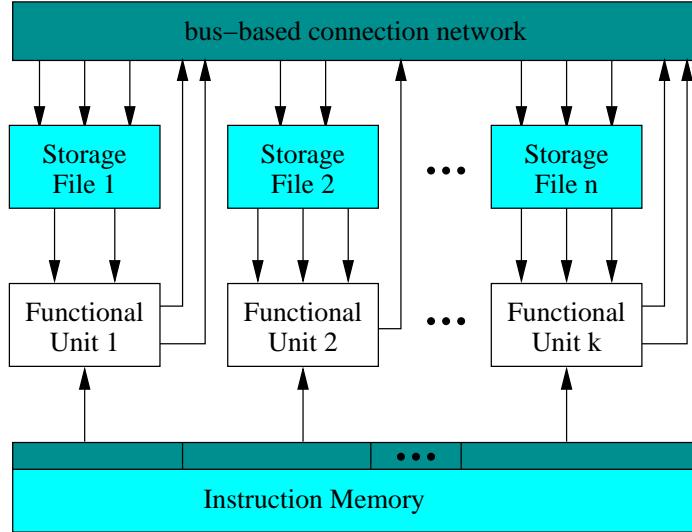


Figure 2.1: Architecture template.

file).

The storage file *capacity* $c(SF)$ is the number of units available for assignment in storage file SF . In case of having only one storage unit, it is seen as a storage file with capacity equal to one.

In a storage file, the limited numbers of read and write ports can be modeled as resource constraints. Resource constraints are explained later in this chapter.

It is assumed that the data routing, i.e. the connections between functional units and storage files, has already been solved like it is done in [8], which guarantees that every data transfer can be performed within the connection network.

Storage units in a storage file SF are identified by an unique index or address and have a specific access behavior. A storage unit can be:

- *Random-access register*, or simply *register*. This storage is called “random-access” because any single register can be accessed directly. One register can store only one value per clock cycle. Values sharing the same register must have their lifetimes serialized, i.e. a value has to be consumed (read) before another value is produced and written into the same register.
- *Rotating register*. Rotating register files have been designed to allow exploitation of the available ILP by means of reducing the initiation interval of loops. In such a file registers are addressed directly using a “base plus offset” addressing model. The offset comes from the instruction while the base

is inherent to the register file and is decremented each time a new loop iteration starts while maintaining the offset (that comes from the instruction), providing a dynamic register renaming under the control of the compiler.

- *Stack*. A storage unit with multiple registers or locations sharing a read/write port and one address. Several values can be stored in a stack and be simultaneously alive in the same clock cycle. The main characteristic of a stack is that the first value to be read is the last value that was stored (*last-in first-out* access behavior).
- *Fifo*. A storage unit with multiple registers or locations sharing a read and a write ports and one address. Several values can also be stored in a fifo and be simultaneously alive in the same clock cycle. Registers are disposed in a queue-like way, and the main characteristic of a fifo is that the first value to be read from the storage is the first value that was stored (*first-in first-out* access behavior).

It is assumed that storage units have a fixed number of bits per register (fixed *word-size*), and that any incompatibility of operand's size has been resolved.

For stack and fifo units, $q(SF)$ is the number of registers per unit, also called the unit *size* or the depth.

Once the target architecture has been defined, the algorithmic description of the application is specified as follows.

2.3 Data Flow Graph

The algorithmic description of an application is partitioned into *basic blocks*.

A basic block [1] is a maximal length sequence of consecutive statements or operations in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. That is, if the first statement of the block is executed, then all of the others will be executed in sequence until and including the last one in the block. Basic blocks can be computed by first determining entry statements and then associating each statement with one entry.

Each basic block is represented by a data flow graph (*DFG*), which indicates the primitive operations performed in that block and the dependencies between them.

Definition 2.1 A data flow graph $DFG = (V(DFG), E(DFG), \Delta, W)$ is a directed, weighted graph, where:

- $V(DFG)$ is the set of vertices (operations).
- $E(DFG) \subseteq V \times V$ is the set of precedence edges.
 - $E_d(DFG) \subseteq E$ is the set of data dependency edges (values).
 - $E_s(DFG) \subseteq E$ is the set of sequence edges. $E_s \cap E_d = \emptyset$.
- $\Delta : V \rightarrow \mathbb{N}$ where $\Delta = \{\delta(Op) | Op \in V\}$ is a function defining the execution delay in clock cycles associated with operation Op .
- $W : E \rightarrow \mathbb{Z}$ where $W = \{w(Op_1, Op_2) | (Op_1, Op_2) \in E\}$ is a function defining the timing delay in clock cycles associated with each precedence edge between operations Op_1 and Op_2 .

The execution delay is the number of clock cycles needed for the completion of the operation.

Values are consumed at the beginning of an operation and produced at the end of the operation (after the execution delay).

A weight $w(Op_1, Op_2) = w_d$ of a data dependency edge between operations Op_1 and Op_2 implies that the data produced by Op_1 would be consumed by Op_2 at least after w_d clock cycles ($w_d \geq 1$).

A weight $w(Op_1, Op_2) = w_s$ implies that operation Op_2 has to be scheduled at least w_s clock cycles after Op_1 .

Figure 2.2 shows an example of a data flow graph. Operations in the data flow graph are *source*, A, B, C, D, E and *sink*. The execution delays are for example $\delta(\text{source}) = 0$ (dummy operation, see next section), $\delta(B) = 1$, and $\delta(A) = 2$. Data dependency edges are a, b, c and d , the weight associated to each data dependency edge is one, e.g. $w(a) = w(A, C) = 1$. Sequence edges are $(\text{source}, A), (\text{source}, B), (A, B), (B, D), (C, D), (E, A)$ and (E, sink) with weights, e.g. $w(\text{source}, A) = 0, w(B, D) = 3$ and $w(E, A) = -6$.

2.3.1 Vertices and operations

Two vertices (dummy operations) are always assumed to be present in the DFG : *source* and *sink*. All vertices are reachable from the source and the sink is reachable from all vertices. The source and the sink have no execution delays, and represent respectively the first (entry vertex of the DFG) and the last (exit vertex) operations to be executed (see Figure 2.2).

For reasons of simplicity it is assumed that operations are not pipelined, and the data introduction interval (or restart time when the functional unit is ready for reuse) of each operation is equal to its delay. In [53] it is shown how pipelined and other types of operations can be modeled using precedence constraints.

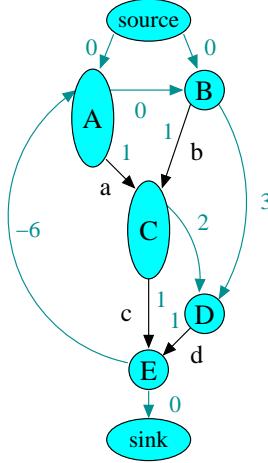


Figure 2.2: Example of a data flow graph.

Operations can have multiple inputs and outputs. The target architecture is assumed to provide functional units and storage files with multiple input and output ports accordingly, to map those operations.

2.3.2 Data edges and values

A data edge $(P^u, C^u) \in E_d(DFG)$ represents a *value* u being produced by operation P^u and consumed by operation C^u . Storage constraint analysis and allocation is performed for *values* and not for *variables*.

A value is produced only once and may be consumed several times, it has an unique lifetime and it is associated with one storage unit.

A variable is also associated with one storage unit. However, it can consist of one or more values (multiple assignments) with non-conflicting lifetimes. Therefore, for applications containing variables a value renaming [38] is assumed to be performed prior any process.

Considering values in our approach helps to reduce the complexity of the construction and coloring of the graphs used for storage allocation. Storing variables in the same unit requires a careful analysis of the values of each variable. Besides, particularly for storage like stacks or fifos the use of multiply assigned variables increases dramatically the complexity of the compiler as well as of the storage controller in the architecture.

Values in loops

In case of program loops values assigned to a storage file are considered *loop-variant*, which means that one instance of a loop-variant value is generated in every loop iteration.

Loop-invariant values (which are already generated outside the loop) are not considered in the storage satisfaction approach. Those values are assigned to storage units prior analysis, and the number of available storage units is therefore reduced.

Although it is possible to assign loop-invariant and loop-variant values to the same stack unit, this case is not considered for reasons of simplicity.

Conventions

- A value u is produced by operation P^u and consumed by C^u . C^u represents one of the consumer operations of u .
- Prod^u is a dummy operation representing the *production* of value u , while Cons^u is the *consumption* of value u . Prod^u and Cons^u are not included in the *DFG* but are used throughout this thesis to represent the instants when a value u is produced and consumed respectively.
- Considering folded cases, u_i is the instance of value u in the i^{th} iteration, and it is produced by operation P_i^u and consumed by C_i^u .
- For a storage file SF , $\Upsilon(SF)$ denotes the set of values assigned to SF .

2.3.3 Distances between operations and timing constraints

The task of scheduling is to assign to each operation $Op \in V(DFG)$ a start time $s(Op)$. Start times are constrained by the precedences. A precedence edge $(Op_1, Op_2) \in E(DFG)$ states that

$$s(Op_2) \geq s(Op_1) + w(Op_1, Op_2) \quad (2.1)$$

The interaction between several precedence constraints becomes clear by combining these precedence edges into a *path*.

A *path* of length d , from operation Op_1 to operation Op_2 , is a chain of precedences $Op_1 \rightarrow \dots \rightarrow Op_2$ that implies $s(Op_2) \geq s(Op_1) + d$.

Definition 2.2 *The distance $d(Op_1, Op_2)$ is the length of the longest path from operation Op_1 to Op_2 .*

A path in the graph thus represents a minimum timing delay. The usual way of administrating these delays is with the *distance matrix* [50], described in Section 2.4.2, that stores the length of the longest path between every pair of operations.

Some timing constraints can directly be expressed at the level of the *DFG* or the distance matrix. Precedence constraints are an obvious example. Other timing constraints that can be expressed, are the latency and the initiation interval.

Latency

The number of available clock cycles in which a data flow has to be scheduled is defined by the *latency* L . This constraint is modeled and expressed in the *DFG* with a precedence edge from the sink to the source vertices with weight $-L$ [53, 50], as illustrated in Figure 2.3a. According to Inequality 2.1, this is interpreted as $s(\text{source}) \geq s(\text{sink}) - L$, which is equivalent to $s(\text{sink}) \leq s(\text{source}) + L$. Because the source is always scheduled in clock cycle zero this formula expresses that the sink should be scheduled in clock cycle L or earlier. Furthermore, because all other operations precede the sink implies that all operations have to finish their execution within the first L clock cycles.

Initiation Interval

For architectures with high levels of parallelism *loop pipelining* or *folded schedules* are intended to achieve performance benefits by exploiting the available ILP. Unlike schedules wherein one iteration of a loop is executed strictly after the execution of the previous one, pipelined schedules consist of overlapping loop iterations with the aim of obtaining potentially much more efficient schedules [40, 29]. The same code called *loop body* or *kernel* is executed in every loop iteration. Iterations of a loop body are periodically initiated in a period called the *initiation interval* (II) without having to wait for preceding iterations to complete.

In pipelined schedules consecutive instances of each value are generated every II clock cycles. If values would be assigned to e.g. random-access registers it has to be ensured that any value instance has to be consumed before another instance of the same value is produced in the next iteration. This means that a value (instance) cannot live longer than II clock cycles.

Therefore, this constraint is modeled and represented in the *DFG* through a precedence edge with weight $-II$ from the consumer to the producer operation of a value [53, 50], as illustrated in Figure 2.3b.

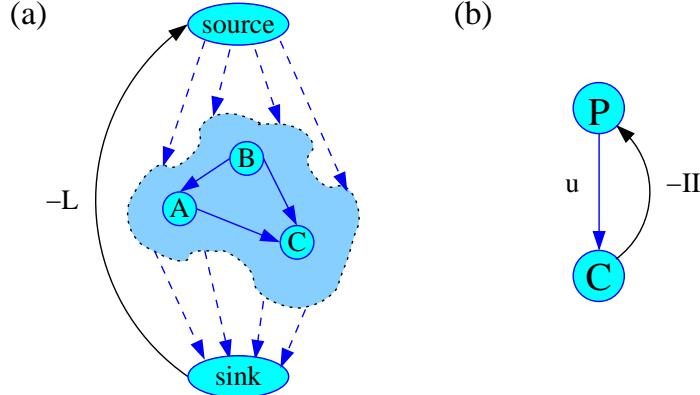


Figure 2.3: Modeling timing constraints in the data flow graph. (a) The latency L , and (b) the initiation interval II .

Distances between operations in loop iterations

Distances between operations from different loop iterations are calculated using the actual distances (considering operations in the same iteration or iteration zero), the initiation interval, and the difference between the respective iteration values.

The distance $d(A_i, B_j)$ from operation A in loop iteration i to operation B in iteration j is deduced as follows:

$$d(A_i, B_j) = d(A, B) - (i - j) \times II \quad (2.2)$$

2.3.4 Resource and storage constraints

The same way as with timing constraints, a schedule also has to satisfy resource constraints. Resource conflicts among operations are modeled by introducing the concept of *resource type* (functional unit) and by defining the *resource usage* of each operation. For this purpose, an *operation type* is associated with each operation. Additionally, each operation type is associated with a unique resource type, which is characterized by a *delay* value, a *data introduction interval* to support pipelined resources, and the number of instances available of that resource type.

The delay value in this model is the number of clock cycles (integer) that the functional unit takes to accomplish its task and to write the result back to a storage unit if required. This is directly associated with the operation execution delay. The data introduction interval is the number of clock cycles that the functional unit takes to be ready for reuse or it is able to read a new set of input operands.

The concept of resource constraints can also be used to model other kinds of

constraints like those arising from instruction sets [81]. Consider for example the case that no instruction exists for the parallel execution of operations Op_1 and Op_2 , so the parallel execution of Op_1 and Op_2 should be prohibited. This can be modeled by generating an *artificial* resource [75] with only one instance that will be “used” whenever operations Op_1 or Op_2 are scheduled. With this addition, constraint analysis techniques (described in Section 2.5) will point out that Op_1 and Op_2 can not be scheduled in the same clock cycles since there is only one artificial resource to be used by Op_1 or Op_2 each time.

A more general constraint arises from the availability of a limited set of *issue slots* [70, 30] to control the data-path of a processor. References [16] and [11] describe a method to completely replace issue slot constraints by artificial resources.

Storage constraints are presented using *memory types* which represent storage files. Each memory type is characterized in terms of its *access* type (random-access register, rotating register, stack or fifo), the number of (available) storage units, and the number of registers per unit in the case of stacks and fifos. No time information or number of read-write ports are included in the memory type.

2.4 Representation of the Schedule Search Space

Our approach uses the *distance matrix* [50] as a representation of the schedule search space.

This is not an obvious choice since in the context of constraint satisfaction other representations of the schedule search space are much more common. However, those representations focus on each individual operation. That is, for each operation either a set [58] or an interval [74] is kept containing the *absolute* clock cycles in which the corresponding operation can be scheduled. One example is the *as soon as possible - as late as possible* interval ([ASAP;ALAP]).

2.4.1 Apparent schedule freedom

As described in the work of Mesman [50], the ranges of possible start times of the operations or *solution space* are approximated to the ASAP-ALAP scheduling intervals of operations, the construction which is solely based on the precedence constraints.

Sequence constraints generated during constraint satisfaction are explicitly added to the data flow yielding a reduction of the ASAP-ALAP intervals. In this way an increasingly more accurate estimate of the set of feasible start times is obtained.

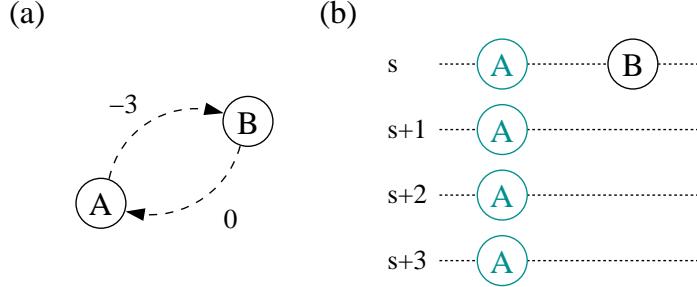


Figure 2.4: (a) In the distance matrix: $d_m(A, B) = -3$ and $d_m(B, A) = 0$. (b) These distances determine the minimum and maximum difference between start times of operations A and B .

In order to measure the effect of the additional precedence constraints on the schedule freedom of operations, the “apparent freedom” or *mobility* of operations is defined. The mobility in a data flow graph is the average difference between the ALAP and the ASAP start times of operations, i.e.

$$\text{mobility}(DFG) = \frac{\sum_{Op \in V(DFG)} (\text{ALAP}(Op) - \text{ASAP}(Op))}{|V(DFG)|} \quad (2.3)$$

2.4.2 Distance matrix

The distance matrix is a bookkeeping method for the minimum and maximum differences between the start times of each pair of operations. A distance in the matrix from operation Op_1 to operation Op_2 is represented by $d_m(Op_1, Op_2)$.

See for example Figure 2.4a, distances found in the matrix between operations A and B indicate that the minimum difference between start times of operations A and B is zero clock cycles while the maximum difference is three. According to this information, A would be scheduled in the same clock cycle as B or up to three clock cycles later (Figure 2.4b).

The results of the constraint analysis techniques [50] are conceptually expressed as additional sequence constraints in the data flow graph. The effects of these additional constraints on the schedule search space are computed by updating the distance matrix: the induced *longest paths* may increment the minimum distance or decrement the maximum distance between two operations.

The distance matrix is calculated using an all-pairs longest-path algorithm [17]. The feature of being able to combine analysis results simply by comput-

ing the longest paths between each pair of operations is one of the motivations for choosing the distance matrix to represent the schedule search space.

The processes related to the distance matrix that take place at the core level of our approach are summarized as follows:

- *Timing constraints* are expressed directly in the distance matrix.
- *Analysis results* are integrated into the distance matrix by means of additional precedence constraints.
- *Precedence constraints*, including the ones resulting from the storage constraint satisfaction process, are combined in the distance matrix such that all implied precedence constraints are also derived.

The distance matrix fits the purposes of this work because of the following reasons:

- The distance matrix administers *relative timing* (order). Practically all code generation constraints have an implication on the *ordering* of operations rather than on their absolute start times. For example, reducing register requirement can be achieved by serializing value lifetimes. Obviously, this is an ordering issue.
- The distance matrix implies an interval representation. The example in Figure 2.5 shows that the reverse is not true: the information in the distance matrix cannot be accurately expressed in terms of intervals. Any interval can be represented in terms of sequence edges expressed directly in the distance matrix. For an interval $[lb;ub]$ of operation A this requires two sequence edges: source $\rightarrow A$ with weight lb and $A \rightarrow$ source with weight $-ub$, which imply $d(\text{source}, A) \geq lb$ and $d(A, \text{source}) \geq -ub$ respectively.

An obvious drawback of maintaining a distance matrix during constraint satisfaction processes is that it is computationally more expensive than an interval representation. However, experimental evidence provided in [77] shows that this overhead is quite acceptable in practice even for large data flow graphs containing hundreds of operations.

Note that adding an extra sequence edge does not mostly require a recalculation of the entire distance matrix but rather an incremental update.

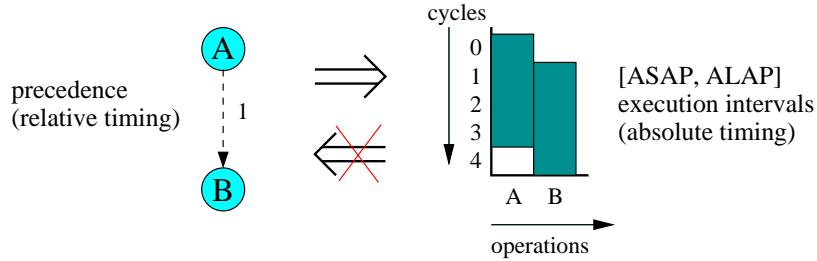


Figure 2.5: The interval (or set) representation does not accurately represent relative timing or precedence between operations.

Distances between productions and consumptions of values

Distances between productions and consumptions of values u and v are calculated from distances in the matrix and execution delays of operations as follows:

$$d(\text{Prod}^u, \text{Prod}^v) = d_m(P^u, P^v) + \delta(P^v) - \delta(P^u) \quad (2.4)$$

$$d(\text{Prod}^u, \text{Cons}^v) = d_m(P^u, C^v) - \delta(P^u) + 1 \quad (2.5)$$

$$d(\text{Cons}^u, \text{Prod}^v) = d_m(C^u, P^v) + \delta(P^v) - 1 \quad (2.6)$$

$$d(\text{Cons}^u, \text{Cons}^v) = d_m(C^u, C^v) \quad (2.7)$$

2.5 Basic Constraint Analysis Techniques

The basic constraint analysis techniques essentially consist of *rules* triggered by the combination of one or more pairs of distances (entries in the distance matrix) and other (often architectural) constraints.

The essence of constraint analysis is that additional sequence constraints are *necessarily implied by the combination of other constraints*. In this way, the schedule search space is pruned to prevent the scheduler from finding infeasible solutions without eliminating feasible solutions.

2.5.1 Execution interval analysis

Execution interval analysis is implemented to analyze resource constraints, and is based on examining the intervals in which operations can be executed [74, 73]. The method focuses on the availability of resources: it reduces the execution interval of an operation when it observes that no resource is available for executing that operation in a particular clock cycle. This is illustrated in Figure 2.6.

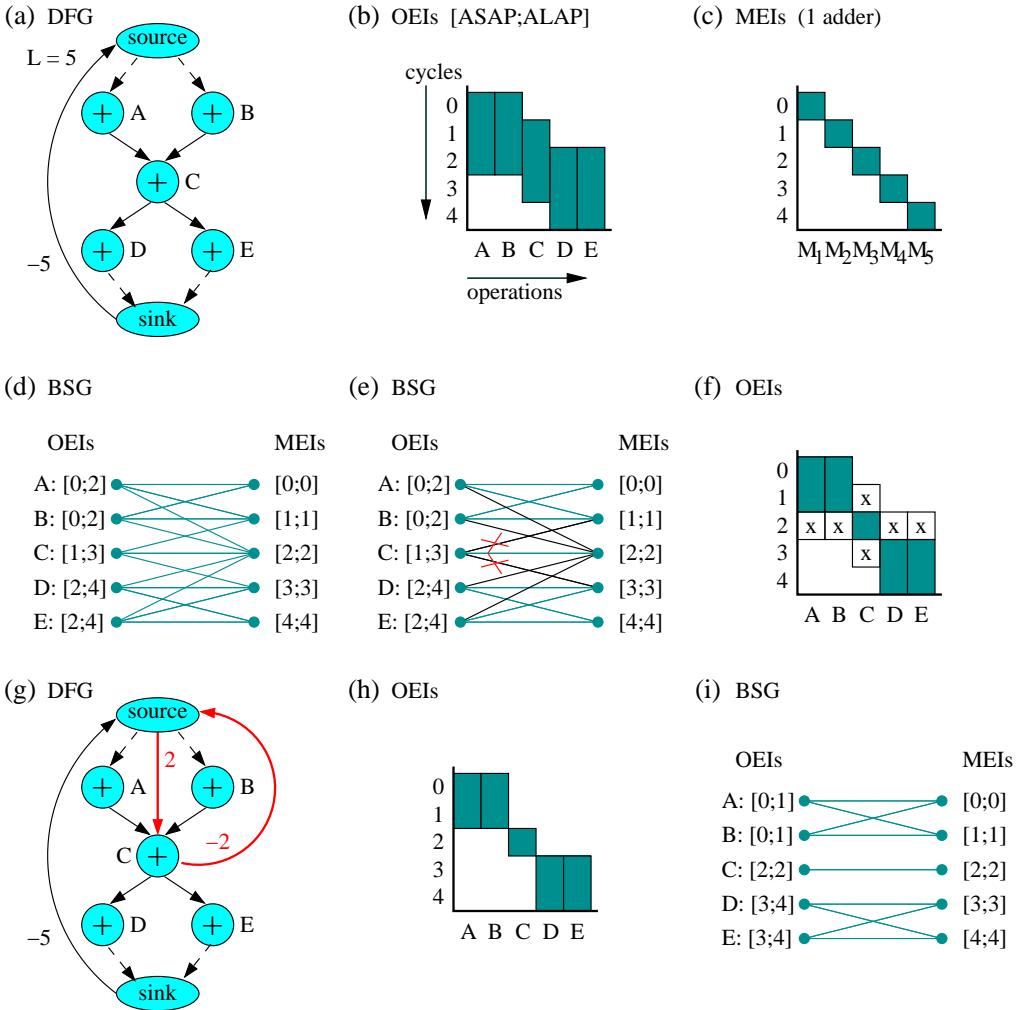


Figure 2.6: Execution interval analysis, (a) a data flow graph *DFG*, (b) initial operation execution intervals *OEIs*, (c) modulo execution intervals *MEIs* of one adder, (d) the corresponding bipartite schedule graph *BSG*, (e) edges in *BSG* that can never be part of a complete matching are eliminated, (f) operation execution intervals are adjusted accordingly, (g) the analysis result is annotated in *DFG*, (h) the pruned *OEIs*, and (i) the updated *BSG*.

In Figure 2.6a a data flow graph is given. The latency is five clock cycles, and one resource of type ‘adder’ is available to execute the operations. The initial *operation execution intervals* (OEIs) are determined by the [ASAP;ALAP] intervals as depicted in Figure 2.6b. The *modulo execution intervals* (MEIs) considering one adder are shown in Figure 2.6c. Each MEI represents the abstract notion that some resource has to execute an operation. In the example, the constraints imply that the available adder has to start executing a new operation every clock cycle.

Execution interval analysis combines the execution intervals of the operations with resource constraints by constructing a *bipartite schedule graph* *BSG* (Figures 2.6d) in the following way: The operations and their corresponding OEIs are shown on the left side. The module execution intervals are shown on the right side. There is an edge between an OEI and a MEI if the intervals overlap, indicating that the corresponding operation can be executed in the designated MEI. In addition, it is required that all preceding operations of the same operation type can be matched with preceding MEIs in the bipartite graph. A similar condition is imposed for all succeeding operations.

The key observation of the analysis is that for every feasible schedule there exists a *complete matching* in the *BSG* between the OEIs and the MEIs. That is, every OEI is matched to exactly one MEI and vice versa. The analysis uses the algorithm of [68] to identify edges in the bipartite graph that can never be part of a complete matching. The reader can verify that the bold crossed edges in Figure 2.6e are such edges. Because these edges can never be part of a matching corresponding to a feasible schedule, they are removed from the bipartite graph. Other (bold) edges are also removed as a secondary effect. As a result, operation *C* cannot execute in MEIs [1;1] and [3;3]. The execution interval of operation *C* is therefore adjusted (Figure 2.6f). This adjustment corresponds to a pruning of the search space. The result of the analysis is annotated in the data flow graph (and the distance matrix) as indicated in Figure 2.6g.

Therefore, the timing and the resource constraints have restricted the execution of operation *C* to interval [2;2] as well as the execution intervals of the other operations (Figure 2.6h). The updated *BSG* is shown in Figure 2.6i.

Execution interval analysis contains an additional process: to determine the earliest possible start time of an operation a relaxed scheduling problem involving all its predecessors is solved, which determines a lower bound of the first clock cycle in which all predecessors are completed. The scheduling problem is relaxed in the sense that the resource constraints are essentially ignored; it is only enforced that each operation cannot start earlier than the lower bound of its start time. Similarly, all successors of the operation are analyzed. Operation predecessors and successors are determined using the distance matrix, e.g. an operation *Op*₁ is said to precede an operation *Op*₂ if $d(Op_1, Op_2) \geq 1$.

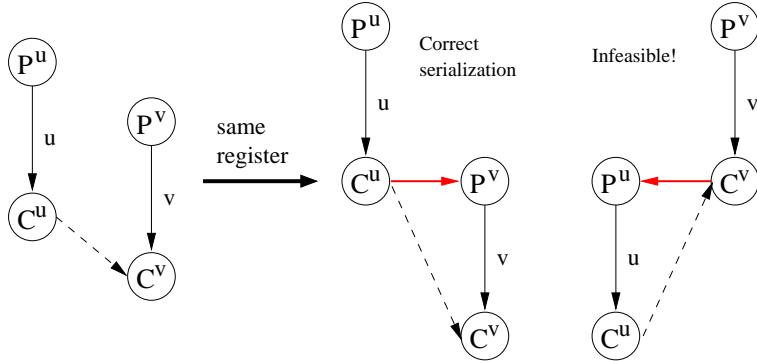


Figure 2.7: To solve the register conflict, since C^u already precedes C^v the feasible solution is that C^u has to precede P^v .

2.5.2 Storage constraint analysis

A relatively simple constraint analysis arises from the decision to assign two values to the same register. This decision can be made by the designer, or by a search strategy for register allocation (Chapter 4), or like in the case of conditional execution of operations where there is only one guard register (predicate) available to store a number of conditional values [82].

Whenever two values are assigned to the same register their corresponding lifetimes are forced to be serialized. In general, this can be done in two ways: value u precedes v or value v precedes u . Sometimes there already exists a precedence between the accesses of u and v that excludes one of these possibilities. One such situation is illustrated in Figure 2.7. Because already exists the precedence $C^u \rightarrow C^v$ the sequence edge $C^u \rightarrow P^v$ is the constraint necessary to solve the register conflict between u and v . Such situations are recognized by simple rules [52] that examine the distance matrix for precedences.

2.5.3 Symmetry detection

Many signal processing algorithms exhibit certain symmetries because their structure is completely or partly regular. Practical examples include algorithms for performing a fast Fourier transform or a discrete cosine transform. The symmetries have a negative impact on the accuracy of constraint analysis techniques. These techniques are not capable of “breaking” the symmetries in order to decrease the apparent scheduling freedom, because doing so would remove feasible schedules. Therefore, van Eijk in [76] developed additional techniques for detecting and utilizing symmetry while preserving feasibility.

In general, the concept of symmetry is strongly related to the property of an object that does not change under a certain transformation. The kind of transformation that is relevant for capturing symmetry in a data flow graph is a relabeling of the operations such that the operation types and the precedence edges are preserved. Such a transformation is called an *automorphism*.

Let T_O denote the set of operation types. Then the function $\tau : V \rightarrow T_O$ defines the operation type of each operation. An automorphism is a bijective function $\alpha : V \rightarrow V$ such that:

- Each operation is mapped to an operation of the same type:
 $\forall Op \in V(DFG) : \tau(Op) = \tau(\alpha(Op)).$
- Each edge is mapped to an edge having the same weight:
 $\forall (Op_1, Op_2) \in E(DFG) :$
 $(\alpha(Op_1), \alpha(Op_2)) \in E(DFG) \wedge w(Op_1, Op_2) = w(\alpha(Op_1), \alpha(Op_2))$

Given an automorphism, the following data flow transformation describes how an extra sequence edge can be added to break the symmetry.

Given an operation Op_1 and an automorphism α that maps Op_1 to another operation Op_2 , i.e. $\alpha(Op_1) = Op_2$ with $Op_1 \neq Op_2$. Introduce a sequence edge in DFG from Op_1 to Op_2 with weight zero.

To illustrate this transformation, consider the data flow graph of Figure 2.6. Because there exists an automorphism that exchanges operations A and B , the transformation allows to introduce a sequence edge with weight zero from A to B . Similarly, it is allowed to put a sequence edge from D to E . In combination with the execution interval analysis, this transformation associates an unique start time with each operation.

For the proof that the transformation preserves feasibility the interested reader is referred to [76].

2.5.4 Infeasibility Analysis

Ordering of operations can result from different processes such as the storage file capacity satisfaction approach, symmetry detection, or scheduling, and is performed through the addition of sequence edges.

The ordering of operations has to be validated. This includes the issue of whether or not the updated information in the distance matrix still represents a feasible scheduling search space.

Constraint analysis detects infeasibility based on the distance matrix in the following way [50]: when a path is found from operation Op to itself (a cycle

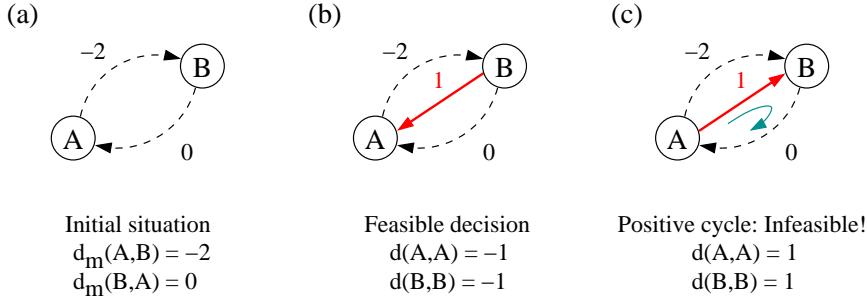


Figure 2.8: Example of infeasibility caused by the ordering of two operations through sequence edges. (a) Initial situation, dashed edges represent the information from the distance matrix. (b) Ordering $B \rightarrow A$ with weight one. No positive cycle is created. (c) Ordering $A \rightarrow B$ with weight one. Cycles with positive weight are created resulting in an infeasible situation.

in the data flow graph) and this path has a positive weight, the operation Op is forced to execute strictly before its own start time which is clearly not possible. Therefore, a precedence cycle of strictly positive weight indicates infeasibility.

To illustrate this process, Figure 2.8 shows a simple situation which consists of the ordering of two operations A and B using a sequence edge with weight one. With the information from the distance matrix (dashed edges in the figure), it is possible to determine the feasibility of each decision made. The first ordering $B \rightarrow A$ (with sequence edge (B, A) with $w(B, A) = 1$) is feasible since no positive cycle path is created between A and B . On the other hand, ordering $A \rightarrow B$ (with sequence edge (A, B) with $w(A, B) = 1$) creates positive cycle paths meaning that each operation must be executed before its own start time, which is infeasible.

During the satisfaction process the positive weight cycles are detected while performing the all-pairs longest-path algorithm to update the distance matrix.

2.6 Conflict Graphs and Coloring

Conflict graphs are intensively used in this work for storage constraint satisfaction and allocation. In this section the basic definitions of graphs, coloring, and an introduction of conflict graphs are presented.

2.6.1 Basic definitions

Definition 2.3 Let $G = (V(G), E(G))$ be an undirected graph, where:

- $V(G)$ is the set of vertices, and
- $E(G) \subseteq V \times V$ is the set of edges.

$N(u)$ is the set of neighbors of a vertex u in graph G , i.e.
 $N(u) = \{v \in V(G) | (u, v) \in E(G)\}$.

The *degree* number $\deg(u)$ of a vertex u is the number of its neighbors or the number of edges incident to it, i.e. $\deg(u) = |N(u)|$.

A *sub-graph* of a graph G is a graph of which vertex and edge sets are contained in the vertex and edge sets, respectively of G .

A *clique* C is a subset of vertices ($C \subseteq V(G)$) that induces a sub-graph of G in which those vertices are completely connected to each other by edges. A *clique cover* of size k is a partition of the vertices such each partition is a clique. The *clique cover number* $k(G)$ is the size of the smallest clique cover of G .

A clique is called the *maximum clique* if it is a clique with the vertex set of largest cardinality. The *clique number* $\gamma(G)$ is the number of vertices of the maximum clique of G .

A *stable set* is a subset of vertices in which no two vertices are connected by an edge. This is also called an *independent set*. The *stability number* $\alpha(G)$ is the number of vertices in a stable set of maximum cardinality.

Vertex coloring of a graph consists of assigning a color to every vertex so that no two vertices connected by an edge have the same color. $\text{color}(u)$ is the color assigned to vertex u . The *chromatic number* $\chi(G)$ is the smallest possible number of colors for coloring G . *Exact coloring* is a coloring that uses $\chi(G)$ colors.

Then, it is easy to see that

$$\gamma(G) \leq \chi(G) \quad \text{and} \quad \alpha(G) \leq k(G)$$

A graph G is said to be 1-perfect if

$$\gamma(G) = \chi(G) \quad \text{and} \quad \alpha(G) = k(G)$$

Definition 2.4 The *saturation number* $\text{sat}(u)$ of a vertex $u \in V(G)$ in a colored graph G is the number of colors used by its neighbors, i.e.
 $\text{sat}(u) = |\{\text{color}(v) | v \in N(u)\}|$

2.6.2 Conflict graph

A well-established approach to model the register allocation problem in many optimizing compilers is as a graph coloring problem [15]. Compared with other register allocation methods, graph coloring provides a relatively simple, conceptually elegant, formulation of the problem of assigning values to a register set. Graph

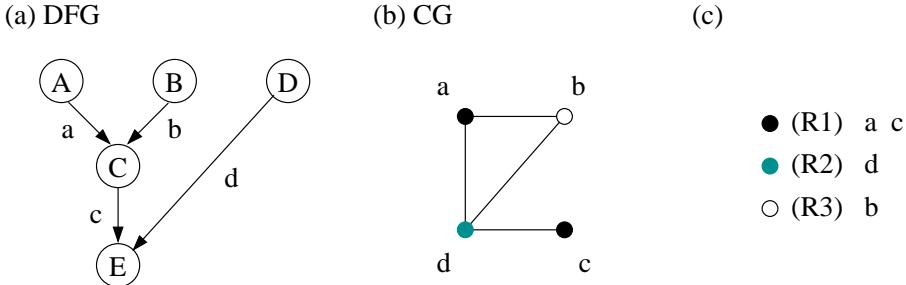


Figure 2.9: Graph coloring for register allocation. (a) *DFG*, (b) a conflict graph with vertices representing values, (c) each color used represents a register in which values are stored.

coloring combines allocation and assignment by allocating values to vertices of a conflict graph and places edges between vertices where the corresponding values are alive simultaneously. The solution then involves finding a k coloring of the graph, where k represents the number of the target machine's available registers (see Figure 2.9).

In a colored graph the color of each vertex differs from the color of each of its neighbors. As an architectural paradigm, this implies that each value is assigned a register different from all other values alive during the same execution cycles.

Definition 2.5 A *conflict graph* $CG(SF) = (V(CG), E(CG))$, for values assigned to storage file SF , is an undirected graph, where:

- $V(CG)$ is the set of vertices. A vertex $u^c \in V(CG)$ represents a value $u \in \Upsilon(SF)$.
- $E(CG) \subseteq V \times V$ is the set of edges. An edge $(u^c, v^c) \in E(CG)$, with $u^c \neq v^c$, exists and represents a conflict if the access mode of values u and v does not match with the access behavior of SF .

The access conflict between two values is obtained by applying the access rules of the corresponding storage file to the relative distances (in the distance matrix) between producer and consumer operations. See the example of Figure 2.10, values u and v will have a conflict if they are assigned to: a random-access register (Figure 2.10a), a stack (Figure 2.10b) or a fifo (Figure 2.10c). The (relative) distances are represented by dashed edges.

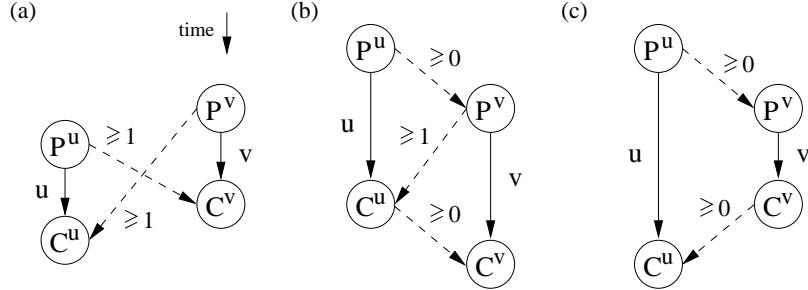


Figure 2.10: Examples of conflicts according to the storage access behavior. Values u and v have a conflict if assigned to (a) registers, (b) stacks, and (c) fifos.

2.7 Description of the FACTS Research Tool

FACTS is a research tool developed to take advantage of timing and resource constraints in order to prune the schedule search space. By exploiting these constraints the scheduler is often prevented from making a decision that violates one or more constraints. The structure of FACTS consists of three layers [77], as depicted in Figure 2.11.

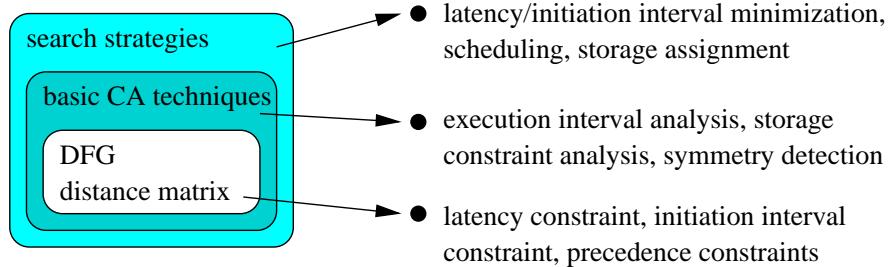


Figure 2.11: The layered structure of FACTS.

The core layer contains the internal representations of the algorithm to schedule and of the schedule search space. Each basic block of the algorithm is represented by an edge-weighted data flow graph. FACTS uses the distance matrix, explained in Section 2.4.2, as a representation of the schedule search space. The results of the constraint analysis techniques in FACTS are expressed as additional precedence constraints in the data flow graph, and updates of the distance matrix.

At the intermediate layer of FACTS, the basic constraint analysis techniques are located like the execution interval analysis, the storage constraint analysis, and the symmetry detection.

At the top level of FACTS search strategies are found. The objective may be to minimize some criterion like latency or initiation interval, or to satisfy “global” constraints like a fixed capacity of a storage file. These constraints are global in the sense that they have a general effect on all timing relations, without affecting any specific timing relation. For example, a constraint on the capacity of a register file limits the amount of values simultaneously alive, which is determined by the timing relations. However, no single value lifetime can be identified that necessarily has to be serialized with some other value lifetime. It is clear that some value lifetimes need to be serialized in order to satisfy the capacity constraints and therefore some choices have to be made regarding the serialization of values. In the search strategies, it is tried to base choices on the identification of bottlenecks for satisfying the corresponding constraints.

In order to expand FACTS to become a full compiler the following items would have to be addressed:

- Support for a high-level description language as C.
- Definition of a machine description language. A machine description would be an input for compiler reconfigurability.
- Include instruction set constraints and code selection.
- Include techniques for global constraint analysis and scheduling in scopes beyond basic blocks, e.g. traces, super-blocks, decision trees, or regions [33].

Given the set of basic definitions, analysis, and tools, next chapter presents the problem definition and the proposed strategy towards obtaining a scheduling solution.

Chapter 3

Storage Constraint Satisfaction

3.1 Introduction

This chapter describes the problem and the proposed approach for scheduling and storage allocation with storage constraint satisfaction.

Our approach is performed in the scope of basic blocks (see the center of Figure 3.1). In a scope beyond basic blocks, like in traces, super-blocks, decision trees, or regions, other techniques like code motion and if-conversion can be applied. Hoogerbrugge in [33] presents an overview of these scopes and the basic techniques applied in each case. Code selection and storage file allocation are assumed to be performed prior our approach. Figure 3.1 also suggests that trade-offs can be made in an extended basic block scope between timing, resource, or storage constraints and the insertion of spill code.

The important characteristics of our approach are: it deals with tight timing, resource, and storage constraints, and the approach proposed is independent of the type of storage.

In the core of our approach, conflict graphs representing potential access conflicts among values, are constructed based on the information extracted from the distance matrix after constraint analysis. Those graphs are used to steer the satisfiability process: to check the feasibility of the schedule, and to identify bottlenecks for storage allocation. Those bottlenecks are treated via ordering of value accesses to allow sharing of storage resources and to prune the scheduling search space towards a solution that will satisfy all constraints.

3.2 Motivation

Consider the data flow graph from Figure 3.2a with six operations and five values (data edges). The latency for the completion of this application is four. All

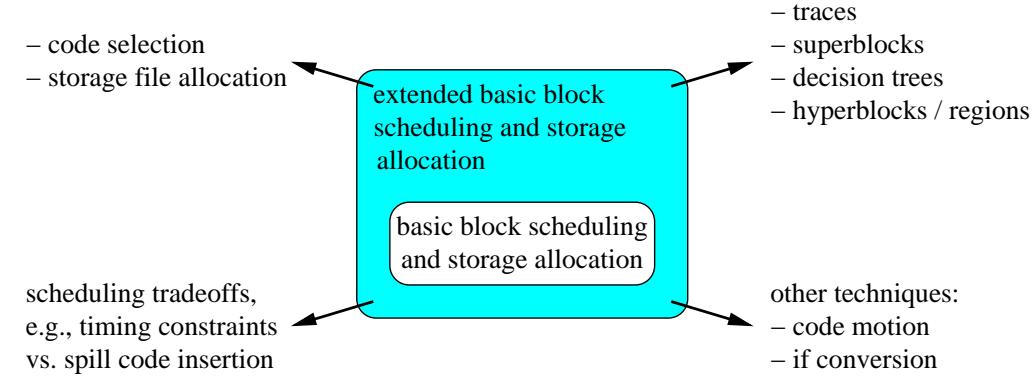


Figure 3.1: Scheduling scope.

values are assigned to one register file with two registers available. It is assumed that the number of functional resources is large enough and does not constitute a constraint.

Figures 3.2b, 3.2c and 3.2d show the different conflict graphs obtained from different scheduling results, i.e. the different clock cycles in which operation E can be scheduled (the rest of operations are already fixed in time). As the use of random-access registers for storing values is assumed, there exists a conflict between two values if their lifetimes overlap.

If operation E is scheduled in clock cycle zero the storage constraint is violated since the number of registers required will be three (for values a , b and e) instead of the available two. Coloring the conflict graph of Figure 3.2b results in three colors, which confirms the constraint violation. Schedule operation E in clock cycles one or two would satisfy the register file constraint. If during scheduling it is possible to prevent operation E from being scheduled in clock cycle zero, the storage constraint would be met.

Since dealing with register allocation and a not yet defined schedule, a way to solve this satisfiability problem would be identifying the bottlenecks for register allocation and try to serialize value lifetimes to allow values to share the same register.

In the example, before operation E is scheduled pairs of values $\{a, e\}$, $\{b, e\}$ can be selected for lifetime serialization, since values a , b and e would require three registers in the worst case. Serialization of pair $\{a, e\}$ results in forcing the scheduling of E (the producer of e) to occur at least at the same clock cycle as C (the consumer of a). This decision is represented by $C \rightarrow E$. Serialization of pair $\{b, e\}$ gives the same result. The resulting conflict graph after serialization $a - e$ or $b - e$ with E scheduled in clock cycle one, is shown in Figure 3.2c. In the graph,

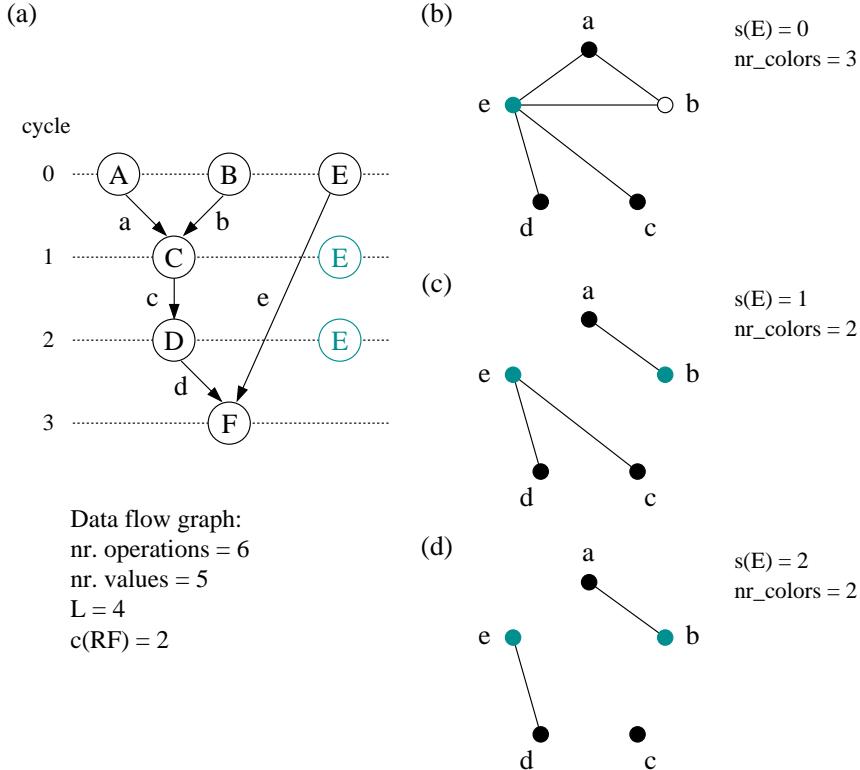


Figure 3.2: Example of a *DFG* and its conflict graphs for each clock cycle in which operation E can be scheduled.

edges (conflicts) (a, e) and (b, e) were eliminated and the new number of colors is two, i.e. two registers are now required, which meets the storage constraint. Another possible serialization is $c - e$, which results in forcing the scheduling of E at the same clock cycle as D (ordering $D \rightarrow E$), i.e. at clock cycle two. The resulting conflict graph is shown in Figure 3.2d. As a consequence, conflicts between values a, b, c with e are eliminated. Any of those orderings applied guarantees that the final schedule will satisfy the storage constraint.

Working with constraint analysis techniques the information of the potential conflicts between values can be translated into a *worst-case* conflict graph for storage allocation, when the scheduling of operations is not defined. With this conflict graph, *bottlenecks* for storage allocation can be identified.

The graph in Figure 3.2b would represent the worst-case conflict graph for the current example. In this graph, bottlenecks can be either edges (a, e) , (b, e) , or (c, e) because those conflicts can be eliminated with a (convenient) value lifetime serialization.

Once a bottleneck has been found, *access ordering* (e.g. lifetime serialization) of the corresponding values is performed by inserting one or more sequence edges among their producer and consumer operations. As a consequence, the extra precedence constraints (sequence edges) will steer the scheduling to find a solution that satisfies all constraints including the storage availability.

Traditional methods deal with scheduling and storage allocation as two separated and independent stages [66, 78]. Although faster, those methods can introduce the problem of *phase coupling* between scheduling and storage allocation, i.e. decisions made in one of the stages can result in an infeasible set of constraints for the other. If scheduling is performed first, the number of required storage units may exceed the amount available in a storage file. On the other hand, if storage allocation is done before scheduling, timing and resource constraint violations may arise.

Spilling values to background memory like performed in [14, 66] could be one solution for the problem of storage constraint satisfaction after scheduling. Some values are selected to be stored in memory, load and store operations are inserted in the code, and operations are re-scheduled. However, the use of additional operations may cause timing constraint violations. Another way to deal with constrained scheduling and storage allocation is simply by relaxing the timing constraints like it is done for the initiation interval discussed in [47].

The proposed approach combines storage allocation and scheduling avoiding the problem of phase coupling and the difficulties of inserting spill code without relaxing constraints. The approach consists in alternating between scheduling and storage allocation sub-problems by making a decision for storage allocation and subsequently analyzing how that prunes the search space for scheduling.

One key advantage of our approach is that it can be easily tuned to work with different types of storage units and access behaviors.

3.3 Problem Statement and Solution Approach

In this section, the storage allocation and scheduling problem is defined and the proposed solution approach is presented. A given assignment of values to storage files is assumed often implied by the assignment of operations to functional units.

Problem Definition 3.1 *Constrained Storage Allocation and Operation Scheduling Problem.* Given a weighted data flow graph DFG , resource constraints, an assignment of values to storage files, for each storage file SF a number of available storage units or capacity $c(SF)$ with a specific access behavior, an initiation interval II , and a latency L . Find an assignment of values to storage units and a schedule s that satisfies the precedence constraints, the resource constraints, the storage file constraints, and the timing constraints II and L .

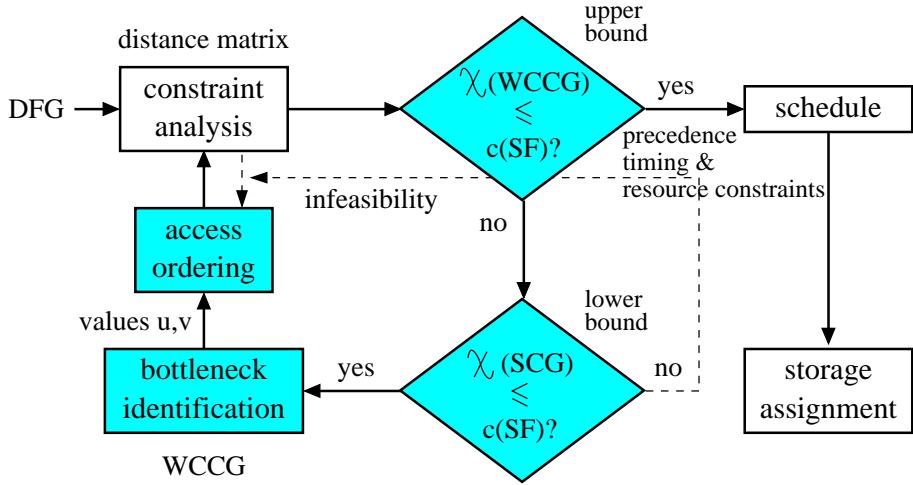


Figure 3.3: Approach for storage constraint satisfaction considering one storage file SF .

The specific storage units considered in this work are: *random-access registers* (the approach for which is presented in Chapter 4), *relative location storage* or *rotating register files* (also in Chapter 4), *stacks* (in Chapter 5), and *fifos* (also in Chapter 5).

Because decisions have to be made that affect the feasible search space in both the domain of storage allocation and the domain of scheduling, the approach is decomposed into steps as depicted in Figure 3.3.

Constraint analysis (explained in Section 2.5) generates additional precedence constraints that are implied by the combination of all constraints including the ones for storage allocation. These additional precedences refine the *distance matrix* (Section 2.4.2) thus providing a much more accurate estimate of the set of feasible start times. The constraint analysis will guide the decisions made in the scheduler and often prevent it from making decisions leading to infeasibility.

After constraint analysis, a storage file SF is selected in the case of distributed storage files (refer to Section 3.4) and the values bound to it are used in the following analysis.

For the selected file SF the worst case or upper bound of required storage units is computed based on the administrative bookkeeping by constraint analysis. Note that an exact figure is unknown because a complete schedule is not yet determined. The worst case situation is found by means of using a *worst-case conflict graph* $WCCG(SF)$ and its exact coloring $\chi(WCCG)$ (Section 3.5.1). When the worst case for the current and all storage files already satisfies their capacity $c(SF)$, the *DFG* and the precedences are transferred to conventional *schedule* and *storage*

assignment phases to complete the process. However, in most cases and especially at the beginning of the process the schedule freedom of the operations can be relatively large, resulting in many potential conflicts hence inevitably violating some storage file capacity constraint.

After that, a lower bound of the number of storage units is determined through the coloring of a *best-case* or *strong conflict graph* $SCG(SF)$ (Section 3.5.1). $\chi(SCG)$ is compared with the respective file's capacity. A lower bound violation determines an infeasible case, if it is detected at the beginning of the process a solution cannot be found.

In order to reduce the storage file requirement, our approach in Figure 3.3 has to reduce the maximum number of conflicts by identifying one or more pairs of values that can potentially be stored in the same unit (*bottleneck identification*, Section 3.6) and then proceed to order their accesses (*value access ordering*, Section 3.7) according to the targeted SF access behavior. After access ordering, constraint analysis calculates the effect of this on the schedule freedom of all operations. This is necessary to guide the access ordering part and to avoid making decisions that lead to infeasible cases. Also, a lower bound violation found with a new SCG indicates that the latest access ordering is infeasible and another ordering with the previous or a different pair of values has to be made.

The file selection, the upper and lower bounds checking, the bottleneck identification, the value access ordering, and constraint analysis keep on alternating until the capacity constraint of each storage file matches the worst case requirements or until a unrecoverable infeasible case is detected.

In fact, just at the beginning of the satisfaction process an infeasible case can be detected, which means that the constraints are too tight and no schedule solution can be found with such requirements. During the satisfaction process an infeasible case may arise as a result of the last access ordering. Infeasible cases can be detected by constraint analysis or by the lower bound checking. In such cases the process backtracks (as indicated by the dashed edges in Figure 3.3) and makes another ordering decision (Section 3.8).

An advantage of our approach is that in practice any conventional scheduling algorithm can be used to complete the schedule. As the scheduler and its heuristics are not critical in this approach, they are not considered in this work.

3.4 Storage File Selection

For distributed storage files the order in which storage files are analyzed is important. After ordering and constraint analysis, the sequence edges added may affect the whole distance matrix making constraints tighter, and consequently affect the result for the next storage file analysis.

A previous approach [3] considers a heuristic in which the selection of register files RF is based on the apparent register pressure, a priority defined as:

$$\text{priority}(RF_n) = \frac{|\Upsilon(RF_n)|}{c(RF_n)} \quad (3.1)$$

where $\Upsilon(RF_n)$ represents the values assigned to the n^{th} register file RF_n , and $c(RF_n)$ is its capacity. The register file selected is the one with the highest priority.

This priority function has three major drawbacks:

- First, the number of values assigned to a storage file says too little about the amount of storage units actually required. I.e. depending on the constraints many values can require fewer registers than few other values.
- Second, this is a static decision. Once a storage file is selected for analysis and capacity satisfaction, the next one will be analyzed only after the process has finished with the first. The process of capacity satisfaction would not be performed gradually for each storage file simultaneously, and decisions made for satisfying one set of storage file constraints at a time can affect negatively the satisfaction process of the others.
- Finally, this criterion is even less clear in the case of using storage files with fifos, stacks or rotating register files, since the number of values assigned to a storage file does not represent the actual storage requirement.

The best solution found so far is to select the storage file for which the unit requirement is the furthest from respecting its capacity, i.e. with priority:

$$\text{priority}(SF_n) = \frac{\chi(WCCG(SF_n))}{c(SF_n)} \quad (3.2)$$

where $WCCG(SF_n)$ represents the worst-case conflict graph for values mapped to storage file SF_n , $\chi(WCCG)$ is the chromatic number of $WCCG$, and $c(SF_n)$ is the capacity of SF_n . Since construction and coloring of $WCCG$ are already part of the satisfaction process the priority function in Expression 3.2 does not add complexity.

Expression 3.2 describes the actual storage pressure in the file. Following this priority function, when a reduction of storage pressure is achieved for one storage file the upper bound requirement is updated for the current file, and then the process can select another one with the highest priority. Thus, capacity satisfaction will be gradually performed for each storage file.

Moreover, note that each storage file may have a different type of storage units. Therefore, this criterion of file selection is independent of the type of storage and directed towards to the satisfaction process itself.

In Section 4.7.1 (Chapter 4 dedicated to register files), experimental results will show the advantages of using Expression 3.2 instead of Expression 3.1.

3.5 Conflict Graphs and Coloring

This section describes how conflict graphs are used for storage satisfaction. The graph coloring model is general enough to be used with different types of storage units, since it takes only into account the access incompatibility (conflict) between values.

3.5.1 Constructing conflict graphs

During the satisfaction process, once the constraint analysis has generated additional precedences and refined the distance matrix, the information contained in the matrix will consist of a set of feasible start times of operations. Using these start times, for each pair of values the relative access conflicts are obtained. Because in this context the producer and consumer operations of values are not fixed yet in time, three different notions of conflict are used.

- Two values have *no access conflict* if the order of their accesses respects the storage access behavior. For example, in the case of registers there is no conflict when the lifetimes of those values are serialized. For stacks, it means that the order of their consumptions is opposite to the order of their productions. For fifos, the order of their consumptions matches the order of their productions.
- Two values have a *strong access conflict* if the order of their accesses definitively does not match the storage access behavior.
- Otherwise, two values have a *weak access conflict*. Depending on the schedule, their accesses may exhibit a conflict or not.

Consider the example of Figure 3.4a without folding ($II = L$). There is no conflict for values c and d since their lifetimes are serialized. There is a strong conflict for values a and b in the clock cycle when operation C executes (both values are consumed at the same time). There is a weak conflict for values b and e . If operation E precedes operation C by one clock cycle lifetimes of b and e overlap (have a strong conflict), otherwise b and e will have no conflict. Since it is not yet determined whether or not E precedes C , b and e have a weak conflict.

The following is the essential difference between a strong and a weak conflict: values with a strong conflict can never reside in the same storage unit, but for

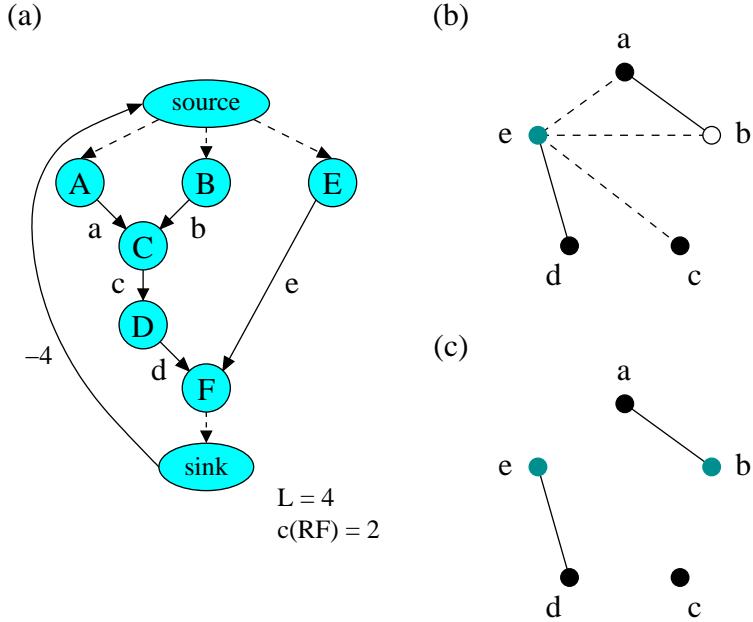


Figure 3.4: Example of (a) a data flow graph DFG , (b) its worst-case conflict graph $WCCG$, and (c) its strong conflict graph SCG , after constraint analysis. Values are assigned to registers.

values with a weak conflict their accesses can still be ordered to match the storage access behavior.

Worst-case conflict graph

A graph $WCCG(SF)$ for values assigned to storage file SF is called the *worst-case conflict graph*. There exists an edge $(u^c, v^c) \in E(WCCG)$ if values u and v have a weak- or a strong access conflict ($u, v \in \Upsilon(SF)$).

Figure 3.4b shows the worst-case conflict graph after constraint analysis of the example in Figure 3.4a. Weak access conflicts are represented by dashed edges, while strong conflicts are represented by solid edges.

As mentioned in the previous section, the exact coloring of $WCCG(SF)$ gives an upper bound of the number of storage units required, and also helps to find and select bottlenecks for storage allocation.

If $\chi(WCCG)$ is greater than the capacity of the file $c(SF)$ the satisfaction process proceeds to identify and reduce bottlenecks. If not, the satisfaction process returns and another file is analyzed or the process performs the final scheduling and storage assignment.

Bottlenecks (for storage allocation) in $WCCG$ are edges that can be removed. Those edges represent weak conflicts and are reduced by ordering the accesses of the corresponding values (vertices in $WCCG$). Access ordering will take edges (conflicts) away from $WCCG$ which reduces the chromatic number and hence, the storage requirement.

Since access ordering can have an effect on the entire distance matrix and subsequently on the access relations between values, a new $WCCG$ is constructed and colored in every iteration of the satisfaction process.

Best-case or strong conflict graph

A graph $SCG(SF)$ for values assigned to storage file SF is called the *best-case* or *strong conflict graph*. There exists an edge $(u^c, v^c) \in E(SCG)$ if values u and v have a strong access conflict ($u, v \in \Upsilon(SF)$).

Coloring of the best-case conflict graph generates a lower bound of the storage requirements. In this graph only the strong access conflict situations are considered. Figure 3.4c shows the strong conflict graph of the example in Figure 3.4a.

The coloring of SCG steers the storage satisfaction process. With this graph it is checked whether the minimum storage requirement of the values assigned to SF is (still) in the range of the available capacity $c(SF)$.

Since access ordering can affect the distance matrix and weak conflicts can become strong ones, a new SCG is constructed and colored in every iteration of the satisfaction process.

3.5.2 Inaccuracies in conflict modeling

Because in the worst-case situation potential access conflicts between values are included, $WCCG$ can be inexact. The modeling of worst-case conflicts and exact coloring of $WCCG$ obtain a pessimistic approximation of the upper bound.

This is especially true in the case of multiply consumed values. In Figure 3.5a a portion of a non-scheduled data flow graph with a latency of four clock cycles is shown. Value a is consumed by operations B and C . If all values are assigned to the same register file Figure 3.5b shows the corresponding $WCCG$ and its coloring. Note that due to the schedule freedom of operations, value a can conflict with either value b or c according to the scheduling of operations B and C . Therefore, there exist weak conflicts between a and b , b and c , and a and c . The resulting number of colors for $WCCG$ is three, however it is easy to see that for any scheduling the number of registers required would be two.

For the best-case situation the coloring of SCG fails being a good lower bound.

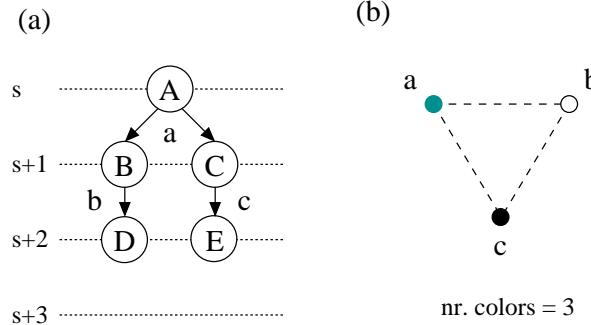


Figure 3.5: $WCCG$ in case of a multiple value consumption, (a) DFG portion, and (b) the corresponding $WCCG$ with $\chi(WCCG) = 3$.

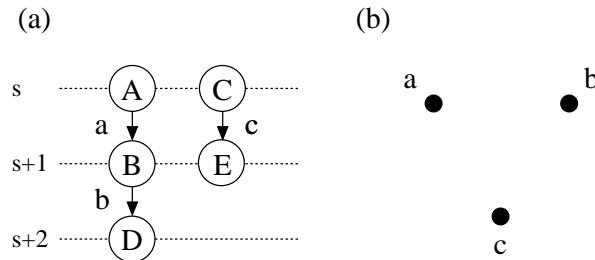


Figure 3.6: SCG does not obtain the tightest lower bound of two, (a) DFG portion, and (b) the corresponding SCG with $\chi(SCG) = 1$.

In Figure 3.6a a different part of a data flow graph is shown. The best-case conflict graph is presented in Figure 3.6b. According to the schedule freedom of the producer and consumer operations of value c , there are no strong conflict situations between this value and values a or b . Value c can conflict with a , b , or both. The tightest lower bound for this example is two and not one as the coloring of SCG suggests. Figure 3.7 shows another example in which the tightest lower bound is not detected.

3.5.3 Conflict graph characteristics

In order to obtain run-time advantages when coloring exactly the worst- and the best-case conflict graphs, it would be useful to find out special properties of these graphs. However, graphs for value access conflicts do not have any recognizable property, since any graph can be shown to be the conflict graph of some data flow graph.

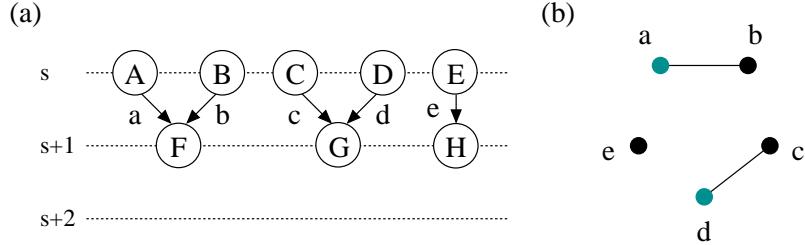


Figure 3.7: Another example of SCG weakness. The minimum requirement of registers is three. (a) DFG portion, and (b) the corresponding SCG with $\chi(SCG) = 2$.

Let $G = (V(G), E(G))$ be an arbitrary graph from which a data flow graph is being derived (e.g. Figure 3.8a), then:

- For all $u^c \in V(G)$, the corresponding value u is created in the data flow graph (Figure 3.8b) as a data edge between two vertices, i.e. the producer and the consumer operations. The weight of each edge u in DFG is one (minimum lifetime).
- For all $(u^c, v^c) \in E(G)$ and assuming random-access registers, sequence edges of weight equal to one are added from producer to consumer operations of values u and v in DFG defining a strong conflict between the lifetimes of those values (Proposition 4.4 for non-folded cases, refer to next chapter).
- Note that the sequence edges added in the data flow graph do not induce paths with positive cycles (from one operation to itself). Otherwise, the data flow graph would be infeasible (refer to Section 2.5.4). In Figure 3.8b all paths go from the set of operations in the top to the set in the bottom.
- Additionally, no paths in DFG induce conflicts not present in G .

With this result it is concluded that, if from any graph a corresponding path-feasible, non-scheduled data flow graph can be obtained, a data flow graph can originate any conflict graph.

Note finally that conflicts derived from G are strong and therefore part of both $WCCG$ and SCG . This proof thus holds for both $WCCG$ and SCG conflict graphs.

In conclusion, no special property can be exploited for coloring efficiently $WCCG$ and SCG .

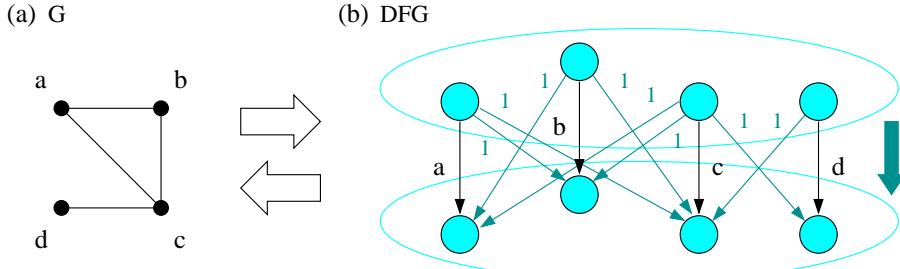


Figure 3.8: Any graph can be the conflict graph of a data flow graph. (a) An initial graph G . (b) For values assigned to random-access registers a data flow graph is build from G .

3.5.4 Exact coloring algorithm

In general, since no property of $WCCG$ or SCG can be exploited, finding a coloring is an NP-hard problem [26]. In order to prevent excessively long run times in finding an exact coloring, the approach presented by Coudert in [18] was selected, which offers a good tradeoff between quality and run time.

Coudert's coloring algorithm is based on finding and coloring a maximum clique of the graph and then sequentially picking up vertices that have the largest saturation number (see Definition 2.4), breaking ties with the largest degree number in the remaining uncolored graph. The process of finding the maximum clique of a graph is based on a simplified branch-and-bound algorithm together with heuristics to prune the search space.

The motivation of Coudert to use heuristics in his algorithm is to exploit the 1-perfectness property (defined in Section 2.6) that appears in many “real-life” examples, but does not always hold (see the example in Figure 3.9). The algorithm works fast and exact for certain cases and slow (but exact) for others. Besides, experimental results presented in the following chapters will show that solutions were found in a reasonable time.

Coudert's algorithm offers the following advantages: after finding the maximum clique of the worst-case conflict graph the clique number $\gamma(WCCG)$ can already be compared with the capacity of the storage file $c(SF)$.

- If $\gamma(WCCG) > c(SF)$, there are already more storage units required than available. Weak conflict edges between values in the clique can be considered to be removed. Nevertheless, not only those weak conflict edges are bottlenecks for storage allocation, but other edges (which are part of many maximum cliques) have to be also considered. The heuristics applied in our approach (explained in the following section) require to completely color

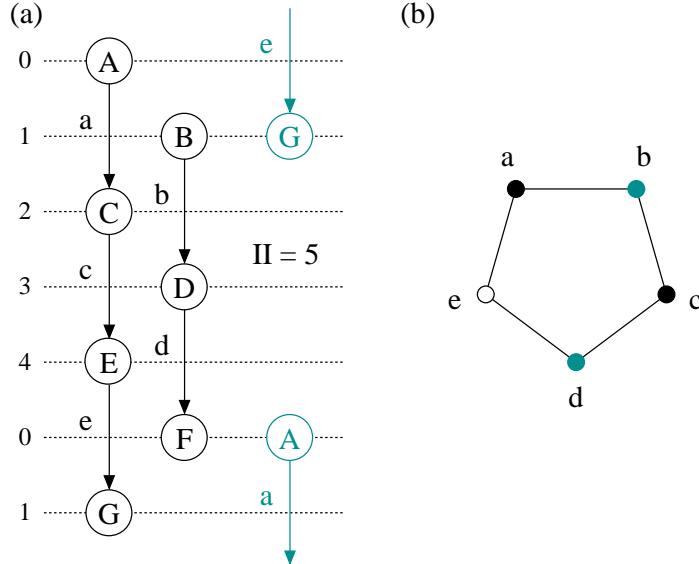


Figure 3.9: A non 1-perfect conflict graph in a loop folded case. (a) A loop folding scheduled data flow graph, (b) the resulted conflict graph with $\gamma(CG) = 2$, and $\chi(CG) = 3$.

WCCG for a better exploration of the search space.

- If $\gamma(WCCG) \leq c(SF)$, the coloring process needs to be completed, and bottlenecks need to be identified with the heuristics proposed.

Similarly, the clique number $\gamma(SCG)$ of the best-case conflict graph can also be compared with the capacity of the storage file $c(SF)$.

- If $\gamma(SCG) > c(SF)$ an infeasible result can already be returned, since the minimum storage requirement is more than the available.
- If $\gamma(SCG) \leq c(SF)$, the coloring process needs to be completed, and the actual number of colors $\chi(SCG)$ will indicate the feasibility direction of the process.

3.6 Bottlenecks for Storage Satisfaction

Reduction of storage file pressure can be obtained by finding a set of values that require the largest amount of storage. Some of those values can be either spilled to

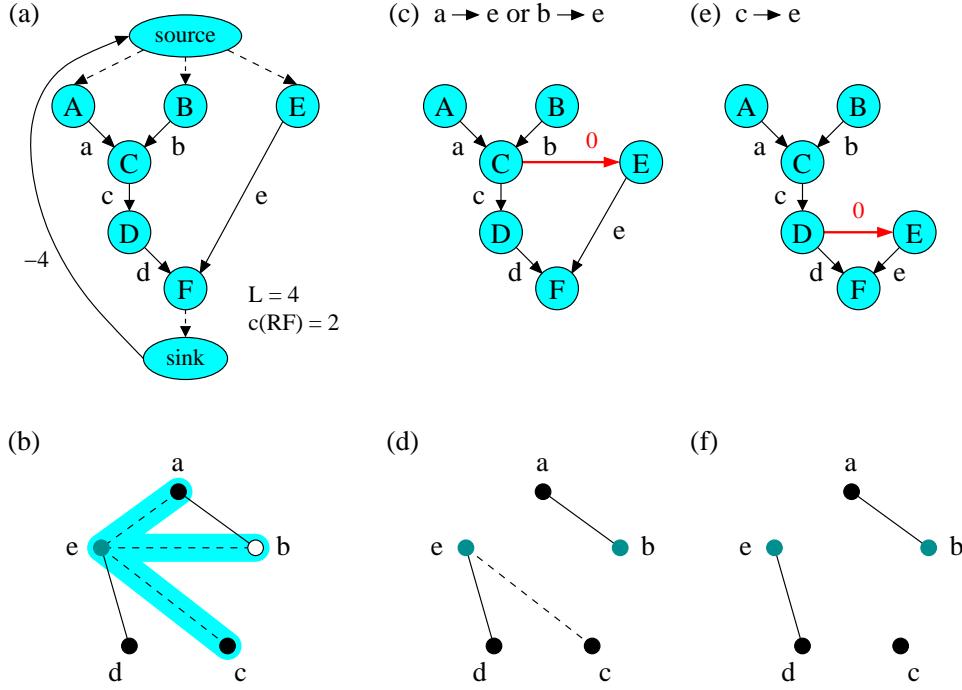


Figure 3.10: Example of bottleneck reduction. (a) Non-folded DFG , (b) $WCCG$ with $\chi(WCCG) = 3$ and the possible edges to remove, (c) serialization of lifetimes of a and e or b and e , (d) resulting $WCCG$ with $\chi(WCCG) = 2$ after ordering $C \rightarrow E$, (e) serialization of lifetimes of c and e , (f) resulting $WCCG$ with $\chi(WCCG) = 2$ after ordering $D \rightarrow E$.

different storage including background memory, or their accesses can be ordered such that they can share a storage unit.

In the domain of conflict graphs, value spilling can be viewed as removing vertices from the graph (and therefore the edges incident to them), while value access ordering removes edges between the corresponding vertices. In both approaches, the aim is to obtain a reduction in the final number of colors to satisfy some “color budget”.

Our approach reduces the storage file pressure until it is possible to store all the assigned values into the available storage units. This is done by means of using $WCCG$ and removing edges that represent weak access conflicts.

Consider the example of Figure 3.10a. $WCCG$ is colored with three colors (Figure 3.10b). Edges (a, e) , (b, e) , and (c, e) represent weak conflicts, and can be selected for removal. The solid edge between vertices a and b indicates a strong access conflict situation, and therefore edge (a, b) cannot be selected.

Now the question arises: how to identify the (best) bottlenecks?. The answer is given in the following section.

3.6.1 Bottleneck identification

Given a data flow graph DFG , a storage file SF , a set of values assigned to SF ($\Upsilon(SF)$), and the capacity of the storage file $c(SF)$, the goal of our approach is to reduce the number of colors in the coloring of $WCCG(SF)$ by removing some of its edges until $\chi(WCCG) \leq c(SF)$.

Removal of an edge (u^c, v^c) in the conflict graph $WCCG(SF)$ implies ordering the accesses of values u and v according to SF access behavior. The removal of an edge in $WCCG$ through the addition of sequence edges in DFG can indirectly imply the removal of other edges, a secondary effect which contributes to speed up the satisfaction process.

Since the removal of an edge in $WCCG$ implies a reduction in schedule freedom it is necessary to carefully select the edges to remove, in order to reduce efficiently the number of colors of $WCCG$ while not too much schedule freedom is sacrificed. For that purpose, the *saturation* number from Definition 2.4 is used to identify bottlenecks.

Consider a colored $WCCG$. When a vertex u^c has a saturation number $\text{sat}(u^c)$ it is equivalent to say that for vertex u^c and its neighbors $(\{u^c\} \cup N(u^c))$, $\text{sat}(u^c) + 1$ colors are assigned in the coloring. As larger the saturation numbers are, more are the colors used by the set.

Therefore, in a colored $WCCG$ the maximum saturation numbers of a vertex u^c and its neighbors indicates that each edge among those vertices is a candidate to be removed from $WCCG$ in order to reduce the number of colors for that vertex set. A reduction of the number of colors in the vertex set with maximum saturation numbers can potentially reduce the total number of colors for coloring $WCCG$.

In conclusion, the largest saturation number of vertices is used as criterion to identify bottlenecks in $WCCG$.

For bigger examples in which many vertices can have the same saturation number, the largest *degree* number of a vertex u^c ($|N(u^c)|$) is used to break ties among vertices with the same saturation number.

The saturation number was used in previous works [4, 51] obtaining acceptable results in terms of accuracy and execution time. In [3] some additional heuristics (including the saturation numbers in SCG) were experimented to refine the selection of the bottlenecks. However, those heuristics did not obtain better results for the analyzed examples.

In the example of Figure 3.10b edge (a, e) is identified as bottleneck since $\text{sat}(a) = \text{sat}(e) = 2$, $\deg(e) = 4$, and $\deg(a) = 2$.

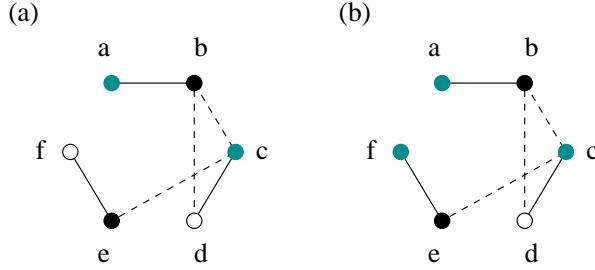


Figure 3.11: Exact colorings of a graph $WCCG$, $\chi(WCCG) = 3$. (a) First coloring in which $\text{sat}(e) = 2$, (b) coloring assumed in which $\text{sat}(e) = 1$.

Disadvantages of the saturation number

The saturation number of vertices, and hence the bottleneck identification, can be sensitive to the way $WCCG$ is colored. This is important to take into account since using the saturation number as a way to identify bottlenecks can mislead the process if the vertices targeted in $WCCG$ do not correspond to a group that requires the largest number of colors (storage).

Look at the example of Figure 3.11. The same $WCCG$ was exactly colored in two ways. The chromatic number is always three, but in the colored graph of Figure 3.11a the saturation number of vertex e is $\text{sat}(e) = 2$. In that graph, the removal of the weak conflict edge incident to e does not reduce the number of colors for coloring $WCCG$. Removing edges between vertices b, c and d does reduce the number of colors. Unfortunately, this situation cannot always be avoided.

Another effect is that, while looking only at the maximum saturation number a solution cannot always be found directly. See the example of Figure 3.12a. The same $WCCG$ was exactly colored in two ways. Using the maximum saturation number, vertices a, e , and d are selected, however they do not have a weak conflict. Meanwhile, the coloring of the graph in Figure 3.12b, allows us to find vertices c and d with a weak conflict.

In the FACTS implementation, to overcome the problems presented, the bottleneck selection proceeds as follows:

1. All vertices in $WCCG$ are considered for reasons of *completeness*.
2. Vertices are listed and sorted according to their largest saturation and degree numbers criteria. Vertex u^c is selected from the top of the list and then the set of its neighbors is obtained.

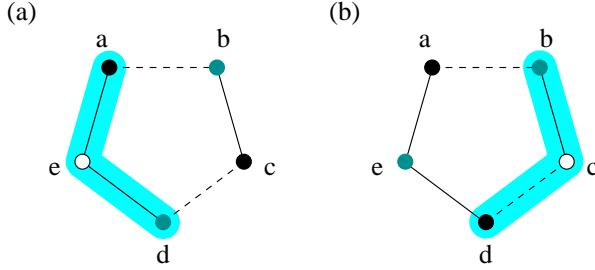


Figure 3.12: Drawback of identifying bottlenecks through maximum saturation number. (a) First coloring, edges between vertices with maximum saturation number do not represent weak conflicts, (b) a second coloring, at least one edge represents a weak conflict.

3. The neighbors $N(u^c)$ are also sorted by their largest saturation and degree numbers. Vertex v^c is selected starting from the top of the list if edge (u^c, v^c) represents a weak conflict.

3.7 Performing Access Ordering

Value *access ordering* is the term used in this work to denote a relative placement of operations (producers and consumers of values) through the insertion of sequence edges according to the access behavior of the targeted storage file. The goal of access ordering is to eliminate a potential conflict between values from the identified bottleneck in $WCCG$.

Value *lifetime serialization* is a particular access ordering performed in the case of having registers as storage.

According to the storage access behavior and the scheduling freedom the accesses of two values can be ordered in different ways (like lifetime serialization can be performed with a value u preceding v or value v preceding u). The ordering process considers one kind of ordering at a time. If constraint analysis or the lower bound checking detect infeasibility as a result of that choice, another ordering is made if possible. If all possible orderings lead to infeasibility, access ordering of those values is discarded and another pair is chosen to repeat the process.

In Figure 3.10 lifetime serialization of, e.g. a and e will result in the addition of sequence edge $C \rightarrow E$ with weight zero in the data flow graph. After serialization, the updated conflict graph $WCCG$ is colored with only two colors as shown in Figure 3.10d. Note that after serialization $a - e$, not only edge (a, e) has been removed but also (b, e) as a secondary effect of that serialization decision. Serialization $c - e$ is not detected by our approach. This serialization removes

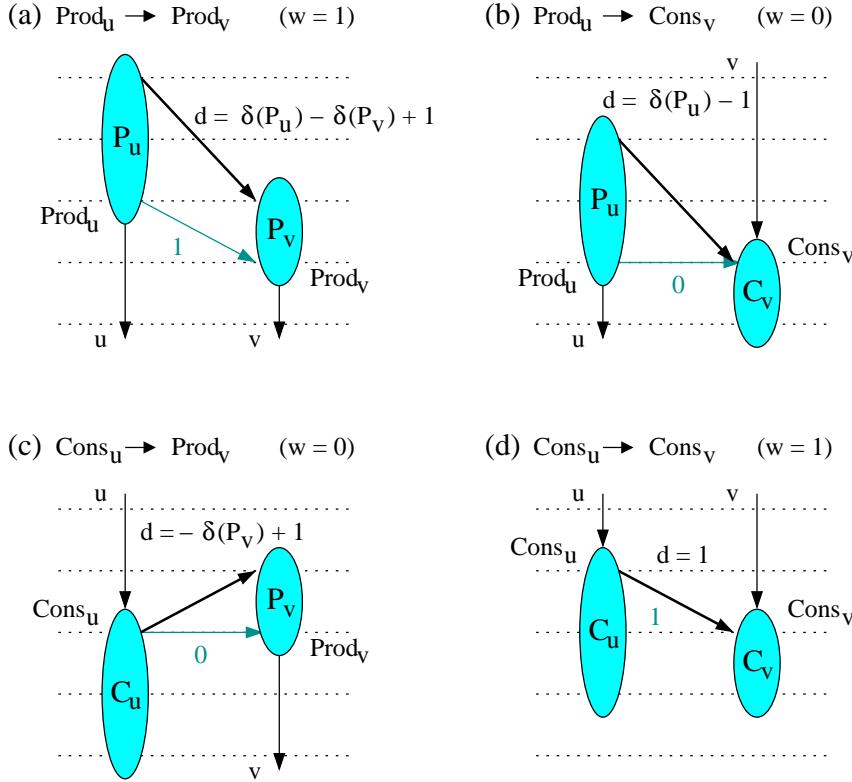


Figure 3.13: Ordering of operations. (a) The production of value u precedes the production of v , (b) the production of value u precedes the consumption of v , (c) the consumption of value u precedes the production of v , and (d) the consumption of value u precedes the consumption of v .

also edges (a, e) and (b, e) but has the drawback of consuming all the scheduling freedom (see Figures 3.10e and 3.10f).

Figure 3.13 shows the different situations when ordering the productions and consumptions of values, taking into account the operation execution delays.

3.8 Branch-and-Bound Approach

A branch-and-bound algorithm is used to solve combinatorial optimization problems with a certain cost function. It is a systematic way to evaluate solutions in a *decision tree* determined by a set of decision variables. Leaves in the tree represent the possible solutions.

For each *branch*, i.e. local decision on the value of a variable, a lower *bound*

is computed on the cost of all solutions in the corresponding sub-tree. If that bound is higher than the cost of any solution found so far the sub-tree is *pruned* (or *killed*), because all its leaves would yield a solution of provably higher cost.

In FACTS, a branch-and-bound approach for storage constraint satisfaction is implemented. In the satisfaction approach there is no cost function, and the bounding mechanism concerns *feasibility* rather than cost. This approach has a systematic way of choosing values for access ordering and the branch-and-bound is restricted to the ordering choices.

For a selected pair of values (output of the *bottleneck identification* process), a *branch* represents a choice regarding the ordering applied to the corresponding producer and consumer operations while the *bounding* consists in the following mechanisms:

- During access ordering, some branches are already excluded when using rules applied to the distance matrix. This is done before any branch is taken. In Figure 3.14 the precedence between consumer operations of values u and v determines only one way of ordering excluding already the other.
- Constraint analysis detects infeasibility as a result of ordering decisions. During ordering some possibilities are excluded but no guarantees regarding feasibility can be given. Constraint analysis provides a more accurate mechanism to exclude ordering possibilities and therefore acts as a bounding mechanism.
- Another bound is determined by the number of colors in the strong conflict graph SCG because this number represents a lower bound on storage allocation. As a result of an ordering decision $\chi(SCG)$ can become greater than the capacity constraint and an infeasibility is returned in this case. In this sense, the importance of the strong conflict graph and its coloring is emphasized to steer the satisfaction process.

If access ordering results in an infeasibility the process *backtracks* and makes another ordering. If all orderings fail, the pair of values is marked and the process restarts with another (unmarked) pair. When there are no more choices it means that there is no solution found for the satisfaction problem.

The branch-and-bound approach is meant to guarantee the *completeness* of our approach regardless of the time consumed to obtain a solution. However, for execution time reasons the maximum number of backtracks is limited by allowing only a limited number of branches visited during the satisfaction process.

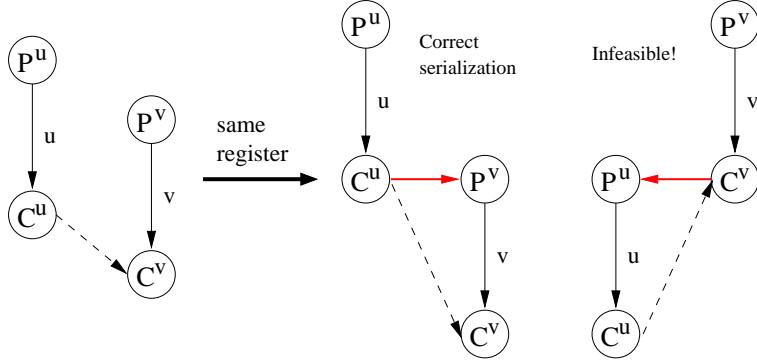


Figure 3.14: Access ordering of values u and v assigned to random-access registers (lifetime serialization). Distance $d_m(C^u, C^v) > 0$ excludes already one of the ordering possibilities.

3.9 Run-time Complexity

Let $V = V(DFG)$, $E_d = E_d(DFG)$, and $N = |\Upsilon(SF)|$ the number of values assigned to a storage file SF , with $|\Upsilon(SF)| \leq |E_d| \leq |V|^2$. $|SF_i|$ is the number of storage files of which constraints have to be satisfied. The complexity of our approach is determined by the following processes:

- *Constraint analysis and distance matrix update.* The run-time complexity of the execution interval analysis is presented in detail in [73]. The dominant factor in execution interval analysis is the construction of the bipartite schedule graph BSG and determining its irreducible components (refer to Section 2.5.1), of which complexity is $|V|^2 + |V|^{1/2} \cdot |\overline{A}|$, where \overline{A} is related to the set of arcs in BSG , with $|\overline{A}| \leq |V|^2$. The complexity of the distance matrix update is determined by the number of paths that need to be updated as a result of a new sequence edge. This process has a complexity of $O(2L \cdot |V|^2)$ [50], where L is the latency.
- *Storage file selection.* Storage file selection is performed using the priority function from Expression 3.2. At the beginning of the process the construction and coloring of $|SF_i|$ worst-case conflict graphs are necessary for the initial file selection. During the satisfaction process storage file selection has a complexity of $O(|SF_i|^2)$ since it uses a sort algorithm. The chromatic numbers $\chi(WCCG)$ for each graph $WCCG(SF_i)$ are updated during the process.
- *Worst-case conflict graph construction and coloring.* For a storage file SF , the construction of $WCCG$ takes $O(N^2)$. The algorithm of Coudert [18] is

used for exact coloring of $WCCG$. This algorithm was built to exploit the 1-perfectness property of most register allocation related conflict graphs. Coloring of 1-perfect graphs is solvable in polynomial time [28] and, although the algorithm of Coudert cannot be bound in polynomial time, it gives good results in practice.

- *Best-case conflict graph SCG construction and coloring.* The construction of SCG also takes $O(N^2)$, and Coudert's algorithm [18] is also used to obtain $\chi(SCG)$.
- *Bottleneck identification.* It uses a sort algorithm for the values assigned to storage file SF . Therefore, it takes $O(N^2)$ [17].
- *Branch-and-bound.* The branch-and-bound complexity for our approach can take exponential run time. To avoid long run times during the experiments the number of backtracks was limited to a small number like 100.

In (most of) the cases in which $|E_d|$ is comparable to $|V|$ or less, the execution interval analysis and the distance matrix update are the critical factors. By making a profile over the routines executed during the storage satisfaction process, it was found that in many cases up to 80% of the time was spent for constraint analysis.

When $|E_d|$ is much larger than $|V|$ the construction and coloring of the conflict graphs become dominant.

The use of distributed storage files is expected to perform in shorter run times because the problem instances are smaller.

The actual complexity of the storage satisfaction process is hard to calculate since it depends not only on the number of operations $|V|$, the number of values $|E_d|$, or the number of storage files $|SF_i|$, but also on how tight are the constraints that prune part of the scheduling search space reducing the time to find a solution.

Experiments showed that for examples with less than 100 operations and values a feasible solution is found in a matter of seconds. However, for some cases a solution could not be found within the limited number of backtracks allowed. Finally, the algorithm of Coudert is not a dominant factor in the run time.

Chapter 4

Register Files

4.1 Introduction

In the previous chapter the approach for storage constraint satisfaction during scheduling was presented. In this chapter our approach is applied to the case of using *registers* in the architecture.

Random-access registers are used to store data that are transferred from a functional unit to another across clock cycle boundaries. This storage is called “random-access” because any single register can be accessed directly and not necessarily in a sequential way like in the case of stacks and fifos (see next chapter). A *register file* denotes a group of registers with common read/write ports or addresses and it is organized and controlled in a way that enables the access to specific registers.

The best reasons for the use of random-access registers in an embedded processor are twofold. First, registers internal to the CPU are faster than external memory. Second, registers are easier for a compiler to use and can be used more effectively than other forms of internal storage like stacks or fifos. Moreover, when registers are used to hold values the memory traffic reduces, the program speeds up since registers are faster than memory, and the code density improves since a register can be addressed with fewer bits than a memory location (the number of registers is always smaller).

From the compiler point of view the ideal situation would be if all registers were equivalent (the same size, same addressing space) and unreserved for special purposes. However, many embedded processors compromise this desire by dedicating registers to special uses, or partitioning the available registers into smaller register files distributed over different parts of the architecture. Embedded processor classes have the following characteristics:

- DSP processors have (distributed) special-purpose registers, like a dedicated

accumulator register. The use of special-purpose registers is dependent on the instruction context, where the source and destination registers are implicit in the instruction code. This leads to shorter delays in the data-path, due to the absence of complex address decoding hardware, and less bits required for encoding. Because of the irregularities in their architecture, compiler construction for DSPs is difficult.

- VLIW multimedia processors have a more regular architecture with preferably one large register file [30] and an orthogonal instruction set, i.e. in which any register can be used by any instruction. This one register file architecture has a negative impact on power consumption, code size (often more than 50% of the instruction bits in VLIW processors are used for register addressing), and clock frequency compared to architectures with distributed register files. Therefore, the tendency is to *partition* the register file into a number of smaller files [22]. The aim of file partitioning is to increase the overall bandwidth by providing multiple paths between registers and functional units. However, this has the major drawback of increasing the complexity of the compiler required to perform register allocation onto a heterogeneous register space, since not all direct communication between functional units and register files is provided.
- Due to the objectives of ASIP design namely small code size, low power consumption, and high performance, ASIPs can have architectures with highly irregular data-paths and relatively few registers, which are far from an ideal compiler target. Due to the large overhead in schedule length and code size of compiler generated code when compared to hand-written assembly for ASIPs, there is an urgent need for compilation methods that can deal with such irregular data-paths.

For these classes of embedded processors the same code generation problem arises: performing scheduling and register allocation for *capacity limited register files* such that potentially severe timing and resource constraints are also satisfied.

This chapter presents the capacity satisfaction process considering register files as shown in Figure 3.3 (repeated for convenience in Figure 4.1). The rules to construct conflict graphs for values assigned to register files are given in Section 4.2. Lifetime serialization as value access ordering is described, and some related heuristics are discussed in Section 4.3. A technique called *value merging* to improve the accuracy and performance of the satisfaction process is presented in Section 4.5. The lower bound of the register count estimated to steer the satisfaction process is also discussed, and different lower bound approaches are described in Section 4.6. The experimental results to test, compare and validate the

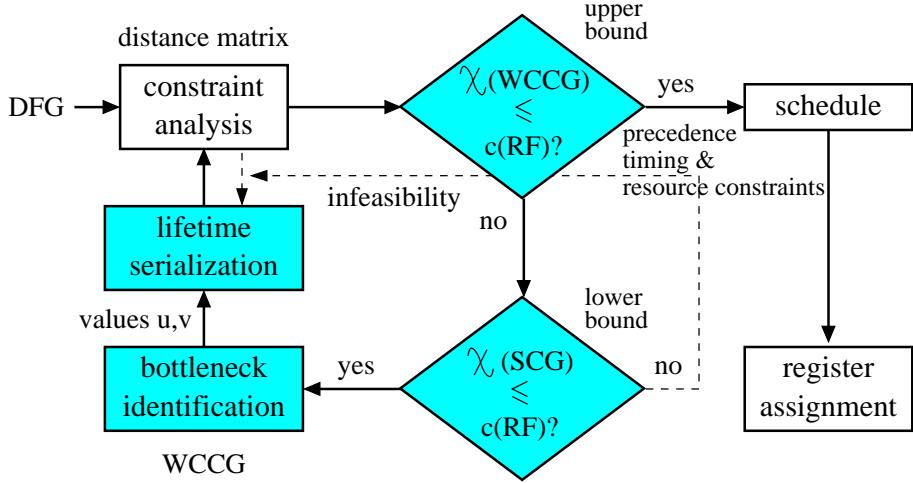


Figure 4.1: Approach for register file constraint satisfaction.

heuristics and techniques proposed are presented in Section 4.7. Finally, capacity satisfaction for *rotating register* files is considered in Section 4.8.

4.2 Constructing Conflict Graphs

The worst-case $WCCG(RF)$ and the best-case $SCG(RF)$ conflict graphs are constructed based on specific rules for accessing values assigned to random-access register files. The access conflict rules are presented in the following section.

4.2.1 Conflict Rules

Consider values u and v produced by operations P^u and P^v and consumed by C^u and C^v respectively. $Prod^u$ and $Prod^v$ represent the productions of u and v respectively, while $Cons^u$ and $Cons^v$ represent their consumptions. For loop folded cases, instances u_i and v_j correspond to values u and v in iterations i and j respectively.

The following propositions consider the loop folded cases in general. For the non-folded cases simply consider $II = L$ and $i = j = k = 0$.

No conflict

Values u and v have no conflict if due to the order of their accesses their lifetimes can never overlap. There is no overlap between values u and v if and only if

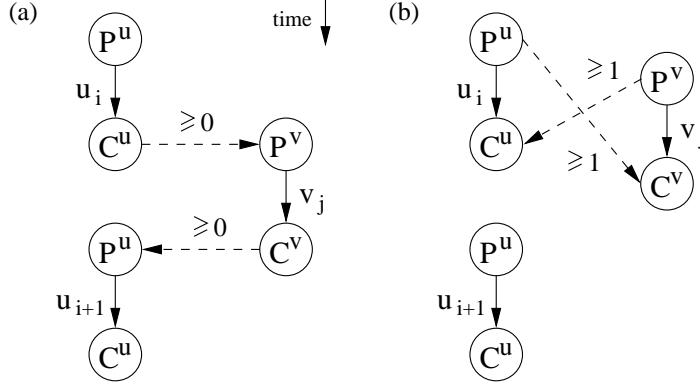


Figure 4.2: Values u and v (a) have no conflict, (b) have a strong conflict.

the lifetime of v is exactly contained in the interval between two successive loop iteration lifetimes of u . This is depicted graphically in Figure 4.2a and captured by the following proposition:

Proposition 4.1 *Values u and v have no conflict if and only if there exist iterations i and j such that $d(\text{Cons}_i^u, \text{Prod}_j^v) \geq 0$ and $d(\text{Cons}_j^v, \text{Prod}_i^u) \geq -II$.*

This corresponds to the situation that a consumer operation of one value precedes the producer operation of the second value by at least zero clock cycles (a value may be read and subsequently overwritten in the same clock cycle).

Because the distance matrix is used for checking a conflict, the equivalent Proposition 4.2 is derived.

Proposition 4.2 *Values u and v have no conflict if there exists $k \in \mathbb{N}$ such that*

$$\begin{aligned} d_m(C^u, P^v) + \delta(P^v) - 1 &\geq k \times II & \text{and} \\ d_m(C^v, P^u) + \delta(P^u) - 1 &\geq -(k + 1) \times II \end{aligned}$$

Strong conflict

Values u and v have a strong conflict if due to the order of their accesses their lifetimes overlap for sure. There is an overlap between values u and v if and only if the lifetime of v can never be exactly contained in the interval between two successive lifetimes of u . This is depicted graphically in Figure 4.2b and captured by the following proposition:

Proposition 4.3 *Values u and v have a strong conflict if and only if there exist iterations i and j such that $d(\text{Prod}_i^u, \text{Cons}_j^v) \geq 1$ and $d(\text{Prod}_j^v, \text{Cons}_i^u) \geq 1$.*

This corresponds to the situation where the producer operation of each value precedes the consumer operation of the other by at least one clock cycle.

Because the distance matrix is used for checking a conflict, the equivalent Proposition 4.4 is derived.

Proposition 4.4 *Values u and v have strong conflict if there exists $k \in \mathbb{N}$ such that*

$$\begin{aligned} d_m(P^u, C^v) - \delta(P^u) &\geq k \times II \quad \text{and} \\ d_m(P^v, C^u) - \delta(P^v) &\geq -k \times II \end{aligned}$$

Weak conflict

There is a weak conflict if conditions of Propositions 4.2 and 4.4 are both invalid.

For the construction of *WCCG* two values have a worst-case conflict if they simply do not have a no conflict situation, i.e. the condition from Proposition 4.2 is invalid. This is because *WCCG* includes weak and strong conflicts. For *SCG* the condition from Proposition 4.4 is directly applied.

Consider the example of Figure 4.3a without folding. The execution delay of the operations is one clock cycle. There is no conflict, e.g. for values a and c , since their lifetimes are already serialized; a strong conflict, e.g. for values b and c in the clock cycle that operation D executes (both values are consumed at the same time); and a weak conflict situation, e.g. for values b and e . If operation E precedes operation D by at least one clock cycle lifetimes of b and e overlap (have a strong conflict), otherwise b and e will have no conflict. Since it is not yet determined whether or not E precedes D , b and e have a weak conflict. The worst- and the best-case conflict graphs are shown in Figures 4.3b and 4.3c respectively.

4.3 Lifetime Serialization

Following the bottleneck identification (see Figure 4.1), values u and v are selected for lifetime serialization.

4.3.1 Heuristic of the minimum sacrificed distance

Lifetime serialization can often be performed in more than one way. At a glance a serialization decision can be based on sacrificing the least possible schedule

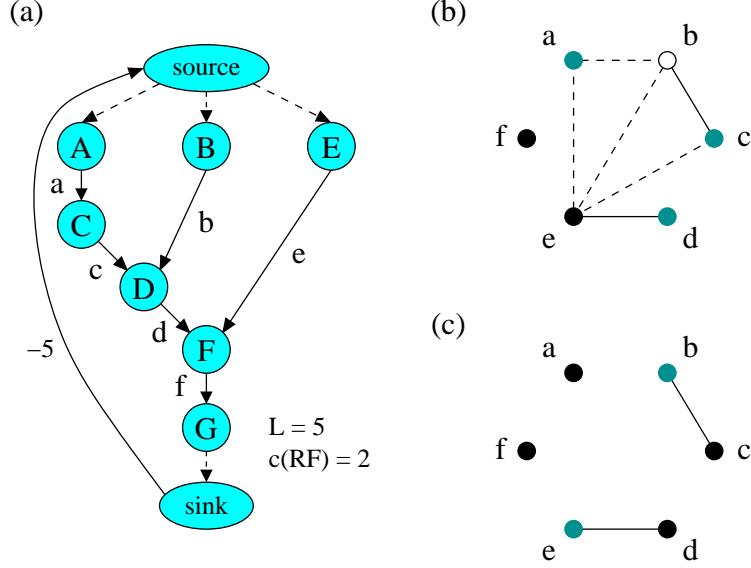


Figure 4.3: Example of (a) a *DFG*, (b) its worst-case conflict graph *WCCG*, and (c) its strong conflict graph *SCG* after constraint analysis and values assigned to random-access registers.

freedom as presented in [3, 51]. When less schedule freedom is sacrificed, more is the chance for subsequent serializations to be performed, and eventually other constraint satisfaction processes can exploit the remaining freedom too.

Considering a non-folded case and execution delays of one clock cycle, lifetime serialization of u and v (with a weak conflict) can be done in two ways: $C^u \rightarrow P^v$ or $C^v \rightarrow P^u$. Sometimes the constraints are such that constraint analysis is able to exclude one of these possibilities. If this is not the case, a heuristic should be applied.

Let $x = d(C^u, P^v)$ and $y = d(C^v, P^u)$. $x < 0$ and $y < 0$, i.e. the weak conflict situation in which serialization of lifetimes of u and v can be performed in two ways. Making a decision is to increase either x or y to zero. The least schedule freedom is sacrificed when the smallest increment is made, therefore if $|x| \leq |y|$ then ordering $C^u \rightarrow P^v$ is performed, otherwise $C^v \rightarrow P^u$.

Consider the example of Figure 4.4. This example has a latency of four, a register file with capacity of two registers. The distance matrix is also shown in 4.4b. The initial worst-case conflict graph is illustrated in Figure 4.4c and has a chromatic number of three. From the list of possible candidates with highest saturation and degree numbers in *WCCG*, suppose edge (b, e) is selected to be reduced. Therefore, lifetime serialization $b - e$ will be performed. In the distance

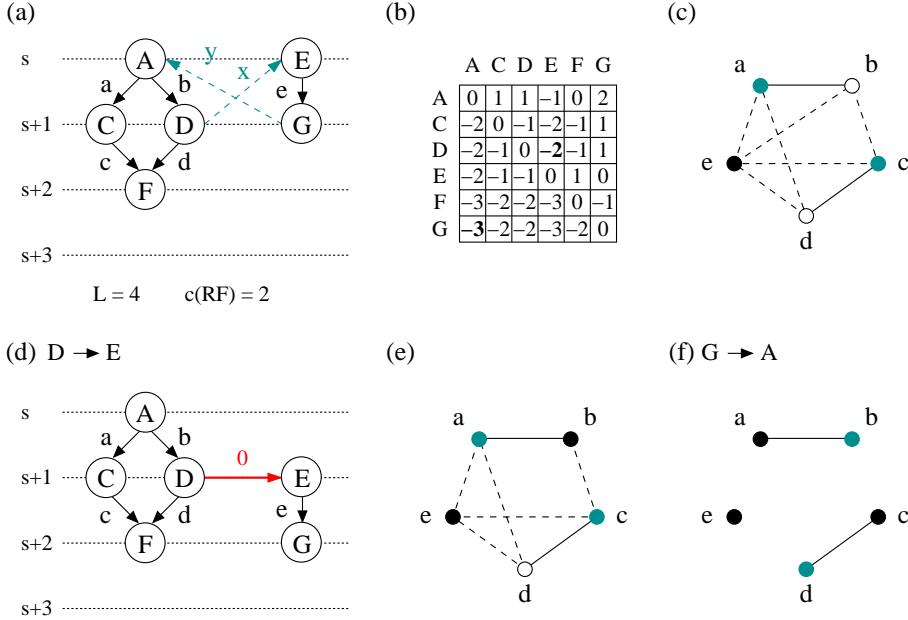


Figure 4.4: Heuristic of the minimum sacrificed distance. (a) DFG, (b) initial distance matrix, (c) WCCG with $\chi(WCCG) = 3$, (d) serialization of values e and b through ordering $D \rightarrow E$, (e) WCCG with $\chi(WCCG) = 3$ after ordering $D \rightarrow E$, and (f) WCCG with $\chi(WCCG) = 2$ after ordering $G \rightarrow A$.

matrix: $x = d_m(C^b, P^e) = d(D, E) = -2$ and $y = d_m(C^e, P^b) = d(G, A) = -3$. In this case $|x| \leq |y| (|-2| < |-3|)$, and the decision would be to insert the sequence edge $D \rightarrow E$ (Figure 4.4d) in order to loose only two clock cycles of schedule freedom. However, looking at the conflict graph in Figure 4.4e there is still a requirement of three registers since $\chi(WCCG) = 3$, having the necessity to repeat the process afterwards and insert another sequence edge $F \rightarrow E$. The decision of serializing lifetimes of e and b with the insertion of sequence edge $G \rightarrow A$ is more effective, as the resulting conflict graph in Figure 4.4f shows.

In conclusion, if any lifetime serialization is possible, looking only at the relative distances between production and consumptions of the corresponding values yields unclear results.

4.3.2 Serialization method

Considering the fact that in the loop folded case one iteration of value v can be scheduled in between any other subsequent iterations of value u , it is necessary to find out the possible “places” where value v can fit without infeasibility, i.e. the

gaps between u iterations where v can be scheduled.

It is tried to serialize lifetimes of u and v by fitting instance k of v between instances $k+i$ and $k+i+1$ of u as depicted in Figure 4.5. i is thus the relative iteration difference between instances of u and v being serialized.

To put Cons_k^v before Prod_{k+i+1}^u a sequence edge with weight zero is introduced from Cons_k^v to Prod_{k+i+1}^u . To prevent a positive weight cycle between operations, which results in an infeasible situation (see Section 2.5.4), the distance from Prod_{k+i+1}^u to Cons_k^v should be less than or equal zero, resulting in the following inequality:

$$d(\text{Prod}_{k+i+1}^u, \text{Cons}_k^v) \leq 0$$

$$\text{Expr. 2.2 : } d(\text{Prod}^u, \text{Cons}^v) - (k+i+1-k) \times II \leq 0$$

$$d(\text{Prod}^u, \text{Cons}^v) \leq (i+1) \times II$$

$$\frac{d(\text{Prod}^u, \text{Cons}^v)}{II} - 1 \leq i$$

$$\left\lceil \frac{d(\text{Prod}^u, \text{Cons}^v)}{II} \right\rceil - 1 \leq i$$

$$\left\lfloor \frac{d(\text{Prod}^u, \text{Cons}^v)-1}{II} \right\rfloor \leq i$$

Finally, using Expression 2.5 we obtain:

$$\left\lfloor \frac{d_m(\text{P}^u, \text{C}^v) - \delta(\text{P}^u)}{II} \right\rfloor \leq i \quad (4.1)$$

This expression gives the lower bound of i .

Similarly, to put Cons_{k+i}^u before Prod_k^v a sequence edge with weight zero is introduced from Cons_{k+i}^u to Prod_k^v . To prevent a positive weight cycle between operations the distance from Prod_k^v to Cons_{k+i}^u should be less than or equal zero resulting in the following inequality:

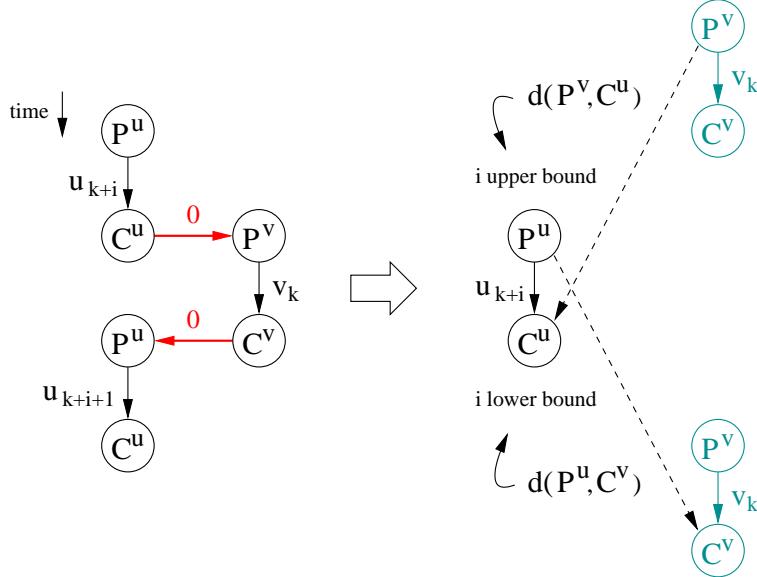


Figure 4.5: Lifetime serialization of values u and v . i is bound by inequalities 4.1 and 4.2.

$$d(\text{Prod}_k^v, \text{Cons}_{k+i}^u) \leq 0$$

$$\text{Expr. 2.2 : } d(\text{Prod}^v, \text{Cons}^u) - (k - (k + i)) \times II \leq 0$$

$$i \times II \leq -d(\text{Prod}^v, \text{Cons}^u)$$

$$i \leq \frac{-d(\text{Prod}^v, \text{Cons}^u)}{II}$$

$$i \leq \left\lfloor \frac{-d(\text{Prod}^v, \text{Cons}^u)}{II} \right\rfloor$$

Finally, using Expression 2.5 we obtain:

$$i \leq \left\lfloor \frac{-d_m(P^v, C^u) + \delta(P^v) - 1}{II} \right\rfloor \quad (4.2)$$

This expression gives the upper bound of i .

For non-folded cases, making $II = L$ in Expressions 4.1 and 4.2 i can take values of minus one (ordering in which the lifetime of value v will precede the lifetime of u) or zero (the lifetime of v will succeed the lifetime of u).

Performing lifetime serialization

With a known relative iteration number i , lifetime serialization is performed with sequence edges:

$$\begin{aligned} \text{Cons}^u \rightarrow \text{Prod}^v : \quad w(C^u, P^v) &= i \times II - \delta(P^v) + 1 \\ \text{Cons}^v \rightarrow \text{Prod}^u : \quad w(C^v, P^u) &= -(i+1) \times II - \delta(P^u) + 1 \end{aligned} \quad (4.3)$$

The value access ordering starts to serialize lifetimes of u and v with the minimum value of i (from Expression 4.1) inserting two sequence edges using Expression 4.3. If constraint analysis or the lower bound checking detect infeasibility as a result of this choice, i is incremented and another serialization (in another gap) is tried. If all possible values of i lead to infeasibility, serialization of u and v is discarded and another pair of values is chosen to repeat the process.

4.4 A Folded-Case Example

To illustrate this process the example in Figure 4.6 is used. This example comprises resource constraints, e.g. $A - D$ means that operation A has a resource constraint with D and they cannot be scheduled in the same clock cycle, a latency of seven, an initiation interval of two, and a register file RF with capacity of four. As explained in Section 2.3.3, the latency and the initiation interval constraints are expressed in the data flow graph, as seen in Figure 4.6a. The resource constraints are also expressed with sequence edges of weights one and three ($w = k \times II + 1$) [50]. The initiation interval constraint is also expressed with precedence edges of weight minus two between consumer and producer operations.

The distance matrix after constraint analysis is shown in Figure 4.6b, while the initial worst-case conflict graph is in Figure 4.6c. The coloring of this graph results in a chromatic number of six, i.e. six registers required, which is more than the available capacity of the register file.

Following the method in Section 3.6.1 suppose edge (a, e) in $WCCG$ is identified as bottleneck. a and e are therefore candidates for lifetime serialization. Using Expressions 4.1 and 4.2 for distances $d_m(A, F)$ and $d_m(E, C)$ respectively (with $u = a$ and $v = e$ and execution delays of one clock cycle), i is equal to one. Therefore, following Expression 4.3 the lifetime serialization process will insert sequence edges $C \rightarrow E$ with weight of two clock cycles, and $F \rightarrow A$ with weight of minus four clock cycles. Figure 4.6d shows the sequence edges added in the data flow. Constraint analysis then reduces the schedule freedom to zero and updates the distance matrix. The worst-case conflict graph is also updated (Figure 4.6e), and the new chromatic number is four which satisfies the storage

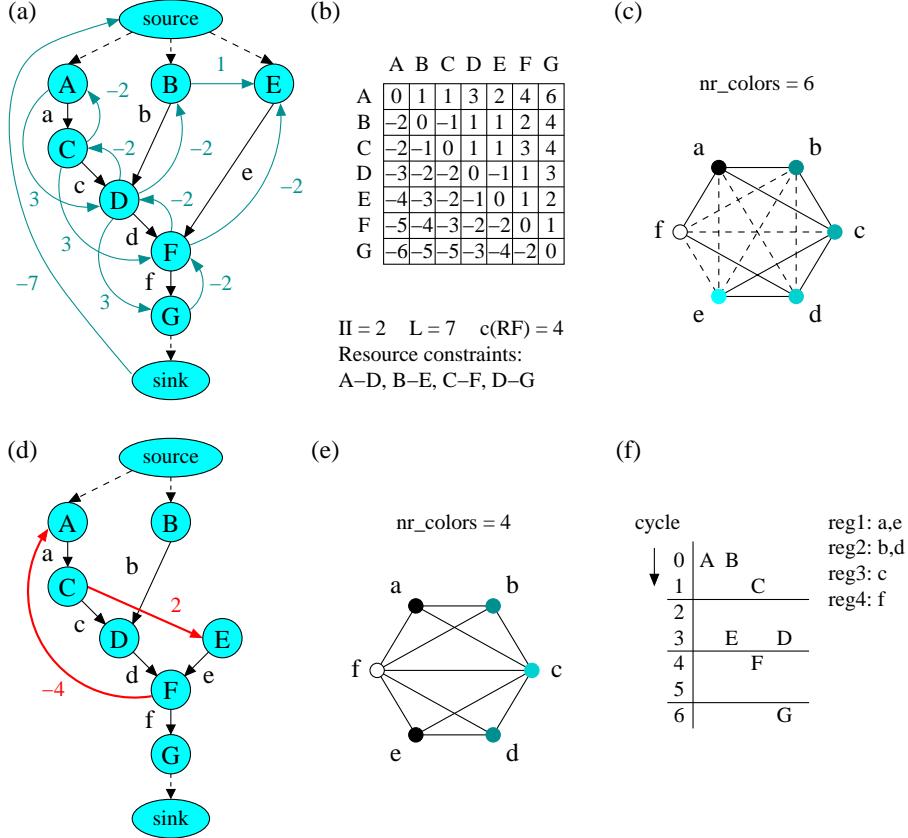


Figure 4.6: Constraint satisfaction for a folded-case (a) initial DFG , values are assigned to RF with $c(RF) = 4$, (b) distance matrix, (c) initial $WCCG$ with $\chi(WCCG) = 6$, (d) lifetime serialization $a - e$, (e) updated $WCCG$ with $\chi(WCCG) = 4$, and (f) final schedule and register assignment.

constraint. As a result the schedule is fixed as depicted in Figure 4.6f, and the register constraint is satisfied.

4.5 Value Merging

In Section 3.5.1 the characteristics and limitations of the worst- and the best-case conflict graphs were presented. However, in many cases the following can be observed. Look at the example of Figure 4.7a. Values a and b could occupy the same register between $s(A)$ and $s(D)$. They can be considered as a single value ab . The advantage is visible in the conflict graphs of Figures 4.7e and 4.7f: there are less vertices and more accurate bounds, which improve the accuracy of the graph

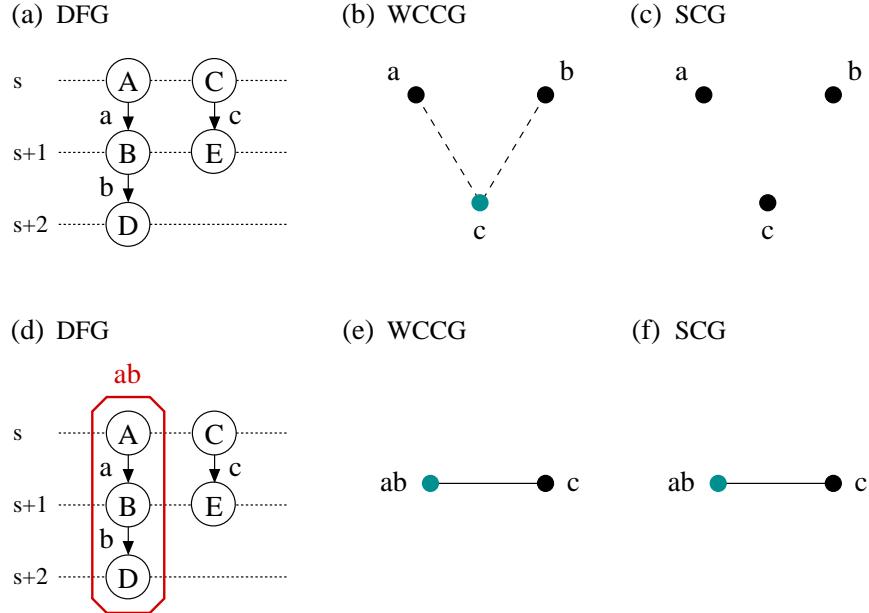


Figure 4.7: Example of the limitations of *WCCG* and *SCG* and the usefulness of value merging. (a) *DFG*, (b) the corresponding *WCCG* with $\chi(WCCG) = 2$, (c) the corresponding *SCG* with $\chi(SCG) = 1$, (d) a merging of values *a* and *b* in *ab*, (e) the new *WCCG* with less vertices and $\chi(WCCG) = 2$, (f) a more accurate lower bound with $\chi(SCG) = 2$.

coloring. Therefore, the *value merging* technique is useful in the construction of worst- and best-case conflict graphs.

Value merging will select a chain of values. All values in the chain are then considered as a single value being produced by the producer operation of the first value and consumed by the consumer operation(s) of the last value. The reasoning behind merging is that the merged values can be mapped onto a single register. However, merging of values at this stage does not necessarily imply that all values in a chain would be assigned to the same register at the end of the final storage allocation process.

With value merging the number of vertices in the conflict graphs is less, it simplifies graph construction, coloring, and hence speeds up the capacity satisfaction process. Moreover, as seen in Figure 4.7 the obtained lower bound is two (which is the tightest). With that lower bound result the capacity satisfaction process can be stopped before it tries unsuccessfully to serialize *a* – *c* or *c* – *b* in order to satisfy the capacity constraint of one register, which is in fact infeasible.

In order to perform merging efficiently and in a conservative way, the follow-

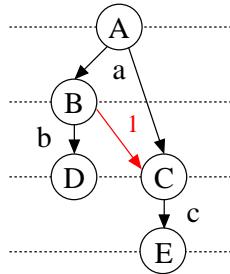


Figure 4.8: Because the lifetime of value a overlaps with its successor value b , a and b cannot be merged. In this case, value c can be part of the chain after a .

ing issues have to be taken into account:

- Values are merged only with others assigned to the same register file.
- To obtain long chains of merged values (with longer lifetimes) the first values should have the earliest production, which corresponds to the earliest ASAP of their producer operations. To break ties among values with the earliest ASAP, the earliest ALAP is used.
- Next value to merge is a direct data successor of the previous one, i.e. it must be produced by an operation that consumes the previous value. Otherwise, some lifetime “gaps” can be introduced which will not be detected and exploited for serialization. If there is more than one successor the value selected is the one with the earliest consumption, i.e. the earliest ALAP of its consumer operations. This is to obtain a longer chain of merged values.
- In addition to the previous item, if the consumer operation of the last value in the chain has an execution delay longer than one the merging process stops for the current chain, since a short value lifetime could fit in the gap created by the operation execution delay.
- Considering multiple consumption of values, the lifetime of the last value in the chain must not overlap the lifetime of the value candidate for merging, see Figure 4.8.

In loop folded cases value merging cannot be directly applied. Values to be merged must not overlap in successive iterations. Also, value merging can add undesirably in the register count. See Figure 4.9, the decision of merging values a and b adds an extra register requirement, as seen by comparing the conflict graphs without and with value merging in Figures 4.9b and 4.9c respectively.

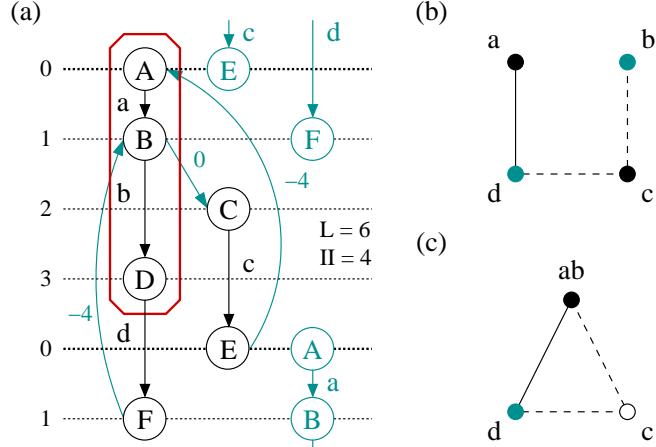


Figure 4.9: An example of value merging in a loop folded case. (a) A data flow graph (assuming $d_m(B, D) = 2$, $d_m(D, F) = 2$, and $d_m(C, E) = 2$). (b) Initial $WCCG$ without value merging with $\chi(WCCG) = 2$. Values a and b are merged, value d is not considered because it has a conflict with a in the next iteration. (c) Resulted $WCCG$ after merging, $\chi(WCCG) = 3$.

One way to avoid this problem is to create chains with fixed lifetime of II clock cycles, which is difficult to find when the lifetimes are not fixed yet.

Another solution is to unroll the loop first [40], create new (unfolded) values and apply value merging as in the non-folded case. This is shown in Figure 4.10. Note that unfolded instances of the same value can be mapped to different registers once the loop is unroll (as c and c_1 or abd and abd_1).

A third solution is the use of rotating register files, a hardware solution that is presented in Section 4.8.

To illustrate the efficiency of value merging, look now at the example of Figure 4.3 repeated for convenience in Figure 4.11. After applying value merging in the data flow graph the new values are b , e , and the chain $acdf$. The conflict graphs from Figures 4.11b and 4.11c consist only of three vertices, and have the same chromatic numbers as compared to the ones from the graphs of Figures 4.3b and 4.3c.

4.6 Lower Bound Register Estimation

As a part of the satisfaction process shown in Figure 4.1 a lower bound of the number of registers is determined through the coloring of a best-case conflict graph $SCG(RF)$ ($\chi(SCG)$).

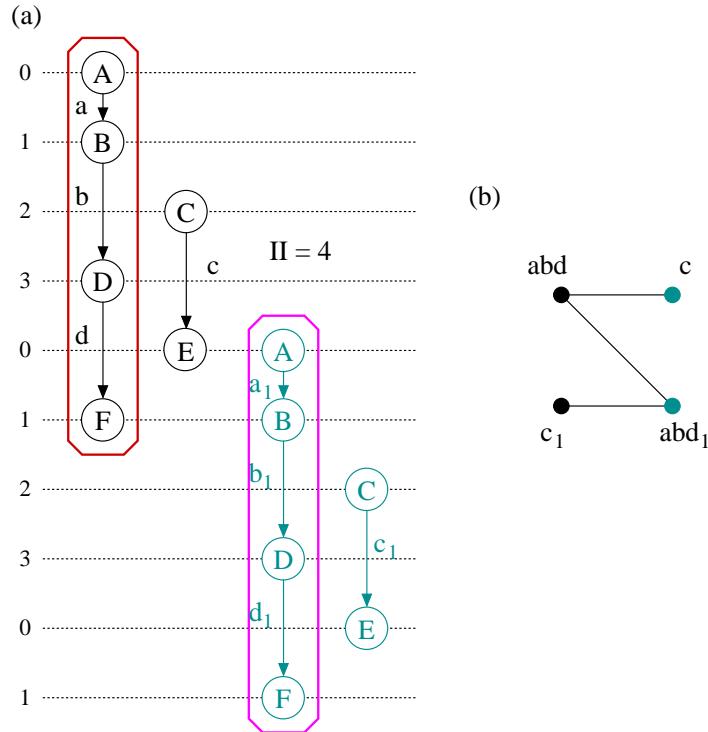


Figure 4.10: Value merging for a folded case. (a) The data flow graph with values *unfolded*. In this case it is possible to merge instances of values a , b and d . (b) CG after unfolding and merging with $\chi(CG) = 2$.

4.6.1 The importance of a good lower bound

A lower bound violation determines an infeasible case in the satisfaction process. When this violation is detected just before the process no solution can be found to the scheduling and register allocation problem, and no time is consumed. Meanwhile, if this is detected during the satisfaction process, a backtrack is initiated to undo the previous lifetime serialization(s). The sooner an infeasible case is detected the faster the solution space exploration, and the better the chance to find a solution in short time. Therefore, it is important to find an as tight as possible lower bound.

For example, look at the data flow graph of Figure 4.12a. This graph has a latency of five, and values are assigned to a register file RF with capacity $c(RF) = 2$. The worst-case conflict graph is shown in Figure 4.12b, and its chromatic number is $\chi(WCCG) = 4$. Therefore, some lifetime serialization has to be performed. The best-case conflict graph in Figure 4.12c has a chromatic number of $\chi(WCCG) = 2$, which allows the satisfaction process to continue. However,

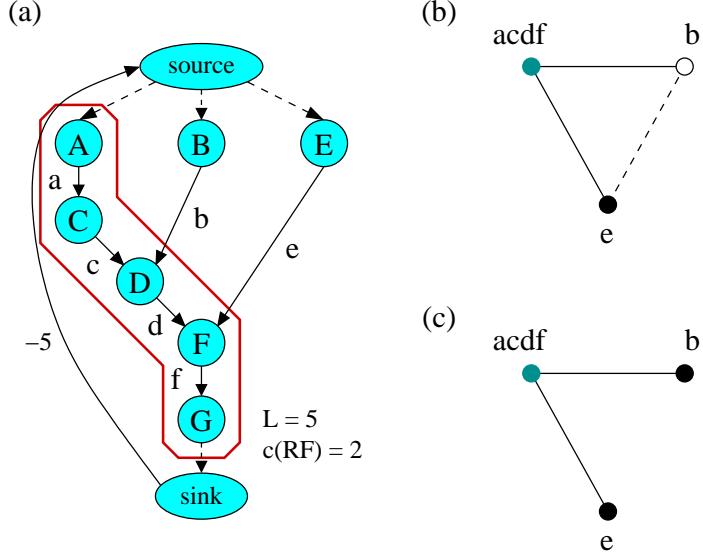


Figure 4.11: Resulting conflict graphs after value merging. (a) A non-folded *DFG* with merged values, (b) its *WCCG* with $\chi(WCCG) = 3$, and (c) its *SCG* with $\chi(WCCG) = 2$.

it is easy to verify that in the best case the minimum register requirement is three. If this would be detected at the beginning of the process an infeasible case would be returned saving time.

Value merging helps to improve the best-case conflict graphs and their coloring to find tighter lower bounds like in the example of Figure 4.7. Nevertheless, value merging cannot always be applied. From the example of Figure 3.7, repeated for convenience in Figure 4.13, it is shown that no value merging can be performed and the lower bound obtained continues to be inaccurate. Thus the need of a better lower bound estimation approach is obvious.

4.6.2 Lower bound register estimation over time intervals

Ohm in [59, 60] and Sharma in [71] present similar approaches aimed at estimating lower bounds of hardware resources needed to implement a behavioral description within a given amount of time (latency). The register estimation is based on the principle that if N objects are distributed over K slots, at least $\lceil N/K \rceil$ objects are assigned to the same slot within those K slots.

The weight of the value v denoted by W_v represents the *minimum lifetime* of v , i.e. the number of clock cycles v is guaranteed to be alive after being produced. If the weight of the value is determined, $\sum_v M_{v,Z}$ is computed for each interval

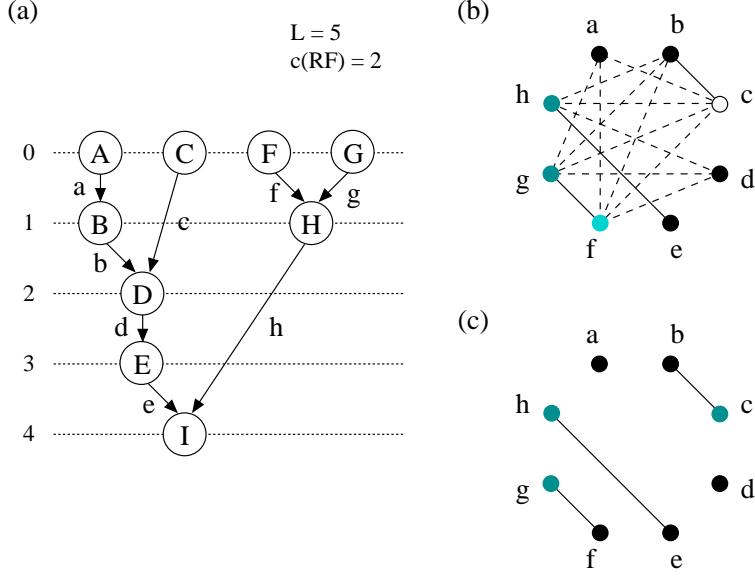


Figure 4.12: A case in which a tight lower bound is important. (a) *DFG* with values assigned to register file *RF*, (b) *WCCG* with $\chi(WCCG) = 4$ (b) *SCG* with $\chi(SCG) = 2$, when the tightest lower bound is three.

Z , where $M_{v,Z}$ is the portion of value v 's lifetime guaranteed to be included in interval Z , taken over all possible schedules.

Since $\sum_v M_{v,Z}$ is the number of registers required for interval Z , $\lceil (\sum_v M_{v,Z}) / |Z| \rceil$ represents the minimum number of registers required, i.e. the lower bound. This is computed for all intervals $Z \subseteq [0, L - 1]$ to obtain a tighter lower bound and then select its maximum as lower bound of the register count.

In order to optimize his approach, Ohm applies *fanout reduction* and *value merging*. Fanout reduction considers only the latest consumer of the value and is used to gain in performance. Value merging is used to refine the lower bound having longer lifetimes.

For the aim of this work, W_v is easily found as the maximum distance (taken from the distance matrix) between the producer and the consumer operations.

Ohm refines the register estimation to get a tighter lower bound [60]. In that case, the maximum number of values which can be active during some step s is restricted to the initial lower bound. The refined algorithm consists of tentatively schedule the producer operation of each value in its ASAP clock cycle. Then update that decision in the whole data flow graph, and check if the lower bound is altered with that tentative decision. If the lower bound is altered, the execution interval of the producer operation is refined excluding the initial scheduling point

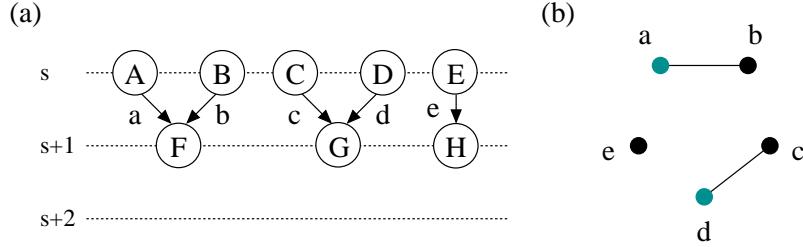


Figure 4.13: No value merging can be performed. (a) DFG , (b) SCG with $\chi(SCG) = 2$. The minimum requirement of registers is three.

assumed. After that the tentative schedule is undone. The same is applied for the value consumer operations but considering their ALAP execution times. After adjusting the time frames for every value, if there is an infeasible case caused by the pruning of the execution intervals, i.e. when the ASAP of some operation is greater than its ALAP, the assumed lower bound is incremented and the process continues until there is no variation.

4.6.3 Register estimation using probabilities

Moreno in [55] presents a method for register estimation in unscheduled data flow graphs. The strategy consists of calculating the probability that an edge (value) between two vertices (operations) crosses the boundary between two control steps (clock cycles). The method is based on a model that associates probabilities with the different scheduling alternatives of each vertex.

Assuming in this case a mapping from operations to resources, the method starts by calculating the probability for each operation Op to be executed in a clock cycle s .

The $Maximum_Concurrence(Op, s)$ is the maximum number of operations similar to Op (including Op itself) which will concur for the same functional unit in clock cycle s . This is calculated as the total number of the corresponding operation execution intervals that overlap in clock cycle s .

$Probability(Op, s)$ is the probability of an operation Op being scheduled in control step s and is calculated as follows:

$$Probability(Op, s) = \frac{Normalization_Factor(Op)}{Maximum_Concurrence(Op, s)}$$

where $s \in [ASAP(Op); ALAP(Op)]$, otherwise $Probability(Op, s) = 0$. The $Normalization_Factor(Op)$ is given by:

$$\text{Normalization_Factor}(\text{Op}) = \frac{1}{\sum_{s=\text{ASAP}(\text{Op})}^{\text{ALAP}(\text{Op})} \frac{1}{\text{Maximum_Concurrence}(\text{Op}, s)}}$$

so that $\sum_{s=\text{ASAP}(\text{Op})}^{\text{ALAP}(\text{Op})} \text{Probability}(\text{Op}, s) = 1$.

The probability of a value u produced by operation P^u and consumed by C^u of being alive between the boundary of control steps s_i and $s_i + 1$ is the sum of probabilities of P^u being scheduled in steps before and including s_i multiplied by the probability of C^u being scheduled after step s_i .

$$\text{Probability_Value}(u, s_i) =$$

$$\left(\sum_{s=\text{ASAP}(P^u)}^{s_i} \text{Probability}(P^u, s) \right) \times \left(\sum_{s=s_i+1}^{\text{ALAP}(C^u)} \text{Probability}(C^u, s) \right)$$

This approach lacks accuracy that depends on obtaining a tighter measure of *Maximum_Concurrence*.

In order to compare the different methods presented (including the coloring of the best-case conflict graph), Section 4.7.2 shows the results obtained by applying these methods in a group of examples.

4.7 Experimental Results Using FACTS

This section presents the experimental results obtained with the proposed capacity satisfaction method and heuristics implemented in FACTS for random-access register files.

The aim of the experiments in Section 4.7.1 is to compare the register file selection criteria described in Section 3.4. The second set of experiments in Section 4.7.2 tests the different lower bound register estimators from Section 4.6, considering the merging of values. Section 4.7.3 presents the results of the capacity satisfaction approach applied to tight constrained application examples. Finally, Section 4.7.4 presents the capacity satisfaction results using value merging and symmetry detection.

All experiments were run on a machine with a Pentium II processor running at 350 MHz.

Instances of the following examples were used:

- A fast discrete cosine transform (**fdct**) used in [73].
- A fast Fourier transform (**fft**) used in [50].
- A finite impulse response filter kernel (**fir**) found in [21] and translated from C to a data flow description.
- An inverse fast Fourier transform (**ifft**) generated by Mistral2 [79] at Philips Research Laboratories.
- An infinite impulse response filter kernel (**iir**) also found in [21] and translated from C to a data flow description.
- A Loeffler algorithm that performs an one-dimensional inverse discrete cosine transform (**loef**) from [48].
- A loop body of an industrial example (**loop**) that combines a fast Fourier transform with differential modulation [34].

Instances of each of these examples differ in the number of resources and timing constraints. The characteristics of the examples are shown in Table 4.1.

In the first column of the table the example names are listed. The second and third columns show the number of vertices (operations) and data edges (values) respectively of the corresponding data flow graphs.

$|FU|$ is a reference of the available instruction level parallelism or functional units, e.g. '1' means the availability of one adder, one multiplier, one load/store unit, '2' means that the initial number of functional units was doubled, and so on. $|SF|$ is the number of storage files of which capacities have to be satisfied.

The latency L shown in the sixth column is the minimum obtainable with the set of resource and data precedence constraints.

Finally in the last two columns the minimum obtainable initiation intervals are shown. II_R corresponds to the cases that use random-access registers or stacks, while II_{RR} corresponds to the cases that use rotating registers or fifos as storage.

4.7.1 Experiments with register file selection criteria

Table 4.2 shows the results of using two different criteria for register file selection during capacity satisfaction. The two criteria were presented in Section 3.4. The example instances considered here are thus the ones with distributed register files.

The first column of the table shows the instance name with the associated resource and timing constraints. The second column shows the minimum register file capacities c ($c(RF)$) obtained with the capacity satisfaction approach using any of the file selection methods.

Table 4.1: Example characteristics.

<i>DFG</i>	$ V $	$ E_d $	$ FU $	$ SF $	L	II_R	II_{RR}
fdct	42	34	1	1,2	18	—	—
			2		11	—	—
			4		8	—	—
fft	30	37	1	1,2	13	4	4
			2		11	3	2
fir	16	11	1	1	6	3	3
			2			—	2
			3			—	1
ifft	73	75	1	1,2	36	26	26
			2		23	14	14
iir	27	21	1	1	9	4	4
			2			—	2
			4			—	1
loef	46	48	1	3	28	—	—
			2		15	—	—
			4		10	—	—
loop	30	26	1	1	11	4	4
			2		7	2	2

The third column presents the results of the static storage file selection, i.e. selecting first the files with the highest priority from Expression 3.1. The capacity satisfaction process is once for each register file until reaching its goal for that selected storage file. After that, another file with high priority is selected. The third column is divided in two, one sub-column indicating if a solution was found and the other sub-column indicating the time in seconds spent to obtain the solution.

The fourth column presents the results of the dynamic storage file selection, i.e. alternating the file selection during the process by taking each iteration a storage file with the highest priority from Expression 3.2. This column also shows the cases in which a solution was found and the time required to find the solution.

The branch-and-bound satisfaction process was set to work with a limited amount of backtracks (up to 100). That means that if the number of backtracks has reached the limit an infeasible case is returned and no solution is found.

Evaluating the results from Table 4.2 it is observable that in most of the cases the approach using the dynamic file selection performed better in finding solutions, and in a comparable time for cases when both heuristics found a solution.

Table 4.2: Register file selection.

$DFG_{ FU , SF ,L,II}$	c	Expr. 3.1		Expr. 3.2	
		found?	$t(s)$	found?	$t(s)$
fdct _{1,2,18,-}	8,2	no	—	yes	0.87
fdct _{2,2,11,-}	6,2	yes	0.59	yes	0.45
fdct _{4,2,8,-}	8,4	yes	0.18	yes	0.17
fft _{1,2,13,-}	1,4	yes	0.10	yes	0.09
fft _{1,2,13,4}	4,6	yes	0.24	yes	0.24
fft _{2,2,11}	2,4	yes	0.08	yes	0.08
fft _{2,2,11,3}	6,6	yes	0.31	yes	0.22
ifft _{1,2,36,-}	5,4	no	—	yes	2.43
ifft _{1,2,36,26}	4,5	yes	8.40	no	—
ifft _{2,2,23,-}	4,4	no	—	yes	3.40
ifft _{2,2,23,14}	6,4	no	—	yes	4.63
loef _{1,3,28,-}	4,6,8	no	—	yes	1.85
loef _{2,3,15,-}	4,2,8	yes	1.24	yes	1.30
loef _{4,3,10,-}	3,5,8	no	—	yes	0.52

The approach using the static file selection could not find a solution in 40% of the cases (within the limited amount of backtracks allowed).

In conclusion, the effectiveness of the dynamic selection, plus its cheaper cost (only by reusing the chromatic numbers from the worst-case conflict graphs), have motivated its usage in the capacity satisfaction process for distributed storage files.

4.7.2 Comparison between lower bound algorithms

Another set of experiments was run in order to compare the accuracy of the lower bound estimation approaches presented in Section 4.6.

Table 4.3 shows the obtained results prior the scheduling and satisfaction process (when the most scheduling freedom exists), and considering instances with one register file and non folded cases. For loop folded cases the algorithms applied present relatively similar results in relation to the non-folded cases. For each approach two situations are considered: without and with value merging.

Following the column with the instance names and constraints, the second column shows the tightest lower bound (feasible minimum capacity) found by observation in small examples or by running the capacity satisfaction approach without time limitation, i.e. allowing the branch-and-bound algorithm to run as it was necessary to find a solution.

Table 4.3: Comparison of lower bound approaches.

$DFG_{ FU , SF , L }$	tightest	$\chi(SCG)$		Intervals		Probabilities	
		orig.	merg.	orig.	merg.	orig.	merg.
fdct _{1,1,18}	9	2	2	2	2	10	10
fdct _{2,1,11}	7	2	2	4	4	9	9
fdct _{4,1,8}	8	8	8	8	8	10	10
fft _{1,1,13}	5	2	4	2	4	6	5
fft _{2,1,11}	6	2	4	4	4	6	6
fir _{1,1,6}	3	2	2	3	3	4	4
ifft _{1,1,36}	7	3	3	3	3	12	9
ifft _{2,1,23}	7	3	6	4	5	11	10
iir _{1,1,9}	5	2	3	3	3	6	6
loop _{1,1,11}	5	4	4	4	4	8	8
loop _{2,1,7}	7	4	7	5	7	9	9

The first results correspond to the exact coloring of a best-case conflict graph $SCG(RF)$. The second ones are the results of applying Ohm's algorithm [59, 60] (from Section 4.6.2), and the third correspond to apply Moreno's algorithm [55] as described in Section 4.6.3.

Table 4.3 shows that the coloring of SCG and Ohm's algorithm present similar results, often lower than the tightest lower bounds. Those results were improved when value merging was applied, which indirectly reduces the scheduling freedom. Moreno's algorithm based on probabilities obtained poor results as lower bound. In most of the cases the estimation obtained was greater than the tightest lower bound.

Due to the small difference between the results using coloring of SCG or using the approach of Ohm, the general applicability of SCG , and in order to keep concordance between methods of finding bounds, the usage of best-case conflict graphs was preferred in the satisfaction approach presented in this thesis.

4.7.3 Experiments for register file capacity satisfaction

To evaluate the proposed capacity satisfaction method, it was applied to the examples of Table 4.1, and the results are shown in Table 4.4. This table shows first the results obtained by the resource constrained scheduler from [75] followed by a graph coloring based register allocation and assignment. c_{req} is the capacity of the register file required after scheduling. $t(s)$ is the CPU time in seconds spent

Table 4.4: Results of proposed method for random-access register files. Capacity satisfaction versus scheduling and allocation a posteriori.

$DFG_{ FU , SF , L, II}$	Sch. & Alloc.		Capacity satisfaction		
	c_{req}	$t(s)$	c	$t(s)$	mobility
fdct _{1,1,18,-}	10	0.34	9	0.75	9.52 → 0.17
fdct _{2,1,11,-}	8	0.39	8	0.87	2.90 → 0.07
fdct _{4,1,8,-}	10	0.21	8	0.20	0.76 → 0.24
fdct _{1,2,18,-}	8,4	0.34	8,2	0.87	9.52 → 0.71
fdct _{2,2,11,-}	6,4	0.39	6,2	0.45	2.90 → 0.31
fdct _{4,2,8,-}	10,4	0.21	8,4	0.17	0.76 → 0.24
fft _{1,1,13,-}	5	0.06	5	0.11	3.17 → 2.23
fft _{1,1,13,4}	10	0.11	10	0.12	2.30 → 0.00
fft _{2,1,11,-}	6	0.06	6	0.11	2.17 → 1.33
fft _{2,1,11,3}	15	0.12	12	0.12	1.20 → 0.23
fft _{1,2,13,-}	1,4	0.05	1,4	0.09	1.90 → 0.73
fft _{1,2,13,4}	4,6	0.10	4,6	0.24	2.30 → 0.00
fft _{2,2,11,-}	4,4	0.05	2,4	0.08	2.17 → 1.23
fft _{2,2,11,3}	8,7	0.12	6,6	0.22	1.20 → 0.23
ifft _{1,1,36,-}	13	1.82	7	2.84	13.9 → 1.12
ifft _{1,1,36,26}	13	3.39	8	5.56	13.9 → 1.21
ifft _{2,1,23,-}	11	1.69	7	1.79	6.34 → 0.86
ifft _{2,1,23,14}	10	2.71	10	3.42	6.30 → 0.82
ifft _{1,2,36,-}	7,8	1.80	5,4	2.43	13.9 → 1.25
ifft _{1,2,36,26}	7,8	3.38	5,5	4.06	13.9 → 1.23
ifft _{2,2,23,-}	6,6	1.68	4,4	3.40	6.34 → 0.82
ifft _{2,2,23,14}	6,6	2.70	6,4	4.63	6.30 → 0.48
loef _{1,3,28,-}	8,8,9	0.95	4,6,8	1.85	14.4 → 1.61
loef _{2,3,15,-}	8,8,8	0.75	4,2,8	1.30	6.11 → 0.32
loef _{4,3,10,-}	8,8,8	0.67	3,5,8	0.52	2.43 → 0.20
loop _{1,1,11,-}	9	0.05	5	0.13	4.40 → 1.67
loop _{1,1,11,4}	12	0.14	10	0.40	4.00 → 1.00
loop _{2,1,7,-}	11	0.05	7	0.11	2.00 → 1.33
loop _{2,1,7,2}	17	0.12	15	0.21	1.60 → 0.60

Table 4.5: Capacity and timing constraints tradeoff.

$DFG_{ FU , SF }$	L	II	c	$t(s)$	mobility
fir _{1,1}	6	3	6	0.02	1.44 → 1.44
		4	5	0.02	1.56 → 0.75
		5	4	0.02	1.56 → 1.31
		–	3	0.01	1.56 → 0.38
	9	–	2	0.04	4.75 → 0.69
iir _{1,1}	9	4	8	0.27	1.56 → 0.22
		5	7	0.14	2.07 → 0.15
		6	6	0.09	2.07 → 0.19
		–	5	0.06	2.07 → 0.15
	11	–	4	0.64	4.15 → 0.70

Table 4.6: Distributed register file capacities tradeoff.

$DFG_{ FU , SF , L}$	c	$t(s)$	mobility
loef _{1,3,28}	3,4,10	2.02	14.4 → 1.75
	4,6,8	1.84	14.4 → 1.61
	5,7,6	2.10	14.4 → 0.89
loef _{2,3,15}	4,2,8	1.30	6.10 → 0.32
	4,6,7	1.08	6.10 → 0.36
loef _{4,3,10}	2,4,10	0.84	2.43 → 0.21
	3,5,8	0.51	2.43 → 0.20
	3,6,7	2.78	2.43 → 0.21

by the scheduling and register allocation.

The third column shows the results obtained by applying capacity satisfaction, scheduling and register allocation. This column shows the minimum capacities c ($c(RF)$) for which our approach could find a feasible solution. The CPU time in seconds $t(s)$ spent to find the solution (including scheduling and register allocation), and the impact of access ordering on the schedule freedom (using the mobility defined in Section 2.4.1) are shown in the last two sub-columns. The numbers before and after the arrow denote the mobility before and after the satisfiability process respectively.

From the results of Table 4.4 the following aspects can be observed:

- By taking the register file capacities into account this method is able to

reduce the register pressure compared to the approach that schedules and performs register allocation *a posteriori*. For the instance $\text{loef}_{2,3,15,-}$ this results in a reduction from 24 (eight in all files) to 14 (four, two and eight) in the total number of registers.

- The initial mobility of instance $\text{fft}_{1,1,13,-}$ (of 3.17) is larger than the one of instance $\text{fft}_{1,2,13,-}$ (of 1.90). Instances $\text{fft}_{1,1,13,-}$ and $\text{fft}_{1,2,13,-}$ differ only in the number of register files in the architecture and their initial scheduling freedom should be the same. The difference on the initial mobility of those instances is a good example of the effect of constraint analysis. It is seen that one of the register files of instance $\text{fft}_{1,2,13,-}$ has only one register available (see column c of table). Therefore, storage constraint analysis (presented in Section 2.5.2) was already able to make serialization decisions with the values assigned to that register. Those decisions affected the initial scheduling freedom prior capacity satisfaction process.
- The number of infeasible cases encountered during the satisfaction process was usually small. This shows the effectiveness of our approach on reaching the feasible solution space. For these experiments, a maximum of 100 backtracks were allowed and an infeasible result (no solution found) was returned when reached that limit. Therefore, the time overhead introduced by the satisfaction process is minimum. It is seen in the table that the time to perform direct scheduling and register allocation is comparable to the time spent for the whole satisfaction process (including final scheduling and register allocation), and in cases like for instances $\text{fdct}_{4,2,8,-}$, and $\text{loef}_{4,3,10,-}$, the whole satisfaction process spent less time. This proves that the intensive use of constraint analysis, update of the distance matrix and, furthermore the use of conflict graphs have a little impact on the scheduling time for data flow graphs with less than 100 operations. Moreover, the satisfaction process actually helps scheduling by pruning the search space with the addition of constraints.
- The use of distributed register files does not add complexity to the approach. In many examples, capacity satisfaction run times for single and distributed register files were comparable, and in some cases using distributed files speeded up the process.

Table 4.5 shows that it is possible to use our approach to make tradeoffs between criteria like timing and capacity constraints. The results obtained for the fir and iir examples show that it is possible to make tradeoffs between the capacity and the latency or the initiation interval constraints. By weakening the timing

Table 4.7: Results applying value merging and symmetry detection.

				breaking symmetries	
		no merging	merging	no merging	merging
$DFG_{ FU , SF ,L}$	c	$t(s)$	$t(s)$	$t(s)$	$t(s)$
fdct _{1,2,18}	8,2	0.87	1.01	nsf	0.82
	7,2	nsf	nsf	nsf	1.27
fdct _{2,2,11}	6,2	0.45	nsf	0.51	0.50
	8,4	0.17	0.14	0.33	0.19
fft _{1,1,13}	5	0.11	0.09	—	—
	6	0.11	0.08	—	—
ifft _{1,1,36}	7	2.84	2.48	4.40	2.29
	7	1.79	2.42	1.68	1.07
loef _{1,3,28}	4,6,8	1.85	nsf	1.33	1.25
	3,2,7	nsf	nsf	nsf	1.45
loef _{2,3,15}	4,2,8	1.30	nsf	0.80	0.89
	3,5,8	0.52	nsf	0.51	0.59
loef _{4,3,10}	2,4,6	nsf	nsf	nsf	0.58

constraints, register file capacities can be further reduced. This is useful for architectural synthesis.

The proposed method can also help to reduce the register requirement for register files with low capacity by means of increasing the number of registers available in other register files. This is seen for the loef instances in Table 4.6, which shows the solutions found making tradeoffs among the capacities of the register files. When the capacity is tight in one file the satisfaction process uses the available freedom of the other files to find a solution for the first. Therefore, a relaxation of the constraint of some file can allow the constraint satisfaction of others. This is important for handling heterogeneous register file architectures. Traditional methods lack the capability of making tradeoffs between different register files. Furthermore, this also shows that the method can be used to do design space exploration for storage file architectures.

4.7.4 Experiments with value merging and symmetry detection

Value merging (Section 4.5) and symmetry detection (Section 2.5.3) help to speed up the satisfaction process.

Symmetry detection and breaking the detected symmetries in data flow graphs can help our approach to find solutions more efficiently and in shorter time, since

it prunes the scheduling search space while preserving feasibility. This process is applied once before the satisfaction process starts (also preceding value merging) and its result is evaluated with constraint analysis.

For comparison reasons Table 4.7 presents first the original capacity satisfaction results c ($c(RF)$) from Table 4.4. Then the results in terms of time $t(s)$ by applying value merging and symmetry detection are shown.

Only non folded instances for which value merging or symmetry detection have changed the final result are shown in the table. For the fft examples symmetries were not detected, so only the original and the solutions using value merging are shown. “nsf” means “no solution found” in less than 100 iterations of the satisfaction process.

From Table 4.7 it is seen that although value merging or symmetry detection can in fact reduce the time of finding a solution, when applied separately there are cases in which a solution cannot be found like for $fdct_{1,2,18}$ (capacities of eight and two), $fdct_{2,2,11}$, and loef examples.

The best situations are the ones in which symmetry detection and value merging are used together, as seen in the last column of the table. In addition, it was possible to find a solution for tighter capacities, especially in the case of instances $fdct_{1,2,18}$ (capacities of seven and two), $loef_{1,3,28}$ (capacities three, two and seven), and $loef_{4,3,10}$ (capacities two, four and six) for which no solution was found using without symmetry detection, or value merging.

4.8 Rotating Register Files

Using random-access registers the lifetime of values is upper bound by the initiation interval II . With reduced II s, software techniques like modulo variable expansion of Lam [40], or hardware resources like *relative location storage* [78] or *rotating register files* [69] have to be used.

This section presents the constraint satisfaction technique for rotating register files [2], a type of storage intended to obtain the maximum usage of resources during the execution of tight timing constrained program loops.

Consider the example of Figure 4.14 (borrowed from [69]). This example consists of a loop with a load and an add operations, which are scheduled in clock cycles s and $s + 2$. The figure shows the execution of two consecutive iterations of the loop. The initiation interval is $II = 1$.

Assuming that values are assigned to random-access registers, the load operation executed in cycle s will load a value in register r_b . The lifetime of this value is extended to clock cycle $s + 2$ when it is consumed by the add operation. However, one cycle later in $s + 1$ the load operation will be executed again and will overwrite the value in r_b while it is alive, thereby inserting a wrong result that will

clock cycle	iteration i	iteration $i + 1$
s	$r_b = \text{load } (r_a);$	
$s + 1$...	$r_b = \text{load } (r_a);$
$s + 2$	$r_d = \text{add } (r_b, 7);$...
$s + 3$		$r_d = \text{add } (r_b, 7);$

Figure 4.14: Example of overlapped lifetimes, $II = 1$ and random-access registers are used as storage.

be used by the add operation in $s + 2$. Therefore, it is seen that random-access registers have limitations when it is tried to make an aggressive pipeline [40].

In order to exploit better the available ILP by means of reducing the initiation interval, the rotating register files [69] were introduced.

Rotating register files also have a limited capacity and, depending on the application and resource constraints, the problem consists on performing scheduling and register allocation such that all those constraints are satisfied. Therefore, our approach is adapted in this context.

4.8.1 Storage model

In Figure 4.15 the assumed rotating register file RRF is shown. The total number of registers of this file is $c(c(RRF))$.

Consider a value u assigned to rotating register file RRF . Conceptually, once u has been assigned to a register, the register address specified in an instruction $I(u)$ (source or target) is added to the *base* to derive the physical register address in the file. The physical register address $P(u)$ is the modulus of the previous addition and the file capacity, i.e.

$$P(u) = (base + I(u)) \bmod c(RRF)$$

Once a new loop iteration starts the base address is decremented by one.

For the example presented in Figure 4.14, using a rotating register file with $c(RRF) = 8$, the load operation specifies rotating register $rr3$ as its target, as shown in Figure 4.16. In the i^{th} iteration it writes to the physical register number three. At the beginning of a new iteration the *base* is decremented. As a consequence, the load operation in the next iteration writes to physical register two, although it specifies $rr3$ as the target. Therefore, distinct registers are given for the lifetimes of a value in successive iterations.

Since *base* is decremented when a new loop iteration starts, the proper offsets in the producer and consumer operations must be maintained. For example, the

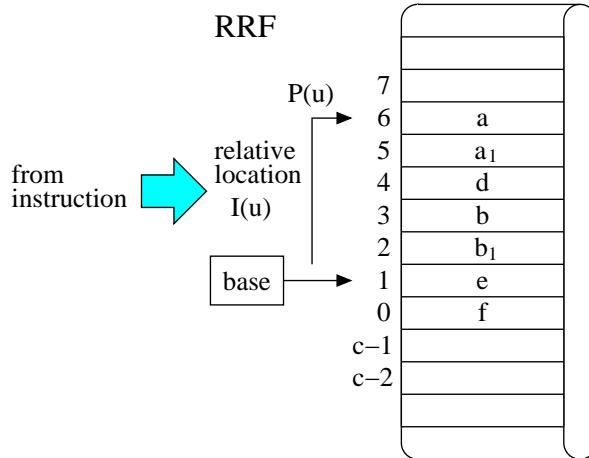


Figure 4.15: A rotating register file.

clock cycle	iteration i	iteration $i + 1$	physical registers accessed
s	$rr3 = \text{load}(r_a);$		$(0 + 3) \bmod 8 = 3$
$s + 1$...	$rr3 = \text{load}(r_a);$	$(0 - 1 + 3) \bmod 8 = 2$
$s + 2$	$r_d = \text{add}(rr5, 7);$...	$(0 - 2 + 5) \bmod 8 = 3$
$s + 3$		$r_d = \text{add}(rr5, 7);$	$(0 - 3 + 5) \bmod 8 = 2$

Figure 4.16: The use of rotating registers. $II = 1$, and $c(RRF) = 8$.

add operation specifies $rr5$ (and not $rr3$) as the source operand in order to access the value produced by the load, because the add operation is scheduled two clock cycles later after the load operation and $base$ was decremented twice between the two operations.

In conclusion, rotating register files provide a dynamic register renaming under the control of the compiler.

4.8.2 Constructing conflict graphs

Like it is performed for random-access registers, the capacity satisfaction process uses a worst-case conflict graph $WCCG(RRF)$ and a best-case conflict graph $SCG(RRF)$ for values assigned to a rotating register file RRF .

$WCCG$ and SCG are constructed based on specific rules like the ones for random-access registers, with the difference that multiple value instances must be

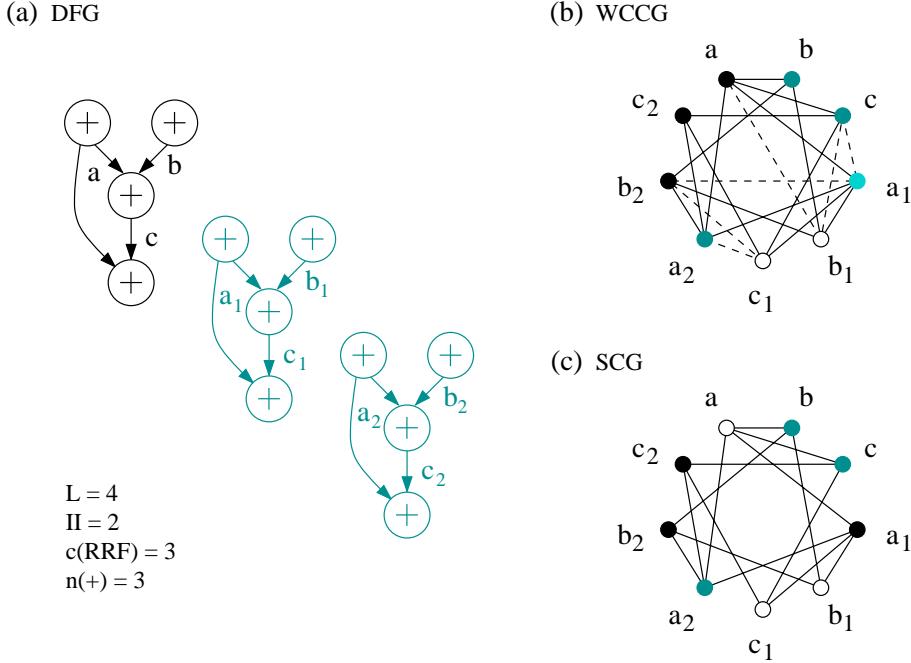


Figure 4.17: Example of graphs for rotating register files. In order to construct *WCCG* and *SCG*, multiple instances of values in *DFG* have been generated. (a) *DFG* and copies to illustrate the overlap between value instances of successive loop iterations. (b) *WCCG* with vertices representing value instances. Dashed edges in conflict graph represent weak conflicts. (c) *SCG*.

considered for loop folded cases.

Multiple value instances in a conflict graph

In loop folded schedules, value lifetimes from different loop iterations overlap, i.e. several instances of values assigned to rotating register files can coexist in some clock cycle.

Since the register requirement is modeled with conflict graphs, value instances from successive loop iterations require a corresponding vertex representation in those graphs (see Figure 4.17).

The number of vertices representing the instances of one value is related to the longest value lifetime and the initiation interval:

$$\text{number_instances} = \left\lceil \frac{-d_m(C^u, P^u) + \delta(P^u) - 1}{II} \right\rceil + 1 \quad (4.4)$$

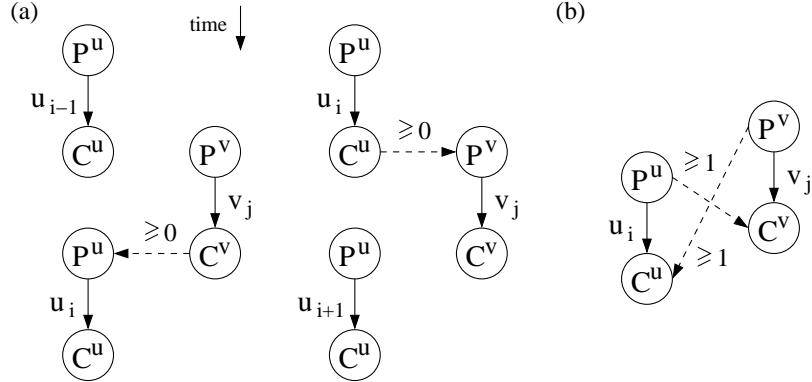


Figure 4.18: Instances u_i and v_j (a) have no conflict, (b) have a strong conflict.

This number is based on the maximum number of overlaps among the lifetimes of values, and it is upper bound by $\lceil (L - 1)/II \rceil + 1$ which considers the longest possible lifetime in a data flow graph. It is clear that the run-time complexity for constructing conflict graphs will be incremented by a factor of $number_instances^2$, and the extra amount of vertices can make coloring a dominant run-time factor in the satisfaction process (refer to Section 3.9).

Conflict rules

No conflict. Instances u_i and v_j have no conflict if their lifetimes can never overlap. The no conflict situation is depicted graphically in Figure 4.18a and captured by the following proposition:

Proposition 4.5 *Instances u_i and v_j have no conflict if $d(\text{Cons}_i^u, \text{Prod}_j^v) \geq 0$ or $d(\text{Cons}_j^v, \text{Prod}_i^u) \geq 0$.*

Because the information from the distance matrix is used for checking a conflict, the equivalent Proposition 4.6 is derived.

Proposition 4.6 *Instances u_i and v_j have no conflict if*

$$\begin{aligned} d_m(C^u, P^v) + \delta(P^v) - 1 &\geq (i - j) \times II \quad \text{or} \\ d_m(C^v, P^u) + \delta(P^u) - 1 &\geq (j - i) \times II \end{aligned}$$

Strong conflict. Instances u_i and v_j have a strong conflict if their lifetimes overlap for sure. The strong conflict situation is depicted graphically in Figure 4.18b and captured by the following proposition:

Proposition 4.7 *Instances u_i and v_j have strong conflict if $d(\text{Prod}_i^u, \text{Cons}_j^v) \geq 1$ and $d(\text{Prod}_j^v, \text{Cons}_i^u) \geq 1$.*

Because the information from the distance matrix is used for checking a conflict, the equivalent Proposition 4.8 is derived.

Proposition 4.8 *Values u_i and v_j have a strong conflict if*

$$\begin{aligned} d_m(P^u, C^v) - \delta(P^u) &\geq (i - j) \times II \quad \text{and} \\ d_m(P^v, C^u) - \delta(P^v) &\geq (j - i) \times II \end{aligned}$$

Additionally, instances u_i and u_j of the same value u always have a strong conflict, because the accessing mechanism of a rotating register file assigns different locations for each value instance. This is formalized with the following proposition:

Proposition 4.9 *Instances u_i and u_j always have a strong conflict.*

Weak conflict. Two value instances u_i and v_j have a weak conflict if Propositions 4.6 (no conflicts), and Propositions 4.8 and 4.9 (strong conflicts) are invalid.

Consider the data flow graph of Figure 4.17a, e.g. instances b_1 and c_1 have no conflict, instances a_2 and b_2 have a strong conflict in the clock cycle that the operation that consumes both executes, and instances c_1 and a_2 have a weak conflict because it is not yet determined whether their lifetimes overlap or not. In Figure 4.17b the corresponding worst-case conflict graph $WCCG$ is shown. The number of instances considered per value is three according to Expression 4.4. Coloring this graph results in a chromatic number of four, while the rotating register file capacity is three, therefore some lifetime serialization has to be performed. In Figure 4.17c, the strong conflict graph SCG is shown. In the example, edges (a, b_1) , (c, a_1) , (c, b_1) , (a_1, b_2) , (c_1, a_2) and (c_1, b_2) can be bottlenecks. Then, instance a_1 is chosen first because its saturation and degree number of three and six respectively, and instance c because it has a weak conflict relation with a_1 .

4.8.3 Lifetime serialization

There are two ways that lifetimes of u_i and v_j can be serialized: the lifetime of u_i precedes lifetime of v_j or vice versa. Often, the distance relations between producer and consumer operations exclude one possibility. In Figure 4.19, a distance $d(P_i^u, C_j^v) > 0$ determines the only possible serialization $C_i^u \rightarrow P_j^v$.

Lifetime serialization $\text{Cons}_i^u \rightarrow \text{Prod}_j^v$ is performed with sequence edge:

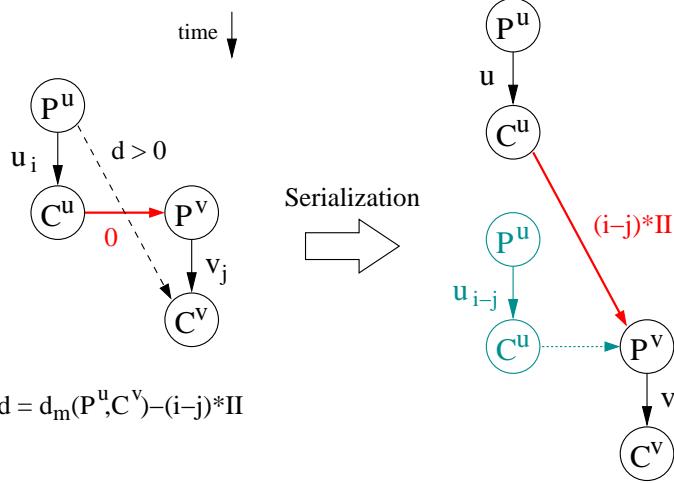


Figure 4.19: Lifetime serialization of instances u_i and v_j . (a) The dashed edge with weight d shows the condition that determines this serialization, (b) the actual sequence edge $C^u \rightarrow P^v$ of weight $(i - j) \times II$ is inserted in the DFG.

$$\text{Cons}^u \rightarrow \text{Prod}^v : \quad w(C^u, P^v) = (i - j) \times II - \delta(P^v) + 1 \quad (4.5)$$

$\text{Cons}_j^v \rightarrow \text{Prod}_i^u$ is performed with sequence edge:

$$\text{Cons}^v \rightarrow \text{Prod}^u : \quad w(C^v, P^u) = (j - i) \times II - \delta(P^u) + 1 \quad (4.6)$$

Figure 4.20 shows the result from lifetime serialization of c and a_1 . The updated worse-case conflict graph in Figure 4.20d requires only three colors, which matches the capacity of three.

4.8.4 Experiments with rotating register files

The set of experiments tests the capabilities of the constraint satisfaction approach to find solutions for tight constrained examples, and using rotating registers as storage.

In the first column of Table 4.8 instance names have the format $DFG_{|SF|,L}$, where $|SF| = 1$ and L is the latency. The second column shows the initiation interval II which is the minimum obtainable with the available ILP ($|FU|$) shown in the next column. Initiation intervals with an '*' are the minimum obtainable using random-access registers.

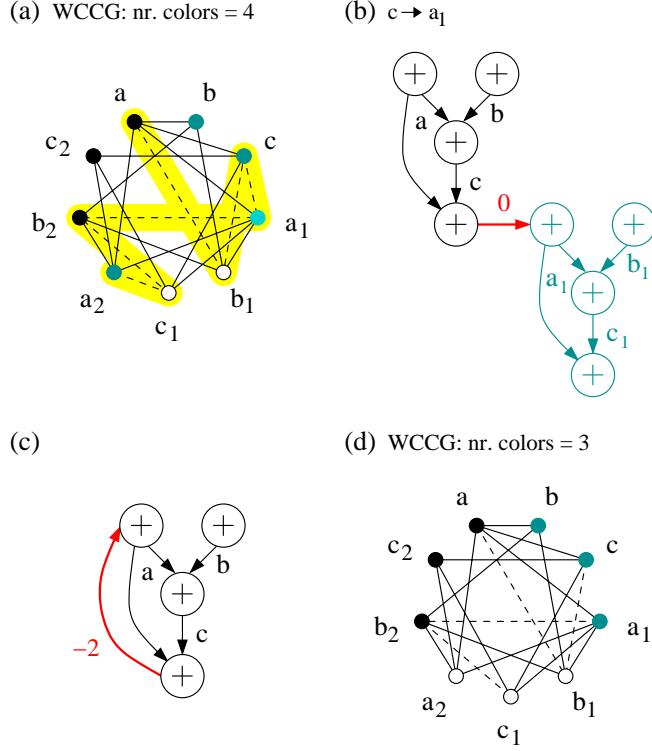


Figure 4.20: Results for the example of Figure 4.17. (a) *WCCG* with bottlenecks (shaded region). (b) Serialization of value instances c and a_1 . (c) Actual sequence edge added in *DFG* to serialize $c - a_1$. (d) Updated *WCCG*.

Next, the table shows the results obtained by the resource constrained scheduler from [75] followed by the rotating register assignment approach from [78]. c_{req} is the capacity of the file required after scheduling. The CPU time in seconds $t(s)$ spent by scheduling and register allocation and assignment is also shown.

The last part of the table shows the results of applying storage constraint satisfaction. The resource constrained scheduler was used to complete the partial schedule resulting from the satisfaction process. The column c shows also the minimum capacities ($c(RRF)$) for which our approach could find a feasible solution. #ins corresponds to the initial number of instances per value in the conflict graphs from Expression 4.4. The CPU time in seconds $t(s)$ spent to find the solution (including scheduling and register assignment) and the impact of access ordering on the schedule freedom of the operations are shown in the last two columns. The number before and after the arrow denote the mobility before and after the satisfaction process respectively.

The results from Table 4.8 show that our approach is able to satisfy the con-

Table 4.8: Results for rotating register files. Capacity satisfaction versus scheduling and allocation a posteriori.

			Sch. & Alloc.		Capacity satisfaction			
$DFG_{ SF ,L}$	$ FU $	II	c_{req}	$t(s)$	c	#ins	$t(s)$	mobility
fft _{1,13}	1	4*	11	0.09	9	4	0.40	3.10 → 0.00
	2	3*	16	0.10	12	5	0.60	2.17 → 0.17
	2	2	18	0.09	17	6	1.06	2.17 → 0.00
fir _{1,6}	1	3*	6	0.01	6	3	0.05	1.44 → 0.48
	2	2	9	0.01	8	4	0.02	1.94 → 0.19
	3	1	15	0.01	15	6	0.73	1.94 → 0.19
ifft _{1,36}	1	26*	13	1.73	12	3	2.46	13.9 → 4.16
iir _{1,9}	1	4*	9	0.05	9	3	0.18	1.59 → 0.33
	2	2	19	0.06	15	5	0.25	2.22 → 0.11
	4	1	32	0.04	32	9	2.11	2.22 → 0.07
loop _{1,11}	1	4*	17	0.07	16	4	263.	4.40 → 1.00
	2	2*	24	0.07	15	4	0.54	2.00 → 0.60

(*) Indicates the minimum obtainable II using random-access registers.

straints of rotating register files. By taking the rotating register file capacity into account, this method is able to reduce the register pressure compared to the approach that performs register allocation a posteriori. For loop_{1,7} with $II = 2$ this results in a reduction from 24 to 15 in the total number of registers.

Additionally, the use of rotating registers offers the opportunity to reduce the initiation intervals. Our approach is also able to find solutions with tighter timing constraints. This is shown in the table specifically for fft, fir and iir examples.

Considering run times, it is seen that although the increment of the number of vertices in the conflict graphs is significant, capacity satisfaction run times are still acceptable. In the case of loop_{1,11} with $II = 4$ the CPU time spent by the constraint satisfaction is two orders of magnitude (263 seconds) longer than the other cases. In that example, coloring $WCCG$ with the algorithm of Coudert [18] was the critical run-time factor in the satisfaction process (for more information refer to Section 3.9).

Until now, we have studied the cases of registers in which the access to values is random. Next chapter deals with other types of storage in which the access of values is in a (predefined) sequential way. The capacity satisfaction approach is then applied with some little adaptations.

Chapter 5

Sequential-Access Storage

5.1 Introduction

In the previous chapter, register files were considered for capacity constraint satisfaction. In this chapter, the storage satisfaction method is applied to storage files with multiple registers that are accessed in a sequential way. Particularly, storage files with stacks or fifos are considered for which the number of units and registers per unit are the constraints to satisfy.

Stacks

A *stack* consists of multiple registers sharing a read/write port and one address. The main characteristic of a stack is that the first value to be read is the last stored.

Processors using stacks as storage are commonly known as *stack machines* [37]. Stack machines are efficient at running certain types of programs than random-access register-based ones, particularly programs that are based in “recursions”. Stack machines can be simpler than other machines, and provide computational power using little hardware. A favorable application area for stack machines is in real time embedded control applications, which require a combination of small size, high processing speed, and support for interrupt handling. Procedure calls are cheaper on stack machines.

The 4stack processor [62] uses stack based instructions for a four data-path VLIW processor and DSP applications. Those instructions with implicit addressing reduce instruction memory bandwidth, since multiple registers can be pointed with one address, an advantage over register-based machines.

However, it will be seen later that the characteristics of the DSP kernels used and the architecture template (VLIW with distributed storage files with stacks) are not necessarily favorable for the exploitation of stack features. Nevertheless,

some value access ordering can be performed in order to obtain some reduction in the stack requirement and exploitation of their multiple registers.

Fifos

A *fifo* consists also of multiple registers or locations sharing read and write ports and one address. The main characteristic of a fifo is that the first value to be read from the storage is the first stored. The regularity of value accesses presented on many DSP computations makes embedded system designers think about the use of queues or fifo-like storage in their ASIPs in order to save area, power, overhead of memory address generation or encoding [6].

In the case of multimedia processors (VLIW) code size can be critical. Taking into account that the main contributor of code size is register addressing, an alternative is the use of fifos which offers a larger storage capacity for the same addressing cost. Besides, for pipeline-scheduled loops queue-like storage offers the advantage of having value lifetimes longer than the initiation interval [23], just like rotating registers.

The drawback of stacks and fifos is their lack of flexibility in comparison with register files because compilers are not prepared to deal with their strict access behaviors. This introduces the problem of phase coupling between scheduling and storage allocation even more severely than the case of scheduling and register allocation.

In basic blocks, which can be data intensive, value accesses have to be ordered in a way to guarantee the use of no more than the available storage units. The satisfaction process in the case of stacks and fifos has to deal not only with the interaction of timing, resource, and available units (capacity) constraints, but also with the number of registers in each storage unit or *size* [5].

This chapter introduces the stack and fifo models in Section 5.2. Section 5.3 presents the extended satisfaction approach for storage files with stacks or fifos. In Section 5.4, the rules to construct conflict graphs for values assigned to stacks or fifos are presented. Access ordering for capacity satisfaction is described in Section 5.5. Section 5.6 explains the construction of conflict graphs, bottleneck identification, and solutions applied for unit size satisfaction. Finally, in Section 5.7 storage constraint satisfaction results are presented.

5.2 Storage Model

In Figure 5.1a, a storage file SF with stacks is shown. Storage file SF has c ($c(SF)$) stacks available for assignment. Each unit in file SF corresponds to a

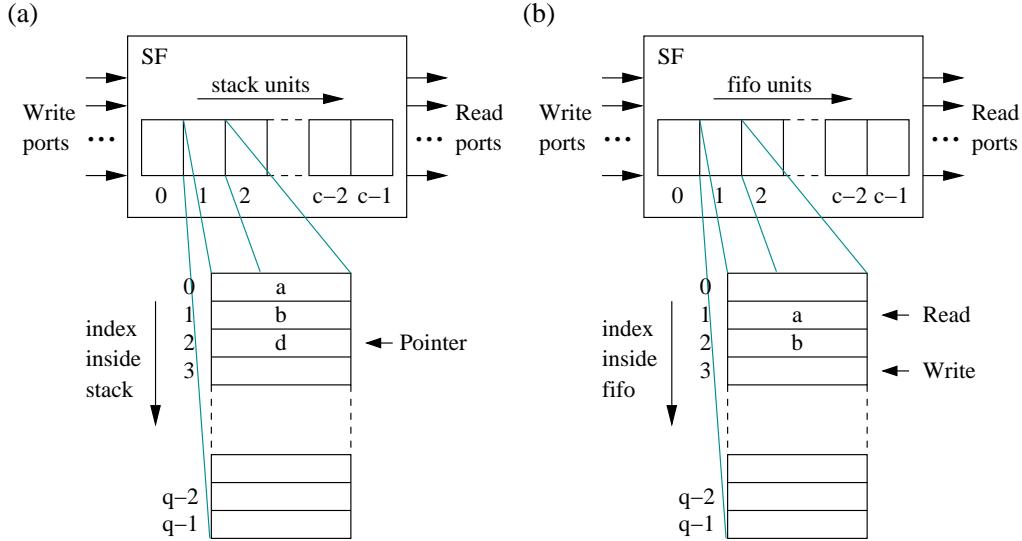


Figure 5.1: Storage file with (a) stacks, (b) fifos.

stack with q ($q(SF)$) registers.

Stacks organize register accesses as last-in-first-out. Addressing of registers is implicit through the stack pointer, so register numbers are not encoded in the instruction word. Before a value is written into a stack its pointer is incremented to point to the next empty register. A value read from the stack is fetched from the last index and its pointer is decremented. It is assumed also that one stack allows one read or one write access per clock cycle.

In Figure 5.1b, a storage file SF with fifos is shown. Storage file SF has c ($c(SF)$) fifos available for assignment. Each unit in the file corresponds to a fifo with q ($q(SF)$) registers.

Once a value is written in a fifo at some index its write pointer is incremented to point to the next empty register. A value read from the fifo is fetched from a lower index (no empty register) and its read pointer is incremented. It is assumed also that one fifo allows one read or one write access per clock cycle. The important characteristic of fifos regarding loop folded cases is that the lifetime of values stored in fifos can be longer than the initiation interval constraint II . This is because a value that belongs to the next iteration would be written at a different pointed register inside the fifo.

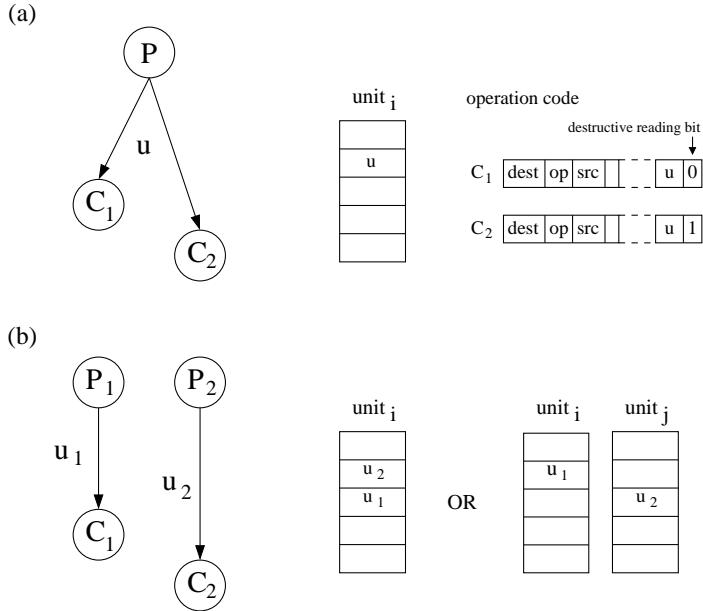


Figure 5.2: Solutions for multiple consumption of values assigned to stacks or fifos: (a) using (non-) destructive reading bits in the instruction, or (b) creating multiple copies of the values.

5.2.1 Multiple consumption of values

One problem that arises using stacks or fifos is the multiple reading or consumption of values. As described in the previous section, a value read from a stack is fetched from the last index and the stack pointer is decremented. A value read from a fifo is fetched and the read pointer is incremented to point to another register. Therefore, a value read once cannot be read again if another operation in another clock cycle needs to access it.

With multiple consumption of values assigned to stacks or fifos two alternatives are available:

- A control bit in the instruction word can indicate whether the read access is “destructive” (the value will not be read again) or not, affecting the decrement of the pointer. This can be seen in Figure 5.2a.

If a multiple consumption of one value happens at the same clock cycle, the connection network between storage files and functional units is assumed to guarantee the broadcasting of that value.

As a consequence, the compiler must take care that all readings (consumptions) of one value will precede all readings of any other value assigned to

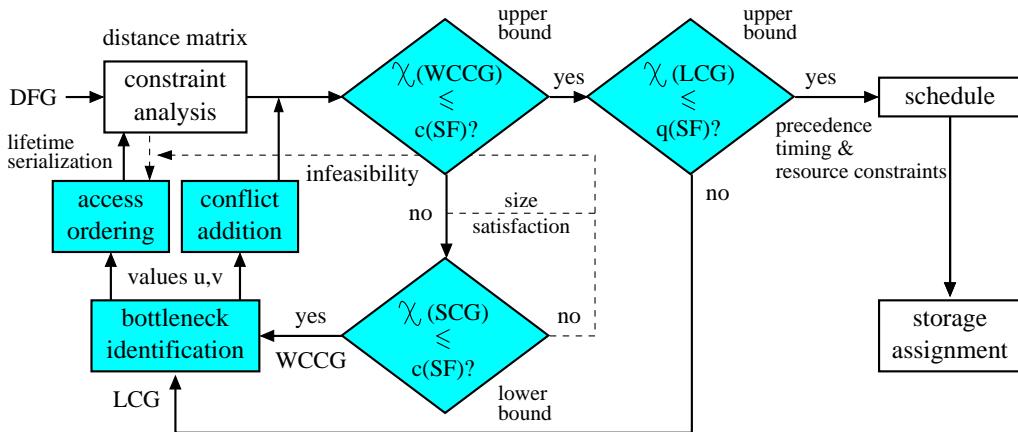


Figure 5.3: Approach for stacks and fifos.

the same stack or fifo.

- A multiple consumption of values can be avoided by generating copies of those values by copying their producer operations as many times as consumers exist. This is depicted in Figure 5.2b, and this alternative will create new values written and read only once, changing the original *DFG*.

Those new value copies can be stored in the same storage for which some access ordering would be needed, or in different storage units. However, with this alternative the number of additional operations can grow exponentially violating timing, resource and even storage constraints. Therefore, designers might be forced to look for convenient tradeoffs among constraints.

The approach implemented in FACTS deals with multiple consumptions in general and can be easily tuned to any of the two alternatives.

5.3 Satisfaction Approach for Stacks and Fifos

The storage satisfaction approach presented in Chapter 3 is extended now for the case of storage files with stacks or fifos. This is because not only the capacity $c(SF)$, but also the size $q(SF)$ is a constraint to satisfy. The approach is depicted graphically in Figure 5.3.

When the worst-case unit requirement for each storage file already respects their available capacity ($\chi(WCCG) \leq c(SF)$), the data flow graph is transferred to the storage unit size satisfaction.

In order to prevent a stack or fifo of size $q(SF)$ from overloading, no more than $q(SF)$ values can be simultaneously alive in the storage unit. To satisfy this requirement the *worst-case size conflict graph* $LCG(SF)$ is used. The coloring of LCG gives an upper bound of the number of registers required for the units in storage file SF .

When the worst case location requirement for each storage unit already respects their size ($\chi(LCG) \leq q(SF)$), the data flow graph with the additional precedences is transferred to the schedule and the storage assignment processes. If not, a bottleneck identification is performed to find the values that can overload a storage unit.

In order to reduce the storage size requirement *lifetime serialization* is performed between values if possible. Serializing lifetimes will guarantee that one value will be consumed before the other is produced, hence using only one register. The constraint analysis checks afterwards the feasibility of this solution.

If serialization cannot be performed, a strong conflict at the capacity level (in $WCCG$) is forced in order to make the storage allocation process to assign those values to different units (*conflict addition*). The forced conflict is preferable between values with a weak conflict. If not, the impact of the conflict addition is checked with a new coloring of $WCCG$, and the capacity $c(SF)$ is compared with the new $\chi(WCCG)$ in order to preserve the capacity satisfaction. If the new $\chi(WCCG)$ is larger than $c(SF)$ an infeasible case is returned and another pair of values is chosen.

Instead of conflict addition, lifetime serialization is the preferred solution in our approach since it exploits the remaining schedule freedom to satisfy also the size constraint.

5.4 Conflict Graphs for Capacity Satisfaction

The capacity satisfaction process shown in Figure 5.3 uses two conflict graphs for values assigned to storage file SF with stacks or fifos: a worst-case conflict graph $WCCG(SF)$ and a best-case conflict graph $SCG(SF)$.

$WCCG$ and SCG are constructed based on specific rules for accessing values assigned to storage files with stacks or fifos. The access conflict rules are presented in the following sections.

Consider values u and v produced by operations P^u and P^v and consumed by C^u and C^v respectively. $Prod^u$ and $Prod^v$ represent the productions of u and v respectively, while $Cons^u$ and $Cons^v$ represent their consumptions. For loop folded cases, instances u_i and v_j correspond to values u and v in iterations i and j respectively.

The following propositions consider the loop folded cases in general. For the non-folded cases simply consider $II = L$ and $i = j = k = 0$.

5.4.1 Stack access conflict rules

No stack conflict

Stacks can store values of which lifetimes appear in sequence, therefore two values have no stack conflict if their lifetimes are serialized (Proposition 4.2 presented in Section 4.2.1). This situation is shown in Figure 5.4a and repeated again for convenience in Proposition 5.1.

Proposition 5.1 *Values u and v have no stack conflict if there exists $k \in \mathbb{N}$ such that*

$$\begin{aligned} d_m(C^u, P^v) + \delta(P^v) - 1 &\geq k \times II & \text{and} \\ d_m(C^v, P^u) + \delta(P^u) - 1 &\geq -(k+1) \times II \end{aligned}$$

In addition, values u and v have no stack conflict if the order of their productions opposes the order of their consumptions. The following proposition formalizes this:

Proposition 5.2 *Values u and v have no stack conflict if there exist iterations i and j such that $(d(\text{Prod}_i^u, \text{Prod}_j^v) \geq 1 \text{ and } d(\text{Cons}_j^v, \text{Cons}_i^u) \geq 1)$ or $(d(\text{Prod}_j^v, \text{Prod}_i^u) \geq 1 \text{ and } d(\text{Cons}_i^u, \text{Cons}_j^v) \geq 1)$.*

The possible non-conflict cases from Proposition 5.2 are depicted graphically in Figures 5.4b and 5.4c. Because the distance matrix is used for checking a conflict, the equivalent Proposition 5.3 is derived.

Proposition 5.3 *Values u and v have no stack conflict if there exists $k \in \mathbb{N}$ such that*

$$\begin{aligned} \left(\begin{array}{l} d_m(P^u, P^v) + \delta(P^v) - \delta(P^u) \geq k \times II + 1 \\ d_m(C^v, C^u) \geq -k \times II + 1 \end{array} \right) \text{ or} \\ \left(\begin{array}{l} d_m(P^v, P^u) + \delta(P^u) - \delta(P^v) \geq k \times II + 1 \\ d_m(C^u, C^v) \geq -k \times II + 1 \end{array} \right) \end{aligned}$$

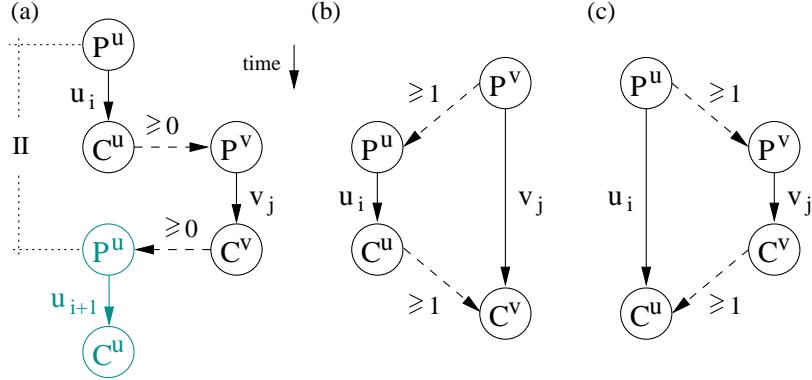


Figure 5.4: u and v have no stack conflict. (a) Serialized value lifetimes, (b),(c) typical stack access behavior.

Strong stack conflict

Values u and v have a strong stack conflict if their lifetimes overlap and the order of their productions matches the order of their consumptions. This order of accesses corresponds to typical fifo-like access behavior. Additionally, because single-port stacks are considered values simultaneously produced or consumed cannot be assigned to the same stack, and therefore they have a strong conflict. Both conditions are grouped, depicted graphically in Figure 5.5a, and formalized with the following proposition:

Proposition 5.4 *Values u and v have a strong stack conflict if there exist iterations i and j such that $(d(\text{Prod}_j^v, \text{Cons}_i^u) \geq 1 \text{ and } d(\text{Prod}_i^u, \text{Prod}_j^v) \geq 0 \text{ and } d(\text{Cons}_i^u, \text{Cons}_j^v) \geq 0)$ or $(d(\text{Prod}_i^u, \text{Cons}_j^v) \geq 1 \text{ and } d(\text{Prod}_j^v, \text{Prod}_i^u) \geq 0 \text{ and } d(\text{Cons}_j^v, \text{Cons}_i^u) \geq 0)$.*

Because the distance matrix is used for checking a conflict the equivalent Proposition 5.5 is derived.

Proposition 5.5 *Values u and v have a strong stack conflict if there exists $k \in \mathbb{N}$ such that*

$$\begin{aligned}
 & \left(\begin{array}{l} d_m(P^v, C^u) - \delta(P^v) \geq -k \times II \\ d_m(P^u, P^v) + \delta(P^v) - \delta(P^u) \geq k \times II \\ d_m(C^u, C^v) \geq k \times II \end{array} \right) \text{ or} \\
 & \left(\begin{array}{l} d_m(P^u, C^v) - \delta(P^u) \geq -k \times II \\ d_m(P^v, P^u) + \delta(P^u) - \delta(P^v) \geq k \times II \\ d_m(C^v, C^u) \geq k \times II \end{array} \right)
 \end{aligned}$$

A strong conflict also occurs when the productions or consumptions of two values that apparently do not overlap, happen simultaneously when loop folded, i.e. they would be scheduled at the same clock cycle. This is depicted in Figure 5.5b.

Proposition 5.6 *Values u and v have a strong stack conflict if there exist iterations i and j such that $d(\text{Prod}_i^u, \text{Prod}_j^v) = 0$ or $d(\text{Cons}_i^u, \text{Cons}_j^v) = 0$.*

From the distance matrix, Proposition 5.7 is derived.

Proposition 5.7 *Values u and v have a strong stack conflict if there exists $k \in \mathbb{N}$ such that*

$$d_m(P^u, P^v) + \delta(P^v) - \delta(P^u) = k \times II \quad \text{or} \quad d_m(C^u, C^v) = k \times II$$

Additionally, for multiple consumption of values, assuming that a value in a stack can be read multiple times in different clock cycles implies that all consumptions of one value must precede all consumptions of the other. As depicted in Figure 5.5c values u and v have a strong conflict if one consumer of u precedes one of v and one consumer of v precedes one of u .

Proposition 5.8 *Values u and v have a strong stack conflict if there exist consumer operations $\text{Cons_a}_i^u, \text{Cons_b}_i^u, \text{Cons_x}_i^v, \text{Cons_y}_i^v$ and iterations i and j such that $d(\text{Cons_a}_i^u, \text{Cons_x}_j^v) \geq 0$ and $d(\text{Cons_y}_j^v, \text{Cons_b}_i^u) \geq 0$.*

From the distance matrix, Proposition 5.9 is derived.

Proposition 5.9 *Values u and v have a strong stack conflict if there exist consumer operations $\text{Ca}_i^u, \text{Cb}_i^u, \text{Cx}_i^v, \text{Cy}_i^v$ and $k \in \mathbb{N}$ such that*

$$d_m(\text{Ca}_i^u, \text{Cx}_i^v) \geq k \times II \quad \text{and} \quad d_m(\text{Cy}_i^v, \text{Cb}_i^u) \geq -k \times II$$

Weak stack conflict

Two values u and v have a weak stack conflict if Propositions 5.1, 5.3 (no stack conflict) and Propositions 5.5, 5.7, 5.9 (strong stack conflicts) are invalid.

WCCG is constructed just by applying the no stack conflict conditions. If they are false, either a weak- or a strong stack conflict situation occurs. Pairs of values linked by an edge in *WCCG* have a weak stack conflict if the strong

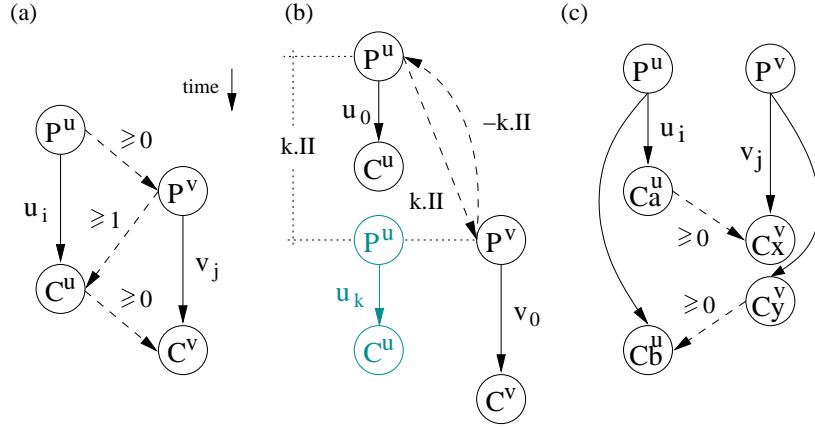


Figure 5.5: u and v have strong stack conflict. (a) Typical fifo-like access behavior, (b) overlap of productions (or consumptions) of values, (c) alternated consumption of values.

conflict condition is false. SCG is constructed considering only the strong stack conflict conditions.

In Figure 5.6 for stack access compatibility, e.g. values r and b have no conflict, a and r have strong conflict in the clock cycle that operation A executes, and values r and s have a weak conflict because it is not yet determined whether or not operation F precedes D .

5.4.2 Fifo access conflict rules

No fifo conflict

Values u and v have no fifo conflict if the order of their productions matches the order of their consumptions. This situation is depicted in Figure 5.7 and formalized with the following proposition:

Proposition 5.10 *Values u and v have no fifo conflict if there exist iterations i and j such that $d(\text{Prod}_i^u, \text{Prod}_j^v) \geq 1$ and $d(\text{Cons}_i^u, \text{Cons}_j^v) \geq 1$ and $d(\text{Prod}_j^v, \text{Prod}_i^u) \geq -II + 1$ and $d(\text{Cons}_j^v, \text{Cons}_i^u) \geq -II + 1$.*

Because the distance matrix is used for checking a conflict, the equivalent Proposition 5.11 is derived.

Proposition 5.11 *Values u and v have no fifo conflict if there exists $k \in \mathbb{N}$ such that*

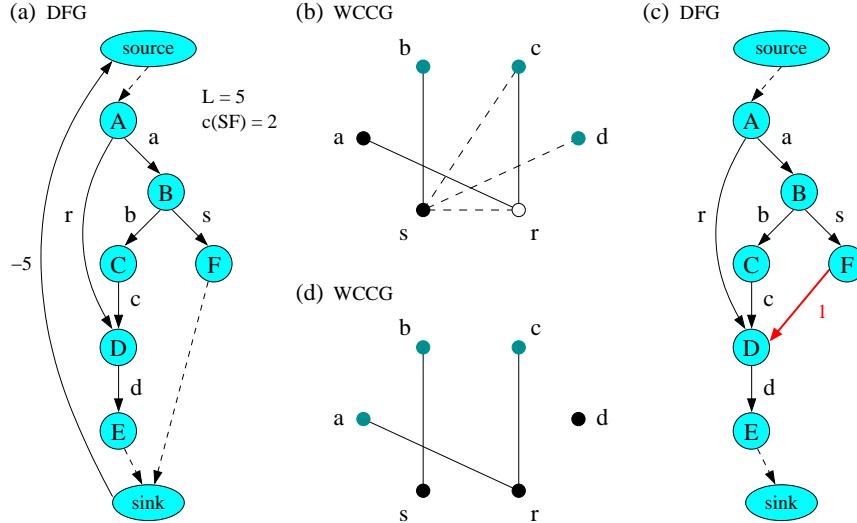


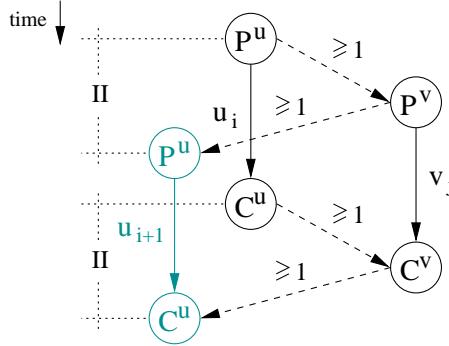
Figure 5.6: Data flow and conflict graphs for values assigned to stacks. (a) *DFG*, (b) initial *WCCG*, dashed edges represent weak conflicts, (c) access ordering of values r and s with sequence edge $F \rightarrow D$, $w(F, D) = 1$, (d) resulting *WCCG* after access ordering.

$$\begin{aligned}
 d_m(P^u, P^v) + \delta(P^v) - \delta(P^u) &\geq k \times II + 1 & \text{and} \\
 d_m(P^v, P^u) + \delta(P^u) - \delta(P^v) &\geq -(k+1) \times II + 1 & \text{and} \\
 d_m(C^u, C^v) &\geq k \times II + 1 & \text{and} \\
 d_m(C^v, C^u) &\geq -(k+1) \times II + 1
 \end{aligned}$$

Strong fifo conflict

Values u and v have a strong fifo conflict if the order of their productions opposes the order of their consumptions. This order of accesses corresponds to typical stack-like access behavior. Additionally, because single-port fifos are considered values simultaneously produced or consumed cannot be bound to the same fifo, and therefore they have a strong conflict. Both conditions are grouped, depicted graphically in Figure 5.8a and formalized with the following proposition:

Proposition 5.12 *Values u and v have a strong fifo conflict if there exist iterations i and j such that $(d(\text{Prod}_i^u, \text{Prod}_j^v) \geq 0 \text{ and } d(\text{Cons}_j^v, \text{Cons}_i^u) \geq 0)$ or $(d(\text{Prod}_j^v, \text{Prod}_i^u) \geq 0 \text{ and } d(\text{Cons}_i^u, \text{Cons}_j^v) \geq 0)$.*

Figure 5.7: u and v have no fifo conflict.

Because the distance matrix is used for checking a conflict, the equivalent Proposition 5.13 is derived.

Proposition 5.13 *Values u and v have a strong fifo conflict if there exists $k \in \mathbb{N}$ such that*

$$\begin{aligned}
 & \left(\begin{array}{l} d_m(P^u, P^v) + \delta(P^v) - \delta(P^u) \geq k \times II \\ d_m(C^v, C^u) \geq -k \times II \end{array} \right) \text{ or} \\
 & \left(\begin{array}{l} d_m(P^v, P^u) + \delta(P^u) - \delta(P^v) \geq k \times II \\ d_m(C^u, C^v) \geq -k \times II \end{array} \right)
 \end{aligned}$$

Similar to the cases for strong stack conflicts, Proposition 5.7 related to time overlap between the execution of operations from different iterations, and Proposition 5.9 related to alternated consumptions of values, constitute also strong conflicts for values assigned to fifos. These situations are depicted graphically in Figures 5.8b and 5.8c respectively.

Weak fifo conflict

Two values u and v have a weak fifo conflict if Proposition 5.11 (no fifo conflict), and Propositions 5.13, 5.7, 5.9 (strong fifo conflicts) are invalid.

$WCCG$ is constructed just by applying the no fifo conflict conditions. If they are false, either a weak- or a strong fifo conflict situation occurs. Pairs of values linked by an edge in $WCCG$ have a weak fifo conflict if the strong conflict condition is false. SCG is constructed considering only the strong fifo conflict conditions.

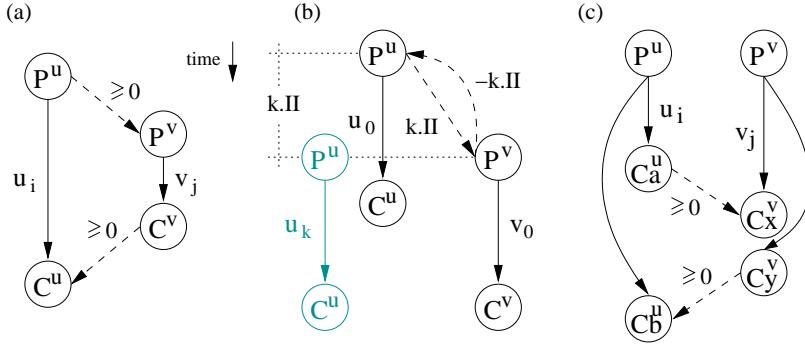


Figure 5.8: u and v have strong fifo conflict. (a) Typical stack-like access behavior, (b) overlap of productions (or consumptions) of values, (c) alternated consumption of values.

Consider Figure 5.9 for fifo access compatibility, e.g. values a and b have no conflict, a and r have strong conflict in the clock cycle that operation A executes, and values r and s have a weak conflict because it is not yet determined whether or not operation D precedes F .

5.5 Access Ordering

Following the bottleneck identification (Figure 5.3), values u and v are selected for access ordering.

Considering the fact that in the loop folded case one iteration of value v can be access ordered with any other iteration of value u , it is necessary thus to find the possible “places” related to u iterations where v can fit without leading to infeasibility. I.e. the lower and upper bounds of u iterations with which v can be ordered. After access ordering, the effect on the storage file pressure is checked using an updated $WCCG$ with a new coloring.

5.5.1 Stack access ordering

Ordering of u and v can be done in one of the three ways presented in Figure 5.10. According to those possible ways, three intervals of iterations i are derived. The first interval is related to serialization of value lifetimes as presented in Section 4.3 which covers the case shown in Figure 5.10a:

$$\left\lfloor \frac{d_m(P^u, C^v) - \delta(P^u)}{II} \right\rfloor \leq i \leq \left\lfloor \frac{-d_m(P^v, C^u) + \delta(P^v) - 1}{II} \right\rfloor \quad (5.1)$$

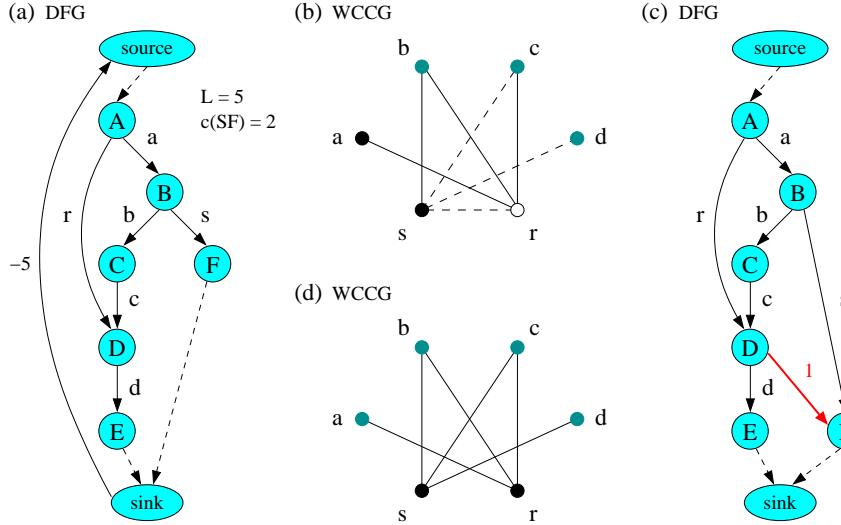


Figure 5.9: Data flow and conflict graphs for values assigned to fifos. (a) *DFG*, (b) initial *WCCG*, dashed edges represent weak conflicts, (c) access ordering of values r and s with sequence edge $D \rightarrow F$, $w(D, F) = 1$, (d) resulting *WCCG* after access ordering.

The second interval is found considering the ordering in which the lifetime of value u is inside the lifetime of value v as shown in Figure 5.10b.

It is tried to order them by fitting instance $k + i$ of u inside instance k of v . For productions, to put Prod_k^v before Prod_{k+i}^u a sequence edge with weight one is introduced from Prod_k^v to Prod_{k+i}^u . To prevent a positive weight cycle between operations which results in an infeasible situation (refer to Section 2.5.4) it is required that: $d(\text{Prod}_{k+i}^u, \text{Prod}_k^v) \leq -1$. Similarly, for consumptions, to put Cons_{k+i}^u before Cons_k^v it is required that: $d(\text{Cons}_k^v, \text{Cons}_{k+i}^u) \leq -1$. From both inequalities and following the process similar to the one for registers in Section 4.3.2 the second interval of i is:

$$\left\lfloor \frac{d_m(P^u, P^v) + \delta(P^v) - \delta(P^u)}{II} \right\rfloor + 1 \leq i \leq \left\lfloor \frac{-d_m(C^v, C^u) - 1}{II} \right\rfloor \quad (5.2)$$

The third interval is found considering the ordering in which the lifetime of value v is inside the lifetime of value u as shown in Figure 5.10c. It is tried to order them by fitting instance k of v inside instance $k + i$ of u . To put Prod_{k+i}^u before Prod_k^v it is required that: $d(\text{Prod}_k^v, \text{Prod}_{k+i}^u) \leq -1$. Also, to put Cons_k^v before Cons_{k+i}^u the last distance requirement is obtained: $d(\text{Cons}_{k+i}^u, \text{Cons}_k^v) \leq -1$. From both inequalities the third interval of i is:

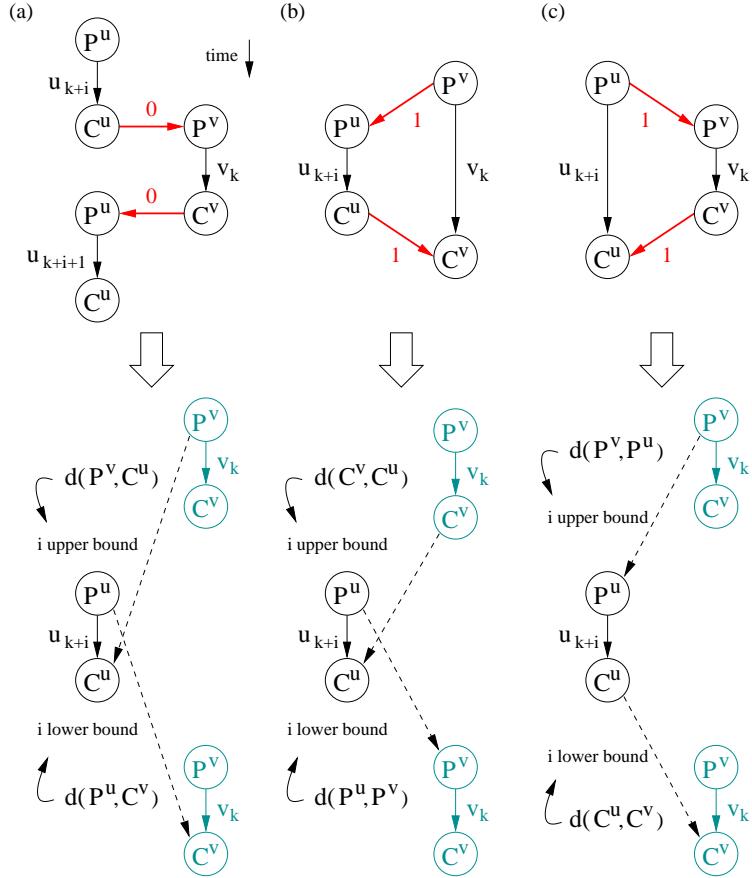


Figure 5.10: Stack ordering of u and v . (a) Lifetime serialization, i is bound by Expression 5.1, (b) lifetime of value u inside lifetime of v , i is bound by Expression 5.2, and (c) lifetime of value v inside lifetime of u , i is bound by Expression 5.3.

$$\left\lfloor \frac{d_m(C^u, C^v)}{II} \right\rfloor + 1 \leq i \leq \left\lfloor \frac{-d_m(P^v, P^u) - \delta(P^u) + \delta(P^v) - 1}{II} \right\rfloor \quad (5.3)$$

The minimum and maximum values of i from Expressions 5.1, 5.2 and 5.3 are the lower and upper bounds respectively of u iterations that can be access ordered with iteration zero of value v . For non-folded cases i can take values of -1 and 0 (while $II = L$).

Performing stack access ordering

The value access ordering process starts with the minimum value of i . The way access ordering is performed depends on whether i is between the bounds of the respective expression. Whenever there is more than one option the process proceeds as follows.

- The first ordering tried is lifetime serialization and is performed following Expression 4.3. The reason for trying lifetime serialization first is that it allows a better reutilization of registers in the stack unit.
- The second ordering tried is by making the lifetime of u being inside of the lifetime of v , inserting in the data flow graph the sequence edges:

$$\begin{aligned} \text{Prod}^v \rightarrow \text{Prod}^u : & \quad w(P^v, P^u) = i \times II + \delta(P^v) - \delta(P^u) + 1 \\ \text{Cons}^u \rightarrow \text{Cons}^v : & \quad w(C^u, C^v) = -i \times II + 1 \end{aligned} \quad (5.4)$$

- Finally, the last ordering tried is by making the lifetime of v being inside of the lifetime of u inserting the sequence edges:

$$\begin{aligned} \text{Prod}^u \rightarrow \text{Prod}^v : & \quad w(P^u, P^v) = i \times II + \delta(P^u) - \delta(P^v) + 1 \\ \text{Cons}^v \rightarrow \text{Cons}^u : & \quad w(C^v, C^u) = -i \times II + 1 \end{aligned} \quad (5.5)$$

The value access ordering starts ordering access of u and v with the minimum value of i . Once an access ordering is made, the constraint analysis or the lower bound checking validate the choice. If any of them detects infeasibility as a result of the decision, another type of ordering is made with the same value of i if possible. If none works, then i is incremented and orderings in subsequent iterations are tried. If all possible values of i lead to infeasibility, ordering of u and v is discarded and another pair of values is chosen to repeat the process.

5.5.2 Fifo access ordering

Ordering of u and v can be done in two ways: the production and consumption of value u precede respectively the production and consumption of value v or vice versa. As shown in Figure 5.11, it is tried to order them by fitting instance k of v between instances $k + i$ and $k + i + 1$ of u . For productions, to put Prod_k^v before Prod_{k+i+1}^u a sequence edge with weight one is introduced from Prod_k^v to Prod_{k+i+1}^u . To prevent a positive weight cycle between operations

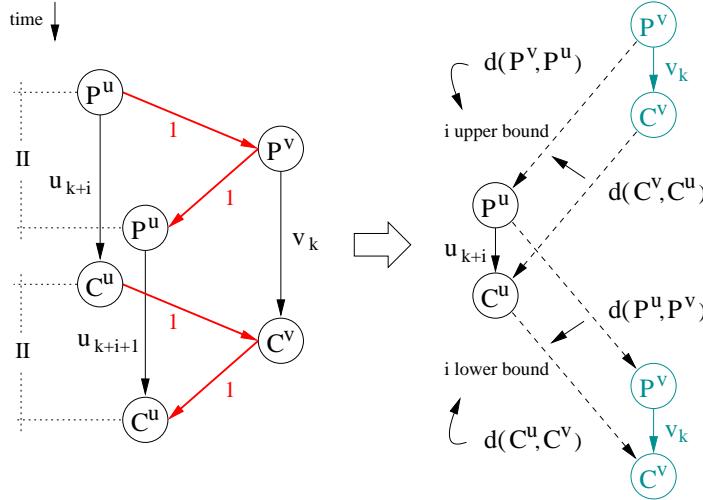


Figure 5.11: Fifo ordering of u and v . i is bound by Expressions 5.6 and 5.7.

which results in an infeasible situation (refer to Section 2.5.4), the distance from Prod_{k+i+1}^u to Prod_k^v must be less than zero, resulting in the following inequality: $d(\text{Prod}_{k+i+1}^u, \text{Prod}_k^v) \leq -1$. For consumptions, to put Cons_k^v before Cons_{k+i+1}^u the distance requirement is: $d(\text{Cons}_{k+i+1}^u, \text{Cons}_k^v) \leq -1$. From both inequalities and following the process similar to the one performed for registers in Section 4.3.2, it is obtained:

$$\max \left(\left\lfloor \frac{d_m(P^u, P^v) + \delta(P^v) - \delta(P^u)}{II} \right\rfloor, \left\lfloor \frac{d_m(C^u, C^v)}{II} \right\rfloor \right) \leq i \quad (5.6)$$

The maximum value from expression 5.6 gives the lower bound of value u iteration that can be access ordered with iteration zero of value v .

Similarly, to put Prod_{k+i}^u before Prod_k^v if is required that: $d(\text{Prod}_k^v, \text{Prod}_{k+i}^u) \leq -1$. Also, to put Cons_{k+i}^u before Cons_k^v the last distance requirement is: $d(\text{Cons}_k^v, \text{Cons}_{k+i}^u) \leq -1$. From both inequalities it is obtained:

$$i \leq \min \left(\left\lfloor \frac{-d_m(P^v, P^u) + \delta(P^u) - \delta(P^v) - 1}{II} \right\rfloor, \left\lfloor \frac{-d_m(C^v, C^u) - 1}{II} \right\rfloor \right) \quad (5.7)$$

The minimum value from expression 5.7 gives the upper bound of value u iteration that can be access ordered with iteration zero of value v . For non-folded cases i can take values of -1 and 0 (while $II = L$).

Performing fifo access ordering

With a known iteration number i , access ordering of values u and v assigned to fifos is performed inserting the following sequence edges in the data flow graph:

$$\begin{aligned} \text{Prod}^u \rightarrow \text{Prod}^v : \quad w(P^u, P^v) &= i \times II + \delta(P^u) - \delta(P^v) + 1 \\ \text{Prod}^v \rightarrow \text{Prod}^u : \quad w(P^v, P^u) &= -(i+1) \times II + \delta(P^v) - \delta(P^u) + 1 \\ \text{Cons}^u \rightarrow \text{Cons}^v : \quad w(C^u, C^v) &= i \times II + 1 \\ \text{Cons}^v \rightarrow \text{Cons}^u : \quad w(C^v, C^u) &= -(i+1) \times II + 1 \end{aligned} \quad (5.8)$$

The value access ordering starts ordering access of u and v with the minimum value of i . Once an access ordering is made, the constraint analysis or the lower bound checking validate the choice. If any of them detects infeasibility as a result of the decision, i is incremented and orderings in subsequent iterations are tried. If all possible values of i lead to infeasibility, ordering of u and v is discarded and another pair of values is chosen to repeat the process.

5.6 Conflict Graphs for Unit Size Satisfaction

After capacity satisfaction there is still the possibility for any storage unit to be overloaded. This can occur when a number of values that can potentially be assigned to the same unit are simultaneously alive, and would demand more registers than the size $q(SF)$ of the stacks or fifos.

The following analysis consists of using again the distance matrix to construct another worst-case conflict graph for size satisfaction LCG . The bottleneck identification is based on the coloring of LCG and the respective values are then candidates for lifetime serialization or for being assigned to different storage units.

Since values should appear in sequence in order to reuse the same register in a stack or fifo, the term *sequential access conflict* is used.

5.6.1 Sequential access conflict rules for stacks

The following propositions consider the loop folded cases in general. For the non-folded cases simply consider $II = L$ and $i = j = k = 0$.

No sequential conflict

Two values u and v have no sequential conflict if they have a strong stack access conflict, i.e. they will be assigned to different stacks anyway, or their lifetimes are serialized. The following proposition formalizes this:

Proposition 5.14 *Values u and v assigned to stacks have no sequential conflict if Propositions 5.6 or 5.8 are valid or there exist iterations i and j such that $(d(\text{Prod}_i^u, \text{Prod}_j^v) \geq 0 \text{ and } d(\text{Cons}_i^u, \text{Cons}_j^v) \geq 0)$ or $(d(\text{Prod}_j^v, \text{Prod}_i^u) \geq 0 \text{ and } d(\text{Cons}_j^v, \text{Cons}_i^u) \geq 0)$.*

Because the distance matrix is used for checking a conflict, the equivalent Proposition 5.15 is derived.

Proposition 5.15 *Values u and v assigned to stacks have no sequential conflict if Propositions 5.7 or 5.9 are valid or if there exists $k \in \mathbb{N}$ such that*

$$\begin{array}{lll} (d_m(P^u, P^v) + \delta(P^v) - \delta(P^u) \geq k \times II \text{ and} \\ d_m(C^u, C^v) \geq k \times II) \text{ or} \\ (d_m(P^v, P^u) + \delta(P^u) - \delta(P^v) \geq k \times II \text{ and} \\ d_m(C^v, C^u) \geq k \times II) \end{array}$$

Strong sequential conflict

Two values have a strong sequential conflict if they have a no stack conflict and their lifetimes overlap. Those values will require different registers if assigned to the same stack. The following proposition formalizes this:

Proposition 5.16 *Values u and v assigned to stacks have strong sequential conflict if Proposition 5.3 (no stack conflict), and Proposition 4.4 (related to lifetimes overlap) are valid.*

Weak sequential conflict

Two values u and v assigned to stacks have a weak sequential conflict if Proposition 5.15 (no sequential conflict) and Proposition 5.16 (strong sequential conflict) are invalid.

Values with a weak sequential conflict can have their lifetimes serialized, and values with a strong sequential conflict could be assigned to different stacks by explicitly adding a conflict in $WCCG$ between them if the updated $\chi(WCCG)$ is not increased as result of that.

Consider in Figure 5.12 the data flow and conflict graphs of a non-folded example during the stack satisfaction process. The capacity of two stacks is already satisfied with the addition of sequence edges of weight one, this result is visualized in the worst-case conflict graph $WCCG$ of Figure 5.12b. For size satisfaction, as shown in the conflict graph of Figure 5.12c, values b and e have no sequential

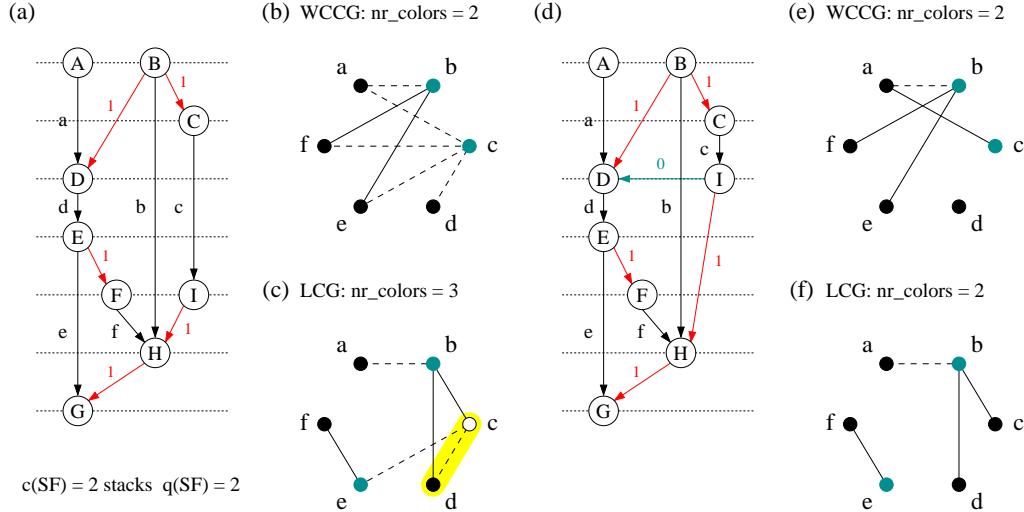


Figure 5.12: Example of size satisfaction for stacks. After capacity satisfaction of SF with two stacks and two registers per stack we obtained: (a) a *DFG* with sequence edges of weight one, (b) a *WCCG* with $\chi(WCCG) = 2$, and (c) a *LCG* with $\chi(LCG) = 3$, which is more than the size of each stack. To satisfy the size constraint values c and d are chosen for serialization, therefore we obtain: (d) the *DFG* with the extra sequence edge of weight zero between operations I and D , (e) the resulting *WCCG* with still $\chi(WCCG) = 2$, and (f) the resulting *LCG* with $\chi(LCG) = 2$, which proves the completion of the size satisfaction process.

conflict since they have a fifo-like access and will be assigned to different stacks (stack strong conflict), values b and c have a strong sequential conflict since their lifetimes overlap, and values c and d have a weak sequential conflict since it is not yet determined whether operation C precedes D and operation E precedes I .

5.6.2 Multiple value instances in conflict graphs for fifos

In loop folded schedules values assigned to fifos can have their lifetimes longer than the initiation interval II . Therefore, multiple instances of the same value can be alive simultaneously at any clock cycle and will occupy multiple registers in a fifo.

In order to construct a worst-case size conflict graph *LCG* for fifos, it is thus necessary to know the number of instances of the same value that can be potentially alive in a single fifo.

As a consequence, extra vertices are included in *LCG* representing value instances from different loop iterations. The number of vertices, representing the

instances of one value, is related to the longest value lifetime and the initiation interval:

$$\text{number_instances} = \left\lceil \frac{-d_m(C^u, P^u) + \delta(P^u) - 1}{II} \right\rceil \quad (5.9)$$

Proof. Assume that a value u has the largest lifetime

$$\text{lifetime}_u = -d(\text{Cons}^u, \text{Prod}^u) = -d(C^u, P^u) + \delta(P^u) - 1. \quad (\text{a})$$

The number of overlaps caused by u instances would be equal to:

$$\text{overlaps}_u = \lceil \text{lifetime}_u / II \rceil. \quad (\text{b})$$

u is assigned to one fifo and the number of registers required by u instances is equal to the number overlaps of these instances, i.e. $q_{req}(u) = \text{overlaps}_u$. (c)

When using a conflict graph CG to generate the requirement of u for registers in a fifo, the number of colors $\chi(CG)$ would be equal to the number of registers required, i.e. $\chi(CG) = q_{req}(u)$. (d)

Furthermore, using only instances of value u to build CG , results in:

$$\text{number_instances}_u = \gamma(CG) = \chi(CG). \quad (\text{e})$$

Therefore, from (a), (b), (c), (d) and (e):

$$\text{number_instances} = \lceil (-d_m(C^u, P^u) + \delta(P^u) - 1) / II \rceil \quad \blacksquare$$

The number of instances from Expression 5.9 is upper bound by $\lceil (L - 1) / II \rceil$ which considers the longest possible lifetime in a data flow graph.

5.6.3 Sequential access conflict rules for fifos

No sequential conflict

Two value instances u_i and v_j have no sequential conflict if u and v have a strong fifo conflict, i.e. they will be assigned to different fifos anyway or their lifetimes are serialized. The following proposition formalizes this:

Proposition 5.17 *Instances u_i and v_j of values u and v assigned to fifos have no sequential conflict if Propositions 5.12, 5.6, or 5.8 are valid for u and v , or $d(\text{Cons}_i^u, \text{Prod}_j^v) \geq 0$ or $d(\text{Cons}_j^v, \text{Prod}_i^u) \geq 0$.*

Because the distance matrix is used for checking a conflict, the equivalent Proposition 5.18 is derived.

Proposition 5.18 *Instances u_i and v_j of values u and v assigned to fifos have no sequential conflict if Propositions 5.13, 5.7, or 5.9 are valid for u and v , or if*

$$\begin{aligned} d_m(C^u, P^v) + \delta(P^v) - 1 &\geq (i - j) \times II \quad \text{or} \\ d_m(C^v, P^u) + \delta(P^u) - 1 &\geq (j - i) \times II \end{aligned}$$

Strong sequential conflict

Two value instances have a strong sequential conflict if they have a no fifo conflict and their lifetimes overlap. Those value instances will require different registers if assigned the same fifo. The following proposition formalizes this:

Proposition 5.19 *Instances u_i and v_j of values u and v assigned to fifos have strong sequential conflict if Proposition 5.10 is valid, and $d(\text{Prod}_i^u, \text{Cons}_j^v) \geq 0$ and $d(\text{Prod}_j^v, \text{Cons}_i^u) \geq 0$.*

Proposition 5.18 is derived when using the distance matrix.

Proposition 5.20 *Instances u_i and v_j of values u and v assigned to fifos have strong sequential conflict if Proposition 5.11 is valid for u and v , and if*

$$\begin{aligned} d_m(P^u, C^v) - \delta(P^u) + 1 &\geq (i - j) \times II \quad \text{and} \\ d_m(P^v, C^u) - \delta(P^v) + 1 &\geq (j - i) \times II \end{aligned}$$

Weak sequential conflict

Two value instances u_i and v_j , of values u and v assigned to fifos, have a weak sequential conflict if Proposition 5.18 (no sequential conflict) and Proposition 5.20 (strong sequential conflict) are invalid.

Consider in Figure 5.13 the data flow and conflict graphs of a non-folded example during the fifo satisfaction process. The capacity of two fifos is already satisfied with the addition of sequence edges of weight one, this result is visualized in the worst-case conflict graph $WCCG$ of Figure 5.13b. For size satisfaction, as shown in the conflict graph of Figure 5.13c, values b and f have no sequential conflict since they are consumed simultaneously and will be stored in different fifos (strong fifo conflict), values f and d have a strong sequential conflict since their lifetimes overlap, and values c and d have weak sequential conflict since it is not yet determined whether operation C precedes D and operation I precedes E .

5.6.4 Bottleneck identification and reduction

One conflict graph is constructed in this situation, the *worst-case size conflict graph LCG(SF)*. Its chromatic number $\chi(LCG)$ determines an upper bound of location requirement for stacks or fifos. If the chromatic number is greater than the actual unit size q ($q(SF)$), the satisfaction process continues (see Figure 5.3).

With a number of units (stacks or fifos) greater than one, a strong sequential conflict graph and its coloring is not needed for a lower bound estimation, since

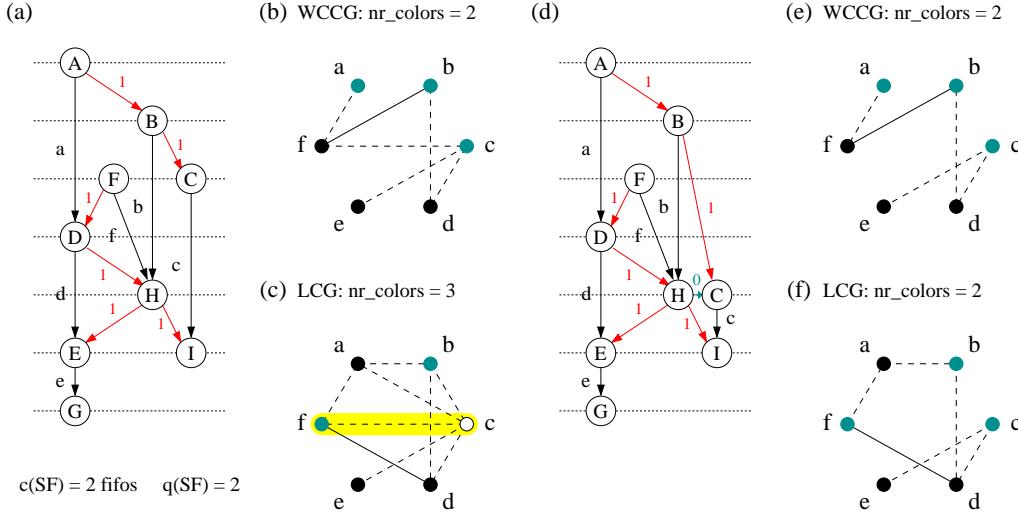


Figure 5.13: Example of size satisfaction for fifos. After capacity satisfaction of SF with two fifos and two registers per fifo we obtained: (a) a DFG with sequence edges of weight one, (b) a $WCCG$ with $\chi(WCCG) = 2$, and (c) a LCG with $\chi(LCG) = 3$, which is more than the size of each fifo. To satisfy the size constraint values c and f are chosen for serialization, therefore we obtain: (d) the DFG with the extra sequence edge of weight zero between operations H and C , (e) the resulting $WCCG$ with still $\chi(WCCG) = 2$, and (f) the resulting LCG with $\chi(LCG) = 2$, which proves the completion of the size satisfaction process.

strong sequential conflicts can also be eliminated by introducing strong conflicts at the capacity level (in $WCCG$), which will force the values to be assigned to different storage units.

Bottlenecks for size satisfaction are identified using the *saturation* and *degree* numbers of LCG , the same way as to identify bottlenecks for capacity satisfaction in $WCCG$.

Lifetime serialization of u and v is performed following the method presented in Section 4.3.2 for stacks and Section 4.8.3 for fifos ([3]).

From the worst-case conflict graph in Figure 5.12c it is observable that values b , c and d could overload a stack of size two if they are stored together. A possible solution is to serialize lifetimes of c and d because they have a weak sequential conflict. In Figure 5.13c, the set of values $\{a, b, c\}$, $\{a, c, f\}$, and $\{c, d, f\}$ can overload a fifo of size two if they are stored together. A possible solution is to serialize lifetimes of f and c .

Once all possibilities for serialization have been used (constraint analysis has detected infeasibility as a result of serializing lifetimes), a strong conflict in the

worst-case conflict graph (and not in the data flow) can be explicitly added between the candidate values. As a result, the values will be assigned to different units which eliminates the conflict in LCG . The best candidates for this conflict addition are the ones with a weak conflict in both LCG and $WCCG$ since the change from weak to strong conflict does not imply an increment in the number of colors for $WCCG$. Otherwise, if the conflict addition is performed between values with no conflict in $WCCG$ the enforcement of the conflict has to be validated with a coloring of the updated $WCCG$ to avoid decisions that will not respect the already satisfied capacity.

In the example of Figure 5.12 for stacks, the serialization of values c and d results in the conflict graphs of Figures 5.12e and 5.12f of $WCCG$ and LCG respectively. This satisfies the capacity and size constraints of the example. Otherwise, the weak stack conflict (c, d) in $WCCG$ can be forced to become strong, which will only take away edge (c, d) from LCG . Meanwhile, in the example of Figure 5.13 for fifos, the serialization of values c and f results in the conflict graphs of Figures 5.13e and 5.13f. If a strong fifo conflict is forced, only edge (c, f) will be eliminated from LCG which will still leave LCG with $\chi(LCG) = 3$ and the constraint satisfaction will continue to iterate.

Lifetime serialization and the conflict addition will alleviate the number of sequential conflicts and coloring of LCG . At the end of the satisfaction process the extra conflicts are sent to storage allocation and assignment after operation scheduling.

5.7 Experimental Results Using FACTS

This section presents the experimental results obtained with the proposed method. The aim of the set of experiments is to test the capabilities of the constraint satisfaction approach to find solutions for tight constrained examples using stacks and fifos as storage.

Instances of the examples with one storage file, presented in Table 4.1 were used, and the storage satisfaction results are shown in Tables 5.1 and 5.2 for stacks and fifos respectively.

Following the column with the instance names, the second column of the tables show the storage requirements obtained by the resource constrained scheduler from [75] followed by an exact storage allocation and assignment. c_{req} is the capacity and q_{req} is the size of the storage units required after scheduling. $t(s)$ is the CPU time in seconds spent.

The third column of both tables shows the results of applying storage constraint satisfaction. The scheduler from [75] was used to complete the partial schedule resulting from the satisfaction process. The tables also show the mini-

Table 5.1: Results for storage files with stacks.

$DFG_{ FU , SF ,L,II}$	Sch. & Alloc.			Capacity satisfaction				c_R
	c_{req}	q_{req}	$t(s)$	c	q	$t(s)$	mobility	
fdct _{1,1,18,-}	7	3	0.34	7	2	0.93	$9.52 \rightarrow 0.17$	9
fdct _{2,1,11,-}	8	2	0.39	8	1	0.40	$2.90 \rightarrow 0.02$	8
fdct _{4,1,8,-}	10	2	0.21	8	1	0.22	$0.76 \rightarrow 0.14$	8
fft _{1,1,13,-}	4	2	0.06	3	2	0.16	$3.17 \rightarrow 0.03$	5
fft _{1,1,13,4}	9	2	0.11	7	2	0.53	$2.30 \rightarrow 0.00$	10
fft _{2,1,11,-}	6	2	0.06	4	2	0.10	$2.17 \rightarrow 0.27$	6
fft _{2,1,11,3}	13	2	0.12	9	2	0.18	$1.20 \rightarrow 0.07$	12
fir _{1,1,6,-}	3	2	0.01	3	1	0.01	$1.56 \rightarrow 0.38$	3
fir _{1,1,6,3}	5	2	0.02	5	2	0.30	$1.44 \rightarrow 1.13$	6
ifft _{1,1,36,-}	11	3	1.82	6	2	2.83	$13.9 \rightarrow 0.21$	7
ifft _{1,1,36,26}	11	3	3.39	6	2	7.62	$13.9 \rightarrow 0.16$	8
ifft _{2,1,23,-}	9	3	1.69	6	2	1.90	$6.34 \rightarrow 0.51$	7
ifft _{2,1,23,14}	10	2	2.71	9	2	3.10	$6.30 \rightarrow 0.41$	10
iir _{1,1,9,-}	6	2	0.06	5	1	0.08	$2.07 \rightarrow 0.15$	5
iir _{1,1,9,4}	8	2	0.10	7	2	0.12	$1.55 \rightarrow 0.19$	8
loop _{1,1,11,-}	6	3	0.05	5	2	0.14	$4.40 \rightarrow 1.17$	5
loop _{1,1,11,4}	11	2	0.14	10	1	1.42	$4.00 \rightarrow 1.07$	10
loop _{2,1,7,-}	9	2	0.05	7	1	0.12	$2.00 \rightarrow 1.33$	7
loop _{2,1,7,2}	17	1	0.12	15	1	0.22	$1.60 \rightarrow 0.60$	15

mum capacities $c(c(SF))$ and sizes $q(q(SF))$ for which our approach could find a feasible solution. The CPU time in seconds $t(s)$ spent to find the solution including scheduling and final storage allocation, and the impact of access ordering on the schedule freedom of the operations using the mobility are shown in the last two sub-columns. The numbers before and after the arrow denote the mobility before and after the satisfaction process respectively.

For comparison reasons, the last column of the tables shows the minimum capacity c_R obtained by applying the storage satisfaction process to the examples with random-access register files (from Table 4.4).

Tables 5.1 and 5.2 show that our approach can deal also with stacks and fifos as storage units, together with timing and resource constraints. By taking the storage file constraints into account, this method is able to reduce the storage pressure compared to the approach that performs storage allocation a posteriori. For example, for instances ifft_{1,1,36,-} and ifft_{1,1,36,26} in Table 5.1 a reduction from

Table 5.2: Results for storage files with fifos.

$DFG_{ FU , SF ,L,II}$	Sch. & Alloc.			Capacity satisfaction				c_R
	c_{req}	q_{req}	$t(s)$	c	q	$t(s)$	mobility	
fdct _{1,1,18,-}	6	3	0.34	7	2	1.07	9.52 → 0.14	9
fdct _{2,1,11,-}	6	2	0.39	8	1	0.80	2.90 → 0.05	8
fdct _{4,1,8,-}	10	1	0.21	8	1	0.26	0.76 → 0.05	8
fft _{1,1,13,-}	4	3	0.06	3	2	0.15	3.17 → 0.33	5
fft _{1,1,13,4}	6	3	0.11	6	4	0.16	3.10 → 0.00	10
fft _{2,1,11,-}	6	2	0.06	4	2	0.13	2.17 → 0.60	6
fft _{2,1,11,3}	10	2	0.12	9	2	0.26	2.17 → 0.03	12
fir _{1,1,6,-}	3	3	0.01	2	3	0.01	1.56 → 0.50	3
fir _{1,1,6,3}	4	2	0.02	3	3	0.02	1.44 → 0.38	6
ifft _{1,1,36,-}	7	5	1.82	6	3	5.04	13.9 → 0.29	7
ifft _{1,1,36,26}	7	4	3.39	7	2	7.62	13.9 → 0.18	8
ifft _{2,1,23,-}	7	4	1.69	6	2	3.12	6.34 → 0.53	7
ifft _{2,1,23,14}	8	3	2.71	8	3	6.51	6.34 → 0.40	10
iir _{1,1,9,-}	5	3	0.06	4	3	0.07	2.07 → 0.22	5
iir _{1,1,9,4}	7	3	0.10	6	2	0.89	1.59 → 0.15	8
loop _{1,1,11,-}	6	4	0.05	5	2	0.24	4.40 → 1.00	5
loop _{1,1,11,4}	10	6	0.14	9	2	0.72	4.00 → 1.00	10
loop _{2,1,7,-}	9	2	0.05	7	1	0.18	2.00 → 0.60	7
loop _{2,1,7,2}	16	5	0.12	15	2	0.25	2.00 → 0.60	15

11 to six stack units is obtained. In Table 5.2, for instance fft_{2,1,11,-} a reduction from six to four, and for loop_{2,1,7,-} a reduction from nine to seven fifo units are obtained.

Comparing the minimum capacities obtained by using random-access registers (in column c_R) and by using stacks or fifos (columns c in tables), a reduction in the number of storage units has often been obtained, specially for example fft. This reduction can be useful if the number of bits to encode instructions is critical. Unfortunately, this is not always the case. In some cases the best results found for stacks or fifos were like having random-access registers as storage (stack or fifo units obtained with only one register).

Comparing the usage of stacks and fifos, the results in the tables show a certain advantage of fifos over stacks. This is due to the fact that DSP examples selected do not have recursive characteristics favorable to stacks, and additionally for folded cases it is more advantageous the usage of fifos.

Table 5.3: Results exploiting available ILP using fifos.

$DFG_{ SF ,L}$	$ FU $	II	Sch. & Alloc.			Capacity satisfaction			
			c_{req}	q_{req}	$t(s)$	c	q	$t(s)$	mobility
fft _{1,11}	2	2	11	3	0.16	8	5	1.15	2.17 → 0.07
fir _{1,6}	2	2	5	3	0.01	4	3	0.02	1.94 → 0.19
	3	1	7	3	0.01	7	3	0.70	1.94 → 0.19
iir _{1,9}	2	2	11	4	0.05	10	4	0.12	2.22 → 0.15
	4	1	17	4	0.04	17	4	1.55	2.22 → 0.11

Similar to rotating registers, fifos offer the chance to reduce the initiation interval and exploit the available ILP. Table 5.3 shows the results of capacity satisfaction for examples in which a reduction of II was obtained. In each of those examples, value lifetimes are longer than the reduced initiation interval making it impossible for the respective values to be stored in random-access registers or stacks.

Chapter 6

Storage Files of Different Access Types

6.1 Introduction

Previous chapters dealt with constraint satisfaction for storage files with units having a specific access type, i.e. random-access registers, rotating registers, stacks, or fifos. Storage file allocation for values was assumed prior scheduling.

In this chapter all the previously presented rules and methods for each particular storage are put together to obtain a method that satisfies the constraints during scheduling of architectures containing storage files of different access types, without any previous file allocation for values.

Storage files of different access types can be used to exploit the value access regularity presented in many applications, to reduce the total number of storage units (and bits for addressing), or to exploit the available ILP by reducing the initiation interval (when using rotating registers or fifos).

In such storage scope, Aloqeely in [6] presents a technique for architectural synthesis which exploits the regularity of value accesses that exists in many DSP and matrix computations. After scheduling of operations, the storage assignment procedure of Aloqeely consists of three stages. In the first stage, it is tried to assign all values to queue-like storage. Then, based on a rejection criterion like the size of the queue, some of the queues that do not meet the minimum allowed utilization are rejected. The values that have been assigned to rejected queues will be used as inputs to the next stage in which they are tried to be assigned to stacks. Similarly, stacks that do not meet the minimum allowed utilization are rejected and the values are sent to the last stage that will assign them to random-access registers.

Similar to the work of Aloqeely our approach follows a predefined analysis

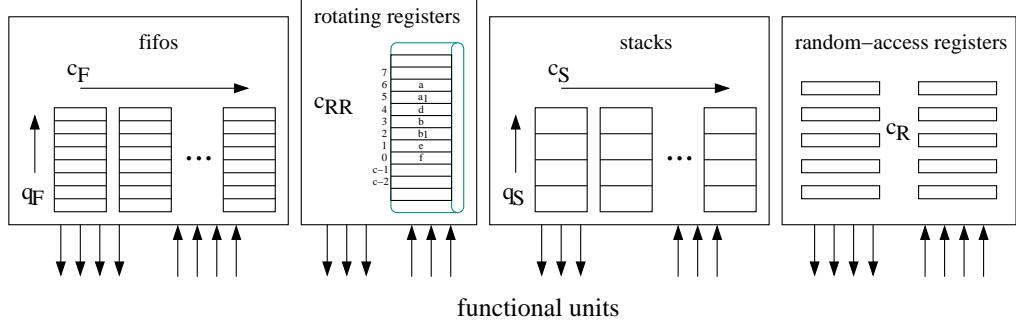


Figure 6.1: A storage architecture with files of different access types.

order. For every file with a specific access behavior a limited *capacity* (and *size* for stacks and fifos) is associated and constitutes a constraint to satisfy during the scheduling process.

Our approach makes use of two different methods for constraint satisfaction when units of a specific storage file and access type are about to be overloaded. The first method is *access ordering* of values, as has been described in previous chapters. The second method is *spilling* values to other storage files (and other access types). Bottlenecks for storage allocation are found in both the worst- and the best-case conflict graphs. Bottlenecks found in the worst-case conflict graph are *edges*, and the respective values are candidates to have their accesses ordered accordingly. Bottlenecks found in the best-case conflict graph are *vertices* representing values that will be spilled to another storage file. Both methods will remove edges and vertices respectively from the graphs until the upper and lower bounds of required storage are satisfied.

This chapter is organized as follows. In Section 6.2, the model assumed in this case for the storage architecture is shown. Section 6.3 explains the proposed constraint satisfaction approach. Finally, in Section 6.4 experimental results are presented.

6.2 Storage Model

In Figure 6.1, the assumed storage architecture is illustrated. It has a random-access register file with capacity c_R , a rotating register file with capacity c_{RR} , a file of stacks with capacity c_S and size q_S , and/or a file of fifos with capacity c_F and size q_F .

It is unlikely however to have architectures with units having four storage files each with a different access type, since their implementation can become

complex and expensive. From the compiler point of view, to exploit the storage characteristics of those architectures is also difficult to implement.

6.3 Proposed Approach

The goal of our approach with this kind of storage architecture is to find an assignment of values to storage files (*pre-assignment*), an assignment of values to storage units, and a schedule that satisfy the storage, precedence, timing and resource constraints.

Constraint satisfaction and pre-assignment of values are performed considering one access type at a time. Initially, all values are assumed to be pre-assigned to one storage file (one access type). During the satisfaction process, when the minimum requirement of a particular file is more than the available, some values are selected to be spilled to another available file (hence another access type).

The criteria about which access type would be considered first take into account the characteristics of each particular storage as follows:

- Random-access registers are the most flexible storage from the compiler point of view, since access to values is straightforward. However, they are costly in terms of control and encoding (observation mainly considered for architecture design). For loop folded cases, values assigned to random-access registers have their lifetimes upper bound (constrained) by the initiation interval II .
- Rotating registers are the most costly storage units since their control is more complex related to their base plus offset addressing mode. For the compiler rotating registers are as flexible as random-access registers and their advantage is that they can be used in applications with a reduced II , since values assigned to rotating register files do not have their lifetimes upper bound by II .
- Stacks are less costly than random-access registers because of their multiple registers sharing one address. Their control is simpler, since the stack pointer is updated automatically in every access. However, for the compiler stacks are not as flexible if it is assumed a destructive reading of values. Furthermore, for loop folded cases, values assigned to stacks have also their lifetimes upper bound by II .
- Fifos are also less costly than random-access registers because of their multiple registers and their simpler controllability. They are also less flexible

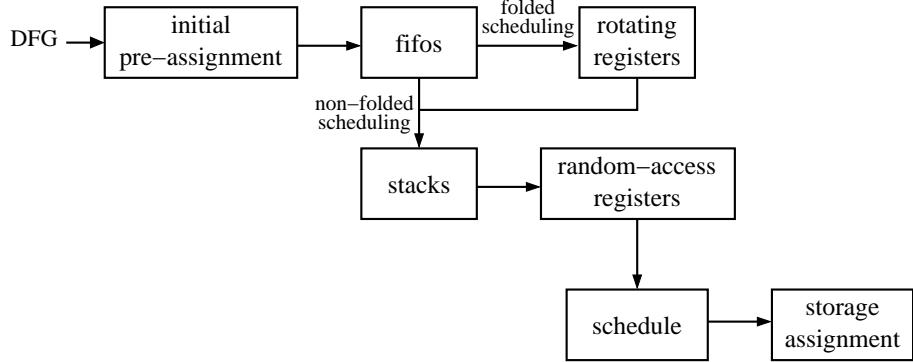


Figure 6.2: Constraint satisfaction and scheduling flow for storage files of different access types.

than random-access registers from the compiler point of view, if a destructive reading of values is assumed. On the other hand, values assigned to fifos do not have their lifetimes upper bound by II .

With this description, the order in which the files are analyzed and their constraint satisfied, depends specifically on two situations (see the flow in Figure 6.2):

- Loop folded cases. It is always desirable to reduce the schedule freedom gradually, and to satisfy the less flexible storage first. Therefore, for folded cases the following order is considered: constraint satisfaction for fifos first, then for rotating registers. By this stage the initiation interval constraint binds the lifetimes of the remaining values. After that, the process continues with stacks and finally with random-access registers.
- Non-folded cases. For non-folded cases the advantage of rotating registers is reduced, since they become the same as random-access registers. Therefore, the following order is considered: constraint satisfaction for fifos first, then for stacks, and finally for registers.

The satisfaction approach depicted graphically in Figure 6.2, starts with an *initial pre-assignment* of values.

For folded cases this process checks whether there are values with lifetimes longer than II . If yes, those values are pre-assigned to fifos or rotating registers (in that order) according to the availability of those units. The process also checks whether values are consumed more than once (for stacks and fifos). When an initial pre-assignment cannot be performed because the lack of units with a specific access type, infeasibility is reported.

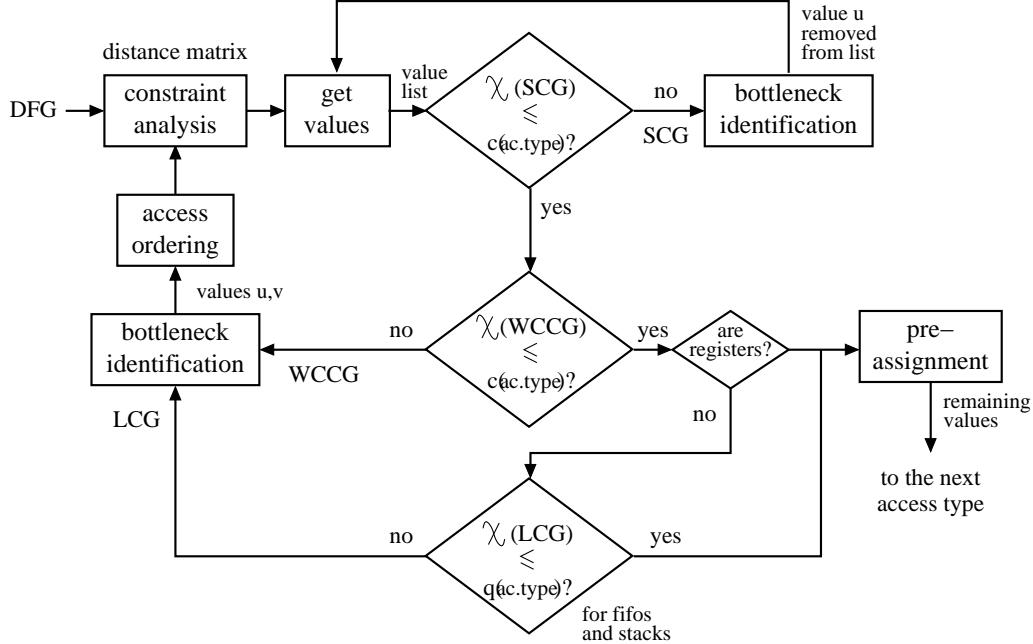


Figure 6.3: Storage satisfaction for one storage file.

With a list of value candidates for pre-assignment, the satisfaction algorithm analyses the storage file with a specific access type. The satisfaction process is therefore called, which is depicted graphically in Figure 6.3.

In the satisfaction process a lower bound is calculated first by coloring a best-case conflict graph SCG . If this bound is larger than the number of units available (capacity), a vertex (value) is selected from its largest saturation and degree numbers in the colored SCG . The selected value is thus removed from the list of candidates.

When the number of available units fulfills the minimum requirements, the process checks the upper bound of storage requirement using a worst-case conflict graph $WCCG$, identifies bottlenecks in $WCCG$, and selects pairs of values to order their accesses.

After access ordering the satisfaction process checks whether there is a variation on the bounds. If the lower bound requirement becomes greater than the capacity, a value is selected and removed from the list. Every time a value is removed, all orderings for the current storage file are discarded, inserted sequence edges are removed from the DFG , and the satisfaction process restarts with an updated list of values.

The satisfaction process continues until lower- and upper bounds for the ca-

pacity (and size for fifos and stacks) are satisfied. Only when the satisfaction of constraints is guaranteed, the values on the list are then pre-assigned to the file, and the next file (another access type) is analyzed for the remaining values.

For loop folded cases, before the algorithm deals with stacks or random-access registers the initiation interval constraint will bind the lifetimes of the remaining values as described in Section 2.3.3.

For the last storage file analyzed, if it is not possible to satisfy its constraints infeasibility is reported.

After constraint satisfaction was successfully accomplished and values are already pre-assigned, the precedences are transferred to conventional schedule and storage assignment phases to complete the process.

6.4 Experimental Results Using FACTS

This section presents the experimental results obtained with the proposed method. Instances of the examples presented in Table 4.1 were used, and the respective storage satisfaction results are shown in Table 6.1.

The first column of Table 6.1 shows the instance names with the format DFG_L . L is the latency. The second column shows the initiation interval II which is the minimum obtainable with the available ILP ($|FU|$) shown in the third column. For each storage architecture the table shows the fifo capacities c_F and sizes q_F , the rotating register capacities c_{RR} , the stack capacities c_S and sizes q_S , and the random-access register capacities c_R .

The CPU time in seconds $t(s)$ spent to find the solution (including scheduling and final storage assignment), and the impact on the schedule freedom (mobility after the satisfaction process) are shown in the last two columns of the table.

The resource constrained scheduler from [75] was used to complete the process. Storage assignment for rotating registers is performed using the method for relative locations assignment in [78], while for the other storage types a graph-coloring based storage allocation is performed.

As a reference, also the minimum capacities obtained by applying the satisfaction approach to storage files having a specific access type are included in Table 6.1 at the same row following the instance name or a new ($|FU|, II$) set of constraints. Fifos and stacks are assumed to have a destructive reading and to obtain the capacity and size references, copies of multiply consumed values were generated (refer to Section 5.2.1). Note that for certain values of II no solution could be found using stacks or random-access registers. At the end of the row the initial mobility of operations prior the satisfaction approach is included.

The results in Table 6.1 are the minimum obtainable with FACTS making trade-offs among the different constraints for each file with a specific access type. Com-

binations of fifos, stacks, and random-access registers, or rotating and random-access registers were tested (which are the most likely to be used together in an architecture).

As the results in the table show, it is possible to make tradeoffs among the constraints of different storage files. By weakening the constraints for one file the capacity requirement for the others can be reduced. Also, with the use of fifos or rotating registers a better use of the available ILP can be made by reducing the initiation interval (see the results for instances fir_6 and iir_9).

In many cases presented in the table a reduction in the number of storage units can be observed when using fifos, stacks and random-access registers than when using only registers. One such example is fir_6 with $II = 3$ for which the number of units were reduced from six (random-access registers) to e.g. four (one fifo plus three random-access registers), or five (two stacks plus three random-access registers). However, this is not always the case especially when considering fifos and stacks with destructive reading. Tradeoffs between rotating and random-access registers are straightforward as seen for fir_6 and iir_9 .

In conclusion, our approach is able to find solutions for tight constrained examples, and having storage files of different access types in the architecture. Using this kind of storage, reductions in the number of units and bits for addressing are achieved, as well as exploitation of the available ILP in the processor architecture.

Table 6.1: Results for architectures with storage files of different access types.

DFG_L	$ FU $	II	c_F	q_F	c_{RR}	c_S	q_S	c_R	$t(s)$	mobility
fft_{13}	1	-	5	3	-	5	2	5	-	3.17
			2	2	-	-	-	2	0.75	0.30
			2	4	-	-	-	1	0.75	0.10
			-	-	-	2	2	3	0.47	0.37
			-	-	-	3	2	2	0.41	0.43
			1	2	-	1	2	2	0.80	0.13
			2	2	-	2	2	-	0.75	0.30
fir_6	1	3	4	3	6	5	2	6	-	1.44
			1	3	-	-	-	3	0.05	0.94
			-	-	-	2	2	3	0.07	0.38
			-	-	4	-	-	2	0.13	1.06
			-	-	2	-	-	4	0.07	0.69
	2	2	6	3	8	-	-	-	-	1.94
			2	3	-	-	-	3	0.02	0.31
			-	-	6	-	-	2	0.05	0.19
	3	1	7	3	15	-	-	-	-	1.94
			5	3	-	-	-	2	0.03	0.25
			-	-	13	-	-	2	0.15	0.19
$ifft_{36}$	1	-	7	3	-	7	2	7	-	13.8
			3	2	-	-	-	4	33.3	2.58
			-	-	-	4	3	4	5.31	3.01
			2	2	-	2	2	5	41.0	1.84
iir_9	1	4	7	2	9	7	2	8	-	1.59
			4	3	-	-	-	3	0.17	0.19
			-	-	-	4	2	6	0.37	0.41
			3	2	-	3	2	2	0.34	0.22
			-	-	7	-	-	2	0.70	0.19
			-	-	2	-	-	6	0.26	0.22
	2	2	12	4	15	-	-	-	-	2.22
			8	4	-	-	-	3	0.11	0.11
			-	-	10	-	-	5	0.31	0.17

Chapter 7

Conclusions and Further Research

This thesis presents an approach for storage allocation and operation scheduling for code generation in embedded processor compilers, and for the architectural synthesis of DSP and multimedia applications. DSP and multimedia application kernels characterized by intensive computation, are mapped onto a VLIW embedded processor architecture (or architecture template for synthesis), which has a limited number of resources including distributed and capacity constrained storage files.

Storage file constraints are taken into account from the first phase of scheduling while there is enough freedom to reduce storage pressure. Constraint analysis techniques are used to capture the interaction among the precedence, timing and resource constraints. By constructing and coloring a worst-case conflict graph that models the strong and weak conflicts between value accesses, the bottlenecks for storage allocation are identified. These bottlenecks are subsequently reduced by ordering value accesses accordingly. This results in a partial schedule that can be completed by a conventional scheduler without violating the storage file constraints.

Although the problem of spilling values to background memory was not directly addressed, the proposed method can help to avoid unnecessary spill code.

The experimental results in this work clearly show the advantages of the approach:

- This approach is able to satisfy storage file constraints under tight timing, precedence, and resource constraints. The method provides a good balance between solution quality and run time.
- By taking the storage file constraints into account, this method is able to reduce the storage pressure compared to an approach that performs scheduling first and storage allocation *a posteriori*.

- The proposed method is also able to reduce the storage pressure for files with low capacity at the expense of an increase in the amount of storage used in other storage files. This is important for handling heterogeneous storage file architectures. Traditional methods lack the capability of making tradeoffs between different storage files. Furthermore, this also shows that the method can be used to do design space exploration for storage file architectures.
- It is possible to make a tradeoff between storage pressure and timing constraints. By weakening the latency or the initiation interval constraints, storage file capacities can be further reduced.

Moreover, the satisfaction approach presented also works in the context of rotating registers, stacks, fifos, and enables an integrated approach for storage architectures with different types of storage. The coloring approach was reused effectively in all the treated storage allocation problems.

This work has been implemented in the FACTS research tool. The FACTS tool is used at the Eindhoven University of Technology as a vehicle for research in code generation and architectural synthesis. FACTS functionality is being integrated in the A|RT tool set from Adelante Technologies (a company resulted from the merging of Frontier Design and Philips Semiconductors' EPD), in order to design and program ASIPs with a VLIW architecture. At Philips Research, FACTS is applied in the COCOON and the ERC (Embedded Reconfigurable Computing) projects as part of the compiler targeted at VLIW architectures.

Current research related to code generation and implemented in FACTS focuses on the following topics:

- The work of Bekooij [8] presents an approach that uses constraint analysis for the assignment of operations to functional units (data routing through a limited connection network between functional units and register files).
- For conditional constructions, the work of Zhao in [82] presents an approach that performs *if-conversion* and creates predicates that are included in the code. After if-conversion, scheduling and storage allocation is performed considering the property of *mutual exclusivity* of values, i.e. because they belong to exclusive conditioned basic blocks, mutual exclusive values will never coexist and can be assigned to the same storage.
- Zhao et al. in [81] present an approach that reduces the need for explicit instruction selection by transferring constraints implied by the instruction set to virtual resource constraints, that conventional resource constrained schedulers can cope with.

To guarantee compiler retargetability, there is a necessity to define a complete machine description in FACTS. The compiler retargetability is limited when tuning the processor to a specific application (domain), because no complete machine description is defined in the input description file for FACTS.

Possible extensions include the use of the techniques presented in this thesis in a broader scheduling scope (beyond the scope of basic blocks). In that case, a global constraint analysis is necessary as well as the ability to make tradeoffs between the use of the techniques presented in this thesis and the insertion of spill code.

Bibliography

- [1] A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.
- [2] C. Alba Pinto, B. Mesman, and J. Jess. Constraint satisfaction for relative location assignment and scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001*, San Jose, CA, Nov. 2001. ACM and IEEE Computer Society.
- [3] C. Alba Pinto, B. Mesman, and K. van Eijk. Register file capacity satisfaction during scheduling. In *Proceedings of the ProRISC/IEEE workshop on Circuits, Systems and Signal Processing*, pages 1–8, Mierlo, The Netherlands, Nov. 1999. STW Technology Foundation.
- [4] C. Alba Pinto, B. Mesman, and K. van Eijk. Register files constraint satisfaction during scheduling of dsp code. In *Proceedings of the XII Symposium on Integrated Circuits and Systems Design, SBCCI'99*, pages 74–77, Natal, Brazil, Oct. 1999. IEEE Computer Society Press.
- [5] C. Alba Pinto, K. van Eijk, B. Mesman, and J. Jess. Satisfaction of constraints for storage files with fifos or stacks during scheduling. In *Proceedings of the ProRISC/IEEE workshop on Circuits, Systems and Signal Processing*, pages 171–180, Veldhoven, The Netherlands, Nov. 2000. STW Technology Foundation.
- [6] M. Aloqeely and C. Roger Chen. A new technique for exploiting regularity in data path synthesis. In *Proceedings of the European Design Automation Conference with EURO-VHDL*, pages 394–399, Grenoble, France, Sep. 1994. ACM and IEEE Computer Society.
- [7] S. Bashford and R. Leupers. Constraint driven code selection for fixed-point dsps. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 817–822, New Orleans, CA, Jun. 1999. ACM and IEEE Computer Society.

- [8] M. Bekooij, B. Mesman, J. van Meerbergen, and J. Jess. Constraint analysis for operation assignment in FACTS. In *Proceedings of the ProRISC/IEEE workshop on Circuits, Systems and Signal Processing*, pages 229–236, Veldhoven, The Netherlands, Nov. 2000. STW Technology Foundation.
- [9] D. Berson, R. Gupta, and M. Soffa. Hare: A hierarchical allocation of registers. Technical Report 95-06, University of Pittsburgh, Computer Science Department, Feb. 1995.
- [10] D. Bradlee, S. Eggers, and R. Henry. Integrating register allocation and instruction selection for riscs. In *Proceedings of the 4th International Conference on Architectural support for Programming Languages and Operating Systems*, pages 122–131, Santa Clara, CA, Apr. 1991. Association for Computer Machinery.
- [11] R. Braspenning. Modeling issue slot constraints with resources. Technical report, Eindhoven University of Technology, The Netherlands, 1999.
- [12] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, 1992.
- [13] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtegale, and A. Vandecapelle. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.
- [14] G. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 98–105, Boston, MA, Jun. 1982. Association for Computer Machinery.
- [15] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, Oct. 1981.
- [16] Z. Chamski, C. Eisenbeis, and E. Rohou. Flexible issue slot assignment for vliw architectures. In *Proceedings of the 2nd International Workshop on Compiler and Architecture Support for Embedded Systems, CASES'99*, Washington, DC, USA, Oct. 1999.
- [17] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, Cambridge, second edition edition, 2001.
- [18] O. Coudert. Exact coloring for real-life graphs is easy. In *Proceedings of the 34th ACM/IEEE Design Automation Conference*, pages 121–126, Anaheim, CA, Jun. 1997. ACM and IEEE Computer Society.

- [19] S. Davidson, D. Landskov, B. Shriver, and P. Mallet. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C-30(7):460–477, Jul. 1981.
- [20] J. Dehnert and R. Towle. Compiling for the cydra 5. *The Journal of Supercomputing*, 7(1/2):181–228, May. 1993.
- [21] P. Embree. *C Algorithms for Real-Time DSP*. Prentice Hall, 8 edition, 1995.
- [22] P. Faraboschi, G. Desoli, and J. Fisher. Clustered instruction-level parallel processors. Technical Report HPL-98-204, Hewlett Packard Laboratories, Cambrigde, Dec. 1998.
- [23] M. Fernandes, J. Llosa, and N. Topham. Using queues for register file organization in vliw architectures. Technical Report ECS-CSG-29-97, Computer Systems Group. University of Edinburgh, Feb. 1997.
- [24] J. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 263–273, 1983.
- [25] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, Englewood Cliffs, 1994.
- [26] M. Garey and D. Johnson. *Computers and Intractability. A guide to the Theory of NP-Completeness*. W. Freeman, San Francisco, 1979.
- [27] C. Gebotys and M. Elmasry. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 2–7, San Francisco, CA, Jun. 1991. ACM and IEEE Computer Society.
- [28] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
- [29] G. Goossens, J. Vandewalle, and H. De Man. Loop optimization in register-transfer scheduling for dsp-systems. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 826–831, Las Vegas, Jun. 1989. ACM and IEEE Computer Society.
- [30] Trimedia Product Group. *Trimedia TM-1 Media Processor Data Book*. Philips Semiconductors, 1997.
- [31] M. Heijligers. *The Application of Genetic Algorithms to High-Level Synthesis*. PhD thesis, Eindhoven University of Technology, 1996.

- [32] J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, second edition edition, 1996.
- [33] J. Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, 1996.
- [34] J. Huisken, F. van de Laar, M. Bekooij, G. Gielis, P. Gruijters, and F. Welteren. A power-efficient single-chip ofdm demodulator and channel decoder for multimedia broadcasting. *IEEE Journal of Solid-State Circuits*, 33(11):1793–1798, Nov. 1998.
- [35] J. Janssen and H. Corporaal. Registers on demand, an integrated region scheduler and register allocator. In *Proceedings of the 7th International Conference on Compiler Construction*, Lisbon, Portugal, Apr. 1998.
- [36] D. Kolson, A. Nicolau, N. Dutt, and K. Kennedy. Optimal register assignment to loops for embedded code generation. In *Proceedings of the 8th International Symposium on System Synthesis, ISSS'95*, pages 42–47, Cannes, France, Sep. 1995. IEEE Computer Society Press.
- [37] P. Koopman. *Stack Computers: the new wave*. Ellis Horwood, 1989.
- [38] D. Ku and G. De Micheli, editors. *High-level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Dordrecht, 1992.
- [39] S. Kumar, J. Aylor, B. Johnson, and W. Wulf. *The Codesign of Embedded Systems*. Kluwer Academic Publishers, 1996.
- [40] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, Jun. 1988. Association for Computer Machinery.
- [41] E. Lee. Programmable dsp architectures i. *IEEE ASSP Signal Processing Magazine*, 5(4):4–19, Oct. 1988.
- [42] E. Lee. Programmable dsp architectures ii. *IEEE ASSP Signal Processing Magazine*, 6(1):4–14, Jan. 1989.
- [43] R. Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, Boston, Nov. 2000.

- [44] R. Leupers. Code selection for media processors with simd instructions. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, DATE 2000*, pages 4–8, Paris, France, Mar. 2000. IEEE Computer Society Press.
- [45] R. Leupers and P. Marwedel. Instruction selection for embedded dsp's with complex instructions. In *Proceedings of the European Design Automation Conference with EURO-VHDL*, pages 200–205, Geneva, Switzerland, Sep. 1996. ACM and IEEE Computer Society.
- [46] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction selection using binate covering for code size optimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD'95*, pages 393–399, San Jose, CA, Nov. 1995. ACM and IEEE Computer Society.
- [47] J. Llosa, M. Valero, and E. Ayguade. Heuristics for register-constrained software pipelining. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-29*, pages 250–261, Dec. 1996.
- [48] C. Loeffler, A. Ligtenberg, and G. Moschytz. Practical fast 1d-dct algorithms with 11 multiplications. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing, ICASSP-89*, volume 2, pages 988–991, 1989.
- [49] M. McFarland, A. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, Feb. 1990.
- [50] B. Mesman. *Constraint Analysis for DSP Code Generation*. PhD thesis, Eindhoven University of Technology, The Netherlands, May. 2001.
- [51] B. Mesman, C. Alba Pinto, and K. van Eijk. Efficient scheduling of dsp code on processors with distributed register files. In *Proceedings of the 12th International Symposium on System Synthesis, ISSS'99*, pages 100–106, San Jose, CA, Nov. 1999. IEEE Computer Society Press.
- [52] B. Mesman, M. Strik, A. Timmer, J. van Meerbergen, and J. Jess. A constraint driven approach to loop pipelining and register binding. In *Proceedings of the Design, Automation and Test in Europe Conference, DATE'98*, pages 377–383, Paris, France, Feb. 1998. IEEE Computer Society Press.
- [53] B. Mesman, A. Timmer, J. van Meerbergen, and J. Jess. Constraint analysis for dsp code generation. *IEEE Transactions on Computer-Aided Design*, 18(1):44–57, Jan. 1999.

- [54] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.
- [55] R. Moreno, R. Hermida, and M. Fernandez. Short note: Register estimation in unscheduled dataflow graphs. *ACM Transactions on Design Automation of Electronic Systems*, 1(3):396–403, Jul. 1996.
- [56] C. Norris and L. Pollock. Register allocation over the program dependence graph. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 266–277, Orlando, FL, Jun. 1994. Association for Computer Machinery.
- [57] C. Norris and L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 169–179, Ann Arbor, MI, Nov. 1995. IEEE Computer Society Press.
- [58] W. Nuijten. *Time and Resource Constrained Scheduling*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1994.
- [59] S. Ohm, F. Kurdahi, and N. Dutt. Comprehensive lower bound estimation from behavioral descriptions. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD’94*, pages 182–187, San Jose, CA, Nov. 1994. ACM and IEEE Computer Society.
- [60] S. Ohm, F. Kurdahi, and N. Dutt. A unified lower bound estimation technique for high-level synthesis. Technical report, University of California, Irvine, CA, Apr. 1997.
- [61] P. Paulin, M. Cornero, and C. Liem. Trends in embedded systems technology: An industrial perspective. In M. Sami and G. De Micheli, editors, *Hardware-Software Codesign*, pages 311–337. Kluwer Academic Publishers, Boston, 1996.
- [62] B. Paysan. *Implementation of the 4stack Processor Using Verilog*. PhD thesis, Technical University Munich, Germany, Aug. 1996.
- [63] S. Pinter. Register allocation with instruction scheduling: A new approach. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 248–257, Albuquerque, NM, Jun. 1993. Association for Computer Machinery.
- [64] J. Rabaey, H. De Man, J Vanhoof, G. Goossens, and F. Catthoor. Cathedral ii: A synthesis system for multiprocessor dsp systems. In D. Gajski, editor, *Silicon Compilation*. Addison-Wesley, 1988.

- [65] B. Rau, C. Glaeser, and E. Greenawalt. Architectural support for the efficient generation of code for horizontal architectures. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 96–99, Palo Alto, 1982.
- [66] B. Rau, V. Kathail, and S. Aditya. Machine-description driven compilers for epic processors. Technical Report HPL-98-40, Hewlett Packard Laboratories, Sep. 1998.
- [67] B. Rau, V. Kathail, and S. Aditya. Machine-description driven compilers for epic and vliw processors. In *Design Automation for Embedded Systems*, volume 4. Kluwer Academic Publishers, 1999.
- [68] A. Sangiovanni-Vincentelli. A note on bipartite graphs and pivot selection in sparse matrices. *IEEE Transactions on Circuits and Systems*, CAS.23(12):817–821, 1976.
- [69] M. Schlansker, B. Rau, S. Mahlke, V. Kathail, V. Johnson, S. Anik, and S. Abraham. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Technical Report HPL-96-120, Hewlett Packard Computer Research Center, Nov. 1994.
- [70] TI Technical Documentation Services. *TMS320C62xx CPU and Instruction Set Reference Guide*. Texas Instruments, Inc., Jul. 1997.
- [71] A. Sharma and R. Jain. Register estimation from behavioral specifications. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 576–580. IEEE Computer Society Press, Oct. 1994.
- [72] A. Sharma and R. Jain. Insyn: Integrated scheduling for dsp applications. *IEEE Transactions on Signal Processing*, 43(8):1966–1977, Aug. 1995.
- [73] A. Timmer. *From Design Space Exploration to Code Generation: a constraint satisfaction approach for the architectural synthesis of digital VLSI circuits*. PhD thesis, Eindhoven University of Technology, The Netherlands, Apr. 1996.
- [74] A. Timmer and J. Jess. Execution interval analysis under resource constraints. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD'93*, pages 454–459, Santa Clara, Nov. 1993. ACM and IEEE Computer Society.

- [75] A. Timmer, M. Strik, J. van Meerbergen, and J. Jess. Conflict modeling and instruction scheduling in code generation for in-house dsp cores. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 593–598, San Francisco, CA, Jun. 1995. ACM and IEEE Computer Society.
- [76] K. van Eijk, E. Jacobs, B. Mesman, and A. Timmer. Identification and exploitation of symmetries in dsp algorithms. In *Proceedings of the Design, Automation and Test in Europe Conference, DATE'99*, pages 602–608, Munich, Germany, Mar. 1999. IEEE Computer Society Press.
- [77] K. van Eijk, B. Mesman, C. Alba Pinto, Q. Zhao, M. Bekooij, J. van Meerbergen, and J. Jess. Constraint analysis for code generation: Basic techniques and applications in FACTS. *ACM Transactions on Design Automation of Electronic Systems*, 5(4), Oct. 2000.
- [78] J. van Meerbergen, P. Lippens, W. Verhaegh, and A. van der Werf. Relative location assignment for repetitive schedules. In *Proceedings of The European Conference on Design Automation with The European Event in ASIC Design*, pages 403–407, Paris, France, Feb. 1993. IEEE Computer Society Press.
- [79] 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.
- [80] R. Woudsma. Epics, a flexible approach to embedded dsp cores. In *Proceedings of the 5th International Conference on Signal Processing, Applications and Technology*, volume 1, pages 506–511, Dallas, Texas, 1994.
- [81] Q. Zhao, T. Basten, B. Mesman, K. van Eijk, , and J. Jess. Static resource modeling of instruction sets. In *Proceedings of the 14th International Symposium on System Synthesis, ISSS 2001*, Ontario, Canada, Oct. 2001. IEEE Computer Society Press.
- [82] Q. Zhao, K. van Eijk, C. Alba Pinto, , and J. Jess. Register binding for predicated execution in dsp applications. In *Proceedings of the XIII Symposium on Integrated Circuits and Systems Design, SBCCI 2000*, pages 113–118, Manaus, Brazil, Sep. 2000. IEEE Computer Society Press.

Biography

Carlos Antonio Alba Pinto was born on June 3rd, 1971 in Lima, Peru. He concluded his high school in his family town Huaral, Peru in 1987.

He studied Electronics Engineering at the Catholic University in Lima, Peru, from where he graduated with honors on February 1994. He received the degree of Master in Computer Sciences on June 1996 at the Federal University of Rio Grande do Sul, in Porto Alegre, Brazil.

From June 1996 to May 1997, he worked on hardware/software co-design methods and tools at the Federal University of Rio Grande do Sul.

From June 1997 to August 2001, he worked towards a doctorate in the Information and Communication Systems Group, Department of Electrical Engineering of the Eindhoven University of Technology, the Netherlands. He expects to receive this degree based on the work presented in this thesis on June 20th, 2002.

Since September 2001, he is a member of the Embedded Systems and Architectures on Silicon group at Philips Research Laboratories in Eindhoven.