Constraint Analysis for DSP Code Generation

proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 23 mei 2001 om 16.00 uur

door

Bart Mesman

geboren te Eindhoven

Dit proefschrift is goedgekeurd door de promotoren:

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

Druk: Universiteitsdrukkerij, Technische Universiteit Eindhoven

CIP DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Mesman, Bart

Constraint Analysis for DSP Code Generation / Bart Mesman.-
Eindhoven: Eindhoven University of Technology
Thesis Eindhoven. -With summary in Dutch
ISBN 90-74445-52-7
Subject headings: scheduling, code generation, high-level synthesis, compilers, digital signal processing

Acknowledgements

I like to thank Jochen Jess for giving me the opportunity to perform research and the means to share the fruits with many people in the CAD community. I thank Jef van Meerbergen for his mentorship and support. I am gratefull to Koen van Eijk for our co-operation and for the creation of the highly valued FACTS software. I owe thanks to Ad-win Timmer for our many technical discussions and for his help in writing papers. I want to express my gratitude to the philips research lab and the members of the ESAS group and the former digital VLSI group for providing an excellent environment to perform scientific research on industrially relevant topics. I am also thankfull to the ICS group that I still feel part of. I thank my parents and friends for their support. I like to thank Grace and the personnel of La Folie. Many of the ideas expressed in this thesis where conceived there, and unfortunately, many more went up in smoke.

I owe thanks to many other people. I even owe thanks to Toin.

Summary

Code generation methods for digital signal processors are increasingly hampered by the combination of tight timing constraints imposed by signal processing applications and resource constraints implied by the processor architecture. Limited resource availability in the context of pipelined loop schedules poses a problem for greedy scheduling heuristics. Limited and distributed register capacity poses a problem for traditional methods that perform scheduling and register binding in successive stages. This separation often results in suboptimality (or even infeasibility) of the generated solutions because it ignores the problem of phase coupling; since value lifetimes are determined by the schedule, scheduling affects the solution space for register binding. As a result, traditional 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 efficiently cope with the different constraints and the problem of phase coupling.

The approach proposed in this thesis is based on analyzing the constraints to prune the schedule search space. In this way, the scheduler is often prevented from making a decision that inevitably violates one or more constraints. The main aspect of our model of the schedule search space is the distance matrix, which holds the minimum and maximum timing delay between each pair of operations within a Basic Block. Low-order polynomial algorithms identify additional precedence (sequence) constraints that result from the distance relations and the functional resource conflicts. The results of the analyses are combined in the distance matrix by computing the longest paths induced by the precedence constraints. Constraint Analysis interacts with the scheduler by expressing schedule decisions in terms of additional sequence relations and updating the distance matrix.

In order to minimize the register requirements or to satisfy register capacity constraints, the freedom available for scheduling is exploited to serialize value lifetimes. Values are identified that constitute a (potential) bottleneck for register binding, and the corresponding lifetimes are subsequently serialized. Serializations are evaluated in the context of the constraints and the distance matrix is updated accordingly. After the serialization process, each completion of the schedule is guaranteed to induce a valid register binding. In a similar way, the operations that access a register file can be serialized such that the communicated values behave in a streamlined fashion. These values can then be stored in a FIFO. FIFOs have the same addressing cost as registers in terms of instruction bits, but they offer much larger storage capacity. This is convenient because register addressing constitutes about 60% of the code executed on VLIW processors. In a similar way register accesses are serialized in order to store the values in a stack or a FILIFO.

Table of Contents

Chapter 1 Introduction

The last few decades we have witnessed a rapid increase in the number of transistors integrated on a chip. Consequently, we have also witnessed a rapid increase in the amount of man years required to design a complex chip. This increase in design effort must be controlled for at least two reasons. First, chip designers are a scarce resource. Second, there is an enormous pressure to shorten the design time of a chip, because in the consumer electronics market the company with the earliest market introduction of a new product is to expect a large market share. There are basically three major directions in which solutions are sought for controlling the design effort of complex chips:

- Design reuse: This comprises reusing (parts of) a design previously made. This can be done in two ways: either some specification (layout or netlist) called intellectual property [Behn97] is used as a part of a new design, or a design is made programmable so that the chip itself can be used for different applications.
- The use of design tools at increasingly higher abstraction levels: It is clear that designing a chip at the level of transistors is a tedious way of designing, because the complexity is in the order of millions of basic elements. At the other extreme we can specify the functionality of a chip in a high-level programming language such as C, associated with a complexity in the order of hundreds (lines of code). The translation of the C-code (or an intermediate abstraction level) to a transistor-level design is automated using design tools. This translation is called silicon compilation. Designing at a high abstraction level offers a very limited design effort. On the other hand, the implementation is probably not the most efficient in terms of area, power consumption, and performance. These last criteria are however becoming overshadowed by the importance of a limited design effort.
- Programming in a high-level language: Programming a processor requires less effort using a high-level language than using assembly language. Furthermore, high-level code is much more portable and consequently reusable. Here we also need a translation from the high-level language to assembly language (a code compiler).

Surprisingly, silicon compilation and code compilation have a large overlap: In both cases, some form of scheduling and register allocation have to be performed. These two tasks are the subject of this thesis. The application domain on which we focus is called *digital signal processing*.

1.1 Digital signal processing

The area in which digital Very Large Scale Integration (VLSI) chips are applied can be split roughly into two domains as indicated in Figure 1.1: *control processing* and *digital*

signal processing (DSP). Control processing typically involves a lot of decision making; actions are taken based on events generated by the environment in which the chip is applied. Control processing can be further partitioned into the areas of *general purpose computing* and *real-time* applications. In general purpose computing (for personal computers, networks, etc.), processor speed is the main optimization criterion, and there are few hard constraints. Real-Time (RT) applications usually involve hard timing constraints, and hardware cost is a secondary issue. The RT application domain comprises all sorts of regulators: for your heater and vacuum cleaner, but also for safety critical situations like height control in air traffic, anti-skid systems for the brakes in your car, etc. Control tasks are typically executed *sequentially*, mainly because the application involves a lot of control dependencies. As a result, control processors often have limited arithmetic resources, applied in a diversity of computations. This demand for sequential processing and flexibility has led designers to make use of *general purpose processors* (GPP) instead of designing hardware dedicated for the application. As a result, relatively cheap microcontrollers are applied in hundreds of products and sold by the millions each year. When performance is an important issue (e.g. in personal computers and workstations), the GPPs are pushed towards extremely high clock-speeds. This kind of work involves complicated electrical design and tiresome hand-made layout at the physical level. Few companies can afford making their own high-performance GPP, which is justified only when the design is sold in large volume and prices are high.

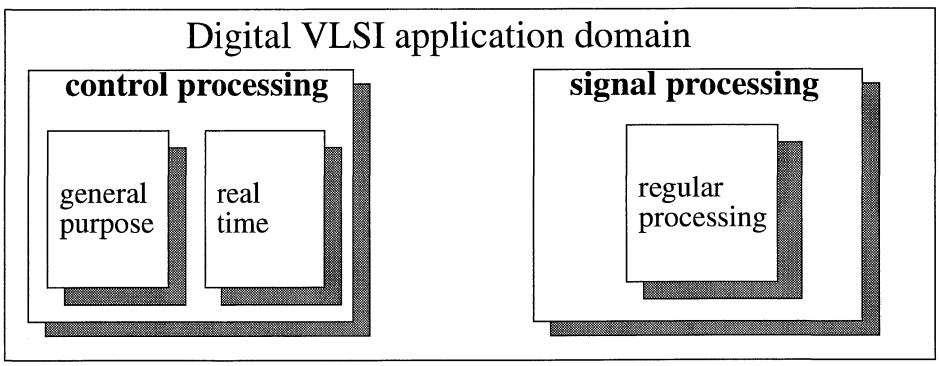


Figure 1.1 Partitioning the application domain of digital VLSI chips

On the other hand, DSP involves a lot of regularity and synchronization; samples are taken and signals are received at fixed periods of time, and the same processing tasks are repeated over and over for each sample or signal. This application domain comprises among others audio and video processing, telecommunication, speech processing and imaging. The regularity in the computations allows parallel processing, and therefore the amount of arithmetic units in DSP processors may be as high as several hundreds, e.g. dedicated processors generated by the automated synthesis toolset Phideo [Meer95]. In this way, performance is obtained by exploiting parallelism rather than high clock speed, which is advantageous for a number of reasons:

- Low clock speed requirements are interesting when considering power consumption. For example, the I.MCiC [Klei97], a single-chip MPEG-2 video encoder runs on 27 MHz. As a result, the power consumption is only 2.1 Watt. The DAB receiver [Huis98], a single-chip for digital audio broadcast runs on just 12 MHz, consuming a mere 0.5Watt.

- Low clock speed requirements also allow the use of widely available standard arithmetic units and automated layout design tools for designing the processor primitives (as opposed to hand-made and heavily pipelined designs). As a result, design activity focuses on high-level and architectural level decisions using automated design tools, which limits the design effort. For example, the I.MCiC mentioned above contains 4.5 million transistors, and is designed within 5 man year using both the Phideo toolset [Meer95] and the Mistral2 toolset [Strik95]. The DAB receiver (also mentioned above) also contains 4.5 million transistors and is developed in 12 man years using the Mistral2 toolset for several parts of the design.

- High clock speed requires a large *pipeline depth* [Henn96]. Instructions remain in the processor for several clock cycles, each representing a stage such as instruction fetch, instruction decode, operand fetch, etc. These stages are called pipeline stages. Consecutive instructions may remain in the processor simultaneously; when instruction *i* fetches its operands, instruction *i+1* is in the decode stage. When the clock speed is high, the number of pipeline stages (the pipeline depth) is necessarily high, which is difficult to oversee both for a programmer and for a compiler. In the case of the TI C60 [TMS97], the performance was boosted with a 200 MHz clock. The result is a pipeline depth between 7 and 11 clock cycles. This leaves the programmer with tedious assembly programming, taking into account overlapping branch delays (5 clock cycles), frequently flushing the pipeline, and looking 7-11 clock cycles ahead which is especially mind-boggling when advanced scheduling techniques like software pipelining are performed.

Summarizing, exploiting parallelism instead of high clock speed as a means to obtain high performance in DSP applications is advantageous for chip design time, power consumption, and some of the complexity of compiler design. However, efficient parallel implementations are (often much) less flexible than using a GPP. Furthermore, a parallel implementation also has some negative effect on the design of a compiler as well. We will focus on these issues in more detail in the following sections, where processor architectures are classified, and the process of mapping an application to a processor is explained. In section 1.3 we will focus more on an alternative processor architecture called VLIW, and see how this architecture relates to classical DSP architectures. Section 1.4 introduces the basic module in the mapping approach taken in this thesis, the constraint analyser. Section 1.5 gives an outline of this thesis.

1.2 Mapping an application to an architecture

In this thesis a method is described for mapping a behavioural specification onto a processor architecture. Processor architectures can roughly be classified using two criteria: *programmability* and instruction set *orthogonality* (Figure 1.2).

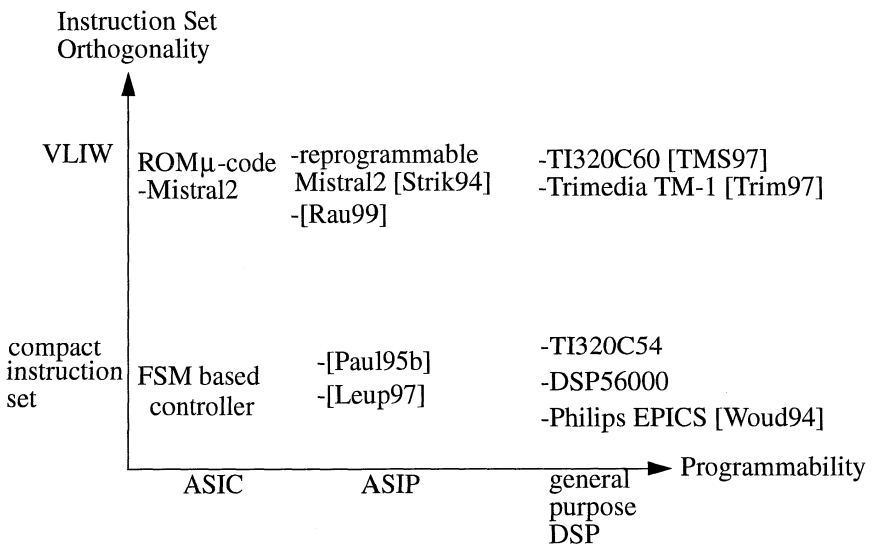


Figure 1.2 Classification of Processor Architectures

There are different degrees of programmability: ASICs are not programmable at all (application specific). ASIPs are programmable, but their performance is tuned to a specific application domain. General purpose DSPs are the most flexible processors that exploit DSP characteristics.

Instruction set orthogonality reflects the ability to control different elements in the data path independently from each other, often by letting independent fields of an instruction control individual data path components [Laps96, p.90]. For example, the register used for storing the result of an operation *v* is independent of the type of operation *v*. In contrast, a non-orthogonal instruction set allows certain operations to be performed on specialized registers only. This introduces a dependency between the instruction field that specifies the operation and the instruction field that specifies the operands for this operation. Although a non-orthogonal instruction set can be encoded very efficiently (in terms of number of instruction bits), this dependency between different instruction fields has to be taken into account by the compiler. A non-orthogonal instruction set is therefore a much more complex compiler target than an orthogonal instruction set. On the other hand, encoding all the possibilities offered by orthogonal processors, necessarily implies a large instruction set, and therefore, wide instruction words.

Because each of the platforms in Figure 1.2 has its characteristic features, we will briefly discuss the architectures and the way their features affect the mapping methodology.

1.2.1 ASICs

An *Application Specific Integrated Circuit* (ASIC) is a chip dedicated to and designed for a single application. The process of translating an ASIC specification to a chip layout is called *silicon compilation.* In Figure 1.3 the design steps of a silicon compiler are depicted. The functionality of the chip is specified using a description language such as VHDL [IEEE88], Silage, or some C dialect. High-Level synthesis, also called architectural synthesis, takes the behavioural description as input, and generates a specification for a so-called data-path and a controller. The data-path consists of functional units (FUs) like multipliers and ALUs, memory, and an interconnection structure. These building blocks are generated using so-called module generators. The controller describes how the flow of data inside the data-path is managed in terms of states and state transitions. The controller description is translated into a configuration of logic gates (or, and, xor, nand, etc.) using logic synthesis. The final synthesis step, called layout synthesis, creates a geometrical description of the layout using placement and routing techniques. The result is a number of layout masks, which are used in an IC foundry to process silicon to chips.

In the case of high-level synthesis the following tasks have to be performed [McFa88]:

- FU Selection: What kind and how many functional units are used in the data-path?
- FU Binding: To which functional units will operations be assigned?
- Scheduling: When will operations from the functional description be executed?
- Register Binding: To which registers will values be assigned?

These four tasks are interrelated, but are difficult to perform simultaneously. Therefore, high-level synthesis strategies solve each problem or a small combination of these problems separately. Most high-level synthesis tools perform these tasks in the order represented above. Interaction with the designer is essential, because as a result of the heuristic nature of the underlying mapping algorithms, the compiler will most likely make some decisions that do not comply with the designer's objectives.

1.2.2 General purpose DSPs

General purpose DSPs (GPDSPs) [Laps96] are the most flexible processors used for DSP applications (although general purpose CPUs are also making steps towards the DSP domain with special multi-media instructions such as Intel's MMX). Characteristic for programmable processors is that the compiler has to deal with a fixed architecture, notably the number of functional units and registers, and the interconnect structure. The controller of a programmable processor is micro coded. Instructions can

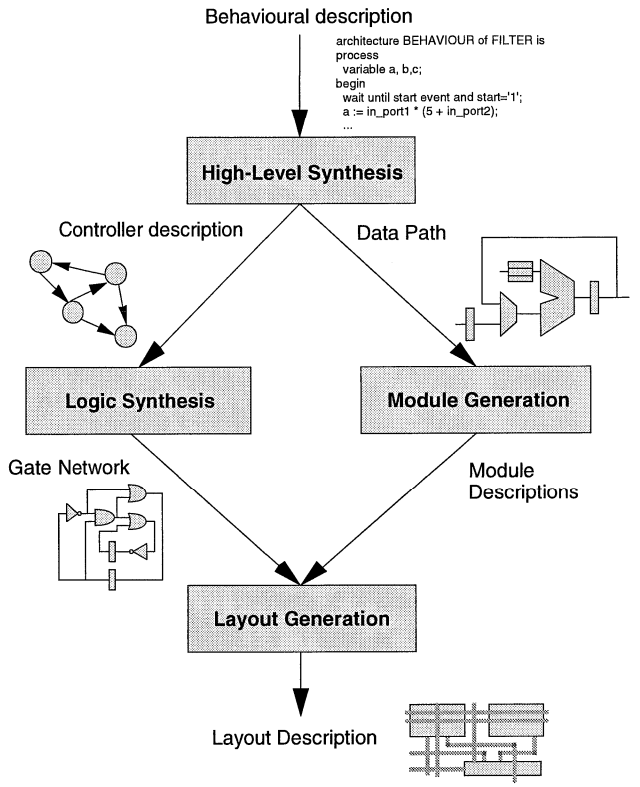


Figure 1.3 Silicon Compiler Overview taken from [heij96]

be loaded into the instruction memory containing all the information the processor needs for proper execution. The register binding and the schedule can be altered by altering the instruction code.

The first GPDSPs were not much more than simple general purpose microcontrollers (like the MIPS R1000 or the Philips 80C51) extended with hardware performing instructions that frequently occur in DSP applications. In filter applications for example, multiplication of a value is most often succeeded by addition of the result with an accumulate register. As a result, a fast *Multiply Accumulate (MAC) unit* and a corresponding single-cycle MAC instruction are part of most GPDSP architectures and instruction sets respectively. To further increase the performance of GPDSPs, the DSP application domain was analysed for even more patterns: the high amount of *regularity* in DSP computations was soon exploited by allocating additional functional units on the DSP, thus enabling more computations simultaneously (in *parallel*). For example, on the DSP5600x a multiply can be performed in parallel with a memory to register move. Soon even more *memory access bandwidth* was required. Bandwidth to background memory was increased (and access latency decreased) by integrating most of the memory *on-chip*. Foreground memory was increased by either *multi-port* register files or with a *distributed* register file architecture. To alleviate the resulting pressure on

the communication bus, larger *communication networks* were allocated with multiple on-chip buses. These extended memory architectures were subsequently augmented with *special addressing modes* such as circular and bit-reversed addressing for FFT computations. Dedicated *address computation units* (ACUs) serve the memories that can handle register-indirect addressing with post-increment for repetitive computations on sequentially stored data. Regular loop structures present in most DSP algorithms are supported and exploited by *hardware loops* and a *repeat* instruction. When used in a small application area, application specific units boost the performance of GPDSPs substantially, as do dedicated peripheral I/O devices.

Besides the exploitation of knowledge of the DSP domain, DSP processor development remains affected by the developments in computer architecture [Henn96]. Most notable are the effects of *heavily pipelining* the processor in order to obtain high clock speed. Since the control hardware is also pipelined, a delayed branch control has to be dealt with, including *flushing the pipeline* when the wrong branch has speculatively been chosen. In order to circumvent some of the pipeline stages, intermediate results are quickly available on bypass networks before writing them to a register file. These bypasses are coordinated at run-time because all kinds of uncertainties (cache misses, data dependent conditions, etc.) are difficult to anticipate at compile-time. For the same reasons, a weak kind of run-time resource scheduling is performed using *reservation tables*. This also offers a larger *window* of instructions to choose from [Henn96], thus increasing the opportunity for more parallel computation. Often programmability is facilitated by allocating an expensive single register file with a relatively large capacity and high access bandwidth.

These performance boosts for GPDSPs have characteristics that may be very disadvantageous for some application areas, because of:

- Power consumption: reservation tables, bypass networks, a large multi-port register file, and dynamic scheduling all consume an amount of power that is not really necessary in the sense that it is not used solely for computation. All this power overhead makes many general purpose DSPs unsuitable for mobile applications, where battery lifetime dictates the usability of an apparatus.

- The hardware features mentioned above occupy valuable chip area and require a large effort for designing the chip layout manually, as already mentioned in the introduction. This is affordable in an industry with very high profits such as the (Intel-type) microprocessor industry. In the consumer electronics industry on the other hand, profits are just a fraction of the cost price of a chip. In this area, general purpose DSPs are used only for prototyping and for a fast introduction of the first generation of a new product in order to gain a profitable market share for later generations of the same product (with cheaper implementations).

For programming GPDSPs roughly the same tasks are identified as for the high-level synthesis of ASICs (section 1.2.1). However, due to the fixed data path, FU selection is

not part of the mapping process. In DSP compilation code selection is considered the most dominant step [Rau99]. The tasks are usually executed in the following order:

- Code selection: Which machine instructions implement the specified behaviour?
- Instruction Scheduling: When will selected instructions be executed?
- Register Binding: To which registers will values be assigned?

Note that in the case of parallel processors an instruction may consist of a number of elementary (arithmetic, load, etc.) operations that are fetched in the same clock cycle. Code selection is the task of determining a set of instructions such that all operations that have to be performed are contained in some instruction. Code selection has to be performed prior to scheduling, because the scheduler is constrained by the instruction set: the operations that are scheduled in the same clock cycle are not guaranteed to combine to a single instruction unless *instructions* are scheduled rather than *operations*. The large overhead in both performance and code size of compiler generated code over manually coded assembly [Paul96] is for a large part due to disappointing results of code selection methods. However, the effectiveness of these methods (and of code selection in general) depends highly on the structure of the instruction set. The transparency of the instruction set for code selection is usually denoted by the rather subjective term 'orthogonality', explained in Section 1.2. Depending on the availability of memory space for certain application areas, either very compact instruction sets are chosen and programming is done manually for the larger part [Woud94], or an orthogonal instruction set is chosen and a large instruction memory is required [Schlan94].

1.2.3 ASIPs

In the previous two subsections we have seen that on the one hand ASICs lack flexibility but offer the most efficient solution in terms of performance, area, and power dissipation. General Purpose DSPs on the other hand offer a lot of flexibility, but are often not able to satisfy performance, area, or power dissipation requirements. Application-Specific Instruction Processors (ASIPs) have become popular due to their advantageous trade-off between the ASIC characteristics and the GPDSP's flexibility. An ASIP is a programmable processor tuned to a specific application domain. Often a large part of the functional units consists of application specific units (ASUs), that efficiently perform computations characteristic for the application domain. It appears that the use of these ASUs can reduce the power consumption of general purpose DSPs by factors in the order of 10-100 [Meer99, section 5.5].

A micro coded controller provides flexibility, but as explained in the previous subsection, the necessary instructions occupy valuable chip area. This is especially true for embedded applications (for which ASIPs are mostly used) where instruction memory is kept on the same chip as the ASIP itself, together with other processors. Because on-chip memory requires more area than off-chip stand-alone memory (mostly because the memory has to be implemented by logic technology rather than the more area effi-

cient memory technology), there is a hard pressure for minimizing the required amount of instruction memory. The number of bits required to encode an instruction set is proportional to the cardinality of the instruction set. ASIP designers have therefore carefully selected an instruction set based on profiling information of the application domain. There is even some research effort to do this automatically[Alom93]. The instruction set for an ASIP is thus a trade-off between *cost* measured by the instruction width, and *performance* measured by the number of operations that a single instruction encodes [Paul95a]. The result is an instruction set with very little regularity or structure, which does not provide a simple transparent processor model to a compiler. Programmability of ASIPs is often considered as an afterthought, partly because they are meant to be programmed in assembly as a result of the pressure on highly optimized code with high-volume electronics. Current compilers for these processors (and more general fixed point DSPs) tend to produce an intolerably large overhead in code size and performance [Zivo94].

However, due to the increasing competition in the consumer electronics sector, time to market is gaining priority, which puts a lot of pressure on design productivity. Programming ASIPs in a higher programming language like C is therefore becoming a necessity, and research efforts in automated compilation techniques for ASIPs have increased during the last decade.

The compiler steps are the same as for general purpose DSPs:

- Code selection: Which instruction will be executed?
- Instruction Scheduling: When will this instruction be executed?
- Register Binding: To which registers will values be assigned?

An orthogonal instruction set provides a transparent processor model for the scheduler [Timm95], [Strik95], so that the task of code selection is alleviated and the emphasis is placed on scheduling and register binding. However, for ASIP compilation there is usually an even larger emphasis placed on the task of code selection [Marw95], because ASIP instruction sets and architectures typically exhibit more irregularity than general purpose DSPs. This is amplified by the requirement of *retargetability* [Lann95], [Paul95b]: A specific ASIP is designed for a narrow application domain. However, making a compiler for each separate ASIP is simply too much effort. Instead, a single 'parameterizable' or retargetable compiler is designed, that makes certain assumptions on the topology of the architecture, and the rest of the architecture information is read from a machine description file, as depicted in Figure 1.4.

The processor architecture is specified using a machine description language such as nML. The following architecture aspects are typically specified in such a language [Rau99]:

- number of functional units

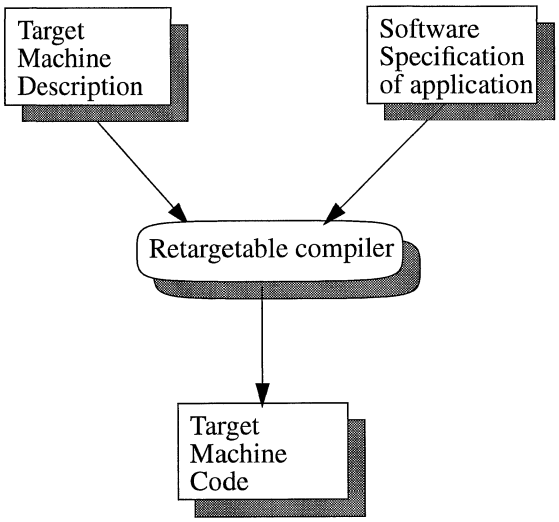


Figure 1.4 Retargetable Compilation

- FU pipeline structure
- FU latencies and throughput
- set of opcodes that each FU can execute
- number of register files
- number of registers per register file
- addressability of the registers
- interconnect between the register files and FUs

Retargetability has a large effect on the range of techniques applied in code selection. A processor specific compiler can exploit instructions that are very specific for the processor. Suppose for example, that a processor is able to encode two parallel moves from memory to register in a single instruction, provided that the first move targets either register r0 or r1, and the second move targets register r2 or r3. This highly efficient instruction can be exploited by the compiler, but it requires processor specific ad-hoc techniques to test the possibilities of exploiting this parallel instruction. It would cost an intolerable amount of effort to retarget such a compiler. As a result, retargetable compilers can only afford to employ generic methods such as graph matching and covering algorithms [Liem94], [Praet94]. This has resulted in poor performance: Despite all the research effort spent on the subject of code selection, current ASIP compilers perform 2 to 8 times worse than manually written assembly on both speed and code size [Paul96].

In the next section we will consider the VLIW processor architecture that provides a compiler friendly processor model at the cost of larger code size.

1.3 The Very Large Instruction Word architecture

The first generation of Very Large Instruction Word (VLIW) processors were developed with the specific goal of making the architecture suitable for automatic code generation [Fish83] [Rau82] by providing a highly orthogonal instruction set. These processors typically provide higher levels of instruction-level parallelism (ILP), more registers, and a regular interconnect. In this way a compiler is able to generate high-quality code using systematic rather than ad hoc techniques. The data-path of a typical ASIP VLIW architecture is given in Figure 1.5. A number of functional units (FUs) executes in parallel, each fetching its operands from dedicated or 'weakly' shared register files (RFs) at the beginning of a clock cycle, and writing the result to another RF at the end of the clock cycle. General purpose DSPs with a VLIW architecture often have one large register file, such as the Trimedia TM-1 [Trim97], or two large register files, such as the TI320C60 [TMS97]. This will provide an easier compiler target for most tradi-

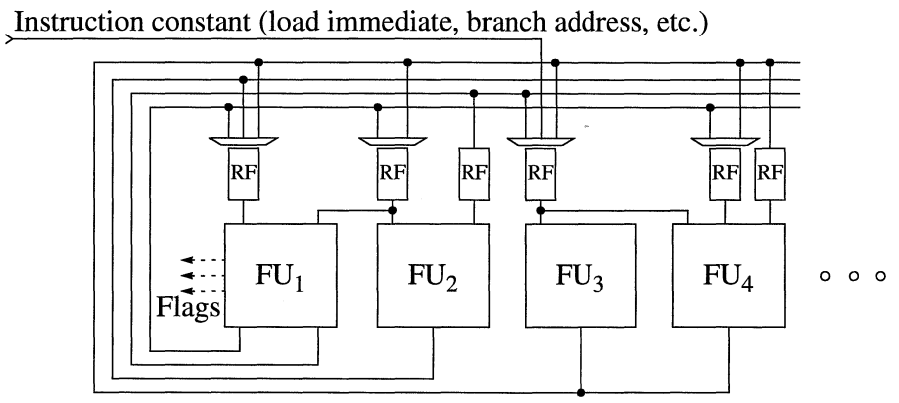


Figure 1.5 Data-path of a typical VLIW architecture

tional compiling techniques at the cost of expensive, slower, power inefficient hardware and wider instructions, as will be shown in section 1.3.2. Distributed register files such as in Figure 1.5 are more typical of ASIPs [Strik94].

1.3.1 Code generation for VLIW processors

In its 'ideal' form [Rau81], each functional unit is controlled by dedicated instruction bits that are completely independent from the bits controlling other FUs. Therefore **every** combination of operations is guaranteed to be encoded by an instruction word, provided that these operations execute on different FUs. This has the following major consequences:

- Instruction selection can be performed **after** scheduling and register binding, thus providing much more freedom for both scheduling and register binding.

- Instead of scheduling instructions, we can schedule individual operations.

- Instruction selection has become a trivial task

So the 'ideal' VLIW architecture eliminates all the difficulties that accompany instruction selection for less 'regular' architectures and shifts the emphasis on scheduling and register binding, thus providing more opportunity to exploit the available parallelism. 'Less' ideal VLIW architectures pose only a limited restriction in that sense. The Trimedia TM-1 [Trim97] processor for example, has 27 functional units, but for several reasons each instruction is able to control only 5 functional units in parallel in the following way. Functional units are grouped in clusters, where each cluster is assigned one of the five *issue slots* that comprise an instruction word. The structure of this instruction set architecture can be modelled for the scheduler in terms of 'regular' resource constraints [Bras99]. Furthermore, the compiler can be retargeted by regrouping functional units and introducing resource constraints accordingly. We conclude that 'regular' instruction set restrictions like issue slot constraints, can be taken into account during scheduling without the difficulties associated with explicit instruction selection. More recent research [Leij00] introduces constraints in the interconnect between functional units and register files. Coping with such constraints is a topic of ongoing research [Beko00].

Whereas the trend in general purpose processor design is towards increasingly higher clock speeds for performance, the VLIW architecture is more focused on the parallel execution of operations. Exploiting this ILP however, is a lot more difficult than exploiting clock speed. So the VLIW architecture emphasizes scheduling not only because instruction selection is less of a problem, but also because there is a lot more pressure on the performance of the scheduler, or more specifically: the scheduler's ability to exploit the parallelism available in the VLIW architecture. There are roughly two ways the scheduler can exploit this parallelism: *Global scheduling* and *loop scheduling.*

In order to understand these scheduling mechanisms, a few words are spent on the way an application algorithm is specified for the scheduler. Such an algorithm is divided into so called *basic blocks*, which consist of operations, and possibly other basic blocks (hierarchically). The division into basic blocks is determined by the control flow within the algorithm. For example, an if-then-else construct in basic block A will generate two new basic blocks B and C that are part of A. Basic block B contains the operations (and possibly basic blocks) specified in the then branch, and C contains those specified in the else branch. Another example of a basic block is a while or for loop. Traditional scheduling techniques consider the operations in a single basic block. Therefore the ILP that can be exploited is limited to the ILP present in one basic block. Global scheduling extends the opportunity for exploiting ILP beyond the basic block boundaries induced by if-then-else constructions, whereas loop scheduling extends the opportunity for exploiting ILP by considering more iterations of the same loop.

The best-known implementation of global scheduling is probably Trace scheduling [Fish81]. Most characteristic about global scheduling is that it extends the scheduling *window*, the set of instructions (or operations) examined for simultaneous execution

[Henn96]. This is done using so called *if-conversion:* removing some of the boundaries between Basic-Blocks. Extending the window can be done at compile time or at run time. The idea is that if there are more operations within the scheduling window, the scheduler has more opportunity to find combinations of operations that can execute simultaneously. At any given time t, the window consists for a large part of operations that are conditional, but at time t the value of this condition may not be known. These operations are executed speculatively: only when the condition has been calculated it is known whether the results of these operations will be used or disregarded. General purpose DSPs rely for their performance for a very large part on global scheduling: efficient schedules for general purpose DSPs with a VLIW architecture execute about half of the operations speculatively [Hoog99]. As a result of global scheduling operations are often duplicated and encoded more than once in the program code. Global scheduling therefore tends to increase code size. The importance of global scheduling for general purpose DSPs is explained by the fact that control oriented code (where basic blocks typically contain little ILP) comprises a large part of typical code mapped on these processors.

As depicted in Figure 1.2, more application specific processors (ASIPs and ASICs) may also employ the VLIW architecture. These processors are often embedded on a chip together with instruction memory (or cache), and therefore code size has to be limited. The problems that VLIW architectures have with code size often limit their application to time-critical code segments. On the other hand, time critical code consists mostly of 'regular' loops that are executed many times. (In signal processing a general rule of thumb is that 80% of the execution time is spent in 20% of the code.) These regular loops can be scheduled very efficiently by either *loop unrolling* [Henn96] or *loop pipelining* [Lam88](also called loop folding or software pipelining) or a combination of both. Both techniques try to overlap the schedules of subsequent loop iterations in order to exploit the available architectural parallelism.

Loop unrolling basically copies the operations in the loop body a number of times before scheduling. The resulting loop is scheduled using a conventional scheduler. The advantages of loop unrolling are that the scheduler can be kept simple, and that there is more 'room' for optimization in the sense that every loop copy can be scheduled differently from the others. One disadvantage is that the beginning and end of the resulting schedule will be relatively sparse (few operations can be executed in parallel) because begin and end do not overlap unless loop pipelining is applied. This 'overhead' can be minimized by a large unrolling factor. This measure combines badly with the main disadvantage of loop unrolling: the code size increases with approximately the same factor. Furthermore, since the problem instance also grows with the same factor, only low-complexity scheduling algorithms can be used.

Loop folding demands that all overlapping loop bodies are scheduled in exactly the same way. The advantage of this is that the same code is used for almost every loop iteration (thus code size is limited) and that parallel code is obtained at every point in the loop kernel. The main disadvantages are that special (intelligent) schedule algorithms

are required, and that so called *preamble* and *postamble* code must be added (see section 2.2) outside the loop.

Research [Aiken95] suggests that loop pipelining is as effective as full loop unrolling, while producing less code [Henn96]. In this thesis we will therefore focus on loop pipelining.

1.3.2 Register file architecture

In this section the pros and cons of an architecture with one large multi-ported register file and an architecture with multiple register files are discussed. Traditionally, the 'ideal' VLIW architecture contains a single large register file [Fish83], [Rau82]. From the compiler perspective this architecture is indeed ideal: for each value, the register binder has the full register address range at its disposal, and no copies of values need to be generated, so the register pressure is relatively low and no additional communication is required. Most other criteria are however in favour of distributing the registers over a number of files. Since these criteria have gained importance during the last decade, it is unlikely that in the future VLIW processors will be designed with a single large register file. We mention some of these criteria below. Let W denote the number of words in a register file, and let P denote the number of access ports. Furthermore, "a" is a constant that depends on some design and technology parameters and ranges from .5 to 1.

- *Power consumption*: For a single access to the register file, the power consumption is $O(P \cdot W^a)$.

- *Access delay*: The access delay is of the same order as the power consumption for one access, $O(P \cdot W^a)$. Access delay is often a persistent bottleneck in processor design. In order to keep the delay within limits, parallel memories can be used (with fewer ports), but consistency between these memories has to be maintained, which has serious effects on both the (manual) design effort and the power consumption.

- *Code size*: Code size is an important criterion for different reasons: for off-chip instruction memory, power consumption for off-chip communication is the main reason. For on-chip (embedded) instruction memory, area is more important. Code size is for a large part determined by the *instruction width*. In the following it will become clear that a distributed register file architecture yields a smaller instruction width than a single register file.

If all registers are concentrated in one register file, each access to this file has to provide an address from the range of **all** registers. The number of accesses to this large register file amounts to 3 times the number of functional units that are addressed in one instruction word, because it is assumed that a functional unit fetches two operands and writes one result. Suppose there are 5 instruction slots and 128 registers arranged in one file, such as in the Trimedia TM-1 [Trim97]. These registers are addressed using $^2\log 128 = 7$ bits, so the number of bits used

for register addressing in a single instruction word, amounts to $5 \cdot (3 \cdot 7) = 105$ bits (guard operands are not taken into account in this calculation). If these 128 registers were distributed over 8 register files of 16 registers each, and each operand for a functional unit can be taken from exactly one such register file, then the address for each source operand takes only $^{2}\log 16 = 4$ bits. It is assumed that the result of a computation can be routed to each register file, so the destination operand has the full register address range at its disposal. Now the number of bits used for register addressing in a single instruction word equals $5 \cdot (2 \cdot 4 + 1 \cdot 7) = 75$ bits, so in this example the single register file architecture uses 40% more bits on register addressing than the distributed register file alternative. If the number of register files would be larger, then the difference in instruction width would be even more dramatic.

1.4 Constraint analysis

The aim of this thesis is to describe a good method of scheduling and register binding for VLIW (and similar) architectures, both application specific and programmable. The problems of scheduling and register binding are however fundamentally different for programmable processors and non-programmable processors: For ASICs it is our aim to minimize the number of registers (for the given constraint set). For a programmable processor on the other hand, the number of registers *used* in each register file is actually irrelevant as long as this number does not exceed the number of *available* registers in that file. It could even be advantageous to exploit all available registers in order to give the scheduler more opportunity to satisfy the timing and resource constraints. For our understanding of the similarities and differences between these two problems, we define three types of feasibility:

- T-feasibility: Timing, precedence, and resource constraints are satisfied
- R-feasibility: T-feasibility extended with a register binding that is consistent with the timing, precedence, and resource constraints.
- S-feasibility: R-feasibility, but now the register binding also has to respect fixed individual register file sizes.

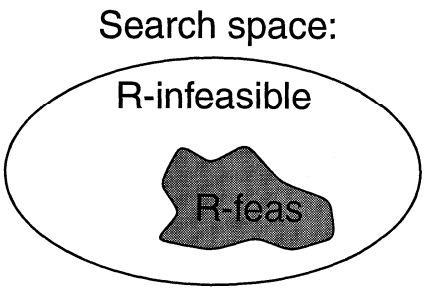


Figure 1.6 The search scope is restricted to the R-feasible region.

For ASICs we try to find an R-feasible solution with the minimum number of registers, and for programmable processors we need to find an arbitrary S-feasible solution. Note that in both cases the solution has to be R-feasible. This observation suggests a modular approach with a basic module that restricts the search scope to the R-feasible region, as depicted in Figure 1.6. This module should also be able to detect infeasibility of the constraint set in order to avoid a lengthy exhaustive search. We call this module the *constraint analyser* [Mesm99]. The techniques in the constraint analyser constitute the major contribution of this thesis.

1.5 Thesis outline

In the next chapter we will define the basic concepts necessary for understanding some scheduling problems. After introducing the data flow graph model and discussing a general scheduling problem, the two main scheduling and register binding problems are formalized, and a solution approach is outlined. In chapter 3 we will see how the constraint analyser handles resource constraints, particularly in the context of loop pipelining. The way a given (partial) register binding is handled by the constraint analyser is treated in chapter 4. The way the constraint analyser is used in finding an efficient R-feasible solution (for ASICs) and an S-feasible solution (for programmable processors) is treated in chapter 4 as well. In chapter 5 we try to enforce lifetimes in such a way that they fit in other types of register files, such as FIFOs and stacks, instead of addressable register files. These (foreground) memory units have the advantage that the address mechanism requires fewer instruction bits, whereas a potentially large storage capacity can be provided. Chapter 6 provides a summary of the thesis.

Chapter 2 Operation Scheduling

In this chapter we will introduce the two fundamental problems that are the subject of this thesis: operation scheduling for minimum register requirements (for ASICs) and operation scheduling for fixed register file sizes (for reprogrammable architectures). In Section 2.1 the basic scheduling model and some definitions are given. Section 2.2 extends the range of possible schedules by introducing the concept of *loop pipelining*. Section 2.3 discusses the traditional high-level synthesis scheduling problem and informally illustrates the difficulty of finding a pipelined schedule. In Section 2.4 it is shown how the constraints and problem specific characteristics are modelled in the *Data Flow Graph* model of Section 2.1. Our two fundamental scheduling problems are defined in Section 2.5. In Section 2.6 some inititialization issues are addressed for our scheduling approach.

2.1 Definitions

We start with the definition of the most widely used RTL-level specification model for an application program: the Data Flow Graph, (DFG) [Ku92].

Definition 2.1 (Data Flow Graph) A data flow graph is a triple $(V, E_d \cup E_s, w)$, where

- V is the set of vertices (operations),
- $E_d \subseteq V \times V$ is the set of directed *data* precedence edges,
- $E_s \subseteq V \times V$ is the set of directed *sequence* precedence edges, and
- w: $E_d \cup E_s \rightarrow Z$ describes the timing delay associated with a precedence edge. ❑

The main difference with DFG models like that from [Ku92] is the emphasis on (sequence) edges. First, minimal delay between operations is associated with the edges rather than the operations. Second, our methodology is heavily based on the concept of precedence, modelled by sequence edges. Note that our definition of a DFG does not require the graph to be acyclic.

An example of a DFG for an IIR filter application is found in Figure 2.1. Typical operations are arithmetic and logical calculations, address computations, memory-reads and writes, i/o operations, and application specific operations.

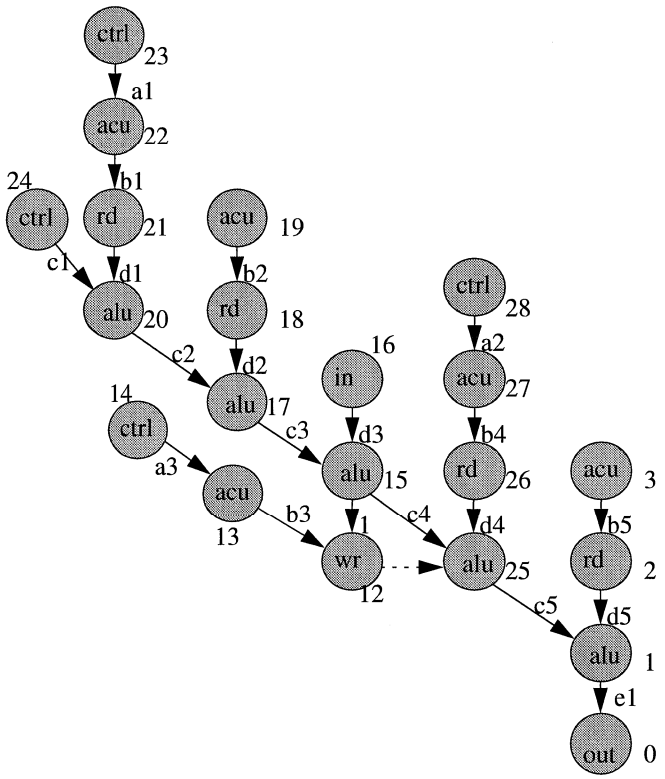


Figure 2.1 : A data flow graph for an IIR filter

The precedence edges define a partial order on the executions of the operations. In this thesis, whenever necessary, a distinction between data edges and sequence edges in a DFG is visualized by drawing data edges with a solid edge, and sequence edges by a dashed edge (e.g. operation 12 to operation 25 in Figure 2.1). Furthermore, data edges have a default delay of one clock cycle and sequence edges have a default delay of zero clock cycles. Most of the constraints that accompany a scheduling problem relate to the start time of an operation, the time that its execution starts. The start times of the operations in a data flow graph comprise a schedule:

Definition 2.2 (schedule) s:V $\rightarrow$ N describes the start times of operations, where N denotes the set of natural numbers. ❑

A schedule is constrained by precedence edges. A precedence (v_i, v_j) with delay $w(v_i, v_j)$ expresses that

$$s(v_j) \geq s(v_i) + w(v_i, v_j)$$

In the text, whenever $w(v_i, v_j) \geq 0$, a precedence (v_i, v_j) will be indicated by $v_i \rightarrow v_j$.

Definition 2.3 (latency) l is the number of clock cycles required to execute a schedule.

A schedule for the DFG in Figure 2.1 is found in Figure 2.2 a). Note that in this example, each operation executes in one clock cycle. In Section 2.4 we will show how multi-cycle executions are modelled using precedence constraints. In the schedule of Figure 2.2 a) the operations are grouped (in columns) with respect to the functional units they are mapped onto. When two operations are mapped to the same functional unit they cannot execute simultaneously. There are other reasons why two operations cannot execute in parallel, e.g. they transport the result of the computation over the same bus. Alternatively, there may exist no instruction in the instruction set of a programmable processor encoding the parallel execution of v_i and v_j although these operations are mapped to different functional units. These constraints preventing parallel execution are called *resource constraints*, and are given by the function $rsc(v_i, v_j)$: $V \times V \rightarrow \{0,1\}$, defined by

$$rsc(v_i, v_j) = \begin{cases} 1 & \text{if } v_i \text{and} v_j \text{have a conflict} \\ 0 & \text{otherwise} \end{cases} \qquad (2.1)$$

A resource constraint $rsc(v_i, v_j)$ expresses that

$$rsc(v_i, v_j) = 1 \Rightarrow s(v_i) \neq s(v_j) \qquad (2.2)$$

A valid schedule has to satisfy the resource constraints. Both the resource constraints and the precedence constraints limit the *Instruction-Level Parallelism* (ILP), the number of operations that can execute in parallel. For loops, a particularly efficient way of scheduling circumvents the limiting effect of most of the precedence constraints.

2.2 Pipelined schedules

In a loop construction the *loop body* (represented by a DFG) is executed a number of times. In a traditional schedule, iteration $i + 1$ of the loop body is executed strictly after the execution of the i^{th} iteration. [Lam88] and [Goos89] demonstrate a practical way to overlap the executions of different loop body iterations, thus obtaining potentially much more efficient schedules. This way of scheduling is called *loop folding, loop pipelining,* or *software pipelining*.

Definition 2.4 (Initiation Interval II) The Initiation Interval (II) is the period between the start times of the execution of two successive loop iterations. ❑

Loop pipelining allows the execution of operations from iteration i in parallel with or even after the execution of other operations from iteration $i + 1$. Compare for example the two schedules in Figure 2.2 for the graph in Figure 2.1, one without pipelining, the other pipelined in such a way that the initiation interval equals 7 clock cycles.

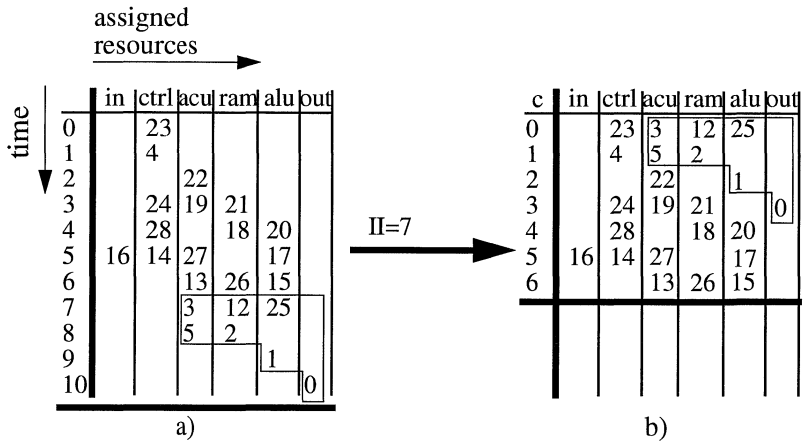


Figure 2.2 a) Schedule for the DFG in Figure 2.1 b) pipelined schedule

The operations of iteration i originally scheduled in the last 4 cycles are now executed simultaneously with the first 4 cycles of iteration $i+1$. In this case, the pipelined schedule is obtained by pipelining an *existing* schedule, and loop pipelining causes a 57% increase in the throughput. Later we will see that an even more efficient schedule is obtained when loops are pipelined in the process of constructing a schedule. However, this can only be done efficiently if the scheduler oversees the constraints present between operations belonging to different loop iterations. Thus we should be able to express precedence and timing relations that cross the loop boundary. It is useful for our purpose to label these operations with their iteration index, so we let C_i denote the i^{th} execution of operation C. Suppose we want to express in the DFG the fact that $C_i \rightarrow P_{i+k}$. This precedence from C_i to P_{i+k} has consequences for the timing relation between C_i and P_i, involving the time between successive initiations of a loop, the initiation interval (II). This timing relation can be derived from the following equations. Equation (2.3) expresses the constraint that the time between two successive loop initiations is fixed to II.

$$s(P_{i+k}) = s(P_i) + k \cdot \mathrm{II} \tag{2.3}$$

Equation (2.4) expresses the consequences of a precedence relation for the starting times of the operations.

$$(C_i \rightarrow P_{i+k}) \Leftrightarrow s(P_{i+k}) \geq s(C_i) \tag{2.4}$$

Substituting (2.3) in (2.4) yields:

$$(C_i \rightarrow P_{i+k}) \Leftrightarrow s(P_i) \geq s(C_i) - k \cdot \mathrm{II} \tag{2.5}$$

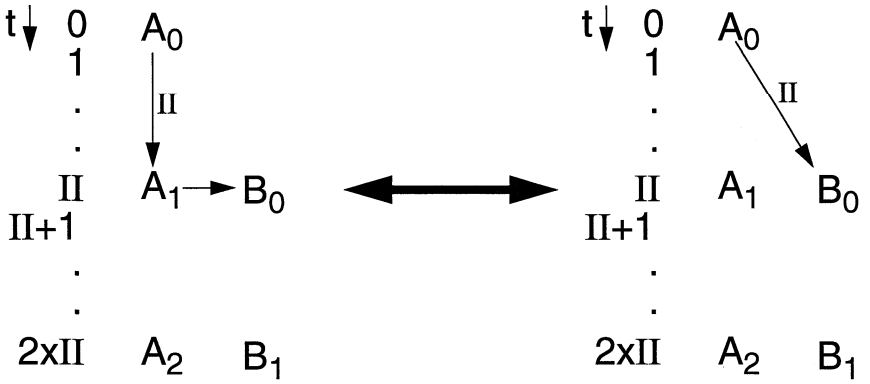


Figure 2.3 Graphical representation of equation (2.5)

Equation (2.5) expresses the effect of a precedence relation $C_i \rightarrow P_i$ which effectively has a delay $-k \cdot II$. When the precedence $C_i \rightarrow P_{i+k}$ has a timing delay w, the projected precedence relation $C_i \rightarrow P_i$ effectively has a delay $w - k \cdot II$. For $k = 1$ equation (2.5) is visualized in Figure 2.3. Note that in equation (2.5) and Figure 2.3 the implication is in two directions. This means that the precedence $C_i \rightarrow P_{i+k}$ is functionally equivalent to a precedence $C_i \rightarrow P_i$ with delay $-k \cdot II$, which can be expressed in the DFG.

We have now found a way to derive so called *inter-iteration dependencies* [Lam88] or *loop carried dependencies* [Govin94] from the normal precedence relations in the data-flow graph and the initiation interval. We should however also be able to express resource conflicts that cross loop boundaries. Therefore Equation (2.2) is generalized to Equation (2.6).

$$rsc(v_i, v_j) = 1 \Rightarrow s(v_i) \mod II \neq s(v_j) \mod II \qquad (2.6)$$

The term $s(v_i) \mod II$ is called the *time potential* of v_i.

We are now ready to introduce the traditional High-Level Synthesis scheduling Problem.

2.3 The High-Level Synthesis scheduling Problem

The general high-level synthesis (feasibility) scheduling problem is formulated as follows:

Definition 2.5 (High-Level Synthesis Scheduling Problem) Given are a DFG, a function $rsc(v_i, v_j)$, an initiation interval II, and a constraint on the latency l

(completion time). Find a schedule s that satisfies the precedence constraints, the resource constraints, and the timing constraints II and l. ❑

The high-level synthesis scheduling problem is NP-complete because it generalizes the NP-complete problem of sequencing with release times and deadlines [Garey79, p.236]. The corresponding optimization problems, minimizing l or II, are therefore NP-hard. As a result, many heuristic approaches can be found in the literature, an early overview of which is given in [McFa90].

By far the most widely used type of schedule heuristic is called *list-scheduling* [Hu61]. List scheduling became well-known after the theoretical treatment in [Coff76], and was introduced in the HLS community by [Girc84]. The basic algorithm works as follows: Starting with clock cycle 0, clock cycles are 'filled' with operations. For each clock cycle, a 'ready-list' is kept, containing those operations that are 'ready' to be scheduled (that is, their predecessors have already been scheduled in earlier clock cycles). Operations are taken from the ready-list and scheduled at the current clock cycle. When some operations in the ready-list have a resource conflict, an operation is selected based on a *priority function*, and the remaining conflicting operations are moved to the ready list corresponding to the subsequent clock cycle. In this way many variations of the basic list-scheduler exist by granting priority based on, among others, the number of predecessors or successors of an operation, the ASAP value, the ALAP value, the mobility, etc. In [Thom90] some experiments are done with different priority functions. An overview of priority functions can be found in [Heij91].

Researchers from both the general-purpose computing [Rau81] and the HLS community [Goos89] have used the list-scheduling principle for generating pipelined loop schedules. It appears to be much more difficult to find good priority functions for generating pipelined schedules than for regular (non-pipelined) schedules. This disparity stems from the fact that for pipelined schedules, resource conflicts have to be solved between operations that belong to different loop iterations. The difficulty in handling these *inter-iteration conflicts* is illustrated with a small example in Figure 2.4. In this figure, a precedence graph of 5 operations is given. In order to meet the constraint of 3 clock cycles on the initiation interval (II), loop pipelining has to be applied (indicated by the arrow in Figure 2.4 b and c). Because pipelining introduces extra code, we do not want to fold more than once, which constrains the latency to 6 clock cycles. In Figure 2.4b the result of a list scheduler is shown. The left column contains the time potential. The list scheduler greedily schedules A, B, and C as soon as possible (ASAP), and concludes that D cannot be scheduled. In Figure 2.4c a valid schedule is given. The key to obtaining this schedule is to postpone B one clock cycle relative to its ASAP value. However, most schedule heuristics (notably list-scheduling) are simply too greedy to postpone operations. In Section 3.5 we will demonstrate that an approach based on analysing the constraints finds the only feasible schedule. In order to this efficiently, care must be taken that the DFG model suffices for expressing most of the constraints and constructions that are allowed.

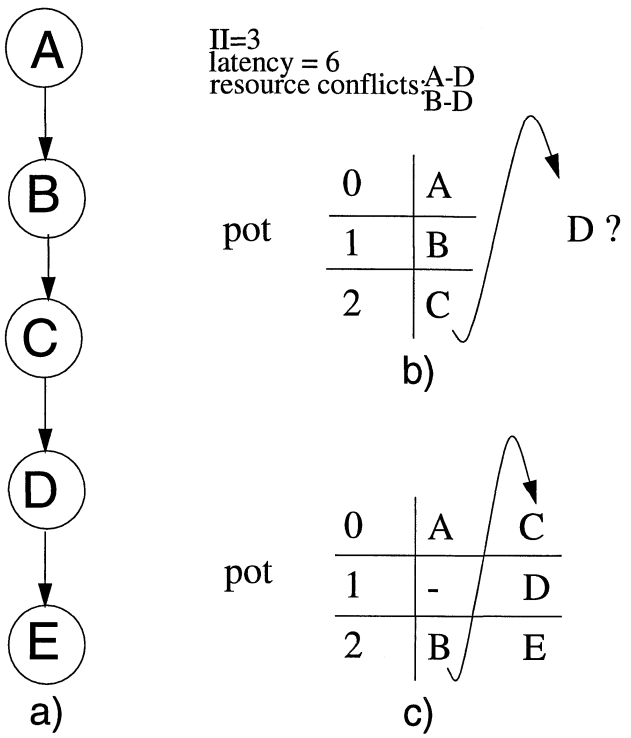


Figure 2.4 Example with loop pipelining. a) precedence graph b) list-schedule c) only feasible schedule in 6 clock

2.4 Modelling the constraints

In this Section we show how some of the constraints can be represented in the DFG model introduced in Section 2.1. We start by expressing the latency in terms of a precedence relation.

- **Latency**. In order to model latency, we introduce two (dummy) operations to our DFG model: the source and the sink. The source operation is the 'first' operation, and the sink operation is the 'last' one, so the start time of each operation is lower bounded by the source operation, and upper bounded by the sink operation: $\forall(v_i \in V)$: $s(\text{source}) \leq s(v_i) < s(\text{sink})$. A constraint l on the latency is now modelled by an arc (sink, source) with w=-l, illustrated in Figure 2.5. This is interpreted as $s(\text{source}) \geq s(\text{sink}) - l$, which is equivalent to $s(\text{sink}) \leq s(\text{source}) + l$, meaning the sink operation may not be executed more than l clock cycles after the start of the source operation.

- **Micro coded controller, randomly addressable register files and loop pipelining.** We assume that the architecture contains a micro coded controller. As a conse-

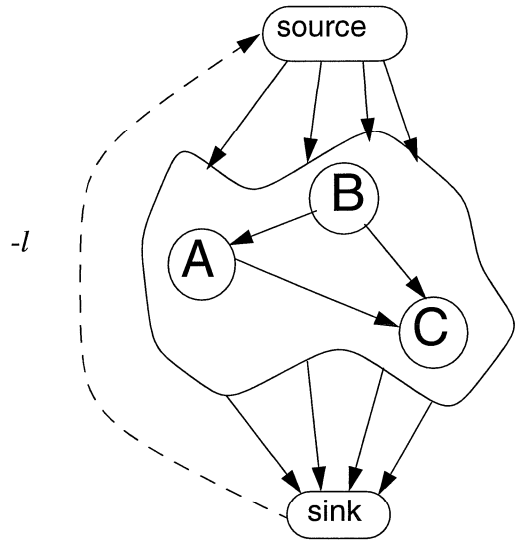


Figure 2.5 Modelling the latency

quence, the same code is executed every loop iteration. This implies that a communicated value is written in the same register each iteration. When loop iterations overlap, we have to ensure that a value is consumed before it is overwritten by the next production. Since subsequent productions are exactly II (initiation interval) clock cycles apart, a value cannot be alive longer than II clock cycles. So the operation C that consumes a value must execute within II clock cycles after the operation P that produced the value. Just like the latency constraint, a necessary and sufficient translation to the precedence model is that for each data dependency (P,C) there is an arc (C,P) with $w = -II$. Note that this constraint is not implied by all register file models; in Section 5.1 we treat fifos which can contain values with a lifetime exceeding the initiation interval.

- **Pipelined executions and multicycle operations**. These are operations that violate our assumption of operations to take one clock cycle to execute. Conceptually, they are split in a number of 'stages' for each clock cycle. In our model, an operation is introduced for each stage of the execution. Subsequent stages are linked in time using two sequence edges as indicated in Figure 2.6. For multicycle operations, A and B occupy the same resource. Pipelined operations are allowed to overlap in time, and therefore imply a resource conflict only between operations that correspond to the same pipeline stage.

- **Scheduling decisions**. When schedule decisions are taken during the process, the schedule intervals of other operations are affected. Therefore it is desirable to be able to express a schedule decision in the DFG, so that its effect can be analysed in the context of the other constraints. Scheduling decisions may take different forms. A

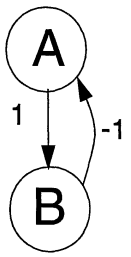


Figure 2.6 Modelling pipelined and multicycle operations

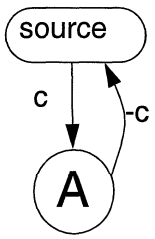


Figure 2.7 Modelling a schedule decision

timing relation between two operations can be directly translated to a sequence edge. When an operation A is fixed at a certain clock cycle c, we need two sequence edges as indicated in Figure 2.7

- **Resource conflicts and instruction set conflicts.** In Section 2.1 the resource conflict model $rsc(v_i, v_j)$ was introduced. In this thesis it is assumed that instruction set conflicts are expressed in the resource conflict model. Relatively simple instruction set conflicts (e.g. that prevent the simultaneous execution of two types of operations) can be expressed in this model by the method explained in [Timm95]. More general issue slot constraints require more modelling effort [Bras99].

2.5 Problem formulation

In order to formulate the problem we need to state some assumptions:

- All operations have been mapped to functional units. This is often the case because instruction selection is usually done prior to the scheduling phase (see for example [Liem94]), thus providing a resource binding.

- All values communicated between operations have been mapped to register files. In ASIP-architectures, a functional unit usually gets its operands from a specified register file, so instruction selection (implicitly) determines the assignment of values to register files. Within a register file there are multiple registers however, and the assignment of values to these registers remains to be decided.

- The controller is micro coded. One consequence is that in a pipelined loop a value cannot reside in a certain register for a period longer than the *initiation interval*, which is the period of initiating the schedule for a loop iteration. Another restriction is that a loop-body execution is the same for each loop index. This is not the case for e.g. the Phideo processor architecture [Meer95], for which potentially better schedules can be obtained.

- The initiation interval II for each hierarchical level is fixed prior to scheduling. It can be fixed by the designer. Otherwise, we start with a lower bound as explained in Section 2.6. When this value of the Initiation Interval is not feasible, it is incremented. Profiling suggests that the optimal II is usually only one or two clock cycles away from the lower bound.

In this thesis two different scheduling approaches are treated: one for **minimizing** the required number of registers (for ASIC design), the other for handling **fixed** register files (for programming ASIPs). We will call them the unconstrained and the constrained "Register Binding and Operation Scheduling Problem", respectively. Both approaches serialize value lifetimes during or prior to scheduling by introducing sequence edges. The terms serializing and sequencing will be used as synonyms.

2.5.1 Minimizing the register count

The design of an ASIC typically concerns satisfying performance constraints while minimizing some cost function. The main criteria involved in the cost function are area, power consumption, and time to market. The register count affects all these criteria:

- area: although a register occupies silicon area, the physical register is *not* the dominant contribution of the register to the silicon area. The control required to address this register is usually the dominant factor. For example, when a micro-coded controller is used, going from 8 to 10 registers in some register file requires an additional instruction bit for the extended address range. If the function is implemented with, say, 5K instructions, the additional 2 registers cost 5K bit (embedded) instruction rom (far more expensive than two physical registers).

- power consumption: power consumption within a register file grows with the size of the register files. The main contribution however is (again) in accessing and communicating the additional instruction bits required for addressing the larger register file.

- time to market: when this is an important criterion programmable processors are usually preferred. If this solution is too expensive however, we rely on the *synthesizability* of an ASIC. Synthesizability refers to the suitability of designing the ASIC with a small effort. It is clear that synthesizability is improved by the use of *standard* components and components that do not require a lot of (manual) effort to 'push' to the required performance (timing). Larger register files require a larger depth of the multiplexer tree in the address decoding part, which is in the critical path of accessing

the register file. Smaller register files therefore require less (manual) manipulation in order to satisfy the timing requirements.

The problem of minimizing the number of registers is defined as follows.

Definition 2.6 (Unconstrained Register Binding and Operation Scheduling Problem): Given a data flow graph (DFG), the function $rsc(v_i, v_j)$, a binding of values to register files, an initiation interval II, and a constraint on the latency l. Find an assignment of values to registers and a schedule s that satisfies the precedence constraints $E_d \cup E_s$, the resource constraints, and the timing constraints II and l, such that the total number of registers is minimized. ❑

Because it is difficult to determine a register binding and a schedule simultaneously, we decompose the problem into separate phases as depicted in Figure 2.8. First an initial II and an initial register binding are constructed. The determination of the initial II is explained in Section 2.6. The initial register binding is such that all values assigned to a certain register file are assigned to the same register. This register binding requires the least number of registers but will usually be overconstrained (infeasible) in the sense that the register binding is inconsistent with the timing constraints. The central part, the constraint analyser (discussed in Chapters 3 and 4), generates additional precedence constraints that are implied by the combination of all constraints, including the given register binding. These additional precedences prune the schedule search space. They will guide the scheduler and often prevent it from making a schedule decision leading to infeasibility. When the constraint set leaves some room for different lifetime serializations, the *lifetime sequencer* (on the right of the constraint analyser) chooses the alternative that implies the smallest loss of schedule freedom. The constraint analyser (together with the lifetime sequencer) completely replaces the register-binding constraints by precedence constraints. These precedences may cause the constraint set to be infeasible. An *infeasibility analysis* (discussed in Section 4.2) uses the administrative bookkeeping done by the constraint analyser to identify the bottleneck in the constraint set and the register binding. The 'change register binding' block in Figure 2.8 subsequently eliminates this bottleneck by placing two values in different registers that were previously assigned to the same register. This scheme is iterated until the constraint set and the register binding are feasible. Finally, the precedences generated by the constraint analyser, are fed to a scheduler. The way the search space is traversed is illustrated in Figure 2.9: Starting from a mostly infeasible register binding, bottlenecks are solved until the solution is just feasible, on the border between the R-infeasible and the R-feasible region (see Section 1.4 for definitions of the different types of (in)feasibility).

An advantage of this approach is that in order to complete the schedule, a rather straightforward scheduler can be used that is unaware of register binding issues. Although the existence of a schedule is not strictly guaranteed after the constraint analyser, a schedule has always been found in practice when the constraints are very tight, that is, when loop pipelining is applied. When the constraints are not very tight, some

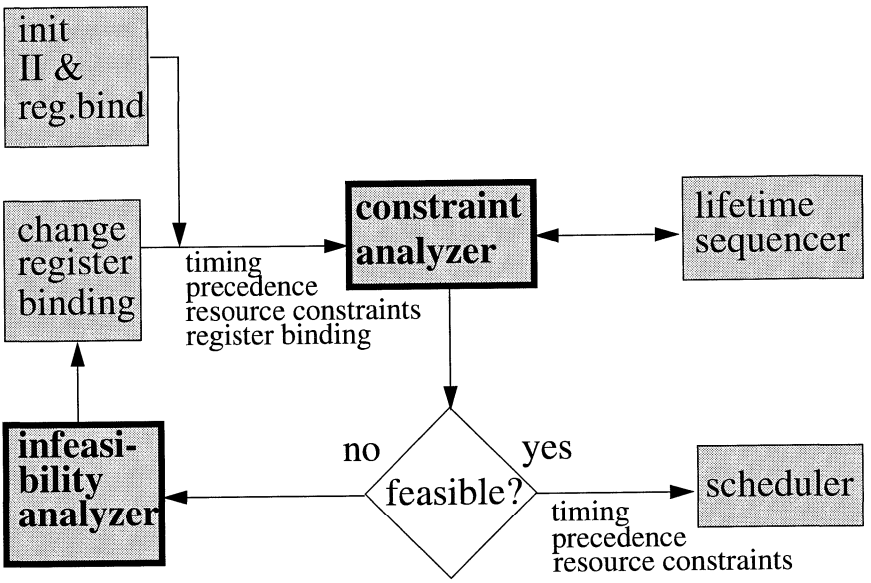


Figure 2.8 Global approach for minimizing the register count

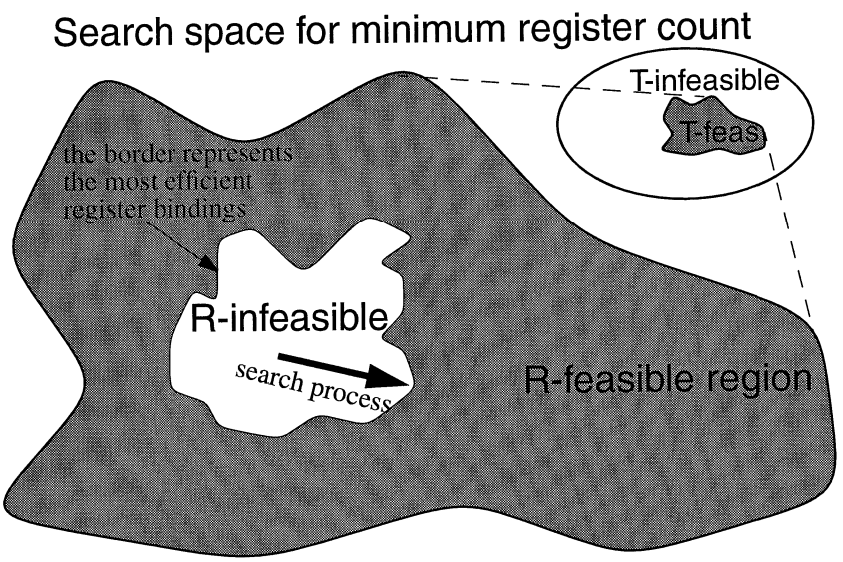


Figure 2.9 In Figure 2.8 the (register binding) is incremented from the centre of the overconstrained (R-infeasible) region to the border with the R-feasible region.

form of backtracking in the scheduler may be desirable. As the scheduler and its heuristics are not critical in this approach, we will not focus on them in this thesis.

Note that a main characteristic of our approach is that we perform register binding *prior* to scheduling. This scheme provides more opportunity for finding an efficient register

binding, because we are not constrained by a given schedule. From the perspective of scheduling this allows to sacrifice schedule freedom with the explicit goal of obtaining an efficient register binding.

After the basic techniques have been discussed in Chapters 3 and Section 4.1, Section 4.2 discusses the infeasibility analyser.

2.5.2 Handling fixed register file sizes

When compiling code for an ASIP (or other programmable processors) using as *few* registers as possible is not the ultimate goal: we would rather use *all* available registers and find a schedule that takes one clock cycle less to execute. Although minimizing the register count is generally considered a wise approach, it may still yield an infeasible register binding: it is conceivable that in some register files very few registers are required at the cost of overloading another (small) register file. Therefore the problem of handling fixed register file sizes is considered a separate problem that requires a tailor-made approach. It is formulated as follows.

Definition 2.7 (Constrained Register Binding and Operation Scheduling Problem): Given a data flow graph (DFG), the function $rsc(v_i, v_j)$, a binding of values to register files, for each register file rf a fixed capacity c(rf), an initiation interval II, and a constraint on the latency l. Find an assignment of values to registers and a schedule s that satisfies the precedence constraints $E_d \cup E_s$, the resource constraints, the register file size constraints and the timing constraints II and l. ❑

The problem is decomposed into separate phases, as illustrated in Figure 2.10. The constraint analyser and the lifetime sequencer work exactly as explained in the previous subsection. The major difference to the approach in the previous section is that the infeasibility analyser is replaced by the incremental register binder. Furthermore the register binding is initialized such that the binding is R-feasible (see Figure 2.11), but will possibly not respect the fixed register file sizes. Contrary to our approach for minimizing the register count, we will remain in the R-feasible region from the start. Instead of looking for an **efficient** solution in the R-feasible region, we are looking for an **arbitrary** solution in the S-feasible region. We do this by incrementally serializing values until all register file sizes are respected. First, the constraint analyser restricts the search scope to the R-feasible region. Similar to the infeasibility analyser in Figure 2.8, the incremental register binder tries to identify a bottleneck, but now the bottleneck is represented by a worst-case overlap of a number of value lifetimes that exceeds the capacity of the corresponding register file. In order to reduce the maximum number of overlapping values, the incremental register binder identifies one or more pair(s) of values that should be serialized. The constraint analyser subsequently calculates the effect of this serialization on the mobility of all operations. This is necessary to prevent the incremental register binder (in subsequent iterations) from making serializations that are not possible. The process is repeated until the register requirements respect the cor-

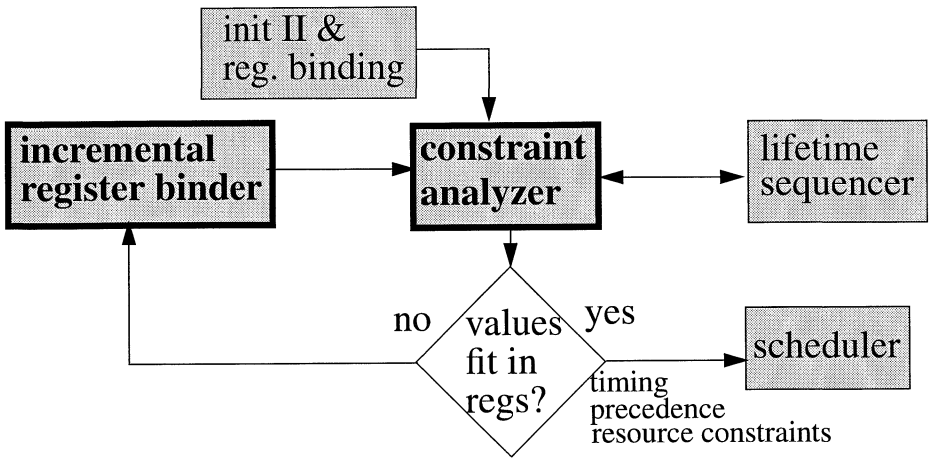


Figure 2.10 Global approach for mapping to fixed register files

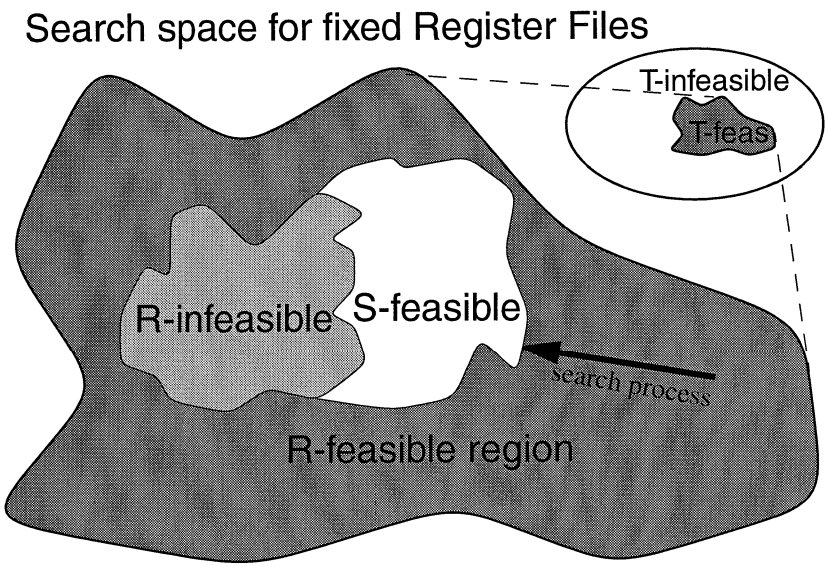


Figure 2.11 In Figure 2.10, the register binding is refined from the R-feasible region to the border with the S-feasible region.

responding capacity. Serializations may be undone in a branch & bound manner in case the R-infeasible region is reached.

Figure 2.11 depicts the search space and the way it is traversed in the approach in Figure 2.10. A major difference with our approach for the minimization problem (Section 2.5.1) is that we remain in the R-feasible region. The incremental register binder is discussed in Section 4.4 after our treatement of the basic techniques in Chapters 3 and Section 4.1.

2.6 Initialization of the initiation interval

The initiation interval is initialized with a lower bound, and incremented if the bound cannot be met. A lower bound on the initiation interval results both from the resource constraints and from the precedence edges.

First consider the resource constraints $rsc(v_i, v_j)$. We associate a so called *conflict graph* CG with $rsc(v_i, v_j)$ in the following way. A node in CG corresponds to an operation. There is an edge in CG between nodes v_i and v_j if and only if $rsc(v_i, v_j) = 1$. Let γ denote the *chromatic index* of CG, which is the minimum number of colours (time potentials) required to colour CG. A valid schedule with II time potentials exists only if there exists a valid colouring with II colours. Therefore γ is a lower bound to II:

$$\mathrm{II} \geq \gamma \tag{2.7}$$

Another lower bound is determined by the precedences [Reit68]. In Section 2.2 it is derived that for two operations P and C, such that $C_i \rightarrow P_{i+k}$, it is necessary that $s(P_i) \geq s(C_i) - k \cdot \mathrm{II}$. Rewriting this inequality (and rounding) yields

$$\mathrm{II} \geq \left\lceil \frac{s(C) - s(P)}{k} \right\rceil, \tag{2.8}$$

where k equals the number of iterations this dependency crosses ($C_i \rightarrow P_{i+k}$). This is called the iteration distance between C and P, and is denoted by id(C,P). The difference between the start times s(C) and s(P) is lower bounded by the delay of the longest precedence path from P to C in the DFG. This delay is called the *distance* d(P,C). (formally defined in Section 3.3) Because the inequality must hold for each pair of operations, we conclude that

$$\mathrm{II} \geq \max_{(v_i, v_j)} \left\lceil \frac{\mathrm{d}(v_j, v_i)}{\mathrm{id}(v_i, v_j)} \right\rceil \tag{2.9}$$

Combining inequalities (2.6) and (2.9) yields

$$\mathrm{II} \geq \max\left(\gamma, \max_{(v_i, v_j)} \left\lceil \frac{\mathrm{d}(v_j, v_i)}{\mathrm{id}(v_i, v_j)} \right\rceil\right) \tag{2.10}$$

Profiling suggests that the minimum initiation interval is in most cases equal to the lower bound in (2.10), and rarely more than one clock cycle away from it.

Chapter 3 Scheduling with Resource Constraints

Scheduling operations that share a limited number of resources is a task that has received attention from many different, often industrially related research areas. Many of the theoretical results stem from the area of operations research. A rather general and certainly very popular 'model' in this area is called Job Shop Scheduling [Coff76]. In electronic design automation, scheduling is considered to be a dominant step in the high-level synthesis phase [McFa90]. The same is true in modern compiler research, where parallel processors occur frequently as a compiler target. In most practical applications the scheduling problem is interwoven with several other tasks, such as (intermediate) storage assignment and the assignment of operations to resources. In compiler and High-Level Synthesis (HLS) research, these three tasks roughly comprise the code generation phase. In this thesis no attention is paid to the problem of resource assignment. We have argued in Section 1.3.1 that for our target architectures (VLIW), the emphasis in code generation is on scheduling and register binding (storage assignment). The interaction between scheduling and register binding is considered in the following chapters.

This chapter is structured as follows. In Section 3.1 an introduction to the scheduling problem is given, justifying the constraint analysis approach taken in this thesis. The perspective of schedule freedom, essential for understanding the concept of constraint analysis, is introduced in Section 3.2. The distance matrix, the central data structure for storing and combining constraint analysis results, is treated in Section 3.3. Related work is discussed in Section 3.4. The way resource constraints are analysed is discussed in Section 3.5. In Section 3.6 the analysis is integrated with scheduling, which comprises our approach for scheduling with resource constraints [Mesm97a]. In Section 3.8 the complexity of this approach is discussed, and Section 3.9 shows some experimental results.

3.1 Introduction

The general High-Level Synthesis Scheduling Problem (HLSSP), introduced in Section 2.1, is a generalization of SS1, sequencing with release times and deadlines [Garey79, p. 236]. Since SS1 is proven NP-complete in the strong sense, HLSSP is NP-complete in the strong sense as well. In order to keep run times within reasonable limits, most research focuses on heuristics rather than 'exact' methods such as Integer Programming (IP) or Branch & Bound (B&B). Heuristics typically run fast without guaranteeing opti-

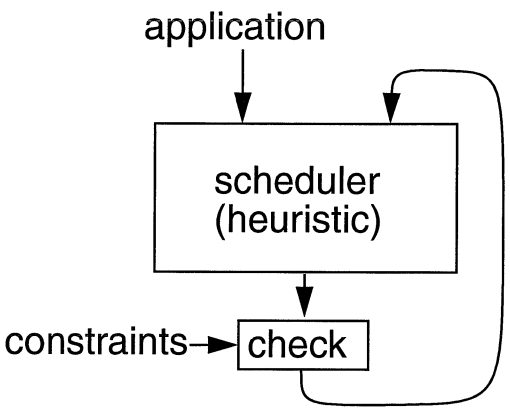


Figure 3.1 Traditional approach for satisfying constraints

mality, whereas exact methods guarantee optimality but may require excessive run times. However, the emphasis of typical scheduling problems is shifting: Feasibility of the schedule (satisfying the constraints) often plays a more important role than 'optimality'. The emphasis on satisfying constraints originates both from the application area (strict timing constraints) and efficient architectures (resource constraints), such as ASIPs. It is no surprise that the early research that focuses on these constraints, still rely on the same old heuristics. This is illustrated in Figure 3.1. A heuristic generates a schedule. It is checked if this schedule satisfies the constraints. If this is not the case, either

- The heuristic is run again, but using different 'priorities' (Section 2.3)
- A bottleneck is searched for, and a repair action is taken to make the schedule valid.
- The designer or programmer is asked for 'hints' on how to solve certain conflicts.

This process may iterate many times, which becomes clear from the perspective of the schedule scope, which is the search space depicted in Figure 3.2.

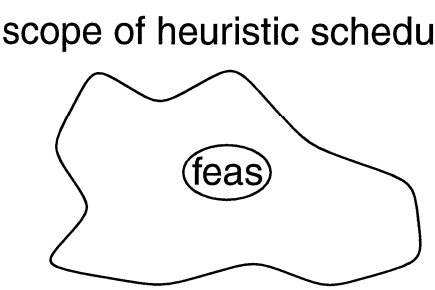


Figure 3.2 The scope of a heuristic scheduler

In this figure, 'feas' indicates the region of solutions that satisfy all constraints. The area surrounding it represents the search scope of a heuristic scheduler. One iteration through the scheduler in Figure 3.1 corresponds with one 'randomly' chosen solution in the search space of Figure 3.2. The probability that this solution is in the feasible region is proportional to the fraction of the search space that is feasible. So if the constraints are very severe the probability of finding a feasible solution in an iteration is extremely small. As a result many iterations may be required, and the scheduler may not even find a feasible solution. Furthermore, there is no way of knowing if there exists a feasible solution at all.

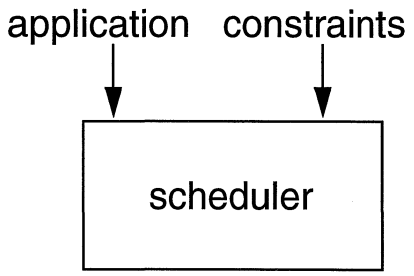


Figure 3.3 Constraint oriented approach for satisfying constraints

For problems with tight constraints it is clearly desirable to have an approach that takes these constraints much earlier and much more directly into account in the course of the scheduling process, such as depicted in Figure 3.3. In this thesis such an approach is proposed. Constraints are analysed and exploited to prune the search space, ideally to the one depicted in Figure 3.4. Either a (traditional) heuristic or a Branch & Bound

Figure 3.4 Ideal search space

method can then be used to traverse this space by making schedule decisions. Care should be taken that after each decision the search space is pruned to eliminate solutions inconsistent with this decision. This pruning of the search space has to be performed in a way that can be interpreted by the scheduler (heuristic or B&B). In Section 2.4 we saw that most of the practical constraints (except for the resource constraints) can be modelled in terms of precedences. Precedences are therefore a powerful way to express additional constraints that emerge from the combination of precedence and resource constraints, and can therefore be used to make pruning information comprehensible for the scheduler. Providing simple but powerful pruning rules based on this observation, is the main contribution of this chapter.

The rest of this chapter is organized as follows. Section 3.2 provides a perspective of the search space (schedule freedom) that makes sense from the constraint analysis point of view. In Section 3.3 a representation of this search space (distance matrix) is intro-

duced which is sufficient for expressing the information most relevant for scheduling: ordering information. Section 3.4 discusses some related work. In Section 3.5 the actual pruning rules are treated. In Section 3.6 the general scheduling approach is introduced, integrating constraint analysis and scheduling. Section 3.8 discusses the complexity of this approach and Section 3.9 shows some experimental results.

3.2 Schedule freedom

In Section 2.1 we have introduced the High-Level Synthesis Scheduling Problem. In order to solve this problem (and the extended scheduling problem from Section 2.5) it is convenient to describe the set of possible solutions, the *solution space*. In this subsection we will describe the solution space as a range of possible start times for each operation. Because this set of feasible start times is at least as difficult to find as it is to find a schedule, we will approximate it by the so called ASAP-ALAP interval, the construction of which is solely based on the precedence constraints $E_d \cup E_s$. By generating additional precedence constraints that are implied by the combination of all constraints, the ASAP-ALAP interval provides an increasingly more accurate estimate of the set of feasible start times.

We start with a description of the solution space:

Definition 3.1 (set of feasible schedules) The set of feasible schedules S is the set of schedules such that each schedule $s \in S$ satisfies the precedence constraints, the resource constraints, and the timing constraints. ❑

An operation thus has a range of feasible start-times, each corresponding to a different schedule.

Definition 3.2 (set of feasible start times $T(v_i)$) $T(v_i) = \{c \in N | \exists s \in S: s(v_i) = c\}$, where N denotes the set of natural numbers. ❑

Definition 3.3 (actual schedule freedom) The actual schedule freedom is the average size of the set of feasible start times minus one:

$$\frac{1}{|V|} \cdot \sum_{v_i \in V} [|T(v_i)| - 1]$$ ❑

The minus one enforces that the actual schedule freedom equals zero when the schedule is completely fixed. The actual schedule freedom quantifies the amount of choice for making schedule decisions. For traditional schedule heuristics a large actual schedule freedom is advantageous because it gives the scheduler more room for optimization. The actual schedule freedom is however defined by the constraints, so for tightly constrained scheduling problem instances a feasible solution cannot be expected from a heuristic scheduler.

The set of feasible start times is formally as difficult to find as a feasible schedule. Therefore, conservative estimates of the schedule interval are more practical. The most widely used estimate of the set of feasible start times is the so called ASAP-ALAP interval. It is based solely on the precedence constraints $E_d \cup E_s$ and the latency constraint.

For the definition of the ASAP-ALAP interval we need the notion of immediate predecessors and successors:

Definition 3.4 (immediate predecessors, successors)

$$\forall(v \in V): \text{pred}(v) = \{u \in V | (u, v) \in E\}$$

$$\forall(v \in V): \text{succ}(v) = \{u \in V | (v, u) \in E\}$$ ❑

The ASAP (as soon as possible) value is defined as:

Definition 3.5 (ASAP value)

$$\text{ASAP}(v) = \begin{cases} 0 & \text{if pred}(v) = \varnothing \\ \max\limits_{u \in \text{pred}(v)} (\text{ASAP}(u) + w(u, v)) & \text{if pred}(v) \neq \varnothing \end{cases}$$ ❑

The latest possible start time is called the ALAP (as late as possible) value. It exists only if the latency (completion time) of the schedule is bounded. Let l denote the latency constraint. Then ALAP(sink)=l, and for all other operations:

Definition 3.6 (ALAP value)

$$\text{ALAP}(v) = \begin{cases} l - 1 & \text{if succ}(v) = \varnothing \\ \min\limits_{u \in \text{succ}(v)} (\text{ALAP}(u) - w(u, v)) & \text{if succ}(v) \neq \varnothing \end{cases}$$ ❑

The start time of each operation must lie in between the ASAP and ALAP value inclusively:

$$\forall(v_i \in V): \text{ASAP}(v_i) \leq s(v_i) \leq \text{ALAP}(v_i) \qquad (3.1)$$

Therefore the ASAP-ALAP interval is a conservative estimate of (contains) the set of feasible start times.

In this chapter we will extract sequencing constraints that are necessarily implied by the combination of all constraints. These sequencing constraints are then explicitly added

to the DFG as precedence constraints, yielding a narrowing of the ASAP-ALAP intervals. This way an increasingly more accurate estimate of the set of feasible start times is obtained. For most scheduling methods, either the ASAP-ALAP intervals or the precedence constraints are an extremely important guideline. Schedule choices are made with respect to the available resources. When the ASAP-ALAP interval does not reflect the actual schedule freedom very accurately, there will often come a point in the schedule process where there are no available resources for an operation, and the operation cannot be scheduled. In this way, the precedence constraints and the resulting ASAP-ALAP interval implicitly represent the 'search scope' of the scheduler. Therefore we also define the 'apparent freedom', also called mobility or slack.

Definition 3.7 (apparent schedule freedom, mobility, slack) The apparent schedule freedom is the average size of the set of ASAP-ALAP intervals:

$$\frac{1}{|V|} \cdot \sum_{v_i \in V} (\mathrm{ALAP}(v_i) - \mathrm{ASAP}(v_i)) \qquad \square$$

Because the precedences and the ASAP-ALAP interval form the basis for making schedule decisions, the performance of a scheduler depends largely on the accuracy of this interval as an estimate of the set of feasible start times $T(v_i)$. When the ASAP-ALAP interval is an accurate estimate, the mobility is an accurate estimate of the actual schedule freedom and vice versa. Therefore we will use the mobility before and after the constraint analysis as a performance measure of the analysis.

Example. In Section 2.3 we showed an example (Figure 3.5) that illustrates the difficulty of greedy schedulers (particularly list-schedulers) to handle resource conflicts in the context of pipelined loop schedules. We will now make the same observation from the perspective of schedule freedom. In Figure 3.5, the operations are annotated with their ASAP-ALAP intervals. The mobility according to Definition 3.7 equals (1+1+1+1+1)/5= 1 clock cycle per operation, roughly corresponding to the search space in Figure 3.2. In Figure 3.7 the subsequent steps of our scheduling approach are depicted in which sequencing constraints are added. These steps are justified in Section 3.5, but are not relevant for this discussion. In Figure 3.7 b), a sequence edge $A \rightarrow D$ of weight 4 is added. The new ASAP-ALAP intervals are: A=[0;0], B=[1;2], C=[2;3], D=[4;4], and E=[5;5]. The mobility is reduced to (0+1+1+0+0)/5= 0.4 clock cycle per operation. In Figure 3.7 d), a sequence edge $A \rightarrow B$ of weight 2 is added. The new ASAP-ALAP intervals are: A=[0;0], B=[2;2], C=[3;3], D=[4;4], and E=[5;5]. The mobility is reduced to (0+0+0+0+0)/5= 0 clock cycle per operation. That is, every operation is fixed. This corresponds to the search space in Figure 3.4, where the feasible region consists of only one solution. In this particular example the pruning techniques are able to reduce the mobility to the exact schedule freedom.

3.3 Representing the search space: the distance matrix

In the previous section the search space was represented using ASAP-ALAP intervals and the amount of schedule freedom was expressed as the mobility, which is the aver-

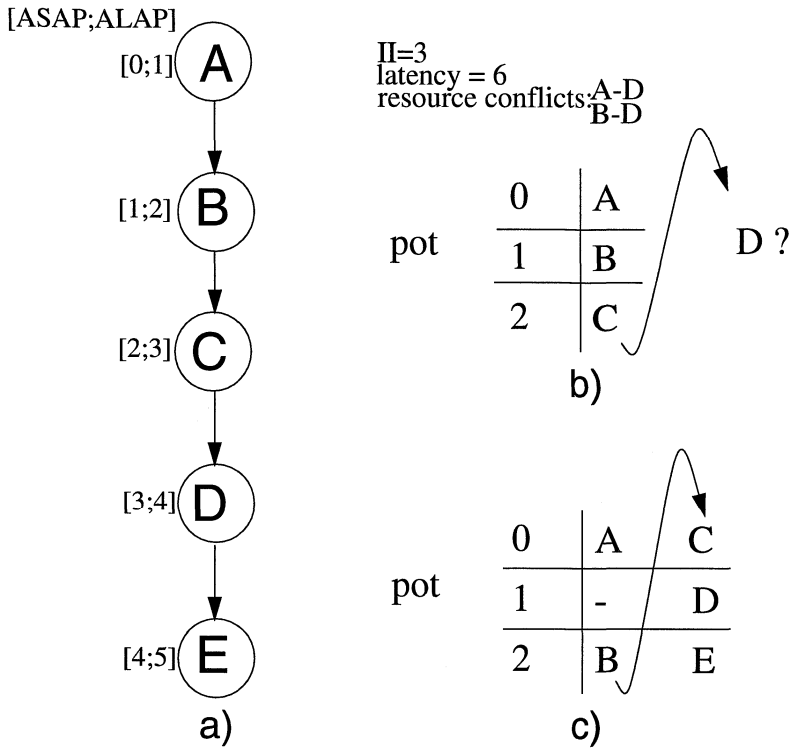


Figure 3.5 Example with loop folding. a) precedence graph b) list-schedule c) only feasible schedule in 6 clock cycles

age cardinality of the ASAP-ALAP intervals. This representation offers the following advantages:

- It is a rather simple representation. For each operation two figures define the ASAP-ALAP interval. The memory requirements for administrating the intervals is therefore in the order O(V).

- The transparency of the terminology appeals to the human mind and is therefore suitable for discussing and explaining the concepts of schedule freedom and search space pruning.

- This representation is easy to derive. Essentially a *depth first search* [Corm90, p. 477] has to be performed with a complexity of only O(V+E).

- It allows for a simple infeasibility check: if, for an interval [lb;ub], $ub \leq lb$, no solution exists.

However, the interval representation is not able to accurately represent the most basic and important type of constraints for scheduling: ordering (precedence) information.

This is illustrated in Figure 3.6. Figure 3.6a) specifies that operation A should start executing at least one clock cycle before operation B. The ASAP-ALAP intervals, as derived from this precedence constraint and the latency constraint, are depicted in Figure 3.6b). The interval representation suggests that operation B may start executing strictly before operation A, which contradicts the precedence specified in Figure 3.6a).

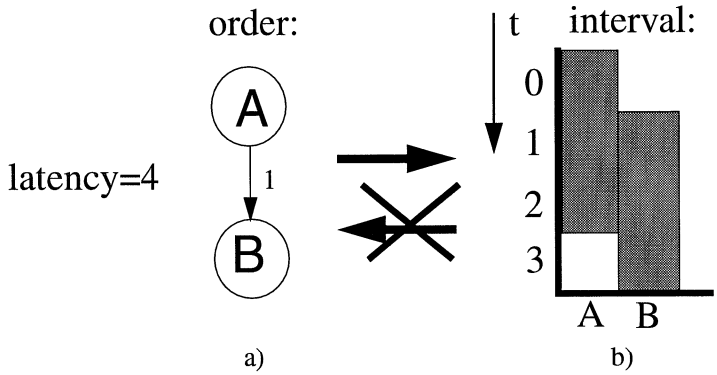


Figure 3.6 The interval representation does not accurately represent precedences

In this section the *distance matrix* is introduced, and it will be shown that this representation is *strictly* more accurate than the interval representation. In order to define distance, first the concept of a path is introduced:

Definition 3.8 Definition 10 (path): A path of length d from operation v_i to operation v_j is a single chain of precedences $v_i \rightarrow v_k \rightarrow \cdots \rightarrow v_l \rightarrow v_j$ that implies $s(v_j) \geq s(v_i) + d$. ❑

Definition 11 (distance): The distance $d(v_i, v_j)$ from operation v_i to v_j is the length of the longest path from v_i to v_j. ❑

A path in the graph thus represents a minimum timing delay. For example, in Figure 3.5 the path $A \rightarrow B \rightarrow C$ indicates a minimum timing delay of 2 clock cycles between the start times of A and C. Finding the longest paths between each pair of operations is equivalent to the all-pairs shortest path problem. The corresponding algorithmic complexity is less than $O(V^3)$ [Corm90, Ch.26]. The distances corresponding to the longest paths can be administrated in a matrix called the *distance matrix*, where entry d_{ij} represents the distance from operation v_i to operation v_j. In cyclic graphs, a path can exist from an operation to itself. If the constraint set is feasible, the corresponding distance is always less or equal to zero. If not, the constraint set is infeasible.

The distance matrix representation is strictly more accurate than the interval representation. This follows from Theorem 3.1 and the observation that the information expressed in the distance matrix cannot always be accurately expressed in terms of intervals.

Theorem 3.1 Any interval can be represented in terms of precedences.

Proof. An interval [lb;ub] for operation A means that

$$\begin{aligned} s(A) &\geq lb \\ s(A) &\leq ub \end{aligned} \qquad (3.2)$$

To represent this interval in the distance matrix the following precedences are added: A precedence source $\rightarrow$ A with weight lb and a precedence A $\rightarrow$ source with weight -ub. Now by definition s(source)=0. According to inequality , the meaning of the added precedences is:

$$\begin{aligned} s(A) &\geq s(\text{source}) + lb = lb \\ s(\text{source}) &\geq s(A) - ub \Rightarrow s(A) \leq ub \end{aligned} \qquad (3.3)$$

Now inequality (3.3) reduces to inequality (3.2), which proves the theorem. ❑

The proof shows how results from analyses on intervals, like [Timm95], can be represented as precedences, and therefore be combined (using the longest path algorithm) with the results of other types of analyses expressed as precedences. The example in Figure 3.6 shows that the other way around is not true: the information expressed in the distance matrix cannot always be accurately expressed in terms of intervals. We conclude that the distance matrix representation is strictly more accurate than the interval representation.The disadvantage of the distance matrix is that the data structure required to store it has a complexity of order $O(V^2)$, whereas the corresponding complexity for the interval representation is of order O(V). In this thesis we will rely heavily on the process of serializing operations. For determining which serializations are useful it is important to have an accurate description of the relative order of operations. The analyses treated in this thesis therefore work on the distance matrix. The interval representation and the mobility will still be used to explain and discuss some of the concepts of constraint analysis.

3.4 Related work in constraint analysis

[Nuijt94] reports results on the TRCSP, the Time and Resource Constrained Satisfaction Problem. General constraint satisfaction techniques are employed that allow (some) exploitation of the problem specific knowledge. For an instance of the constraint satisfaction problem (CSP) a set of variables is given, a domain of values for each variable, and a set of algebraic inequalities on the assignment of values to variables. In TRCSP, a variable exists for each operation. The domain consists of combinations of a set of resources and a start time. In [Nuijt94] widely used tree search algorithms are employed. The emphasis is on *consistency checking*, removing inconsistent values of unassigned variables when a variable is assigned. In this thesis consist-

ency checking is called search space pruning, which comprises the major contribution. For the other tree search components, *variable and value selection* (making scheduling or assignment decisions) and *dead end handling* (backtracking), relatively simple algorithms suffice according to [Nuijt94].

In [Kuch97] schedule constraints are expressed in Constraint Logic Programming (CLP), a generally applicable programming language for describing (linear) constraints. A branch & bound algorithm with depth-first-search is employed. Timing and resource constraints are satisfied, while minimizing system resources (mostly communication busses). The resource constraints are extended to handle loop pipelined schedules. For relatively small examples without pipelining, run-times are good. However, run-times tend to grow exponentially with the number of constraints, which increase with the number of 'processes', and is much larger for pipelined schedules.

In [Timm95] a bipartite matching formulation is used to analyse the matching of execution intervals of operations to execution intervals of resources. Reductions in the execution intervals are obtained by showing that some matchings can never be part of a complete matching. The bipartite-matching approach is based on the concept of an execution interval. It keeps the resource constraints fixed, while 'relaxing' the precedence constraints. The search space is pruned (intervals are reduced) by incorporating increasingly more precedence constraints. The approach taken in this thesis works the other way around: it is based on sequence relations between operations, and therefore keeps the precedence relations fixed. Additional sequence relations are identified by incorporating increasingly more resource constraints. Theorem 3.1 shows that these two complementary analyses can be combined using the distance matrix. These analyses have been integrated in the FACTS environment [Mesm99c]. Furthermore, the analysis in [Timm95] provides a way to identify bottlenecks in the resource usage. In constraint satisfaction terms this implies that the work is suitable for variable and value selection, and therefore it is currently the main scheduler in FACTS.

In [Eijk99] symmetry in the algorithm specification (in the Data Flow Graph) is exploited to prune the search space. Contrary to [Timm95] and the work in this thesis, feasible solutions in the search space may be eliminated. This is justified by the guarantee that solutions remain which are essentially equivalent to those eliminated. Two techniques are proposed that automatically detect and utilize symmetry. These techniques are based on finding automorphisms (in terms of group theory). Both techniques introduce sequence edges between operations such that the feasibility of the problem is preserved while the symmetry is broken. The analysis results can therefore be combined in a straightforward manner using the distance matrix with the analyses from [Timm95] and the work in this thesis. This work has also been integrated in the FACTS environment.

3.5 Sequencing as a result of resource conflicts

In this section two lemmas are introduced that assert the necessity of an additional sequence constraint, resulting from the combination of existing sequence and resource constraints. The first lemma will help us to solve the schedule problem in Figure 3.5.

Lemma 3.2:If $d(v_i, v_j) \equiv 0 \pmod{II}$ and $rsc(v_i, v_j) = 1$, we can add a sequence precedence edge (v_i, v_j) with weight $d(v_i, v_j) + 1$ without excluding any feasible schedules.

Proof: The resource conflict $rsc(v_i, v_j) = 1$ causes the minimum distance $d(v_i, v_j)$ to be infeasible. Therefore the minimum distance is at least one clock cycle larger. ❑

In the schedule problem instance depicted in Figure 3.5, the key decision to obtain a feasible schedule is to put a gap of one clock cycle between A and B. So our goal is to derive that d(A,B) = 2. In Figure 3.7 this derivation is given. Figure 3.7a represents the complete original DFG model. In Figure 3.7a we see a path $A \rightarrow B \rightarrow C \rightarrow D$ of length $3 \equiv 0 \bmod II$ from A to D. According to Lemma 3.2 we can add a sequence edge $A \rightarrow D$ of weight 4 because A and D have a resource conflict. This edge is drawn in Figure 3.7b. There is a path $D \rightarrow E \rightarrow$ sink $\rightarrow$ source $\rightarrow A \rightarrow B$ of length 1+1-6+0+1 =-3 clock cycles. Because of the resource conflict D-B, this length has to be increased by one clock cycle. This gives a sequence edge $D \rightarrow B$ of weight -2, as given in Figure 3.7c. We conclude by finding a path $A \rightarrow D \rightarrow B$ of length 4-2=2 clock cycles. In Figure 3.7d the associated sequence edge (A,B) of weight 2 is explicitly drawn. The precedence relations now completely fix the schedule. The reader can verify that the [ASAP, ALAP] intervals based on the extended DFG of Figure 2.7d all contain just one clock cycle, and the mobility equals zero.

The second lemma we present in this chapter is more complicated, and involves symmetry in the precedence graph. Consider the small piece of precedence graph depicted in Figure 3.8. The distance from A to D is two clock cycles. However, B and C have to be ordered because they have a resource conflict, and both possible orderings will result in d(A,D)=3. Lemma 3.2 will not help us here. In DSP-algorithms this type of symmetry occurs frequently. Lemma 3.3 copes with this issue in the context of loop folding.

Lemma 3.3: For each pair of operations v_i and v_j such that $rsc(v_i, v_j) = 1$, if there is an operation p such that $d(p, v_i) \equiv d(p, v_j) \pmod{II}$, and an operation s such that $d(p, v_i) + d(v_i, s) = d(p, v_j) + d(v_j, s)$, we can add a sequence edge (p, s) with weight $d(p,v_i) + d(v_i,s) + 1$

Proof: The resource conflict $rsc(v_i, v_j) = 1$ causes a minimum distance $d(p, v_i) + d(v_i, s) = d(p, v_j) + d(v_j, s)$ to be infeasible. Therefore the minimum distance is at least one clock cycle larger. ❑

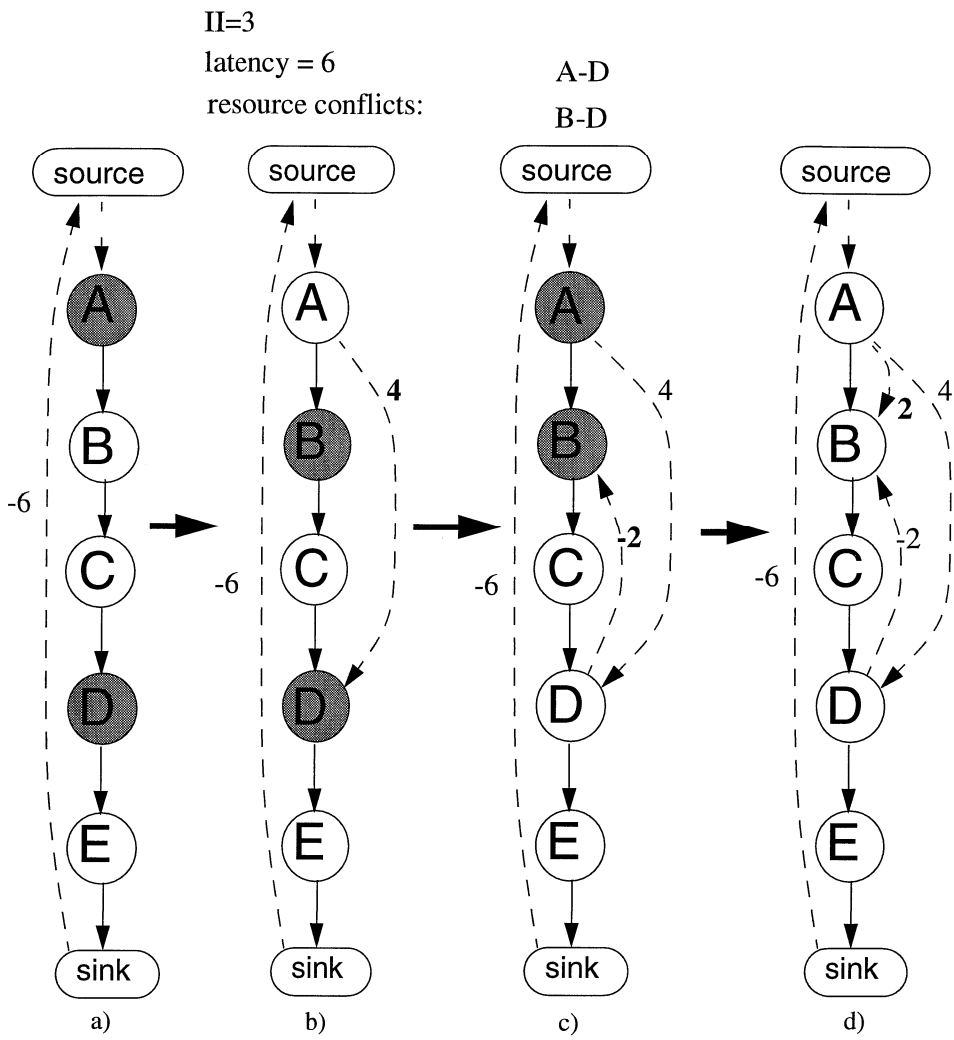


Figure 3.7 Derivation of a schedule for Figure 2.4

In Figure 3.8, operation p is A, and s is D. As a result of lemma Lemma 3.3 a sequence edge A $\rightarrow$ D may be added of weight d(A,B) + d(B,D) + 1 = 3.

In Figure 3.9, the symmetry is of a slightly different kind. As can be seen in the ASAP schedule, the only way the minimum distance of 4 clock cycles from A to H can be realized, is to schedule operations B and G at the same potential. Because B and G have a resource conflict, the distance from A to H is not 4, but 5 clock cycles. This also follows from Lemma 3.3: d(A,B) = 1 and d(A,G) = 3, so d(A,B) $\equiv$ d(A,G) (mod 2). Also d(A,B) + d(B,H) = 4 and d(A,G) + d(G,H) = 4, so we may add a sequence edge (A,H) of weight 5.

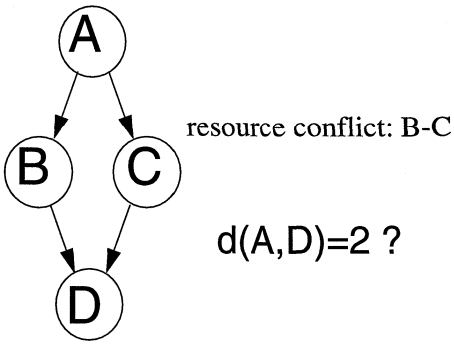


Figure 3.8 Too much apparent mobility due to symmetry

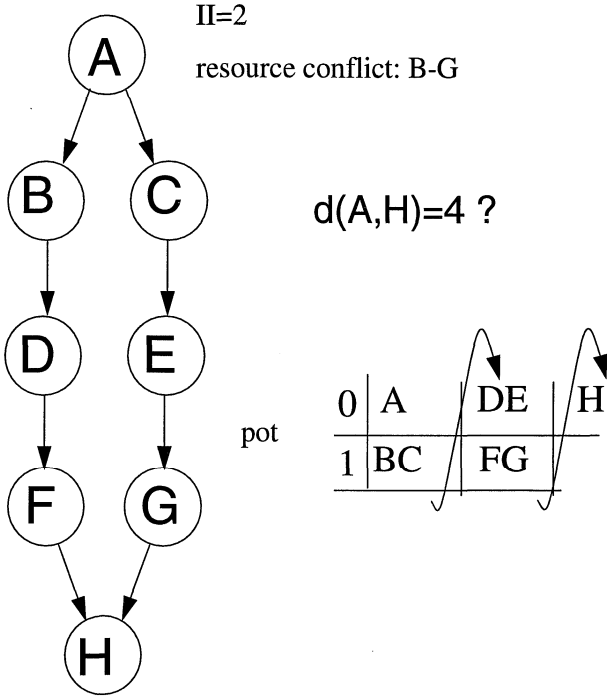


Figure 3.9 Applying rule 2 with loop folding

3.6 Sequencing for an extended resource constraint model

The previous section discussed some rules for serializing operations when two operations have a resource conflict according to the resource conflict model introduced in Section 2.1. An extension of Lemma 3.2 can be derived for a more general model of resource constraints. In this model, also used by [Timm95], an operation is associated with a number of resource usages. These include e.g. addition, read port, communication bus, etc. For each resource usage a number of resource instances may exist that

perform this resource usage. In this thesis it is assumed that the execution delay of a resource usage is independent of the specific resource instance. This model is equivalent to the resource conflict model introduced in Section 2.1 in the following two cases. The first is the situation that only one resource instance exists for a resource usage. The other is the case that resource binding has been performed. In both cases resource usages are associated with a specific resource instance. In Section 3.6.1 the situation is discussed where two resource instances exist. Section 3.6.2 generalizes the idea to n resource instances.

3.6.1 Sequencing for two resource instances

We start with the case that no loop pipelining is applied. Let v_i, v_j, v_k denote three operations that share a resource usage, of which two resource instances are available.

Lemma 3.4 If $d(v_i, v_j) = 0$ and $d(v_j, v_k) = 0$, we can add a sequence precedence edge (v_i, v_k) with weight 1 without excluding any feasible schedules. ❑

Proof: Suppose that the distances $d(v_i, v_j) = 0$ and $d(v_j, v_k) = 0$ are the minimum distances in a feasible schedule. Then v_i, v_j, and v_k all execute in the same clock cycle. But this requires three resource instances, whereas only two are available. Therefore either $d(v_i, v_j) \geq 1$ or $d(v_j, v_k) \geq 1$. Both cases result in $d(v_i, v_k) \geq d(v_i, v_j) + d(v_j, v_k) \geq 1$. ❑

Lemma 3.4 is depicted in Figure 3.10.

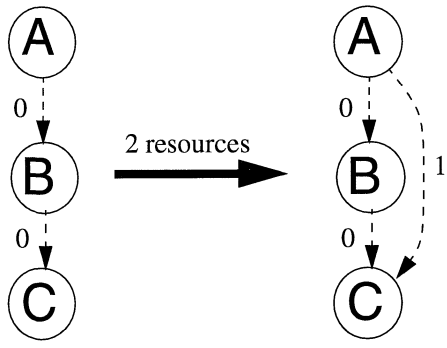


Figure 3.10 Lemma 3.4

We now generalize Lemma 3.4 to the case that loop pipelining is applied. Let v_i, v_j, v_k denote three operations that share a resource usage, of which two resource instances are available.

Lemma 3.5 If $d(v_i, v_j) \equiv 0 \pmod{II}$ and $d(v_j, v_k) \equiv 0 \pmod{II}$, we can add a sequence precedence edge (v_i, v_k) with weight $d(v_i, v_j) + d(v_j, v_k) + 1$ without excluding any feasible schedules. ❑

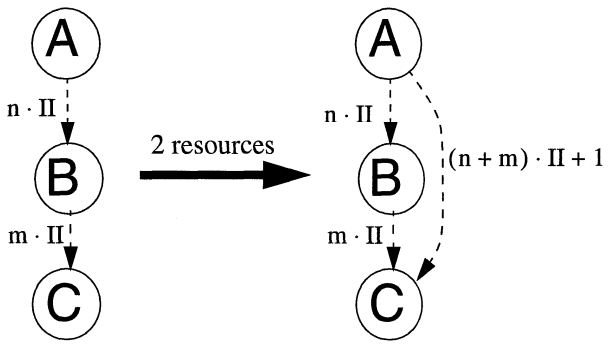


Figure 3.11 Lemma 3.5

Proof: Suppose that $d(v_i, v_j) = n \cdot II$ and $d(v_j, v_k) = m \cdot II$ are the minimum distances in a feasible schedule. Then v_i, v_j, and v_k all execute in the same time potential. But this requires three resource instances, whereas only two are available. Therefore either $d(v_i, v_j) \geq n \cdot II + 1$ or $d(v_j, v_k) \geq m \cdot II + 1$. Both cases result in a lower bound for the distance $d(v_i, v_k) \geq d(v_i, v_j) + d(v_j, v_k) \geq n \cdot II + m \cdot II + 1$ ❑

Lemma 3.5 is depicted in Figure 3.11.

Example. Consider the example depicted in Figure 3.12. Operations F and G model a pipelined multiplication (Section 2.4). We only consider the resource usage of a read port (two instances available) and a write port (two instances available). The use of normal (randomly addressable) registers limits all value lifetimes to two (II) clock cycles (Section 2.4), which justifies all the dashed backward edges with d=-2. This constraint set is infeasible, as the reader may find out when trying to construct a schedule. In order to prove infeasibility, Lemma 3.5 is repeatedly applied to the graph in a cumulative manner, as depicted in Figure 3.13, until . In this figure, each numbered row is one application of Lemma 3.5. In each row, the circled operations denote the role of operations v_i, v_j, and v_k resp. In the first row, for example, there is a path from C to E of length 2 (= 0 mod II), and a path from E to G of length -2 (= 0 mode II). Therefore Lemma 3.5 applies, and a sequence edge from C to G can be added with a delay of 2-2+1=1 clock cycle.

3.6.2 Sequencing for N resource instances

We now generalize Lemma 3.5 to the case where N resources are available. So suppose there exist operations $v_0, v_1, \ldots, v_{N-1} \in V$ that share a resource usage of which N are available. The following is an application of the well-known pigeon-hole principle.

Lemma 3.6 If $d(v_i, v_{i+1}) \equiv 0 \pmod{II}$ for all $0 \leq i \leq N-2$, we can add a sequence precedence edge (v_0, v_{N-1}) with weight $1 + \sum_{0 \leq i \leq N-2} d(v_i, v_{i+1})$ without excluding any feasible schedules. ❑

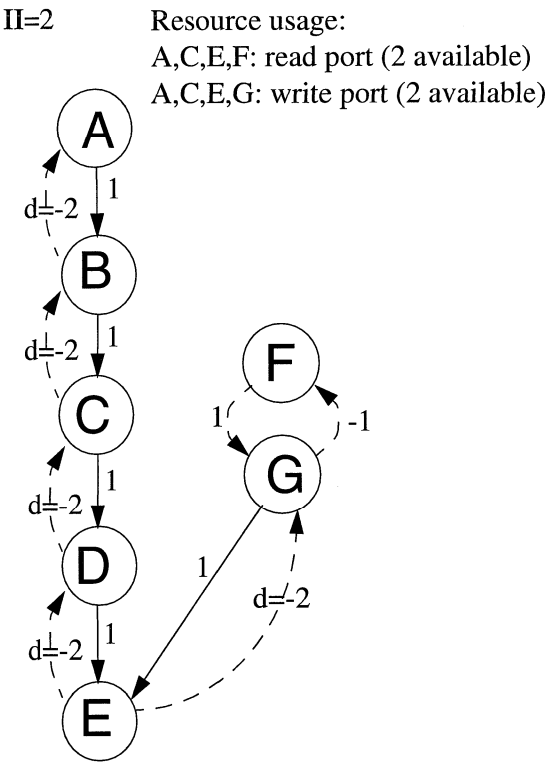


Figure 3.12 The derivation in Figure 3.13 proves infeasibility of the constraint set

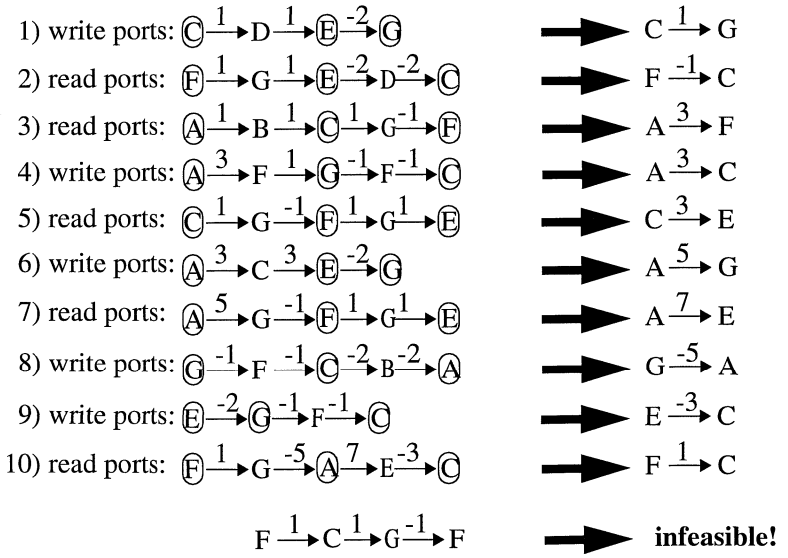


Figure 3.13 Deriving infeasibility of the constraint set in Figure 3.12

Proof: Suppose $d(v_i, v_{i+1}) = k_i \cdot \Pi$ for $0 \leq i \leq N-2$. If these distances are minimal in a feasible schedule, then $v_0, v_1, \ldots, v_{N-1}$ all execute in the same time potential. But this requires N+1 resource instances, whereas only N are available. Therefore in any feasible schedule there is at least one i: $0 \leq i \leq N-2$ such that $d(v_i, v_{i+1}) > k_i \cdot \Pi$. As a result, $d(v_0, v_{N-1}) \geq 1 + \sum_{0 \leq i \leq N-2} d(v_i, v_{i+1})$. ❑

3.7 Schedule approach

In the previous section we have shown some pruning rules for coping with the combination of precedence and resource constraints. The pruning rule resulting from Lemma 3.2 has been implemented and integrated in the FACTS code generation environment [Eijk00], [Mesm01]. In this section the pruning rules are used in an approach to solve the problem of scheduling with precedence and resource constraints. This approach is depicted in Figure 3.14. The idea is as follows: The constraint analyser uses the pruning rules to prune the search space as much as possible. The results of this analysis are expressed in terms of additional sequencing relations which are provided to a scheduler. The scheduler makes a decision, which eliminates solutions previously considered feasible. By expressing the schedule decision in terms of additional sequencing relations (Figure 2.7), the constraint analyser on its turn calculates the effect of this decision on the search space. Additional sequencing constraints are added, which are the essential consequence of both the constraints and the schedule decision(s). This process is iterated until the schedule is fixed. It should be emphasized again that the pruning rules are not guaranteed to eliminate *every* infeasible solution from the search space. As a result, the scheduler may still make decisions that lead to infeasibility. When the constraints are very tight, a Branch & Bound method may therefore be desirable. Fortunately, in this case the constraint analyser will effectively reduce the search space such that only a limited number of schedule decisions are required. Furthermore, support for backtracking is offered by infeasibility detection, see Section 3.3.

The schedule approach is illustrated in Figure 3.15. This is almost the same example as in Figure 3.5, but there is no constraint on the latency, and there is an additional resource conflict C-D. Note that in this case, a list-scheduler would construct a schedule in the same way as in Figure 3.5b), and fail. This failure is not due to a latency constraint, but the result of the resource constraints.

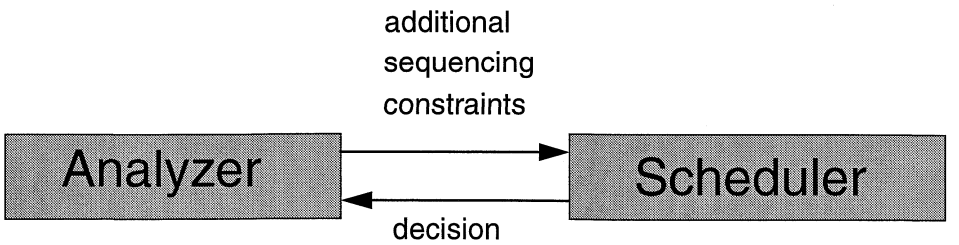


Figure 3.14 Global approach for scheduling

The initial execution intervals (before any analysis or constraint modelling) are A=[0;∞], B=[1;∞], C=[2;∞], D=[3;∞], and E=[4;∞]. The mobility equals ∞. Because it is assumed that randomly addressable register files are used for storing the communicated values, every value lifetime is restricted to the initiation interval II (explained in Section 2.4). In Figure 3.15a) these constraints are added, as is the sequence edge A → D of weight 4=II+1 as a result of the path A → B → C → D of length 3 ≡ 0(mod II) from A to D and the resource conflict A-D (Lemma 3.2).

a) to b). There is a path D → C of length -3 ≡ 0 mod II, and a resource conflict C-D. As a result, a sequence edge D → C of length -3+1=-2 is added.

b) to c). The scheduler schedules operation A at clock cycle 0, and B at clock cycle 1. The execution intervals are A=[0;0], B=[1;1], C=[2;4], D=[4;6], and E=[5;9]. The mobility equals 1.6 clock cycles per operation.

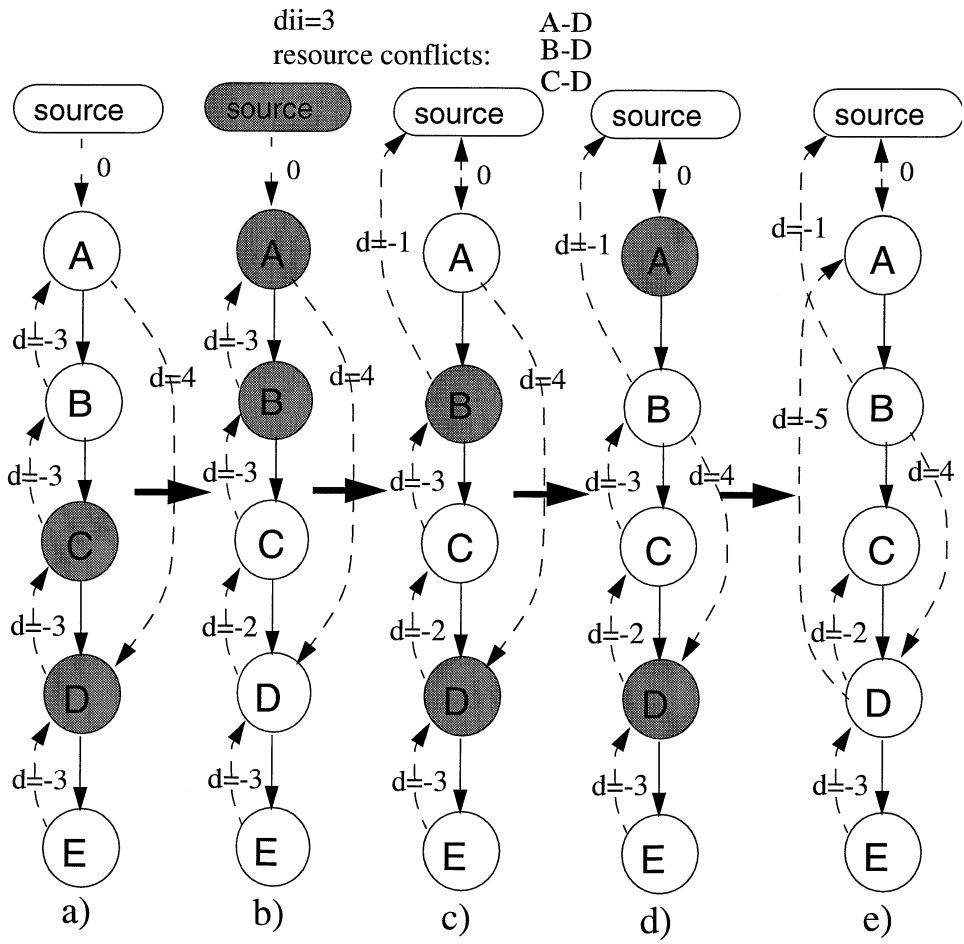


Figure 3.15 Illustrating the schedule approach from Figure 3.14

c) to d). There is a path B → source → A → D of length -1+0+4=3 ≡ 0 mod II, and a resource conflict B-D. As a result, a sequence edge B → D of length 3+1=4 is added.

d) to e). There is a path D → C → B → source → A of length -2+(-3)+(-1)+0=-6 ≡ 0(mod II), and a resource conflict A-D. As a result, a sequence edge D → A of length -6+1=-5 is added. The execution intervals are A=[0;0], B=[1;1], C=[3;4], D=[5;5], and E=[6;8].The mobility equals 0.6 clock cycles per operation.

The remaining search space is now completely feasible and the scheduler (probably) fixes operation C at clock cycle four, and operation E at clock cycle six. The example illustrates that one schedule decision may have a large effect on the mobility (decrease by 63%), and therefore it is useful to iterate between the constraint analyser and the scheduler.

3.8 Complexity

The complexity of the analysis is determined by two factors:

1. Updating the distance matrix

2. The analysis required for determining which sequence edge should be added

We first consider the updates on the distance matrix. In the distance matrix the delay of the longest path between each pair of operations is maintained. So if a new edge is added, the impact on the current longest paths has to be calculated. This complexity is essentially determined by the number of paths that need to be updated as a result of the new sequence edge. Because we are only interested in the longest paths found so far, the number of updates equals V^2 worst case. In most cases, the addition of a sequence edge will affect a few paths. In cases where a lot of paths need to be updated, the reduction in mobility will also be substantial.

An upper bound on the number of path updates (as a result of adding sequence edges) can be derived as follows. A path can have a length between $-l$ and $+l$, where l is the constraint on the latency. Because a path is updated only if its length is increased (by at least one clock cycle), the number of times a path can be updated is at most $2l$. Since the maximum number of paths we keep track of, equals V^2, the number of path updates can be at most $2l \cdot V^2$. A single path update takes constant time.

Now we consider the complexity of applying Lemma 3.2 and Lemma 3.3. Lemma 3.2 is applied in the following way. For each resource conflict v_i - v_j it is checked whether $d(v_i, v_j) \equiv 0 \pmod{II}$ or $d(v_j, v_i) \equiv 0 \pmod{II}$, in which case a sequence edge is added. One such iteration through all resource conflicts has a complexity |rsc|, which denotes the number of resource conflicts. One iteration is usually not sufficient to capture all the reductions attainable with Lemma 3.2. This is because the distance matrix has changed after the sequence edges have been added, thereby providing more oppor-

tunity for applying Lemma 3.2. After a few iterations no additional reductions are obtained. There can be a resource conflict between every pair of operations, so |rsc| is upper bounded by V^2. Therefore the complexity of applying Lemma 3.2 is of order $O(l \cdot V^2 + |rsc|) = O(l \cdot V^2)$.

Lemma 3.3 is applied in the following way. For each resource conflict v_i - v_j it is checked whether there exist p and s such that $d(p, v_i) + d(v_i, s) = d(p, v_j) + d(v_j, s)$. There exist V^2 pairs (p,s), so one iteration through all resource conflicts requires at most $O(|rsc| \cdot V^2)$ computations. Therefore the complexity of applying Lemma 3.3 is of order $O(l \cdot V^2 + |rsc| \cdot V^2) = O(V^4)$. Because this complexity is high and the corresponding search space reduction is small after applying Lemma 3.2, Lemma 3.3 is not applied in the experiments. The lemmas from Section 3.6 on the extended resource constraint model have not been implemented and subsequently, are not applied in the experiments as well.

3.9 Experimental results

Two experiments are reported in this section. The first experiment considers how supplementary our approach is to the approach of [Timm95], discussed in Section 3.4 for both folded and non-folded schedules. The second experiment shows the efficacy and efficiency of our constraint analysis on industrial applications, and it demonstrates the use of integrating constraint analysis and scheduling, as explained in 3.6. The quality of the analysis is measured by the reduction in mobility.

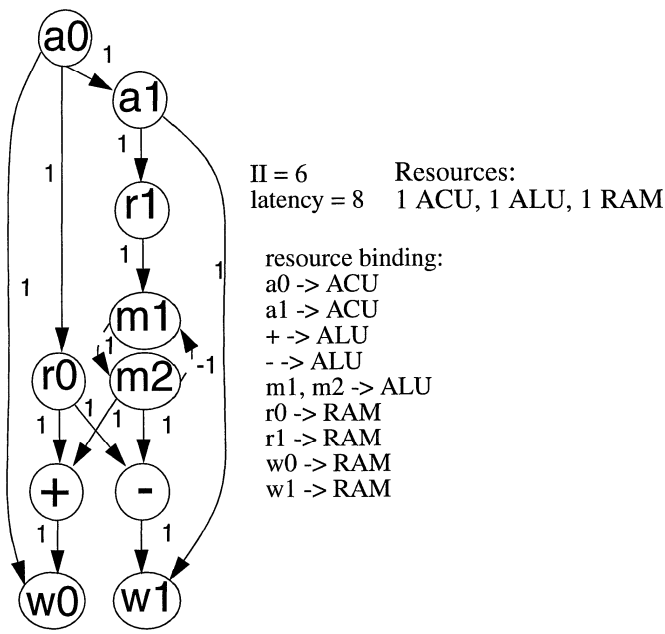


Figure 3.16 Radix-2 butterfly used in first experiment

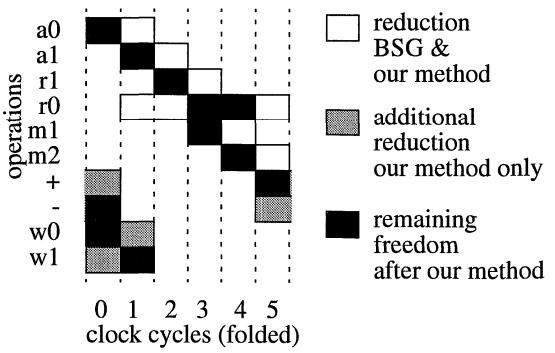


Figure 3.17 Rad2 mobility per operation

The first experiment considers two examples, the first of which is the radix-2 butterfly shown in Figure 3.16. Because the multiplication is a multicycle operation, it is modelled by two stages m1 and m2 as indicated in Section 2.4, making a total of 10 operations. The execution intervals for each operation are given in Figure 3.17 for the folded schedule. In this figure, the time is represented in clock cycles at the horizontal axis. The operations are enumerated vertically. The white area represents the reductions obtained by both BSG and our analysis. For example, the execution interval of operation r0, based on ASAP and ALAP is [1, 5]. Both our method and BSG are able to reduce this interval to [3,4]. The grey area is the reduction obtained by our analysis, that BSG was not able to find.

Notice in this figure how reduction techniques such as BSG and our techniques prevent a greedy scheduler from making a wrong schedule decision. A greedy scheduler would schedule operation r0 in clock cycle 1, leaving no room for w1 in clock cycle 7 (potential 1), which is the only feasible start time for w1. Both our analysis and BSG detect that r0 can not be scheduled in clock cycle 1. However, BSG is unable to detect that operation + must precede operation -. A greedy scheduler can easily go wrong by letting operation - precede operation +.

The second example concerns an IIR filter containing 23 operations, including fetching the coefficients and data from memory. The latency is constrained to the lower bound value of 10 clock cycles. In Table 3.1 the results of the analysis on the radix2 and the IIR example are shown, expressed in the average mobility per operation in clock cycles. For the radix2 schedules BSG is unable to find any reductions additional to the reductions obtained by our method, so for this example there is no gain in accuracy by combining the results of BSG analysis and our constraint analysis. The IIR example shows however that this is not generally true. BSG analysis is capable of deducing some reductions that our techniques can not find, and vice versa. As can be seen in Table 3.1,

combining the analyses provides larger reductions than both analyses separately. The run times for both BSG and our analysis are negligible in this experiment.

Table 3.1 Average mobility for radix-2 butterfly and IIR

	ASAP-ALAP	BSG	our method	combined
rad2 non folded	1.20	.70	.70	.70
rad2 folded	1.20	.50	.10	.10
IIR non folded	2.70	1.61	1.83	1.52
IIR folded	2.70	1.61	1.74	1.43

The second experiment considers only our analysis and concerns the same IIR filter used in the first experiment plus three loops present in FFT algorithms. The first loop (FFTa) contains 40 operations, has a minimum latency of 13 clock cycles, and needs to be folded at least three times to realize an initiation interval of only 4 clock cycles.The second loop(FFTb) contains 60 operations, has a minimum latency of 18 clock cycles, and also needs to be folded at least three times to realize an initiation interval of 8 clock cycles. The third loop (Radix) contains 80 operations, has a minimum latency of 11 clock cycles, and is folded twice to realize an initiation interval of 4 clock cycles. In Table 3.2 the results are shown.The run time of this experiment using the most extensive analysis, is less than a second on a HP 9000/735. The last column depicts the remaining mobility after analysing the first schedule decision a greedy scheduler could make (operation 23 at clock cycle 0). It is clear from the numbers in the sixth column that substantial reductions can be made not only before scheduling, but also during scheduling. This observation strengthens the idea of an interaction between the analyzer and the scheduler.

Table 3.2 Mobility reduction for some folded loops.

experiment	#operations	II	latency	mobility before analysis	mobility resource analysis	mobility after 1 decision
IIR	23	6	10	2.70	1.74	0.56
FFTa	40	4	13	4.46	3.41	2.41
FFTb	60	8	18	6.85	4.53	2.58
Rad4	81	4	11	5.29	2.82	2.08

Chapter 4 Register Binding for Randomly Addressable Register Files

Register binding is one of the three major code generation steps, as introduced in Section 1.2, the other two being code selection and scheduling. We have argued in Section 1.3.1 that for our target architectures (VLIW), the emphasis in code generation is on scheduling and register binding. Traditional approaches deal with scheduling and register binding in separate stages to reduce the complexity of the problem. This introduces the problem of *phase coupling*: a decision made in the first phase may lead to an inefficient or even infeasible constraint set for the second phase. If register binding is performed prior to scheduling, so called *anti-dependencies* are introduced that make it difficult to satisfy tight timing constraints. On the other hand, if scheduling is performed prior to register binding, there is not much opportunity left to make a register binding that respects the capacities of the register files. Therefore, although the separation of scheduling and register binding results in methods that are run-time efficient, it makes it much more difficult to cope with the interaction of timing, resource, and register file capacity constraints.

In this thesis we have taken the perspective of considering scheduling and register binding as a combined problem, and in Section 1.4 we have defined a search space accordingly. In this chapter we discuss the two register binding problems introduced in Section 2.5: Finding an *efficient* (R-feasible) register binding in the context of high-level synthesis, and finding an (S-feasible) binding that also *respects individual register file capacities*. The general approaches for these problems have been discussed in Section 2.5. Both approaches rely heavily on the constraint analyser which has to cope with the constraints associated with a (partially) given register binding. This extension to the constraint analysis techniques discussed in the previous chapter, is treated in Section 4.1. In Section 4.2 these basic techniques are applied to solve the problem of minimizing the register requirements. The corresponding experimental results are reported in 4.3. In Section 4.4 it is shown how to find a schedule that respects individual register file capacity constraints. The experimental results of this approach are given in Section 4.5.

4.1 Lifetime serialization for a given binding

The previous chapter introduced a methodology for finding a schedule that satisfies certain resource-, timing-, and precedence constraints. In this section we will extend the techniques to analyse value conflicts that result from a given register binding. This will

be done by introducing basic lemmas similar to Lemma 3.2. These lemmas provide necessary conditions (in terms of precedence relations) to guarantee the feasibility of a given register binding. Section 4.1.1 is restricted to non-folded schedules in order to explain the concept more clearly. The lemmas will be generalized in Section 4.1.2 for register conflicts that cross loop boundaries, which occur when folded schedules are considered.

4.1.1 Non-folded schedules

In this section two lemmas consider the combination of register, precedence and timing constraints for non-folded schedules. Their use is demonstrated with a small example. What is the exact consequence of binding two values u and v to the same register? Since u and v cannot be alive simultaneously, either u is consumed before v is produced, or vice versa, graphically depicted in Figure 4.1. Figure 4.2 gives a timing perspective of the alternatives in Figure 4.1. In this figure, the solid lines indicate the occupation of the register. The solid line P^v-C^v has to be placed either underneath or above the solid line P^u-C^u, corresponding to the left and right alternative in Figure 4.1 respectively. This process is called *serializing* the value lifetimes of u and v.

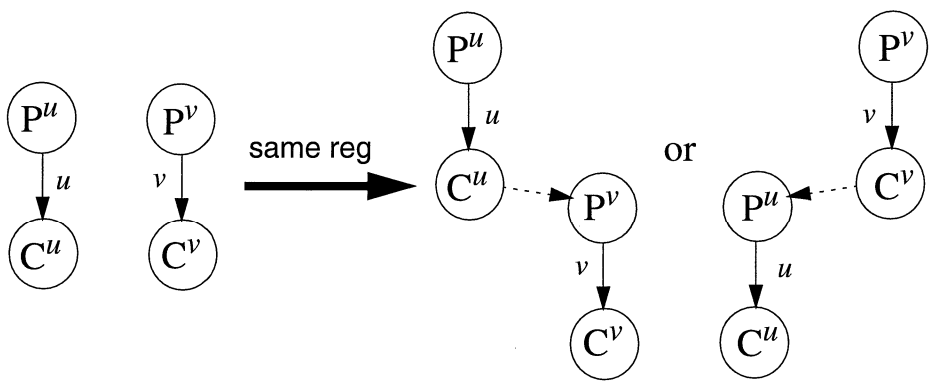


Figure 4.1 Precedence as a result of binding u and v to the same register

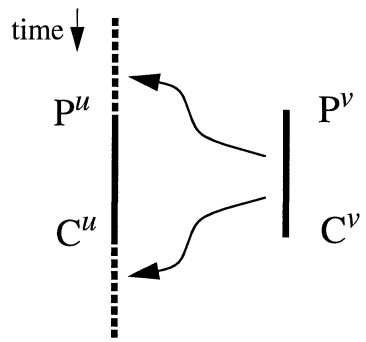


Figure 4.2 Timing perspective of the alternatives in Figure 4.1

The full consequence of binding two values to the same register is thus stated in terms of precedences. The methodology of adding precedences, introduced in Chapter 3, should therefore not be too difficult to extend to the problem of integrating the constraint of a given register binding within the DFG model. This will be done by introducing lemmas similar to Lemma 3.2. The lemmas introduced in this chapter identify situations where one of the alternatives in Figure 4.1 (and thus Figure 4.2) can be eliminated.

Lemma 4.1: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same register. If $d(P^u, P^v) \geq 0$ we can add a sequence precedence edge (C^u, P^v) with weight 0 without excluding any feasible schedules. ❑

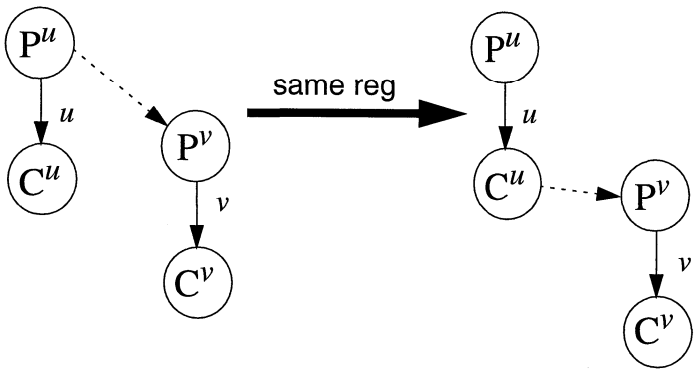


Figure 4.3 Lemma 4.1 for serializing value lifetimes

Lemma 4.1 is illustrated in Figure 4.3. A similar lemma is valid when there is a path between the consumers of the values:

Lemma 4.2: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same register. If $d(C^u, C^v) \geq 0$ we can add a sequence precedence edge (C^u, P^v) with weight 0 without excluding any feasible schedules. ❑

Lemma 4.2 is illustrated in Figure 4.4. The last situation occurs when there is a path between the producer of one value and the consumer of the other. In this case however, we can only exclude a possibility if the delay of the path is strictly greater than zero. Otherwise the alternative sequentialization, $C^v \rightarrow P^u$, could still yield a feasible schedule when P^u and C^v are scheduled in the same clock cycle.

Lemma 4.3: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same register. If $d(P^u, C^v) \geq 1$ we can add a sequence edge (C^u, P^v) with weight 0 without excluding any feasible schedules. ❑

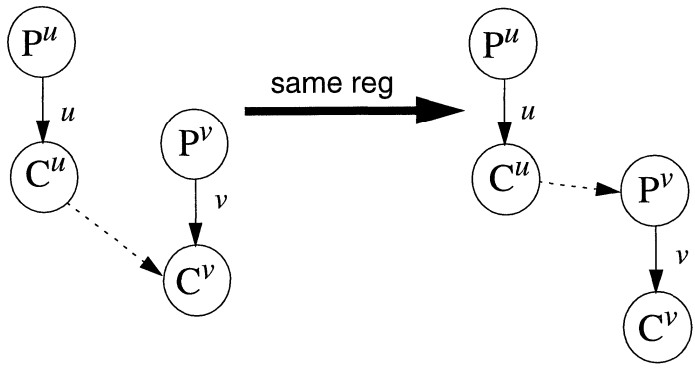


Figure 4.4 Lemma 4.2 for serializing value lifetimes

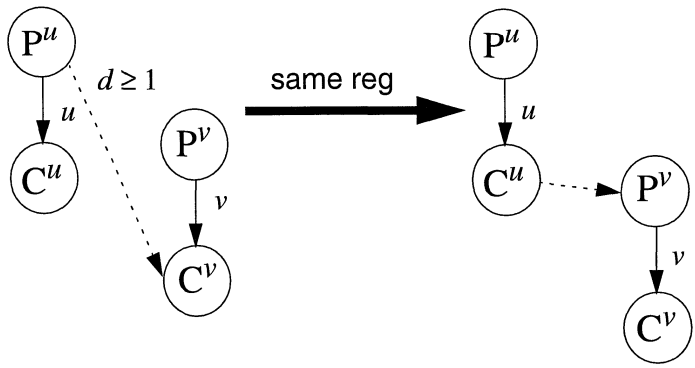


Figure 4.5 Lemma 4.3 for serializing value lifetimes

Lemma 4.3 is illustrated in Figure 4.5. The overall method of analysis is demonstrated in Figure 4.6. In this figure, the sequence edge $1 \rightarrow 7$ is a result of *alias analysis* performed in the front-end of the compiler. Apparently, operation 7 writes to a memory location that could possibly be the same as operation 1 reads from. Values u and v in Figure 4.6 reside in the same register, as do values w and x. Because operation 1 consumes value u and operation 7 consumes value v, the lifetime of u has to precede the lifetime of v as a result of the precedence $1 \rightarrow 7$ (Lemma 4.2 applies). Therefore the sequence edge $1 \rightarrow 8$ is added. Now there is a path $2 \rightarrow 1 \rightarrow 8$ from the consumer of w to the consumer of x and Lemma 4.2 applies again. The sequence edge $2 \rightarrow 9$ is added as a result. Any schedule heuristic can now find a schedule without violating the register binding, which is not the case if the sequence edges were not added.

A larger example is given in Figure 4.7. It is a IIR filter application generated by the Mistral2 toolset, to be scheduled in 11 clock cycles. Again, the sequence $12 \rightarrow 25$ is a result of alias analysis performed in the front-end of the compiler.The ASAP-ALAP intervals prior to analysis are depicted in Figure 4.8. The operations (horizontal axis) are grouped according to the resource binding. The vertical bars indicate the

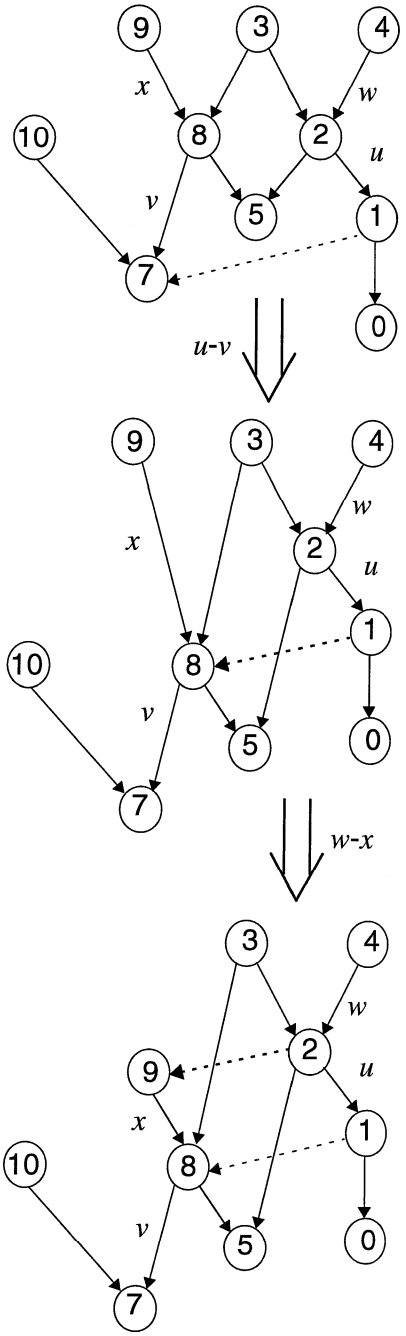


Figure 4.6 Example demonstrating the use of Lemma 4.2

ASAP-ALAP intervals. For example, operation 13 (executed on the ACU) has [ASAP; ALAP] = [1; 7].

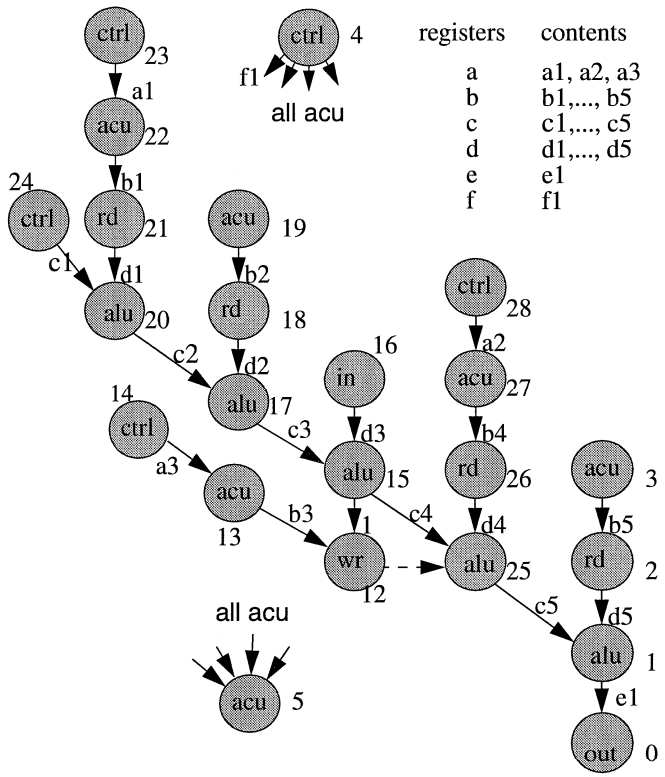


Figure 4.7 : A complete data flow graph for an IIR filter

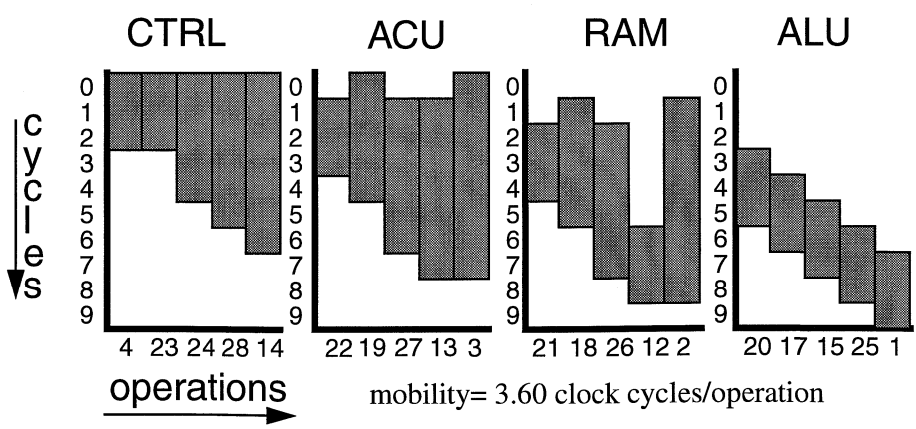


Figure 4.8 ASAP-ALAP intervals for the operations in Figure 4.7

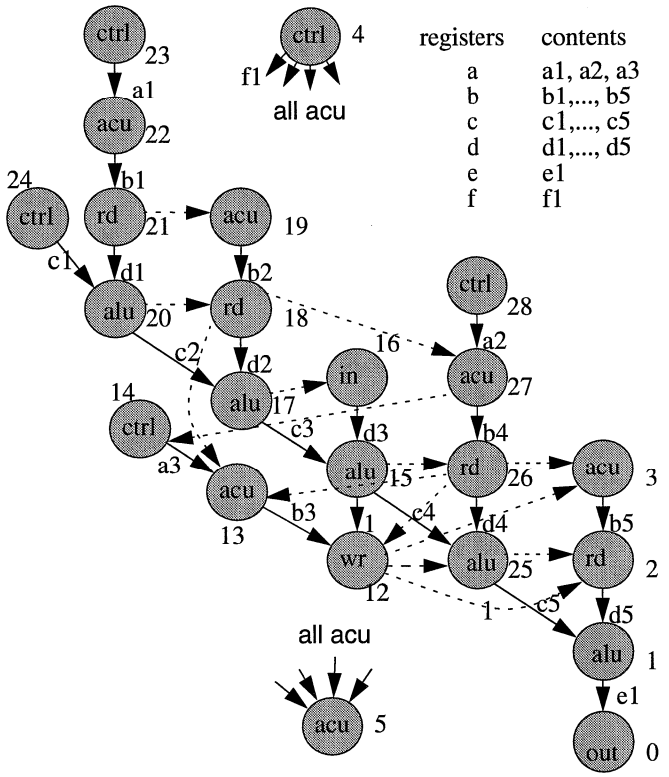


Figure 4.9 The DFG from Figure 4.7 after analysis

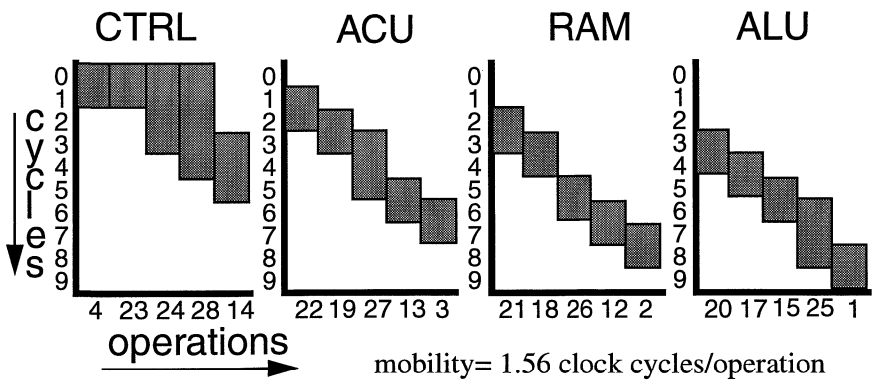


Figure 4.10 ASAP-ALAP intervals for the operations in Figure 4.9

The DFG and the ASAP-ALAP intervals after analysis are depicted in Figure 4.9 and Figure 4.10 respectively. Application of Lemma 4.1 and Lemma 4.2 causes a substantial reduction in the ASAP-ALAP intervals; the mobility drops from 3.60 to 1.56 clock cycles per operation. The reason is that sequence edges accumulate: the introduction of

one sequence edge causes a precedence that may again give rise to a situation suitable for applying one of our lemmas again. New sequence edges result from old ones recursively.

4.1.2 Folded schedules

In this section we extend the lemmas from Section 4.1.1 for serializing value lifetimes, to handle pipelined loop schedules. An example demonstrates the use of the extended lemmas.

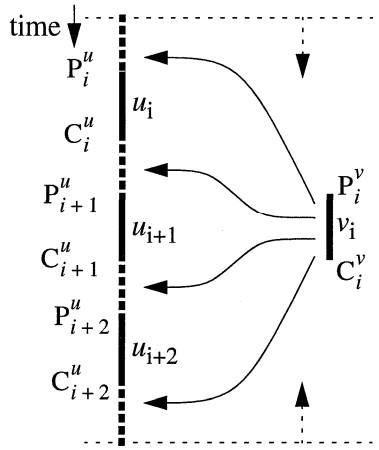


Figure 4.11 4 possible placements of P^v-C^v if the maximum folding factor equals 1

When schedules are not folded it is relatively simple to avoid overlapping lifetimes of values residing in the same register. When loop iterations overlap in time, we also have to take care that the i^{th} lifetime of value v does not overlap with the $i+1^{th}$ lifetime of value u, depicted in Figure 4.11. This means we have to serialize value lifetimes belonging to different loop iterations. The graph model however, makes no difference between operation A_i and A_{i+1} (where A_i denotes the i^{th} execution of A), because it has no notion of loop iteration. In Section 2.2 we showed the equivalence between the relation $C_i \rightarrow P_{i+k}$ and the relation $C \rightarrow P$ with time delay $-k \cdot \Pi$. This equivalence is used to generalize the lemmas from Section 4.1.1. We derive a generalization of Lemma 4.1 in the following way. First, we use the equivalence to translate the timing delay of $k \cdot \Pi$ at the left hand side of the arrow in Figure 4.12 to the iteration indices at the right hand side. In Figure 4.13 we apply Lemma 4.1 directly on the operations that are now annotated with the iteration index. In Figure 4.14 we translate the iteration indices back to a timing relation from C^u to P^v. Lemma 4.1 is now easily generalized to Lemma 4.4:

Lemma 4.4: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same register. If $d(P^u, P^v) \geq k \cdot \Pi$ we can add a sequence edge (C^u, P^v) with weight $k \cdot \Pi$ without excluding any feasible schedules. ❑

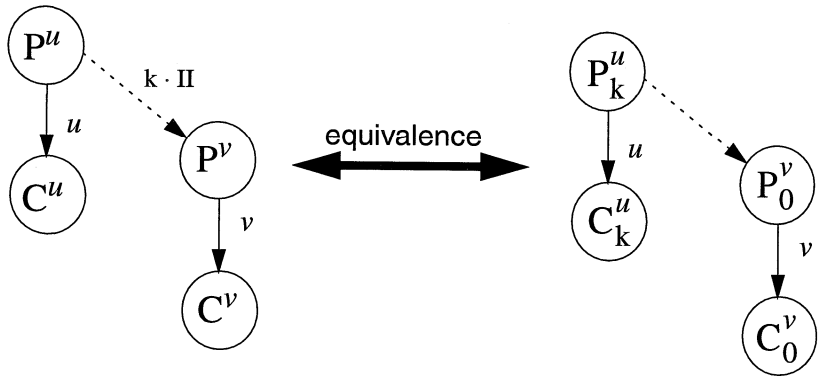


Figure 4.12 First step of generalizing Lemma 4.1.

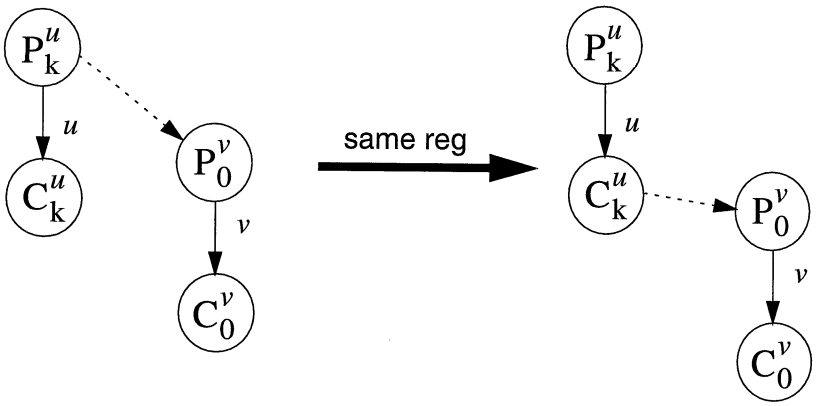


Figure 4.13 Second step of generalizing Lemma 4.1.

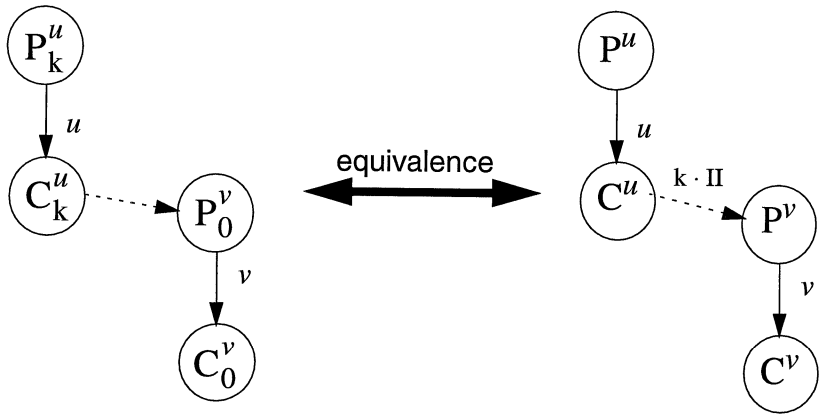


Figure 4.14 Third step of generalizing Lemma 4.1.

Lemma 4.4 is illustrated in Figure 4.15. Lemma 4.2 is generalized to Lemma 4.5:

Lemma 4.5: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same register. If $d(C^u, C^v) \geq k \cdot II$ we can add a sequence edge (C^u, P^v) with weight $k \cdot II$ without excluding any feasible schedules. ❑

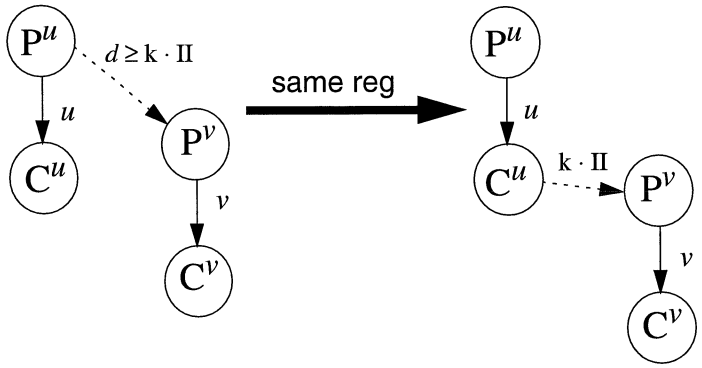


Figure 4.15 Lemma 4.4 for serializing value lifetimes

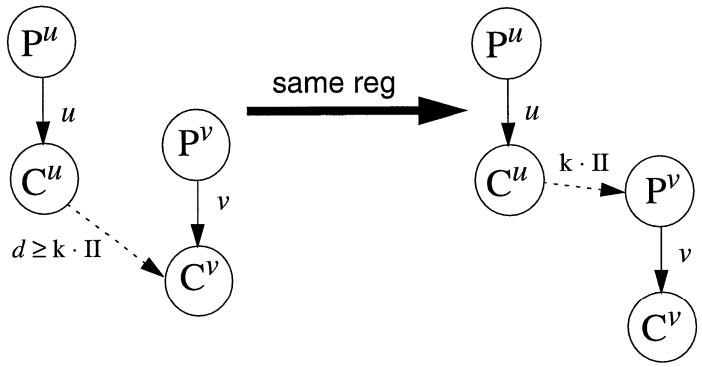


Figure 4.16 Lemma 4.5 for serializing value lifetimes

Lemma 4.5 is illustrated in Figure 4.16. Lemma 4.3 is generalized to Lemma 4.6:

Lemma 4.6: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same register. If $d(P^u, C^v) \geq k \cdot II + 1$ we can add a sequence edge (C^u, P^v) with weight $k \cdot II$ without excluding any feasible schedules. ❑

Lemma 4.6 is presented graphically in Figure 4.17. We illustrate the use of the lemmas in this section with the example in Figure 4.18. It is similar to the example of Figure 2.4, but it is extended with a register binding. Value v, communicated from operation A to B and value w, communicated from operation C to D, are bound to the same register. The same resource conflicts and the same initiation interval are used, but there is no

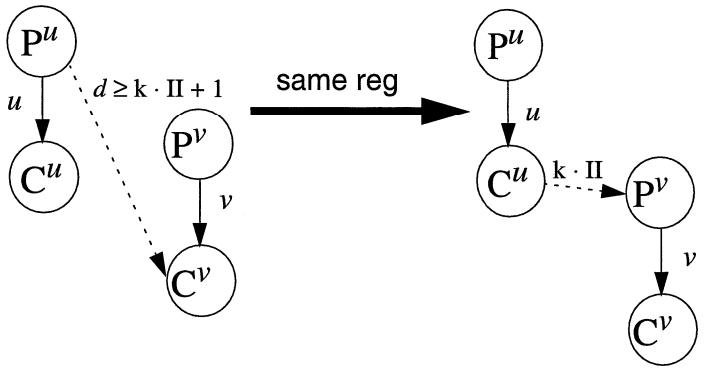


Figure 4.17 Lemma 4.6 for serializing value lifetimes
constraint on the latency. The first step from a) to b) is the same as the first step in Figure 3.7.

from b to c: Value v is produced by A and consumed by B. Value w is produced by C and consumed by D. Lemma 4.6 applies because $d(A, D) \geq 4 = 1 \cdot II + 1$, so we can add a sequence edge (B,C) with weight $1 \cdot II = 3$ without excluding any feasible schedules.

In Figure 4.19 a folded ASAP schedule is given that satisfies the newly added precedence constraints, and thus also the resource constraints and the register binding. In Figure 4.19, the leftmost column indicates the *time potential* (start time modulo II), so operation C is scheduled in clock cycle 4, D in 5 etc. Notice that the constraints have forced a gap of 2 clock cycles between operations B and C. A greedy scheduling approach does not put gaps between operations, and would not have found a schedule that satisfies all constraints.

The last basic lemma we introduce in this chapter generalizes a modelling issue discussed in Section 2.4: Lifetimes of values stored in randomly addressable registers are not allowed to exceed the initiation interval, resulting for each data precedence $P \rightarrow C$ in a precedence constraint $C \rightarrow P$ with weight -II (Section 2.4). When more than one value (the set of values W) is stored in a register r, the sum of the lifetimes lt in this register is not allowed to exceed the initiation interval: $\sum_{v \in W} lt(v) \leq II$. So

$$\forall (u \in W): lt(u) \leq \left\lceil II - \sum_{v \in W/v} lt(v) \right\rceil \leq \left\lceil II - \sum_{v \in W/u} minlt(v) \right\rceil$$

Lemma 4.7: Let W be the set of values that reside in a register r, and let minlt(v) denote the minimal lifetime of value v (the distance from the producer of v to the last consumer of v). Then value $u \in W$ has a maximum lifetime equal to $II - \sum_{v \in W/u} minlt(v)$. ❑

This upper bound on lt(u) can be modelled in the DFG as a sequence edge $C^u \rightarrow P^u$ with weight $\left(\sum_{v \in W/u} minlt(v) \right) - II$.

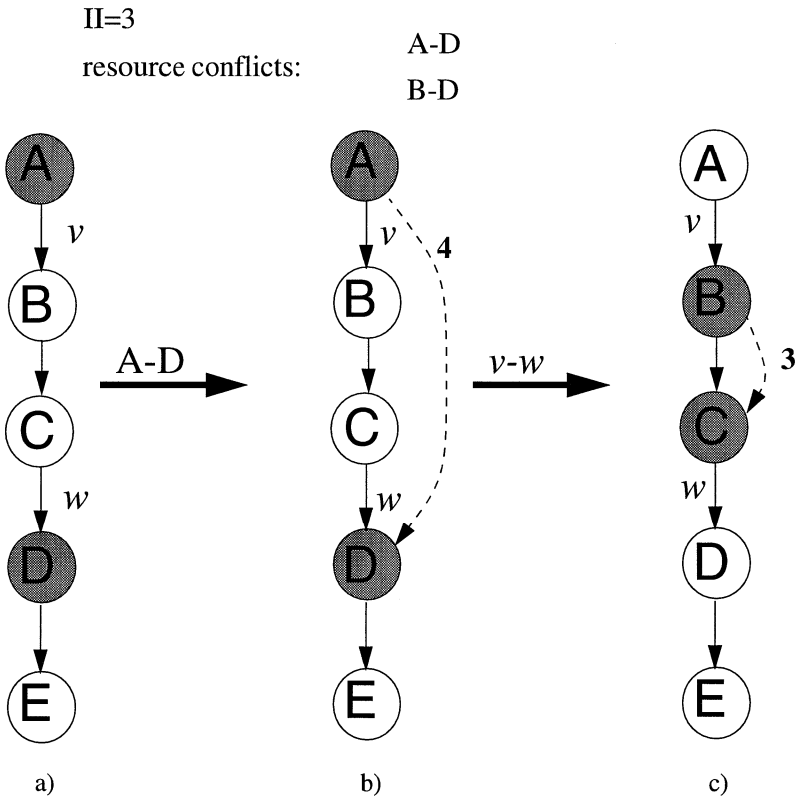


Figure 4.18 Derivation of a partial schedule

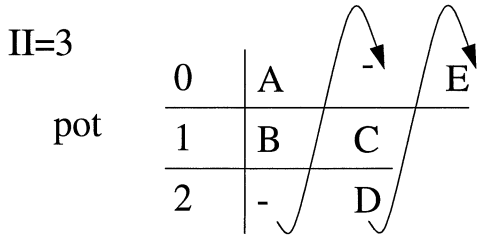


Figure 4.19 Folded ASAP-Schedule for Figure 4.18

We have now covered the basic techniques used in the constraint analyser of Figure 2.8 and Figure 2.10. The next two sections demonstrate the use of these basic lemmas in deriving a register binding.

4.2 Infeasibility Analysis

In this section we tackle the problem of minimizing the register count, as introduced in Section 2.5.1. In that section the general approach is discussed, both using a flow dia-

gram (repeated in Figure 4.20), and from the perspective of how the search space is traversed (Figure 2.9). The process starts with an initial register binding. This register binding requires the least number of registers but will usually be overconstrained (infeasible) in the sense that the binding is inconsistent with the timing constraints.

The schedule analysis is often capable of detecting that the register binding together with the constraint set yields an infeasible result. In order to make a sensible change in the register binding, the infeasibility analyser in Figure 4.20 has to identify the bottleneck in the register binding. More precisely, we want the analyser to give a *smallest infeasible subset of value conflicts*, where a conflict denotes two values residing in the same register. This subset of value conflicts constitutes the cause of infeasibility. Identifying such a subset of conflicts is tightly related to detecting infeasibility. The constraint analyser detects infeasibility based on the distance matrix in the following way: When a path is found from an operation v to itself (a cycle in the precedence graph), and this path has a positive length, the operation v is forced to execute strictly before its own start time, which is clearly not possible. So a precedence cycle of strictly positive length indicates infeasibility.

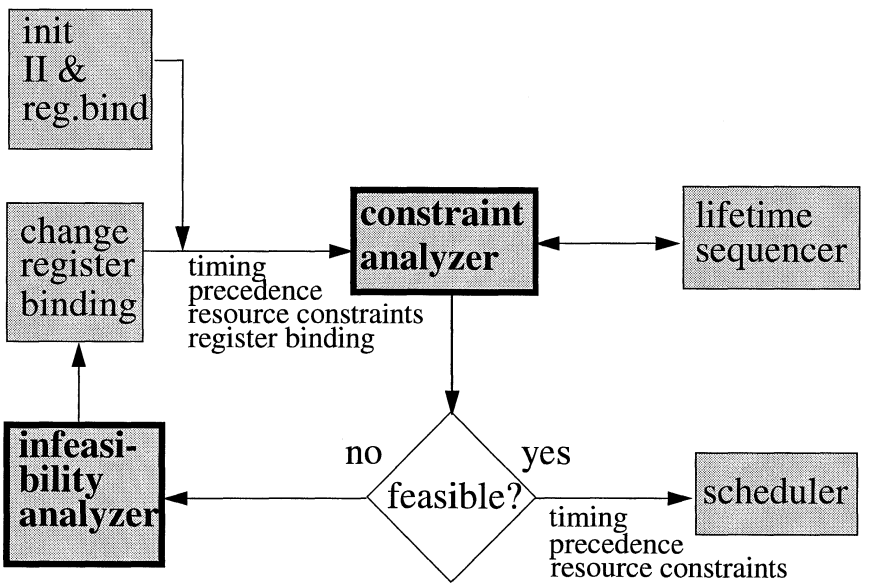
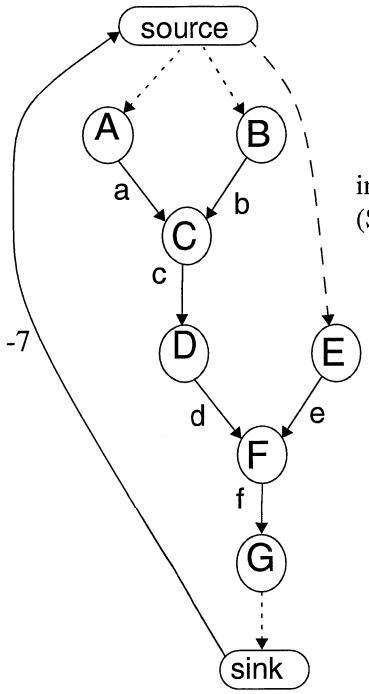


Figure 4.20 Global approach for minimizing the register count

The source of the bottleneck is directly related to the way the positive length cycle came into existence. For example, if in Figure 4.18 the latency is constrained to 6 clock cycles, there was a sequence edge from the sink to the source with a delay of -6 clock cycles. In Figure 4.18c that would yield a positive delay cycle, thus proving infeasibility. Most edges in the precedence cycle involve data precedences, one involves the latency, and one involves a register conflict. The sequence edge $B \rightarrow C$ is a result of two components: 1) the register conflict v-w, and 2) a path of length 4 from A to D. The



II=2
latency=7
Resource conflicts: A-D, B-E, C-F, D-G

initial reg.bind: (Section 4.2)
reg1: a,d
reg2: b,e
reg3: c,f

	A	B	C	D	E	F	G
A	0	0	1	3	3	4	6
B	-1	0	1	2	3	4	5
C	-2	-2	0	1	1	3	4
D	-3	-3	-2	0	0	1	3
E	-4	-3	-2	-1	0	1	2
F	-5	-5	-3	-2	-2	0	1
G	-6	-6	-5	-3	-3	-2	0

Figure 4.21 Example of a precedence graph

path from A to D consists of one sequence edge that is added as a result of the resource conflict A-D and a path $A \rightarrow D$ of length 3 that consists entirely of data precedences. We can thus conclude that infeasibility is caused as a result of the following combination of factors: 1) a register conflict v-w, 2) a resource conflict A-D, 3) the latency constraint, and 4) data precedence. When all constraints are fixed except for the register binding, we conclude that the decision to put the values v and w together in a single register is the cause of infeasibility.

Another example is the graph depicted in Figure 4.21. The constraint set is infeasible with the given register binding, which is derived as follows. The infeasibility analysis is graphically depicted in Figure 4.22. Each block represents a path, and each downward arrow represents an inference. The derivation is top down. The path $D \rightarrow G$ of length 2 (=II) and register conflict c-f lead to the sequence edge $D \rightarrow F$ of weight II=2 as a consequence of Lemma 4.5 (where k=1). The downward arrow shows that this sequence edge is part of the path underneath. The second block from the top indicates a path $C \rightarrow F$ of length 3. Together with the register conflict a-d this yields a sequence edge $C \rightarrow D$ of weight 2 as a result of Lemma 4.5. In the third block the conflict a-d is used again with the path $C \rightarrow F$ of length 4 to add the sequence edge $C \rightarrow D$ of weight 4. The block at the bottom shows that this sequence edge causes a positive precedence cycle $C \rightarrow D \rightarrow C$ with a delay 4 + (-2) = 2 clock cycles. The edge $D \rightarrow C$ with delay -2 is added because the lifetime of each value (in this case value c) cannot exceed II clock

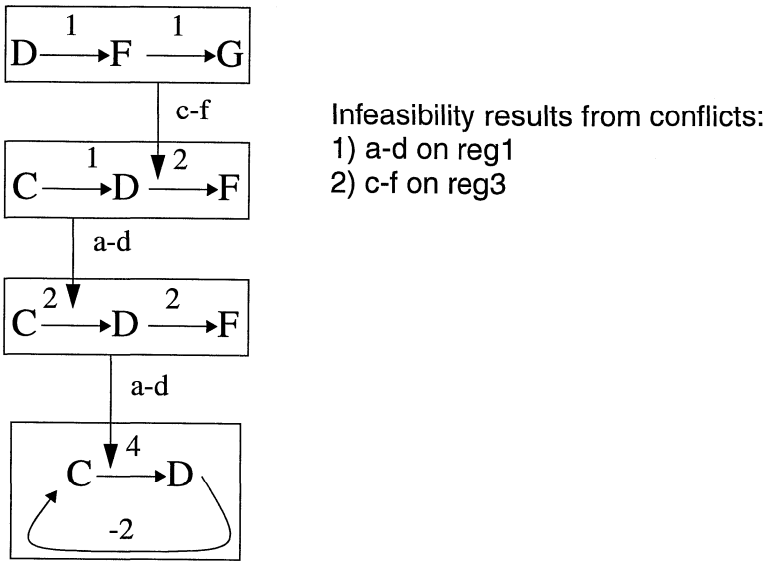


Figure 4.22 Infeasibility analysis for Figure 4.21

cycles, so the consumer (D) must execute within 2 clock cycles after the producer (C). As a result of this positive precedence cycle we conclude that the register binding is infeasible.

The infeasibility analysis is done in bottom-up fashion, to identify exactly those sequence edges and conflicts which have contributed to the positive precedence cycle. The combination of register conflicts that yield infeasibility is identified as 1) a-d on register 1 and 2) c-f on register 3. Note that the conflict b-e on register 2 did not contribute to the infeasibility, and thus it is useless to put the values b and e in separate registers. Instead we have to choose to 'split' either register 1 or register 3. Both decisions yield a feasible schedule, as depicted in Figure 4.23.

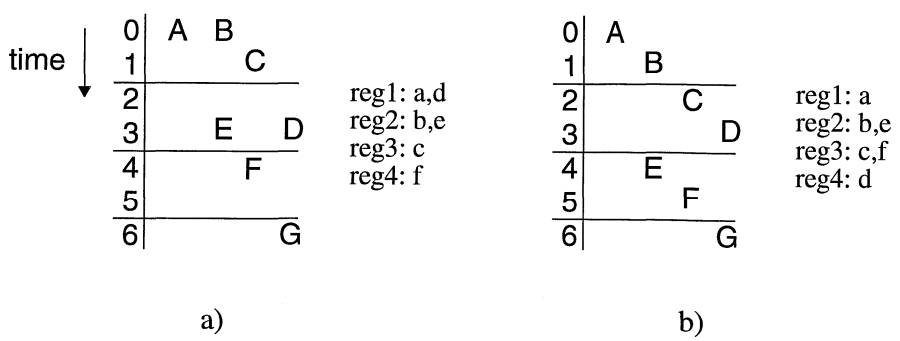


Figure 4.23 The only 2 feasible schedules for Figure 4.21 with changes in the register binding

In this approach a simple heuristic chooses the register conflict to be solved based on the availability of registers in a certain register file, the number of times the conflict appears in the conflict-list, etc.

As the reader may have noticed from the examples, the infeasibility analysis requires a lot of administrative bookkeeping. Almost every path constructed during the longest path analysis has to be kept in memory for reference. A feasible implementation requiring a limited amount of memory to run, is only guaranteed if the storage of a path has a memory cost of O(1). This is possible with the use of an *adjacency matrix* [Corm90], which is based on the following fact of longest paths: if the longest path from A to C travels through B, then the part B to C is the longest path from B to C. As a result, the only administration necessary for the path from A (row of the matrix) to C (column of the matrix) is the first node on the path after A. To facilitate the infeasibility analysis, we also administrate the first edge traversed on the path A to C. Each sequence edge on its turn has a pointer to a register conflict (if there is one) and the matrix entry representing the path that gave rise to the edge. The complexity of the infeasibility analysis is thus linear in the number of edges. Note however, that the amount of edges is not bounded by V^2, since more than one sequence edge (with different delay) may be added between each pair of operations. Also, we assume that the longest paths have already been calculated in the constraint analyser.

4.3 Experimental results

Our implementation on a HP 9000/735 has been tested on the inner loops from 4 different real life industrial examples [Mesm98, Mesm99a]. The results are shown in Table 4.1. The fifth column represents the number of iterations over the schedule analyzer (see Figure 4.20) before a feasible solution was found. The last 2 columns indicate the mobility of the operations in terms of average number of clock cycles per operation. The 7th column indicates the mobility before the analysis, the last column after analysis (what is left for the scheduler to fill in).

The first experiment concerns an IIR filter of 23 operations, including fetching the coefficients and data from memory. The latency is constrained to the lower bound of ten clock cycles. The other experiments concern FFT applications, the largest of which holds 81 operations. Note in Table 4.1 that the run-times are mainly determined by the number of iterations over the schedule analyzer. The number of iterations is a measure of the difficulty of finding a register binding because it reflects the number of changes made to the original binding in order to obtain a feasible schedule.

The mobility is decreased by a factor ranging from 3.6 (Rad4) to 13.2 (FFTb) as a result of the schedule analysis. Because this decrease of mobility is due to the constraints, it is a measure for the analyzers' capability of directing the scheduler and preventing it from making schedule decisions that violate the constraints.

Table 4.1 Results of constraint analysis on DSP loop kernels

experiment	# operations	II constraint	latency constraint	# iterations	Run-time	mobility before analysis	mobility after analysis
IIR	23	6	10	3	0.2 s	2.70	0.13
FFTa	40	4	13	11	17 s	4.46	0.46
FFTb	60	8	18	20	25 s	6.85	0.52
Rad4	81	4	11	1	0.8 s	4.93	1.38

We have included one more experiment to test the performance of our method on a problem instance that was not constrained with respect to timing. It is a preliminary test executed by Frontier Design, who are integrating our method within the Mistral2 toolset [Strik94]. The benchmark, Par2, contains 91 operations. The original schedule, generated by the Mistral2 toolset, counts 61 clock cycles. As a result of the available parallelism and the number of memory accesses the register binder required 6 registers at the address generation unit. The schedule generated by our method, counts only 56 clock cycles and requires only 1 register at the address generation unit. Because of the schedule freedom, a total of 111 schedule decisions had to be made by the lifetime sequencer. Run time is less than a second. The efficient register binding of the new schedule was expected, unlike the reduction in the number of clock cycles. This reduction is explained as follows: Because of the serialization of the address lifetimes the precedence graph became more regular. It is a well-known fact that heuristics such as the list-scheduling are able to find more efficient schedules when the precedence graph contains more regularity.

4.4 Incremental register binding for fixed register files

This section considers the problem of finding a register binding for programmable processors. Contrary to the register binding task in High-Level Synthesis (previous section), we are now dealing with a fixed capacity of the available register files. This characteristic has two major consequences:

- Using as *few* registers as possible is not the ultimate goal: instead of obtaining a minimal register binding we would rather use *all* available registers and find a schedule that takes one clock cycle less to execute.

- The number of registers required in a certain register file is not allowed to exceed the capacity of that file.

The accepted way to deal with fixed register files in a compiler is to do register spilling [Chai82]. When the register binding violates the register file capacity some values are selected that are written to a (background) memory. Load and store operations are inserted and the block of operations is rescheduled. This process is repeated until the

capacity of each register file is respected. Because of the additional spill operations and more extensive usage of load/store resources, severe timing constraints are unlikely to be satisfied in the final schedule. As a result, embedded system designers go through the effort of either 'helping' the scheduler with pragmas (hints) or scheduling time critical code completely by hand. This requires extensive knowledge of the processor architecture and instruction set and is very time consuming. Therefore it is desirable to find an approach avoiding the generation of spill code and coping with severe timing constraints. In this section we focus on such an approach. It integrates scheduling and register binding, and thereby selectively uses the available schedule freedom to satisfy all the constraints, including the timing and capacity constraints. (This work has been done in cooperation with Carlos Alba Pinto and Koen van Eijk from the Eindhoven University of Technology [Mesm99b], [Alba99].)

A formal problem formulation is given in Section 2.5.2. The global decomposition for solving the Constrained Register Binding and Operation Scheduling Problem, is given in Figure 4.24. The incremental register binder has to serialize value lifetimes incrementally until all values assigned to a certain register file actually fit in this register file. This has to be done incrementally, because serializing two values effects the mobility of potentially all operations, and thus may prevent serializing other values. Therefore the effect of serializing two values has to be computed by the constraint analyser before other values are serialized. As a result, the focus of the process alternates between deciding over a register binding and pruning the schedule search space accordingly until the capacity constraints are satisfied. In Section 2.5.2 it is discussed how the search space looks like and how it is traversed. This approach resembles (on an abstract level) the way [Rau98] finds a resource binding and schedule.

The incremental register binder has to act very careful as to which values to serialize. Only those actions should be taken that actually provide a 'better' fit of values to a register file, otherwise schedule freedom is invested for no purpose. Therefore, an essential feature of the incremental register binder is to identify the main bottlenecks violating the register file capacities. This task is performed by analysing potential conflicts between pairs of values before and during scheduling. In Section 4.4.1 it is shown how potential conflicts are identified. In Section 4.4.2 these conflicts are analysed for potential bottlenecks by colouring a 'worst case' and 'best case' conflict graph. The approach is demonstrated using a small example.

4.4.1 Constructing a conflict graph

A conflict graph is an undirected graph $CG(RF) = (V^c, E^c)$, where the nodes in V^c represent the values in register file RF. There is an edge $(u, v) \in E^c$ if the lifetimes of u and v overlap, and there is *no* edge $(u, v) \in E^c$ if the lifetimes of u and v do *not* overlap. The triviality of the latter remark soon fades when we try to construct a conflict graph in the case that the lifetimes are not fixed yet. Consider Figure 4.21 without pipelining; not two, but three different relations may exist between two values:

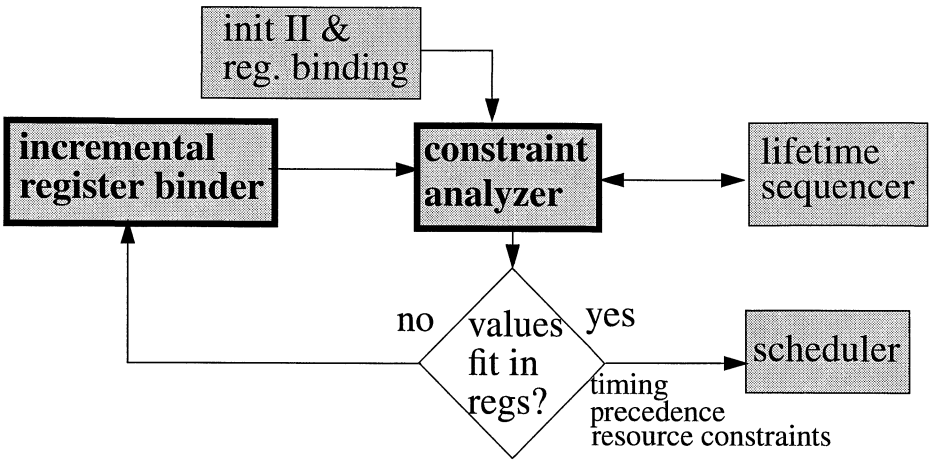


Figure 4.24 Global approach for mapping to fixed register files

- There is no overlap. This is the case e.g. for values a and c.

- There is overlap. This is the case e.g. for values a and b in the clock cycle assigned to the execution of operation C. We call this *strong* overlap.

- Unknown. This is the case e.g. for values b and e: if operation E precedes operation C by at least one clock cycle, b and e overlap. If not, b and e have no overlap. Since it is not yet determined whether or not E precedes C, it is simply unknown if b and e overlap. We call this *weak* overlap.

For our purposes the following is the essential difference between strong and weak overlap: Strongly overlapping values can never reside in the same register, but weakly overlapping values can still be serialized. Serializing eliminates schedule freedom however. Because some distances increase (some paths become longer), the mobility of individual operations is affected. This is disadvantageous because intuitively it becomes 'harder' to find a feasible schedule. Therefore we want to select the values to serialize carefully, such that on one hand, the amount of *potential* overlap (unknown + overlap) in a potentially overloaded register file is reduced, and on the other hand, not too much schedule freedom is sacrificed to obtain that goal. The potential overlap is computed by considering the weak + strong overlap in the conflict graph. The three possible relations between values are distinguished based on information from the distance matrix, our central data base. In the following we will give formal definitions of the properties of non, weak, and strong overlap. Subsequently, criteria based on distances are given and proven to be equivalent to the formal criteria.

Non conflicting values. Values u and v have no conflict if their lifetimes can never overlap. There is no overlap between values u and v if and only if the lifetime of v is contained in the interval in between two successive lifetimes of u. This is depicted graphically in Figure 4.25. The situation is captured by the following definition:

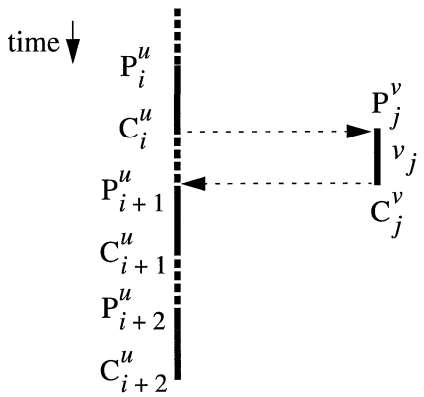


Figure 4.25 Values u and v have no conflict

Definition 4.1 Values u and v have no conflict if and only if for each iteration i there exists a corresponding iteration j such that $d(C_i^u, P_j^v) \geq 0$ and $d(C_j^v, P_{i+1}^u) \geq 0$. ❑

Definition 4.1 is equivalent to the following criterion.

Theorem 4.8: Values u and v have no conflict if and only if

$$\left\lfloor \frac{d(C^u, P^v)}{II} \right\rfloor + \left\lfloor \frac{d(C^v, P^u)}{II} \right\rfloor \geq -1 \qquad (4.1)$$

Proof. Let k be the largest value such that $d(C^u, P^v) \geq k \cdot II$ and let l be the largest value such that $d(C^v, P^u) \geq l \cdot II$. Because we assume that the latency is bounded, such values can always be found. Because for each operation A: $s(A_k) = s(A_0) + k \cdot II$, we have that $d(C^u, P^v) \geq k \cdot II$ is equivalent to $d(C_k^u, P_0^v) \geq 0$, and that $d(C^v, P^u) \geq l \cdot II$ is equivalent to $d(C_0^v, P_{k+1}^u) \geq (k + 1 + l) \cdot II$. Now Definition 4.1 applies if and only if $(k + 1 + l) \cdot II \geq 0$. Because $II > 0$, this condition is equivalent to $k + l \geq -1$. Now by definition

$$k = \left\lfloor \frac{d(C^u, P^v)}{II} \right\rfloor \qquad (4.2)$$

and

$$l = \left\lfloor \frac{d(C^v, P^u)}{II} \right\rfloor \qquad (4.3)$$

so inequality (4.1) follows. ❑

Strongly conflicting values. Values u and v have a strong conflict if their lifetimes overlap for sure. There is overlap between values u and v if and only if the lifetime of v can never be contained in the interval in between two successive lifetimes of u. This is depicted graphically in Figure 4.26. The situation is captured by the following theorem:

Theorem 4.9: Values u and v have a strong conflict if and only if for each iteration i there exists a corresponding iteration j such that $d(P_i^u, C_j^v) \geq 1$ and $d(P_j^v, C_i^u) \geq 1$. ❑

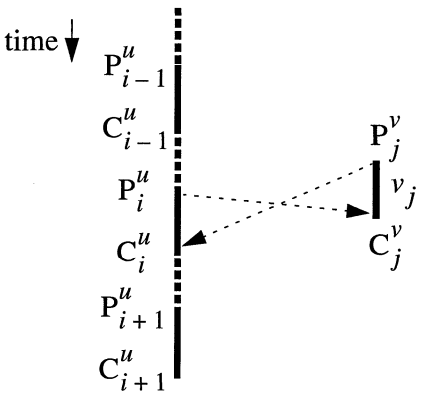


Figure 4.26 Values u and v have a strong conflict

Proof. Suppose the execution order of operations P_i^u, C_i^u, P_j^v, and C_j^v is fixed. The following conditions cover the range of possibilities. Either v_j precedes u_i ($d(C_j^v, P_i^u) \geq 0$), u_i precedes v_j ($d(C_i^u, P_j^v) \geq 0$), or v_j and u_i overlap. Because these situations are mutually exclusive, the condition for overlap is given by

$$\begin{aligned} & \neg(d(C_j^v, P_i^u) \geq 0 \vee d(C_i^u, P_j^v) \geq 0) \\ = \; & \neg(d(C_j^v, P_i^u) \geq 0) \wedge \neg(d(C_i^u, P_j^v) \geq 0) \\ = \; & d(P_i^u, C_j^v) \geq 1 \wedge d(P_j^v, C_i^u) \geq 1 \end{aligned} \tag{4.4}$$

Theorem 4.9 follows. ❑

For the non-folded case we have $i = j$. This corresponds to the case that P^u precedes C^v by one clock cycle **and** P^v precedes C^u by one clock cycle. In Figure 4.21 for example, values a and b have a strong conflict, as depicted in Figure 4.28.

Theorem 4.9 is equivalent to the following criterion.

Theorem 4.10: Values u and v have a strong conflict if and only if

$$\left\lfloor \frac{d(C^u, P^v) - 1}{II} \right\rfloor + \left\lfloor \frac{d(C^v, P^u) - 1}{II} \right\rfloor \geq 0 \qquad (4.5)$$

Proof. Let k be the largest value such that $d(P^u, C^v) \geq k \cdot II + 1$ and let l be the largest value such that $d(P^v, C^u) \geq l \cdot II + 1$. By the definition of II, for each operation A: $s(A_k) = s(A_0) + k \cdot II$. Therefore, $d(P^u, C^v) \geq k \cdot II + 1$ is equivalent to $d(P^u_k, C^v_0) \geq 1$, and $d(P^v, C^u) \geq l \cdot II + 1$ is equivalent to $d(P^v_0, C^u_k) \geq (k + l) \cdot II + 1$. Now Theorem 4.9 applies if and only if $(k + l) \cdot II \geq 0$. Because $II > 0$, this condition is equivalent to $k + l \geq 0$. Now by definition

$$k = \left\lfloor \frac{d(P^u, C^v) - 1}{II} \right\rfloor \qquad (4.6)$$

and

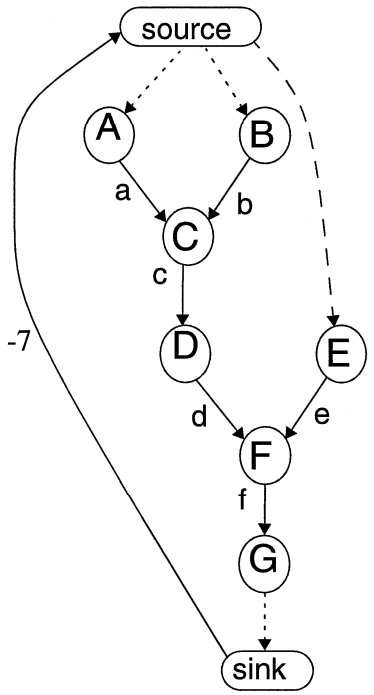
$$l = \left\lfloor \frac{d(P^v, C^u) - 1}{II} \right\rfloor \qquad (4.7)$$

so inequality (4.5) follows. ❑

Weakly conflicting values. There is weak overlap if both inequalities (4.1) and (4.5) are invalid. In Figure 4.21 for example, values a and e weakly overlap, as depicted in Figure 4.28.

4.4.2 Colouring and bottleneck identification

In the previous section we showed how to construct a conflict graph with three possible relations between values. In this section we use the conflict graph to identify two values that should be serialized in order to reduce the potential overload on the corresponding register file. The criteria for selecting these values are derived from a so called *colouring* of a conflict graph. In Figure 4.28 such a colouring is shown, where numbers are used rather than colours. A valid colouring assigns each node in the conflict graph a colour such that conflicting nodes (nodes connected by an edge) have different colours assigned to them. A minimum colouring is a valid colouring with a minimum number of colours. This number is called the *chromatic* number and is a property of the conflict graph. Because edges in the conflict graph model overlap in value lifetimes, a valid (minimum) register binding can be extracted from a valid (minimum) colouring by interpreting the colours as registers [Chai82]. We apply the exact sequential colouring algorithm from [Coud97] to find a minimum colouring. Traditional colouring-based methods for register binding construct a conflict graph *after* the value lifetimes are fixed



II=2
latency=7
Resource conflicts: A-D
B-E
C-F
D-G

	A	B	C	D	E	F	G
A	0	0	1	3	3	4	6
B	-1	0	1	2	3	4	5
C	-2	-2	0	1	1	3	4
D	-3	-3	-2	0	0	1	3
E	-4	-3	-2	-1	0	1	2
F	-5	-5	-3	-2	-2	0	1
G	-6	-6	-5	-3	-3	-2	0

Figure 4.27 DFG used to illustrate the serializing process

by a schedule. In that case, a minimum colouring corresponds directly to a minimum register binding. In our method, a conflict graph is constructed for a *partial* schedule. In the previous section we saw that this results in an additional possible relation between values: a weak conflict. In order to cope with this additional type of conflict, two different conflict graphs are created: a weak conflict graph WCG, that includes both weak and strong conflicts, and a strong conflict graph SCG, that includes only strong conflicts. The weak and strong conflict graphs associated with the DFG in Figure 4.27are depicted in Figure 4.28. A minimum colouring of the weak conflict graph corresponds to a pessimistic or worst case colouring; the chromatic index of WCG is an *upper* bound to the number of registers required in *any* completion of the schedule. Similarly, a minimum colouring of the strong conflict graph corresponds to an optimistic or best case colouring; the chromatic index of SCG is a *lower* bound to the number of registers required in any completion of the schedule. These bounds are used to steer the continuation of the serializing process. If the capacity exceeds the upper bound, no further serialization is required for that register file. If the lower bound exceeds the capacity, the process is in an infeasible region of the search space, and backtracking is performed.

From a minimum colouring, for each node v in the conflict graph we extract the so called *saturation number*, the number of different colours in the neighbourhood of v (the nodes connected to v in the conflict graph). In Figure 4.28a) for example, the neighbourhood of node e consists of nodes a, b, c, and d. Colours 1 and 2 are used to

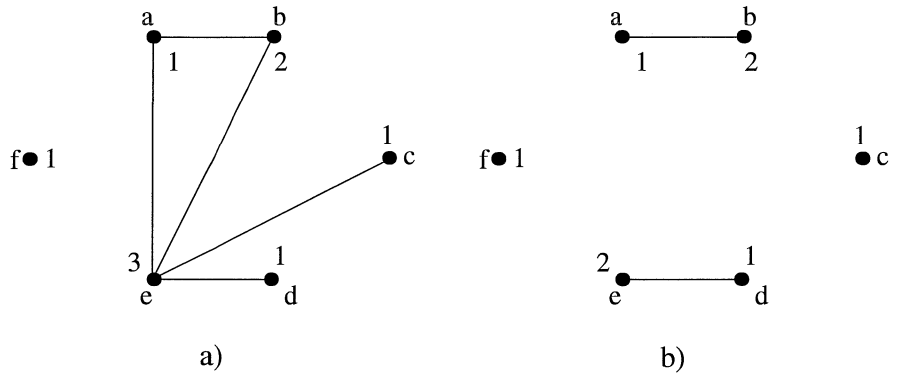


Figure 4.28 Weak conflict (a) and strong (b) coloured conflict graph for Figure 4.21 without pipelining.

colour the neighbourhood of node e, so the saturation number of node e equals 2. Similarly, the saturation number of node a equals 2, and the saturation number of node c equals 1. Saturation numbers are an indication of a bottleneck for the colouring algorithm, because they indicate how many colours the algorithm requires in a subgraph of the conflict graph. Saturation numbers in the *strong* conflict graph indicate bottlenecks that can *not* be solved by serializing value lifetimes, whereas saturation numbers in the *weak* conflict graph indicate bottlenecks that often *can* be solved by serializing value lifetimes. In Figure 4.28 for example, nodes a,b and e all have saturation number two. Because that is the highest saturation number in the graph, these three nodes constitute a bottleneck. This bottleneck can be reduced if one of the conflicts between them is eliminated. The conflict a-e and b-e can be eliminated, but the conflict a-b cannot, because it is a strong conflict.

Now we can explain the process of selecting two values, referred to as u and v, for serialization. For u we choose a node which is primary a bottleneck in WCG. Because many such bottlenecks may exist, we prefer nodes that also constitute a bottleneck in SCG. So we use the highest saturation number in WCG as a first criterion and the highest saturation number in SCG as a second criterion. For value v we choose the highest saturation number in WCG, since we are looking for bottlenecks in WCG. The second criterion however is the *lowest* saturation number in the strong conflict graph, SCG. The rationale behind this is that value v should be such that it has the potential to be serialized with many other values, which is not the case if it constitutes a bottleneck in the strong conflict graph. Furthermore, we maintain the restriction that u and v have a weak conflict, because strong conflicts cannot be serialized, and values having no conflict need not be serialized.

We will use the example in Figure 4.27 to illustrate the binding process. The distance matrix after applying resource constraint analysis (Lemma 3.2) is given in the same figure. It is used to construct the conflict graph in Figure 4.29. Because WCG is complete, the priority function will generate a choice of u and v which is as sensible as any other.

	A	B	C	D	E	F	G
A	0	0	1	3	3	4	6
B	-1	0	1	2	3	4	5
C	-2	-2	0	1	1	3	4
D	-3	-3	-2	0	0	1	3
E	-4	-3	-2	-1	0	1	2
F	-5	-5	-3	-2	-2	0	1
G	-6	-6	-5	-3	-3	-2	0

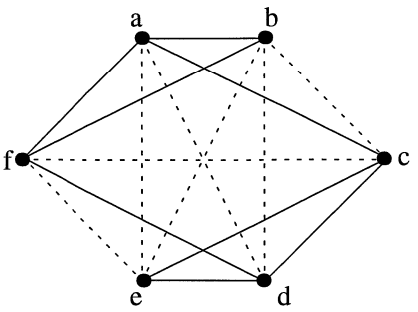


Figure 4.29 Distance matrix and conflict graph for Figure 4.27. A solid edge means strong overlap. A dashed edge means weak overlap. II=2

Suppose that values b and e are selected for serialization. The constraint analysis applies Lemma 4.4 on the distance B → E of 3 (k=1) to serialize C → E with a delay of 2 clock cycles, and on the distance E → B of -3 (k=-2) to serialize F → B with a delay of -4 clock cycles. The distance matrix and conflict graphs are updated, as indicated in Figure 4.30. Now nodes a, c, d, and f have the highest saturation number in WCG. These nodes all have the same saturation number in SCG, so values c and f are chosen arbitrarily. As a result, the constraint analyser applies Lemma 4.4 on the distance C → F of 3 (k=1) to serialize D → F with a delay of 2 clock cycles, and on the distance F → C of -3 (k=-2) to serialize G → C with a delay of -4 clock cycles. The effect on the distance matrix and conflict graph is given in Figure 4.31. Note that the mobility is reduced to zero, so the schedule is fixed. The strong conflict graph contains the clique a, c, d, e, indicating that at least four registers are required. The schedule and the two possible register bindings are given in Figure 4.32b). After the initial choice of serializing b and e (Figure 4.30), we might also have selected a-d instead of c-f. The resulting schedule and two possible register bindings are given in Figure 4.32a). For this particular example, the choice of which values to serialize is not very crucial.

4.4.3 Lifetime sequencing

After the selection of values for serialization it needs to be determined *how* these values are serialized. In Figure 4.33 for example, value v_j can be serialized inbetween u_{i-2} and u_{i-1}, or inbetween u_{i-1} and u_i, or u_i and u_{i+1}, etc. In our approach, we will first try the earliest possibility to schedule value v_j. If that yields infeasibility, then the 2^{nd} earliest possibility is tried, etc.

	A	B	C	D	E	F	G
A	0	0	1	3	3	4	6
B	-1	0	1	2	3	4	5
C	-2	**-1**	0	1	**2**	3	4
D	-3	-3	-2	0	0	1	3
E	-4	-3	-2	-1	0	1	2
F	-5	**-4**	-3	-2	**-1**	0	1
G	-6	-6	-5	-3	-3	-2	0

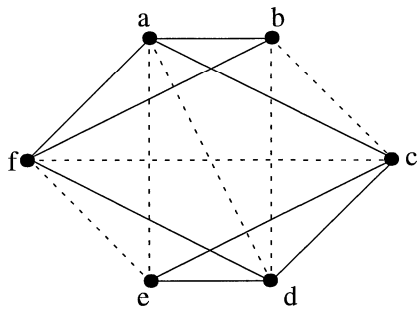


Figure 4.30 The distance matrix and conflict graph corresponding to the example in Figure 4.21 after serializing b-e. Bold faced numbers indicate updated values.

	A	B	C	D	E	F	G
A	0	**1**	**2**	3	**4**	**5**	6
B	-1	0	1	2	3	4	5
C	-2	-1	0	1	2	3	4
D	-3	**-2**	**-1**	0	**1**	**2**	3
E	-4	-3	-2	-1	0	1	2
F	-5	-4	-3	-2	-1	0	1
G	-6	**-5**	**-4**	-3	**-2**	**-1**	0

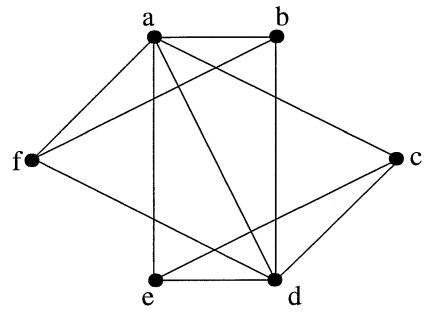


Figure 4.31 The distance matrix and conflict graph corresponding to the example in Figure 4.27 after serializing b-e and c-f. Bold faced numbers indicate updated values.

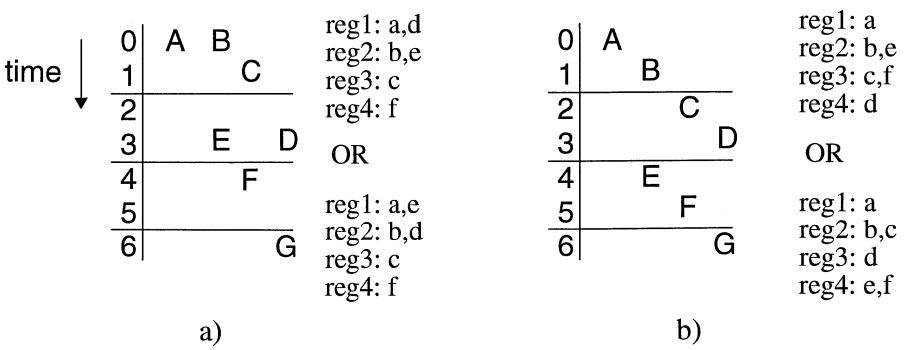


Figure 4.32 The only 2 feasible schedules for Figure 4.28, and corresponding register bindings.

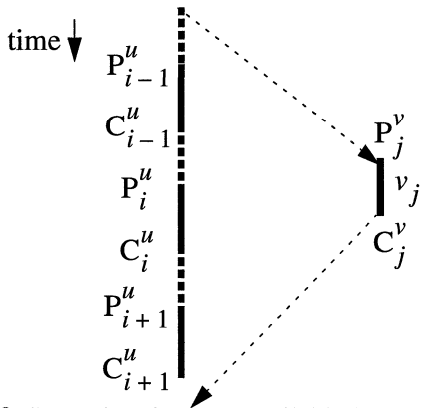


Figure 4.33 Several options are available for sequencing u and v

4.5 Experimental results

In this section, we present the experimental results [Mesm99b] obtained with the proposed method implemented in the FACTS environment [Eijk00]. All experiments are run on a machine with a 233 MHz Pentium II processor.

Because the proposed techniques are especially intended to handle inner loops of DSP algorithms under tight timing constraints, we use the inner loop of a fast fourier transform (FFT) algorithm, a fast discrete cosine transform (FDCT) algorithm and a Loeffler algorithm that performs an 8-point 1-dimensional inverse discrete cosine transform. Each example is mapped to a relatively simple architecture in which each resource type has a dedicated register file. The characteristics of the various examples are shown in Table 4.2. The latency shown in the third column is the minimum latency obtainable for that constraint set. The table also shows the results obtained by a branch-and-bound scheduler [Timm94] followed by a register binder based on exact minimum graph colouring. These results are used as a reference point for the method proposed in this paper.

Table 4.2 Examples and reference results

example	|V|, | E_d |	II/latency	time(s)	RF sizes
fft256	30,43	4/13	0.1	3,3,1,2
fdct	42,43	18/18	0.1	9,4
loef	56,57	26/28	0.4	8,4,10

To evaluate the proposed method, we have applied it to the examples of Table 4.2 with various register file capacity constraints. The branch-and-bound scheduler is used to

complete the partial schedule resulting from value lifetime serialization. The results are shown in Table 4.3.

For each problem instance, Table 4.3 lists the register file capacity constraints, the run time (including the time needed for scheduling), and the impact of serialization on the mobility of the operations (the numbers before respectively after the arrow denote the mobility before and after serialization).

The experimental results for the example fft256 clearly show that the proposed method is steered by the individual register file constraints. Despite the presence of tight timing and resource constraints, the approach is able to generate many different schedules dependent on the settings of the individual capacity constraints. So if a certain register file is potentially overloaded, the method will reduce this load, possibly by exploiting the opportunities offered by other register files. We consider this feature very important for handling heterogeneous register file architectures. By integrating the phases of scheduling and register binding, our method is also able to significantly reduce the register occupation compared to an approach that performs register binding a posteriori. For the example 'loef', this results in a reduction of the total number of required registers from 22 to 15 registers.

Table 4.3 Results of proposed method

example	RF caps	time (s)	mobility
fft256	1, 4, 1, 2	0.1	0.7 → 0.3
	2, 2, 1, 2	0.4	2.3 → 0.0
	2, 3, 1, 1	0.8	2.1 → 0.0
	3, 2, 1, 1	0.9	2.1 → 0.0
	4, 1, 1, 2	0.1	0.7 → 0.4
fdct	9, 4	2.3	9.5 → 4.0
	6, 4	2.7	9.5 → 2.0
	8, 2	0.9	9.5 → 1.4
loef	8, 4, 10	3.5	14.4 → 3.1
	4, 3, 8	4.9	14.4 → 1.0

Chapter 5 Storage Models for Reduced Instruction Width

In the introduction of this thesis it was described how important code size and, closely related, instruction width are. This importance is even stronger for processors with instruction memory embedded on the same chip. The observation was made that often more than half of the instruction bits is used for addressing registers. This suggests that an important decrease in code size can be obtained by questioning the need for the flexibility offered by conventional register addressing mechanisms, and consequently, limiting the range of addressing possibilities, albeit artificially. In this chapter we will consistently apply the same strategy used in this thesis: modelling some constraints in terms of serializing alternatives and rules for choosing those alternatives that keep the solution in the feasible region. In this chapter the "constraints" we wish to model are due to a restricted mechanism for register addressing with the objective of reducing code size. The techniques presented here are limited to serializing operations such that designated values can share the same address. Assigning the values to addresses (similar to assigning values to registers) is considered outside the scope of this thesis and is a topic of ongoing research [Alba00].

The rest of this section is organized as follows. In Section 5.1 we take a FIFO as a storage model and introduce the line of reasoning to extract essential serializing rules. In Section 5.2 a STACK is used as storage medium. In Section 5.3 a new hybrid of a FIFO and a STACK called FILIFO is analysed. Section 5.4 generalizes the analyses of the first three sections to the case that loop pipelining is applied. Section 5.5 discusses some practical issues. A case study is performed in Section 5.6 to show the effectiveness of some of the storage models.

5.1 FIFOs

The FIFO (first in first out) model is illustrated in Figure 5.1. Values are written into the FIFO at the top and read from the bottom. Since we are merely interested in the way the FIFO model affects the ordering of operations accessing the FIFO, we abstract from any implementations of the FIFO model itself. We also assume that the capacity of the FIFO is sufficient for all practical applications. The name "FIFO" is due to the observation that a value first written is consequently first read. Characteristic of this model is that values are not overwritten but rather shifted down. This implies that contrary to randomly accessible registers (Section 2.4), value lifetimes are not limited to the initiation interval in case of loop pipelining, because older values are not overwritten. As a

result, multiple iterations of the same value (u_i, u_{i+1}, u_{i+2}, etc.) can be simultaneously present in the FIFO. So although a FIFO has some restrictions, it also provides a feature that randomly addressable register files do not have.

5.1.1 Analysis of FIFO access ordering

Our analysis is similar to the one in chapter 4. Only conflicts between two values are considered. When conflicts are resolved between each pair of values, the whole set of values is guaranteed to 'fit' in the same storage unit (FIFO). Basically, two overlapping values u and v can relate to each other in four different ways as depicted in Figure 5.2.

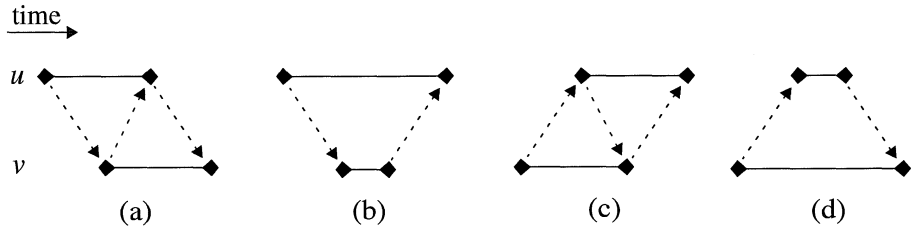


Figure 5.2 Potentially overlapping values u and v

In this figure, a horizontally drawn line represents the lifetime of a value produced by its left side node, and consumed by its right side node. The relation between two values is characterized by the order of their successive write and read accesses. This order is denoted in Figure 5.2 by the diagonal dotted arrows. For example, relation (a) is characterized by three orders: $P^u \rightarrow P^v$, $P^v \rightarrow C^u$, and $C^u \rightarrow C^v$, where P denotes the producer and C denotes the consumer of the corresponding value. When values u and v are addressed in the same way, situations (a) and (c) are equivalent as are (b) and (d). Therefore we restrict our analysis to situations (a) and (b) without loss of generality. Note that situation (a) is feasible for our FIFO model, but situation (b) contradicts the first-in first-out ordering. Situation (b) is characterized by two partial orders: $P^u \rightarrow P^v$ and $C^v \rightarrow C^u$. From the infeasibility of this situation we extract the rule that if either one of these two orders occurs, the other one cannot. Thus the following two lemmas are derived.

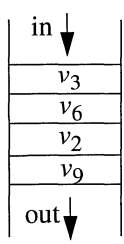


Figure 5.1 FIFO model

Lemma 5.1: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FIFO. If $d(P^u, P^v) \geq 0$ we can add a sequence precedence edge (C^u, C^v) with weight 1 without excluding any feasible schedules. ❑

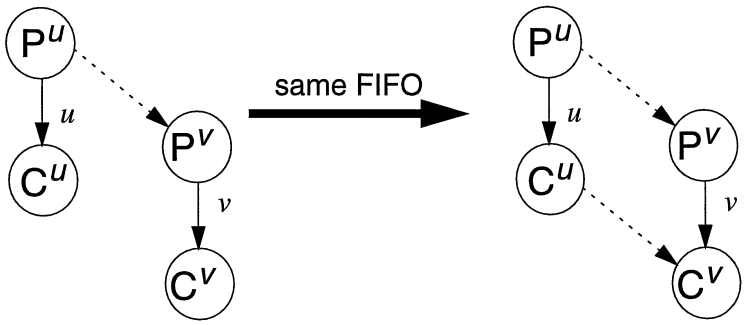


Figure 5.3 Lemma 5.1 for serializing value lifetimes

Lemma 5.1 is illustrated in Figure 5.3. This lemma restricts the possibilities to situation (a) and the situation that the lifetime of value u completely precedes that of value v.

Lemma 5.2: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FIFO. If $d(C^u, C^v) \geq 0$ we can add a sequence precedence edge (P^u, P^v) with weight 1 without excluding any feasible schedules. ❑

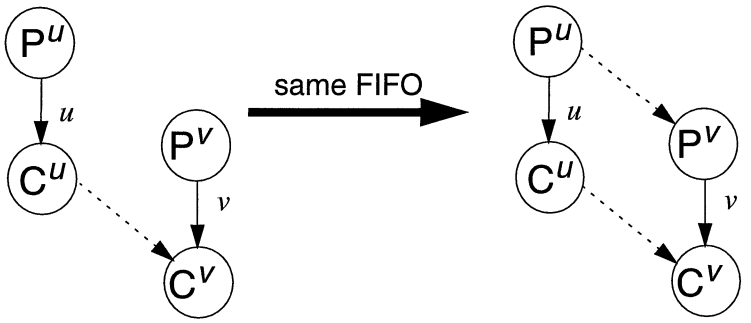


Figure 5.4 Lemma 5.2 for serializing value lifetimes

Lemma 5.2 is illustrated in Figure 5.4. This lemma also restricts the possibilities to situation (a) and the situation where value u completely precedes value v.

5.2 STACKs

The STACK model is illustrated in Figure 5.5. Values are both written to and read from the top of the STACK. A STACK is therefore also called a LIFO (last in first out). Again we assume that the capacity of the STACK is sufficient for all practical applications.

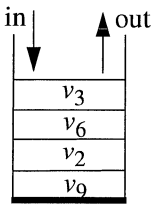


Figure 5.5 STACK model

Because two values can only be read using the same read pointer we restrict the analysis to situations (a) and (b) in Figure 5.6.

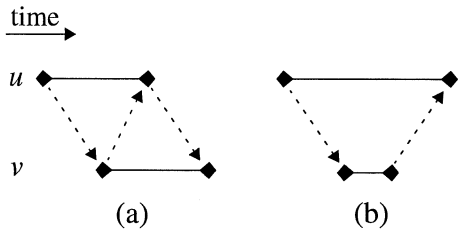


Figure 5.6 Potentially overlapping values u and v

With respect to feasible overlapping lifetimes, the STACK case is the reverse of the FIFO case. Situation (b) is a feasible on the STACK case, but situation (a) is not. Situation (a) is characterized by three orders: $P^u \rightarrow P^v$, $P^v \rightarrow C^u$, and $C^u \rightarrow C^v$, where P denotes the producer and C denotes the consumer of the corresponding value. Infeasibility of situation (a) thus implies that the combination of these three orderings is not allowed. So whenever either two of these orderings are present, the third one has to be negated. This observation yields the following three lemmas.

Lemma 5.3: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same STACK. If $d(P^u, P^v) \geq 0$ and $d(P^v, C^u) \geq 0$ we can add a sequence precedence edge (C^v, C^u) with weight 1 without excluding any feasible schedules. ❑

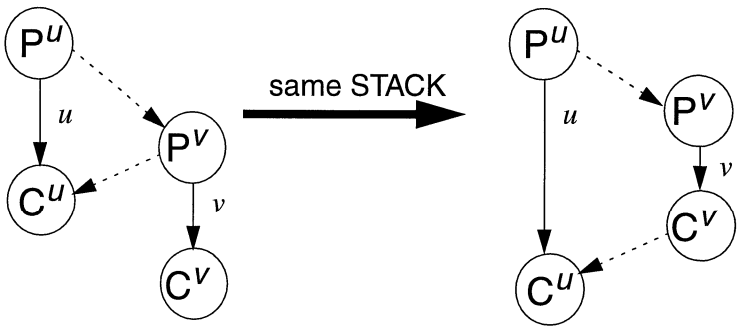


Figure 5.7 Lemma 5.3 for serializing value lifetimes

Lemma 5.3 is illustrated in Figure 5.7. This lemma limits the possibilities to situation (b) in Figure 5.6.

Lemma 5.4: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same STACK. If $d(P^v, C^u) \geq 0$ and $d(C^u, C^v) \geq 0$ we can add a sequence precedence edge (P^v, P^u) with weight 1 without excluding any feasible schedules. ❑

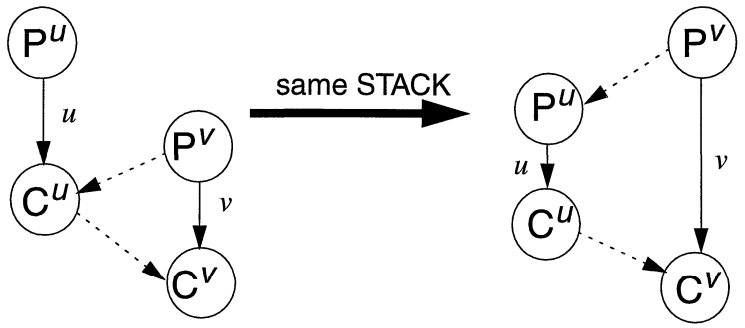


Figure 5.8 Lemma 5.4 for serializing value lifetimes

Lemma 5.4 is illustrated in Figure 5.8. This lemma also limits the possibilities to situation (b) in Figure 5.6, where values u and v are swapped.

Lemma 5.5: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same STACK. If $d(P^u, P^v) \geq 0$ and $d(C^u, C^v) \geq 0$ we can add a sequence precedence edge (C^u, P^v) with weight 0 without excluding any feasible schedules. ❑

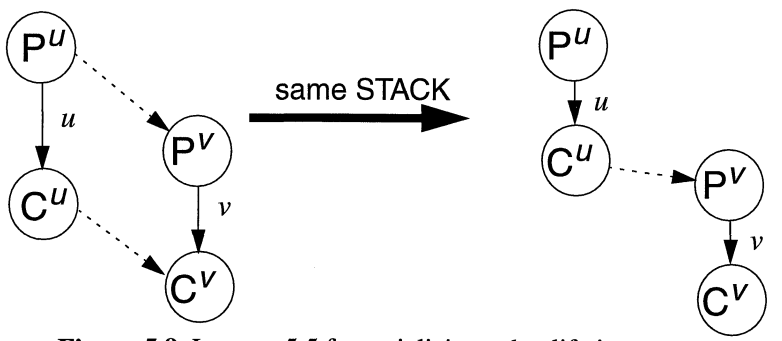


Figure 5.9 Lemma 5.5 for serializing value lifetimes

Lemma 5.5 is illustrated in Figure 5.9. This lemma limits the possibilities to the situation where value u completely precedes value v.

5.3 FILIFO, a hybrid between FIFO and STACK

The FILIFO (first in last in first out) model is designed as a storage unit that provides more flexibility than both a STACK and a FIFO, without increasing the number of control bits too much (at most one additional bit). It is a hybrid of a STACK and a FIFO because two values relate to each other in either a FIFO-like manner (Figure 5.12 a), in a STACK-like manner (Figure 5.12 b), or they simply do not overlap.

The FILIFO is illustrated in Figure 5.10. Values are written into the FILIFO at the top and can be read either from the top or the bottom. When a value is read from the top, we say it is s-read, because it resembles the behaviour of a STACK. When a value is read from the bottom, we say it is f-read, because it resembles the behaviour of a FIFO. The determination whether a value is s-read or f-read we will call *read pointer assignment.*

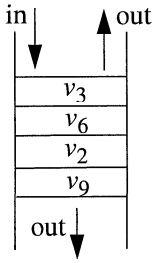


Figure 5.10 FILIFO model

5.3.1 Analysis of FILIFO access ordering

First we observe that the way two value lifetimes are allowed to relate to each other depends on whether these values are s-read or f-read. For example: if both values are f-read they relate to each other as two values in a FIFO (Section 5.1), but if they are both s-read they relate to each other as two values on a STACK (Section 5.2). The reverse is also true. Whether two values can be s-read or f-read depends on the way the corresponding value lifetimes relate to each other. For example, if values u and v relate to each other as in Figure 5.12 (a), value u can only be f-read. We conclude that read pointer assignment and scheduling are interrelated problems, just as register binding and scheduling are interrelated problems (chapter 4). There is however an important difference between register binding and 'read pointer assignment'. In chapter 4 we made explicit register binding decisions (during scheduling) to prevent the scheduler from violating the capacity constraints of the register files. Explicit 'read pointer assignment' decisions are not necessary because we have no constraints similar to the capacity constraints of register files. Instead we allow the scheduler to use the available schedule freedom, and the read pointer assignment will follow implicitly from the schedule. This can only work under the following conditions:

- The serializing (and assignment) rules should describe sufficient conditions to exclude all infeasible situations. The rules thus can serve as a verification. If the completed schedule satisfies these rules, all values assigned to the same FILIFO are

guaranteed to fit in there. This requires a thorough (complete) analysis of all infeasible situations. Furthermore, we should take care that the rules provide *necessary* conditions only.

- As we have seen in the example above, some access orderings imply a certain read pointer assignment and similarly, some (partial) read pointer assignments imply a certain access ordering. In order to cover all infeasible situations, we need to exhaustively consider all combinations of pointer assignments and access orderings.

We use the scheme depicted in Figure 5.11 to cope with the interaction between pointer assignment and scheduling. Because the application of the sequencing rules works accumulative (new precedences and pointer assignments trigger the rules, thus causing new precedences and pointer assignments, etc.), constraint analysis is repeated until no further serializations or pointer assignments are obtained.

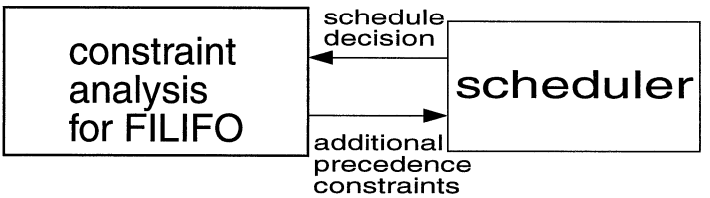


Figure 5.11 Constraint analysis runs along with the scheduler

---

Algorithm 5.1 (constraint analysis for FILIFO).

```
for all v in Y assigned to storage unit SU of type FILIFO
  for all u<>v in Y assigned to SU
    for all lem in Lemmas on FILIFO
      check Lemma lem on v and u and add corresponding sequence edge
```

First we try to find situations where infeasibility is implied by a single value v. Only one situation implies infeasibility: the case where value v is s-read and the lifetime exceeds the initiation interval II. So if either one of these two conditions is true, the other one cannot. This leads to the following two lemmas:

Lemma 5.6: Let value v, produced by operation P^v and consumed by C^v reside in a FILIFO. If $d(P^v, C^v) > II$ we can assign C^v to the f-read pointer without excluding any feasible schedules. ❑

Lemma 5.7: Let value v, produced by operation P^v and consumed by C^v reside in a FILIFO. If C^v is assigned to the s-read pointer, the lifetime of v cannot exceed II, so we can add a sequence precedence edge (C^v, P^v) with weight -II without excluding any feasible schedules. ❑

The analysis for two values will be more complicated. A number of situations should be distinguished with respect to the read pointer assignment. A value can be either s-read (s), f-read (f), or no decision is yet made (x). There are six combinations of accesses on two values *u*/*v*: x/x, x/f, x/s, f/f, s/f, and s/s. These situations are analysed as follows.

***u*/*v* = f/f**: Both values are accessed by the f-read pointer. The corresponding producers and consumers have to satisfy the rules associated with a FIFO, so Lemma 5.1 and Lemma 5.2 apply.

***u*/*v* = s/s**: Both values are accessed by the s-read pointer. The corresponding producers and consumers have to satisfy the rules associated with a STACK, so Lemma 5.3 to Lemma 5.5 apply.

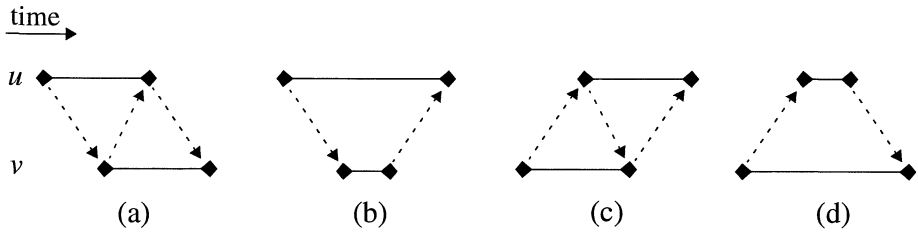


Figure 5.12 Potentially overlapping values *u* and *v*

***u*/*v* = s/f**: Value *u* is s-read and *v* is f-read. Two situations, (a) and (b) in Figure 5.12, are infeasible for the same reason. Value *u* is put on the STACK, and before it is consumed, value *v* is put on top of it; now *u* cannot be read until *v* is gone, and vice versa. So infeasibility is implied by the combination of $P^u \rightarrow P^v$ and $P^v \rightarrow C^u$. From the infeasibility of this situation we extract the rule that if either one of the two orders occurs, the other one cannot. Thus the following two lemmas are derived.

Lemma 5.8: Let value *u*, produced by operation P^u and consumed by C^u, and value *v*, produced by operation P^v and consumed by C^v, reside in the same FILIFO. Let *u* be s-read and *v* be f-read. If $d(P^u, P^v) \geq 0$ we can add a sequence precedence edge (C^u, P^v) with weight 0 without excluding any feasible schedules. ❑

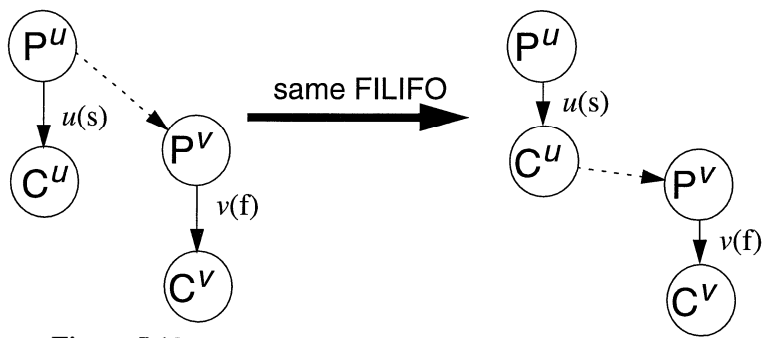


Figure 5.13 Lemma 5.8 for serializing value lifetimes

Lemma 5.8 is illustrated in Figure 5.13. This lemma restricts the possibilities to the situation that the lifetime of value u completely precedes that of value v.

Lemma 5.9: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FILIFO. Let u be s-read and v be f-read. If $d(P^v, C^u) \geq 0$ we can add a sequence precedence edge (P^v, P^u) with weight 1 without excluding any feasible schedules.

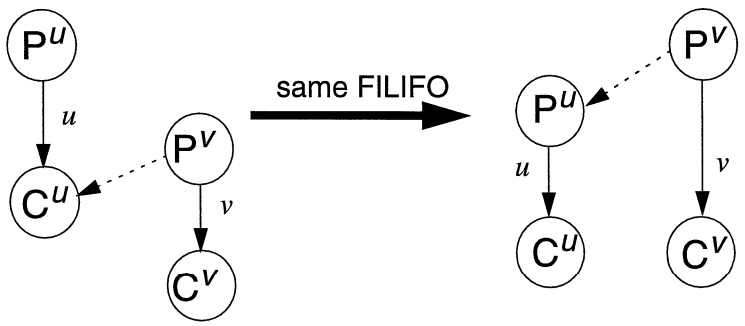


Figure 5.14 Lemma 5.9 for serializing value lifetimes

Lemma 5.9 is illustrated in Figure 5.14. This lemma restricts the possibilities to situations (c) and (d) in Figure 5.12 and the situation where value v precedes value u.

***u/v* = x/x**: Neither value has been assigned a read port. Remember that two value lifetimes relate to each other in either a FIFO-like or a STACK-like manner (or both). This means that no serializing constraints can be derived from any combination of value lifetimes. But some relations do enforce the use of a certain read pointer. In Figure 5.12 (a) value u can only be f-read, and in Figure 5.12 (b) value v can only be s-read.

Lemma 5.10: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FILIFO. If $d(P^u, P^v) \geq 0$, $d(P^v, C^u) \geq 0$ and $d(C^u, C^v) \geq 0$, then C^u can be f-read without excluding any feasible schedules. ❑

Lemma 5.11: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FILIFO. If $d(P^u, P^v) \geq 0$ and $d(C^v, C^u) \geq 0$, then C^v can be s-read without excluding any feasible schedules. ❑

***u/v* = x/f**: Value v is f-read, and the read access on value u is not yet assigned. The only infeasible situation is Figure 5.12 (b). Because this was also the only infeasible situation in the ***u/v* = f/f** case, we refer to Lemma 5.1 and Lemma 5.2. Some situations do enforce a read pointer assignment, but these situations are already covered by preceding lemmas: Situation (d) in Figure 5.12 imposes value u to be s-read, which is already implied by Lemma 5.11. In the case that both: $P^u \rightarrow P^v$ and $P^v \rightarrow C^u$, it is necessary

that $C^u \rightarrow C^v$, and that value u be f-read. This situation is covered by a combination of Lemma 5.1 and Lemma 5.10.

***u/v* = s/x**: Value u is s-read, and the read access on value v is not yet assigned to a read pointer. The only infeasible situation is Figure 5.12 (a). Because this was also the only infeasible situation in the ***u/v* = s/s** case, we refer to Lemma 5.3 to Lemma 5.5. Some situations do enforce a pointer assignment, but these situations are already covered by preceding lemmas: Situation (c) in Figure 5.12 imposes value v to be f-read, which is already implied by Lemma 5.10. In the case that both $P^u \rightarrow P^v$ and $P^v \rightarrow C^u$, it is necessary that value v be s-read. This situation is covered by a combination of Lemma 5.3 and Lemma 5.11.

5.4 Loop pipelining

When a pipelined schedule is desired, we not only have to take care that u and v fit in the same FILIFO (or FIFO or STACK), but also that u_i and v_j fit in the same FILIFO. This is illustrated in Figure 5.15. In this figure, a solid line segment represents a value lifetime, a dotted line segment represents an 'empty' time slot. Values u_i and v_i are serialized and therefore fit in the same STACK, but values u_{i+1} and v_i behave in a typical FIFO-manner and do not fit together in a STACK. We conclude that we have to broaden our scope beyond loop boundaries. We have already done this in Section 4.1.2 for randomly addressable register files, and we will do the same for FIFOs, STACKs and FILIFOs.

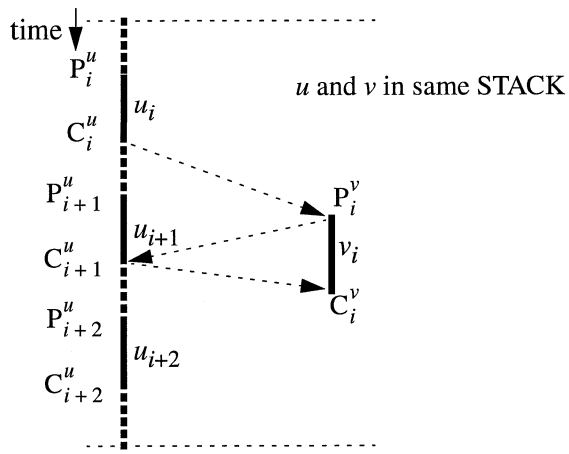


Figure 5.15 Serializing within the same iteration is not sufficient for pipelining

In Section 2.2 we showed the equivalence between the relation $C_i \rightarrow P_{i+k}$ and the relation $C \rightarrow P$ with time delay $-k \cdot II$. This equivalence has been used in Section 4.1.2 to generalize the lemmas from Section 4.1.1. We derive a generalization of Lemma 5.1 in a similar way: First, we use the equivalence to translate the timing delay of $-k \cdot II$ in Figure 5.16 (a) to the iteration indices in Figure 5.16 (b). The additional sequence edge

in Figure 5.16 (c) is obtained from (b) by directly applying Lemma 5.1. In Figure 5.16 (d) the iteration indices have been translated back to the timing domain and a timing relation from C^u to C^v (with delay $-k \cdot II$) is the result. Lemma 5.1 is now generalized to Lemma 5.12.

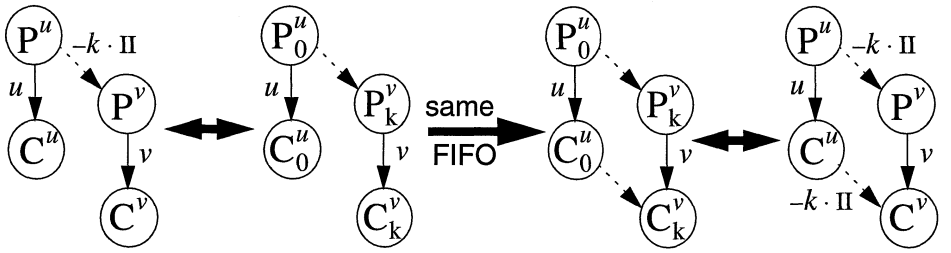


Figure 5.16 Generalization of Lemma 5.1

Lemma 5.12: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FIFO. If $d(P^u, P^v) \geq -k \cdot II$ we can add a sequence precedence edge (C^u, C^v) with weight $1 - k \cdot II$ without excluding any feasible schedules. ❑

The generalization of Lemma 5.2 is now straightforward:

Lemma 5.13: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FIFO. If $d(C^u, C^v) \geq -k \cdot II$ we can add a sequence precedence edge (P^u, P^v) with weight $1 - k \cdot II$ without excluding any feasible schedules. ❑

The STACK lemmas are a little harder to generalize because a sequence edge is added as a result of *two* paths instead of one (which is the case with a FIFO). These paths are indicated in Figure 5.18 (a); one has length $\geq -l \cdot II$, the other $\geq -k \cdot II$. These paths are not completely independent; as a result of $LT(u) \leq II$ and the path $P^u \rightarrow P^v \rightarrow C^u$ of length $-(k + l) \cdot II$, which lower bounds $LT(u)$, it is derived that $k + l \geq -1$. But as long as this is the case, k and l may be arbitrary integers. Because of the relative independence of these two parameters, we can use the equivalence between timing and indexes in two different ways. In Figure 5.18, we observe the conflict between u_0 and v_l, and in Figure 5.17 the conflict between u_k and v_l is analysed. In Figure 5.18 (b) Lemma 5.3 applies if $d \geq 0$, so if $k + l \leq 0$, which finally results in a sequence edge $C^v \rightarrow C^u$ with length $l \cdot II$. In Figure 5.17 (b) Lemma 5.3 applies under the same condition ($k + l \leq 0$), which finally results in a sequence edge $C^v \rightarrow C^u$ with length $-k \cdot II$. Since both sequence edges may be added under the same condition, the generalization of Lemma 5.3 results in Lemma 5.14:

Lemma 5.14: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same STACK. If

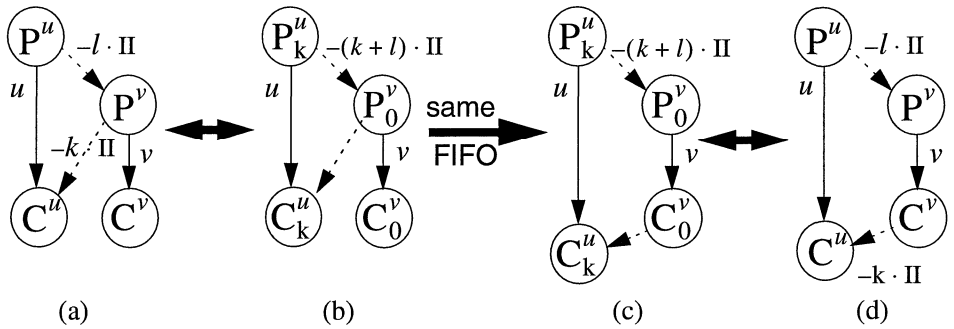


Figure 5.17 First generalization of Lemma 5.3

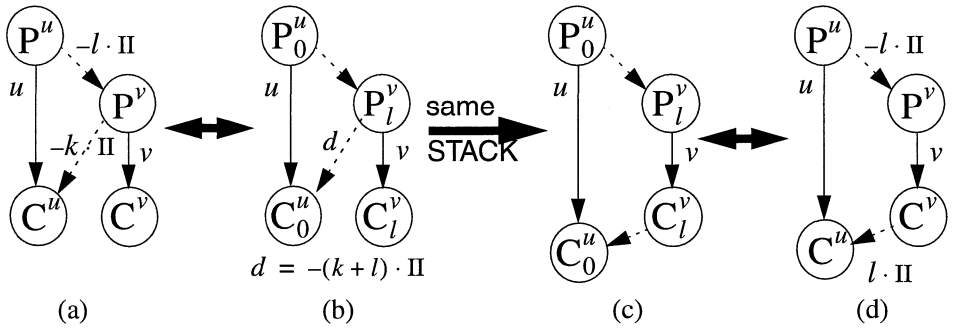


Figure 5.18 Second generalization of Lemma 5.3

$d(P^u, P^v) \geq -l \cdot II$ $d(P^v, C^u) \geq -k \cdot II$ and $k + l \leq 0$ we can add a sequence edge (C^v, C^u) with weight $\max\{l \cdot II, -k \cdot II\}$ without excluding any feasible schedules. ❑

Lemma 5.4 is generalized to Lemma 5.15 following the same line of reasoning.

Lemma 5.15: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same STACK. If $d(P^v, C^u) \geq -k \cdot II$ $d(C^u, C^v) \geq -l \cdot II$ and $k + l \leq 0$ we can add a sequence edge (P^v, P^u) with weight $\max\{l \cdot II, -k \cdot II\}$ without excluding any feasible schedules. ❑

Deriving Lemma 5.16 from Lemma 5.5 also follows this line of reasoning but the weight of the resulting sequence edge is different.

Lemma 5.16: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same STACK. If $d(P^u, P^v) \geq -k \cdot II$ and $d(C^u, C^v) \geq -l \cdot II$ we can add a sequence edge (C^u, P^v) with weight $\min\{-l \cdot II, -k \cdot II\}$ without excluding any feasible schedules. ❑

With respect to the u/v = s/f case for FILIFOs, Lemma 5.8 and Lemma 5.9 are generalized to Lemma 5.17 and Lemma 5.18 respectively.

Lemma 5.17: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FILIFO. Let u be s-read and v be f-read. If $d(P^u, P^v) \geq -l \cdot II$ we can add a sequence precedence edge (C^u, P^v) with weight $-l \cdot II$ without excluding any feasible schedules. ❑

Lemma 5.18: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FILIFO. Let u be s-read and v be f-read. If $d(P^v, C^u) \geq -l \cdot II$ we can add a sequence precedence edge (P^v, P^u) with weight $1 - l \cdot II$ without excluding any feasible schedules. ❑

Lemma 5.10 and Lemma 5.11 for deriving a read port are generalized to Lemma 5.19 and Lemma 5.20 respectively.

Lemma 5.19: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FILIFO. If $d(P^u, P^v) \geq -k \cdot II$, $d(P^v, C^u) \geq -l \cdot II$, and $d(C^u, C^v) \geq -m \cdot II$, where $l \geq k, m$, then C^u can be f-read without excluding any feasible schedules. ❑

Lemma 5.20: Let value u, produced by operation P^u and consumed by C^u, and value v, produced by operation P^v and consumed by C^v, reside in the same FILIFO. If $d(P_u, P_v) \geq -k \cdot II$ and $d(C^v, C^u) \geq l \cdot II$, where $l \geq k$, then C^v can be s-read without excluding any feasible schedules. ❑

We have generalized the rules from the previous section so that we are now able to analyse the constraints generated from the different storage models in the context of loop pipelining. The next section considers some practical issues involved in applying the proposed analysis to real-life applications and a broad range of architectures.

5.5 Some practical issues

The previous sections laid the foundations for coping with unconventional storage models. In this section we make some more generalizations that allow us to apply the analysis rules to a broader range of practical situations. These situations are

- multiple consumers of the same value
- using different RF models in the same architecture

5.5.1 Multiple consumers

When a value in a storage unit is allowed to be consumed more than once (if a read access can be non-destructive), an extensive number of different access orderings become possible. For example, suppose that in some Basic Block, value v is consumed three times, by operations C_1^v, C_2^v, and C_3^v respectively. It is allowed that v is written in a FILIFO and s-read by C_1^v, that subsequently value u is pushed on top of v and popped,

v is s-read again, a number of values, among which w, are pushed on top, and v is f-read destructively. The way v relates to the other values depends on which consumer of v is considered. When considering C_2^v, v/w relate as s/s, but when considering C_3^v, v/w relate as f/s in the example. The way to cope with this ambiguity is now straightforward: instead of considering a value v we consider a *production-consumption pair*, or P-C pair for short. Since a value in our terminology can only be produced once, Algorithm 5.1 is extended to Algorithm 5.2 in order to cope with multiple consumers of the same value.

Algorithm 5.2 (constraint analysis for FILIFO).

```
for all v in Y assigned to storage unit SU of type FILIFO
  for all u<>v in Y assigned to SU
    for all C^v and C^u
      for all lem in Lemmas on FILIFO
        check Lemma lem on P^v, P^u, C^v, and C^u, and add corresponding sequence edge
```

5.5.2 Architectures with mixed storage types

Suppose we have an architecture containing a variety of storage models, so one file is a FIFO, another a STACK or a FILIFO, and some files are randomly addressable. Let type(SU) denote the type of storage unit SU. Algorithm 5.2 is extended to Algorithm 5.3 in order to cope with multiple storage types. Note that register files are not considered in this scheme; for register files an explicit register binding decision has to be taken before the constraint analyser is able to reduce the search space accordingly, see Figure 4.20.

Algorithm 5.3 (constraint analysis for mixed storage types)

```
for all storage units SU of type register, FIFO, STACK, or filifo
  for all v in Y assigned to SU
    for all u<>v in Y assigned to SU
      for all C^v and C^u
        for all lem in Lemmas on type(SU)
          check Lemma lem on P^v, P^u, C^v, and C^u, and add corresponding sequence edge
```

5.6 Case study

In order to understand how the techniques treated in this chapter can be applied, let us spend a few words on the status of the work. Chapter 4 started with a discussion on lifetime serialization for a given register binding. These 'basic' techniques were subsequently used in sections 4.2 and 4.4 to generate a register binding. The status of the techniques presented here are 'basic' in the same sense. The techniques work for values

that have been assigned to a FIFO, STACK, or filifo. However, FIFO or STACK assignment is a topic of ongoing research [Alba00] considered outside the scope of this thesis. In this section we will demonstrate in a high-level synthesis context the effectiveness of FIFO addressing and the tool support for exploring the possibilities offered by these addressing schemes.

For this purpose we will use the example shown in Figure 5.19. It represents the inner loop of an in-place FFT algorithm. The label in-place refers to the fact that the results of the computation are written back to the same address where the samples are fetched (in order to save memory). This explains the long data edges of the addresses a0R1 to a3R2: these addresses are used both for fetching samples at the beginning of the computation, and for storing the results at the end of the computation. The operations required for fetching the multiplication coefficients are omitted. The purpose of the exercise is to obtain a pipelined schedule with an initiation interval of four clock cycles thereby minimizing the number of addresses required for executing the schedule.

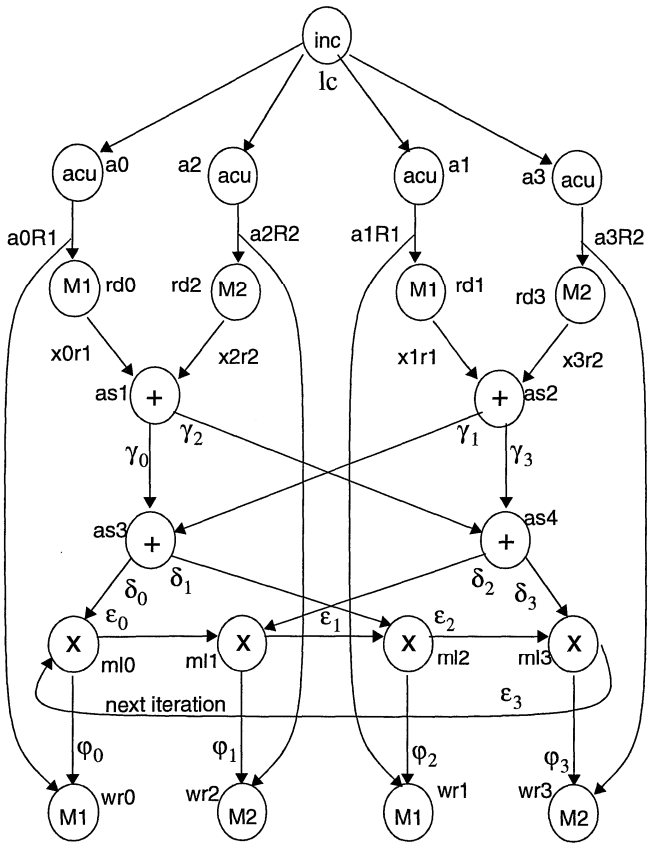


Figure 5.19 DFG of an FFT inner loop

The data-path architecture is depicted in Figure 5.20. The smallest set of functional resources required to obtain this goal consists of an address computation unit (acu), a rd/wr port on memory M1, a rd/wr port on memory M2, a complex adder, a multiplier, and an incrementer for the loop counter. For reasons of convenience we assume an architecture with a single storage file consisting of registers, FIFOs, and STACKs. All functional units have rd/wr access to this file. The instruction encoding is also depicted in Figure 5.20. The opcodes of each functional unit is assumed to take three bits. For each read (write) port, the address of the source (destination) operand in the storage file is encoded. The number of control bits for addressing the storage file equals $\lceil \log a \rceil$, where a represents the number of addresses in the file. For the default architecture the total number of control bits therefore equals $6 \cdot 3 + (10 + 8) \cdot \lceil \log a \rceil =. 18 + 18 \cdot \lceil \log a \rceil$

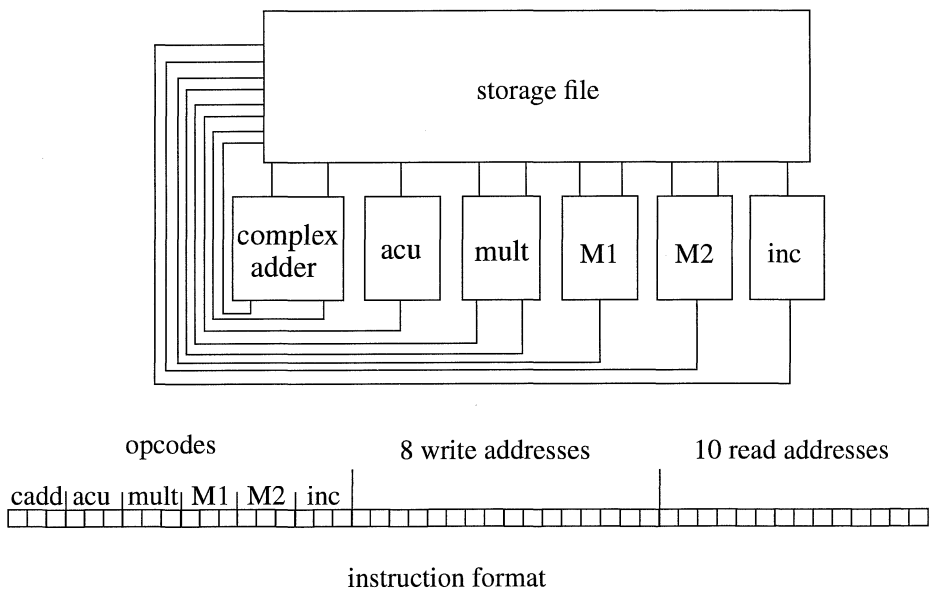


Figure 5.20 Default VLIW architecture and instruction format for mapping the DFG in Figure 5.19

5.6.1 Implementation with randomly addressable registers

This design was originally implemented with the usual randomly addressable registers. Because the lifetimes of the addresses a0R1 to a3R2 necessarily exceed the intended initiation interval of four clock cycles, the designer manually inserted so called *rename* or *move* operations to move a value to another register. Each of the addresses is renamed twice to enable II=4, as indicated in Figure 5.21 by the operations alu10 to alu 23, executed by functional units b1 and b2. Therefore, the default architecture in Figure 5.20 needs to be extended with two functional units b1 and b2, and two read and write ports on the storage file. Therefore the total number of control bits equals $8 \cdot 3 + (12 + 10) \cdot \lceil \log a \rceil = 24 + 22 \cdot \lceil \log a \rceil$.

With this constraint set, the minimum latency of the schedule according to FACTS equals 13 clock cycles. The initial mobility (based on asap-alap) equals 4.56 clock cycles per operation, and is reduced to 2.29 clock cycles after constraint analysis (run-time: 0.1 s). The least amount of registers, for which FACTS is able to find a schedule is 17. The mobility was thereby decreased to 0 in 1.07 seconds run time. Since there are 8 more global variables alive during execution of the DFG which are not represented in Figure 5.21, the total number of control bits equals $24 + 22 \cdot \lceil \log 25 \rceil = 134$.

5.6.2 Implementation with FIFOs and registers

In this section we will add FIFOs to the architecture in order to decrease the number of addresses.The largest gain is obtained by placing the values a0R1-a3R2 in a FIFO. Because these values are read twice and we use a FIFO with destructive read, it is decided that each address is written to both a FIFO and a register. This requires an addi-

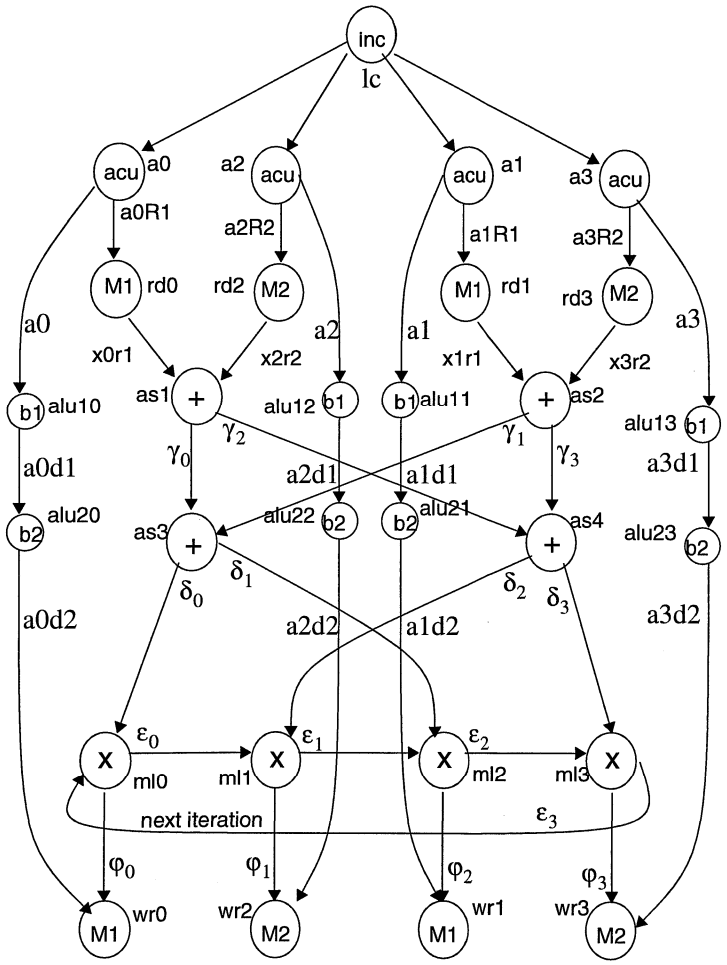


Figure 5.21 Transformed DFG for implementation with registers

tional write port on the storage file. The amount of control bits then equals $18 + 19 \cdot \lceil \log a \rceil$

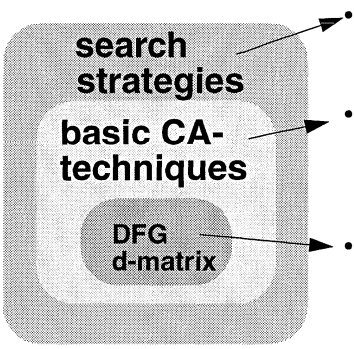
Initially, the mobility equals 3.31 clock cycles per operation. The values a0R1-a3R2 can all be put in one FIFO, but the corresponding constraints reduce the mobility to zero according to FACTS (run-time: 0.1 s). Nine more registers are required to store the remaining values, so the total amount of addresses in the storage file equals ten. Two more FIFOs are added, and iteratively the values with the longest lifetimes are transferred to a FIFO. In this way, a schedule is obtained with three FIFOs and four registers, for a total of seven addresses. Eight more registers are required for the global variables, so the total number of control bits equals $18 + 19 \cdot \lceil \log 15 \rceil = 94$, which is 30 bit less than the situation where only registers were used.

Chapter 6 Conclusions

In this thesis an approach for DSP code generation is presented based on *constraint analysis*. This technique is inspired by the observation that traditional code generation methods require too much help and expertise from a designer to satisfy the combination of timing, resource, and storage constraints encountered when mapping DSP applications onto embedded processors. Using constraint analysis, a scheduler is *guided* rather than *hampered* by these constraints: By using the constraints to prune the schedule search space, the scheduler is often prevented from making a decision that inevitably violates one or more constraints. Some of these techniques have been integrated in the research code generation tool FACTS [Eijk00], [Mesm01] together with some techniques following the same philosophy like execution interval analysis [Timm95] and symmetry analysis [Eijk99].

We have considered the problem of phase coupling: the problem that decisions taken in one phase of the code generation process effect the freedom of movement in the other phases. We have argued that this problem 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 feasible solution. In our approach, the problem of phase coupling is addressed by letting all analyses work on a single unified representation of the schedule search space, the *distance matrix*, which is the core layer in the FACTS hierarchy indicated in Figure 6.1. This is an effective representation because it administrates relative timing, which is important for solving the scheduling problem. Any information regarding relative timing can be directly expressed in the distance matrix, such as schedule decisions, precedences, etc. The results of the analyses discussed in this thesis are also expressed in terms of sequence constraints and can therefore be combined in the distance matrix simply by computing the longest paths between all pairs of operations.

This approach to integrate scheduling and register binding enables a compiler to sacrifice schedule freedom selectively in order to obtain a register binding that respects the individual register file capacities. This feature is considered important given the current trend towards a distributed register file architecture. The efficiency of implementing a strong interaction between several code generation stages is supported by the experimental results (Sections 4.3 and 4.5) that feature reasonable run times for DSP applications that are constrained in the timing, resource, and storage domain. In the current FACTS implementation, constraint analysis is performed by applying all techniques consecutively until no further progress is obtained; the performance can be further improved by developing more efficient strategies to combine the various techniques.



- Lifetime Serialization (Section 4.4), Minimize latency or Initiation Interval
- Execution interval analysis, resource constraint analysis (Ch. 3), symmetry analysis, register (Section 4.1), FIFO, STACK, filifo (Ch. 5)
- Precedence, fixed latency and initiation interval, relative timing, schedule decisions (Section 2.4)

Figure 6.1 The FACTS hierarchy

Current research in the context of constraint analysis and FACTS focuses on the following topics.

- a search strategy for binding values to registers in a *rotating* register file [Rau82]. This storage file is similar to a randomly addressable register file with the added feature of automatic register renaming to allow value lifetimes to exceed the initiation interval.

- a search strategy for assigning values to FIFOs [Alba00]. This could help to reduce code size and to allow value lifetimes to exceed the initiation interval. A similar strategy is planned for STACKs. From our experience with FIFOs and STACKs, we hope to formulate a strategy for targeting storage files that consist of registers, FIFOs, and STACKs.

- a search strategy for the assignment of operations to functional resources, and related to that, the assignment of values to storage files.

An interesting question arises with respect to the general applicability of the colouring approach of Section 4.4. Does this approach work for registers only? Or is it possible to define weak and strong conflicts in the context of FIFOs, STACKs, or functional resources, such that the colouring approach is reused effectively for the above mentioned assignment problems? Preliminary results [Alba00] suggest that a straightforward adaptation of the rules for strong and weak conflicts to a FIFO mechanism works satisfactory.

Another research question relates to the FACTS hierarchy given in Figure 6.1. Apparently, some constraints can effectively be expressed at the level of the distance matrix. Other constraints can be coped with on-the-fly by letting corresponding 'rules' be triggered by refinements in the search space (the distance matrix). This is not true for constraints related to limited storage capacity in the register files. These require an explicit search strategy to satisfy. For a given constraint, no guidelines are available for deter-

mining at which level in Figure 6.1 an approach should be defined. Such guidelines would be helpful in formulating an approach for some architectural features. For example, it is unclear whether a memory addressed indirectly with (post)-modify options [Bart92] should be handled by determining a search strategy such as in Section 4.4 or by defining rules such as in Section 5.1.

The FACTS tool is used at the Eindhoven University of Technology as a research vehicle for research in code generation. FACTS functionality is being integrated in the A|RT toolset from Frontier Design, Leuven, Belgium in order to design and program ASIPs with a VLIW architecture. At Philips research FACTS is applied in the COCOON project as part of the compiler targeted at a VLIW architecture.

Literature

[Aiken95] A. Aiken, A. Nicolau, S. Novack, "Resource-Constrained Software Pipelining", IEEE transactions on parallel and distributed systems, vol. 6, no. 12, pp. 1248-1270, December 1995

[Alom93] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi, "An ASIP instruction set optimization algorithm with functional module sharing constraint", Proc. IEEE/ACM Int. Conf. on Computer-Aided Design, pp. 526-532, 1993

[Alba99] C. Alba-Pinto, B. Mesman, and C.A.J. van Eijk, "Register Files Constraint Satisfaction during Scheduling of DSP code", Proc. Symp. on Integrated Circuits and Systems Design, pp. 74-77, 1999

[Alba00] C. Alba-Pinto, C.A.J. van Eijk, B. Mesman, and J.A.G. Jess, "Address Satisfaction for Fifo and Stack Storage Files during Scheduling of DSP Algorithms", Proc. Symp. on Integrated Circuits and Systems Design, pp. 107-112, 2000

[Bash99] S. Bashford and R. Leupers, "Constraint Driven Code Selection for Fixed-Point DSPs", Proc. ACM/IEEE Design Automation Conference, pp. 817-822, 1999

[Bart92] D.H. Bartley, "Optimizing stack frame accesses for processors with restricted addressing modes", Software-Practice and Experience, vol. 22, no. 2, pp. 101-110, Februari 1992

[Beko00] M.J.G. Bekooij, B. Mesman, C.A.J. van Eijk, J.L. van Meerbergen, and J.A.G. Jess, "Constraint Analysis for operation assignment and scheduling in FACTS", CDROM Proc. Int. Conf. on Signal Processing Applications & Technology, 2000

[Behn97] B. Behnam and G. Saucier, "IP Catalog: The Catalist of Worldwide IP business", Int. Workshop on Logic and Architectural Synthesis, pp. 55-60, 1997

[Bras99] R.A.C. Braspenning, "Modeling Issue Slot Constraints with Resources", trainee report, Eindhoven University of Technology, The Netherlands, 1999

[Catt98] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtegale, and A. Vandecappele, "Custom Memory Management Methodology", Kluwer Academic Publishers, 1998

[Chai82] G. Chaitin, "Register allocation and spilling via graph coloring", ACM Symp. on Compiler Construction, pp. 98-105, 1982

[Coff76] E.F. Coffman Jr, "Computer and Job Shop Scheduling Theory", John Wiley & Sons, New York, 1976

[Corm90] T.H. Cormen, C.E. Leiserson, R.L. Rivest, "Introduction to Algorithms", MIT Press, 1990

[Coud97] O. Coudert, "Exact coloring for real-life graphs is easy," Proc. ACM/IEEE Design Automation Conference, pp. 121-126, 1997.

[Demi94] G. De Micheli, "Synthesis and Optimization of Digital Circuits", New York, McGraw-Hill, 1994

[Depu93] F. Depuydt, "Register optimization and scheduling for real-time digital signal processing architectures", Ph.D. thesis, Katholieke Universiteit Leuven, 1993.

[Eich95] A.E. Eichenberger, E.S. Davidson and S.G. Abraham, "Optimum modulo schedules for minimum register requirements", Proc. Int. conf. on Supercomputing, pp. 31-40, 1995

[Eijk99] C.A.J. van Eijk, E.T.A.F Jacobs, B. Mesman and A.H. Timmer "Identification and Exploitation of Symmetry in DSP Algorithms", Proc. IEEE conf. on Design Automation and Test in Europe, pp. 602-608, 1999

[Eijk00] C.A.J. van Eijk, B. Mesman, C.A. Alba-Pinto, Q. Zhao, M.J.G. Bekooij, J.L. van Meerbergen, and J.A.G. Jess, "Constraint Analysis for Code Generation: Basic Techniques and Applications in FACTS", ACM Trans. on Design Automation of Electronic Systems, vol. 5, no. 4, Oktober 2000

[Fara98] P. Faraboschi, G. Desoli and J.A. Fisher, "Clustered Instruction-Level Parallel Processors", Technical report HPL-98-204, Hewlett-Packard 1998

[Fish81] J.A. Fisher, "Trace scheduling: a technique for global microcode compaction", IEEE Trans. on computers, vol. 30, no. 7, pp. 478-490, July 1981

[Fish83] J.A. Fisher, "Very long instruction word architectures and the ELI-512", Proc. 10th Ann. Int. Symp. on Computer Architecture, pp. 140-150, 1983

[Garey79] M.R. Garey and D.S. Johnson, "Computers and intractability: A guide to the theory of NP-completeness", Freeman, 1979

[Girc84] E.F. Girczyc and J.P. Knight, "An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling", Proc. IEEE/ACM Int. Conf. on Computer-Aided Design, pp. 726-731, 1984

[Goos89] G. Goossens, J. Vandewalle and H. De Man, "Loop optimization in register-transfer scheduling for DSP-systems", Proc. ACM/IEEE Design Automation Conference, pp. 826-831, 1989

[Govin94] R. Govindarajan, E.R. Altman, and G.R. Gao, "Minimizing register requirements under resource-constrained rate-optimal software pipelining", Proc. Int. Symp. on Microarchitecture, pp. 85-94, 1994

[Hart92] R. Hartmann, "Combined scheduling and data routing for programmable ASIC systems", Proc. European Design Automation Conference, pp. 486-490, 1992

[Henn96] J.L. Hennessy and D.A. Patterson, "Computer architecture, a quantitative approach", Morgan Kaufmann 1996

[Heij91] M.J.M. Heijligers, "Time Constrained Scheduling for High Level Synthesis", Masters Thesis, Eindhoven University of Technology, 1991

[Heij96] M.J.M. Heijligers, "The Application of Genetic Algorithms to High-Level Synthesis", Ph.D. Thesis, Eindhoven University of Technology, 1996

[Hoog99] J. Hoogerbrugge and L. Augusteijn, "Instruction Scheduling for TriMedia", Journal of Instruction-Level Parallelism (*http://www.jilp.org*), vol. 1, no. 1, Februari 1999

[Hu61] T.C. Hu, "Parallel sequencing and assembly line problems", Operation Research, no.9, pp. 841-848, 1961

[Huis98] J.A. Huisken, M.J.G. Bekooij, G.C.M. Gielis, P.W.F. Gruijters and F.P.J. Welten, "A Power-Efficient Single-Chip OFDM Demodulator and Channel Decoder for Multimedia Broadcasting", Solid-State Circuits Conference, digest of technical papers pp. 40-41, 1998

[Hwang91] C-T. Hwang, Y-C. Hsu, and Y-L Lin, "Scheduling for functional pipelining and loop winding " IEEE Design Automation Conference, pp. 764-769, 1991

[IEEE88] IEEE standard 1076-1987, IEEE Standard VHDL Language Reference Manual, New York: Institute of Electrical and Electronics Engineers, 1988.

[Klei97] R.P. Kleihorst, A. van der Werf, W.H.A. Bruls, W.F.J. Verhaegh and E. Waterlander, "MPEG2 video encoding in consumer electronics", Journal of VLSI Signal Processing no.17, pp. 241-253, 1997

[Ku92] D.C. Ku and G. De Micheli, "High-level synthesis of ASICs under timing and synchronization constraints", Kluwer Academic Publishers, 1992.

[Kuch97] K. Kuchcinski, "Embedded system synthesis by timing constraints solving", Proc. Int. Symp. on System Synthesis, pp. 50-57, 1999

[Lam88] M. Lam, "Software Pipelining: An effective scheduling technique for VLIW machines", ACM Conf. on Programming Language Design and Implementation, pp. 318-328, 1988

[Lann95] D. Lanneer, J. van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens, "CHESS: retargetable code generation for embedded DSP processors", in [Marw95]

[Laps96] P. Lapsley, J. Bier, A. Shoham and E.A. Lee, "DSP Processor Fundamentals", Berkeley Design Technology, 1996

[Leij00] J. Leijten, M. Bekooij, A. Bink, H. van Gageldonk, J. Hoogerbrugge and B. Mesman, "COCOON: Core and Compiler Codesign for Embedded DSP", CDROM Proc. Int. Conf. on Signal Processing Applications & Technology, 2000

[Leup96] R. Leupers and P. Marwedel, "Algorithms for address assignment in DSP code generation", Proc. IEEE/ACM Int. Conf. on Computer-Aided Design, pp. 109-112, 1996

[Leup97] R.Leupers, "Retargetable code generation for digital signal processors", Kluwer Academic Publishers, 1997

[Liem94] C. Liem, T. May and P. Paulin, "Instruction-set matching and selection for DSP and ASIP code generation", Proc. European Design & Test Conference, pp. 31-37, 1994

[Liao95] S. Liao, S. Devadas, K. Keutzer, S. Tjiang and A. Wang, "Storage assignment to decrease code size", Proc. ACM Conf. on Programming Language Design and Implementation, pp. 186-195, 1995

[Marw95] P. Marwedel and G. Goossens (editor), "Code Generation for Embedded Processors", Kluwer Academic Publishers, Boston, Massachussetts, 1995

[McFa88] M. C.SJ. McFarland, A.C. Parker and P. Camposano, "The High-Level Synthesis of Digital Systems", Proc. of the IEEE, vol. 78, no. 2, pp. 301-318, Februari 1990

[McFa90] M. C.SJ. McFarland, A.C. Parker and P. Camposano, "Tutorial on High-level synthesis", Proc. ACM/IEEE Design Automation Conference, pp. 330-336, 1988

[Meer95] J.L. van Meerbergen, P. Lippens, W. Verhaegh and A. van der Werf, "PHIDEO: High-level synthesis for high-throughput applications", Journal of VLSI Signal Processing, vol. 9, pp. 89-104, 1995

[Meer99] J.L. van Meerbergen, "Embedded Multimedia Systems on Silicon", Nat.Lab. Unclassified Report UR 802/99, Januari 1999.

[Mesm97a] B. Mesman, M.T.J. Strik, A.H. Timmer, J.L. van Meerbergen, and J.A.G. Jess, "Constraint analysis for DSP code generation", Proc. Int. Symp. on System Synthesis, pp. 33-40, 1997

[Mesm97b] B. Mesman, M.T.J. Strik, A.H. Timmer, J.L. van Meerbergen, and J.A.G. Jess, "Constraint Driven Loop Pipelining", International Workshop on Logic and Architectural Synthesis, pp. 205-211, 1997

[Mesm97c] B. Mesman, M.T.J. Strik, A.H. Timmer, J.L. van Meerbergen, and J.A.G. Jess, "An Integrated Approach to Register Binding and Scheduling", Proc. ProRisc, pp. 415-425, 1997

[Mesm98] B. Mesman, M.T.J. Strik, A.H. Timmer, J.L. van Meerbergen, and J.A.G. Jess, "A constraint driven approach to loop pipelining and register binding", Proc. IEEE conf. on Design Automation and Test in Europe, pp. 377-383, 1998

[Mesm99a] B. Mesman, A.H. Timmer, J.L. van Meerbergen, and J.A.G. Jess, "Constraint analysis for DSP code generation", IEEE Trans. on Computer-Aided Design, vol. 18, no. 1, Januari 1999

[Mesm99b] B. Mesman, C.A. Alba-Pinto, and C.A.J. van Eijk, "Efficient Scheduling of DSP Code on Processors with Distributed Register files", Proc. Int. Symp. on System Synthesis, pp. 100-106, 1999

[Mesm99c] B. Mesman, C.A.J.van Eijk, C.A. Alba-Pinto, M.G.J. Bekooy, J.L. van Meerbergen, and J.A.G. Jess, "Constraint analysis for code generation: basic techniques and their applications in FACTS", presented at the Int. Workshop on Software and Compilers for Embedded Systems, 1999

[Mesm01] B. Mesman, C.A.J.van Eijk, and M.G.J. Bekooy, "Constraint Analysis for Scheduling DSP Code", submitted to Int. Static Analysis Symposium, 2001

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

[Papa82] C.H. Papadimitriou and K. Steiglitz, "Combinatorial optimization: algorithms and complexity", prentice-hall, 1982

[Park88] N. Park and A.C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications", IEEE Trans. on Computer-Aided Design, vol. 7, no. 3, pp. 356-370, March 1988

[Paul89] P.G. Paulin and J.P. Knight, "Force-directed scheduling for the behavioural sythesis of ASIC's", IEEE Trans. on Computer-Aided Design, vol. 8, no. 6, pp. 661-679, June 1989

[Paul95a] P.G. Paulin, C. Liem, T.C. May, and S. Sutarwala, "DSP design tool requirements for embedded systems: a telecommunications industrial perspective", J. VLSI Signal Processing, vol.9, no.1-2, pp.23-47, Januari 1995

[Paul95b] P.G. Paulin, C. Liem, T.C. May, and S. Sutarwala, "FlexWare: a flexible firmware development environment", in [Marw95]

[Paul96] P.G. Paulin and C. Liem, "Embedded Systems: Trends and Tools", tutorial notes, European Design & Test Conference, 1996

[Pot92] M. Potkonjak and J. Rabaey, "Scheduling algorithms for hierarchical data control flow graphs", Int. Journal of Circuit Theory and Applications, vol. 20, pp. 217-233, 1992

[Praet94] J.V. Praet, G. Goossens, D. Lanneer and H. de Man, "Instruction set definition and instruction selection for ASIPs", Proc. Int. Symp. on High-Level Synthesis, pp. 11-16, 1994

[Rao99] A. Rao and S. Pande, "Storage Assignment using Expression Tree Transformations to Generate Compact and Efficient DSP Code", ACM Computer Architecture News, vol. 27, no1, pp. 39-42, 1999

[Rau81] B.R. Rau and C.D. Glaeser , "Some scheduling techniques and an easiliy schedulable horizontal architecture for high performance scientific

computing", Proc. of the 14th Workshop on Microprogramming, pp. 183-198, 1981

[Rau82] B.R. Rau, C.D. Glaeser and E.M. Greenawalt, "Architectural support for the efficient generation of code for horizontal architectures", Proc. Symp. on Architectural Support for Programming Languages and Operating Systems, pp. 96-99, 1982

[Rau92] B.R. Rau, M. Lee, P.P. Tirumalai and M.S. Schlansker, "Register allocation for software pipelined loops", Proc. Conf. on Programming Language Design and Implementation, pp. 283-299, 1992

[Rau96] B.R. Rau, "Iterative Modulo Scheduling", Int. Journal of Parallel Programming, vol. 24, no. 1, pp. 3-64, Februari 1996

[Rau98] B.R. Rau, V. Kathail and S. Aditya, "Machine-description driven compilers for epic processors", Tech. Rep. HPL-98-40, Hewlett Packard research labs, 1998

[Rau99] B.R. Rau, V. Kathail and S.A. Gupta, "Machine-description driven compilers for EPIC and VLIW processors", Design Automation for Embedded Systems, vol. 4, no. 3, pp. 71-118, March 1999

[Reit68] R. Reiter, "Scheduling parallel computation", Journal of the ACM, vol.15, pp. 590-599, 1968

[Rimey89] K.E. Rimey, "A compiler for application-specific signal processors", Ph.D. thesis, University of California at Berkeley, 1989

[Romp92] K. van Rompaey, I. Bolsens, and H. De Man", "Just in time scheduling" , Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors, pp. 295-300, 1992

[Schlan94] M.S. Schlansker, B.R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik and S.G. Abraham, "Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity", Tech. Rep. HPL-96-120, Hewlett Packard research labs, 1994

[Strik95] M.T.J. Strik, J.L. van Meerbergen, A.H. Timmer and J.A.G. Jess, "Efficient code generation for in-house DSP-cores", Proc. European Design & Test Conference, pp. 244-249, 1995

[Strik94] M.T.J. Strik, "Efficient code generation for application domain specific processors", Eindhoven University of Technology IVO-report, ISBN 90-5282-390-1, 1994

[Sudar97] A. Sudarsanam, S. Liao and S. Devadas, "Analysis and evaluation of address arithmetic capabilities in custom DSP architectures", Proc. ACM/IEEE Design Automation Conference, pp. 287-292, 1997

[Sugi96] N. Sugino, M. Myiazaki, S. Iimuro and A. Nishihara, "Improved code optimization method utilizing memory addressing and its application to compilers", Proc. IEEE Int. Symp. on Circuits and Systems, vol. 2, pp. 249-252, 1996

[Thom90] D.E. Thomas, E.D. Lagnese, R.A. Walker, J.A. Nestor, J.V. Rajan and R.L. Blackburn, Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench, Kluwer Academic Publisher, 1990

[Timm95a] A.H. Timmer, M.T.J. Strik, J.L. van Meerbergen and J.A.G. Jess, "Conflict modelling and instruction scheduling in code generation for in-house DSP cores", Proc. ACM/IEEE Design Automation Conference, pp. 593-598, 1995

[Timm95b] A.H. Timmer, "From Design Space Exploration to Code Generation", Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 1995

[TMS97] "TMS320C60xx CPU and Instruction Set Reference Guide", Texas Instruments 1997

[Trim97] "Trimedia TM-1 Media Processor Data Book", Philips Semiconductors, Trimedia Product Group, 1997

[Verh97] W.F.J Verhaegh, P.E.R. Lippens, E.H.L. Aarts and J.L. van Meerbergen,"Multidimensional periodic scheduling: A solution approach", Proc. EuropeanDesign & Test Conference, pp. 468-474, 1997

[Woud94] R. Woudsma, "EPICS, a flexible approach to embedded DSP cores", CDROM Proc. Int. Conf. on Signal Processing and Application and Technology, 1994

[Zhao00] Q. Zhao, C.A.J. van Eijk, C.A. Alba Pinto and J.A.G. Jess, "Register binding for predicated execution in DSP applications.", Proc. Symp. on Integrated Circuits and Systems Design, pp. 113-118, 2000

[Zivo94] V.Zivojnovic, J.M. Velarde, C. Schlaeger and H. Meyr, "DSPStone - A DSP oriented Benchmarking Methodology", CDROM Proc. Int. Conf. on Signal Processing and Application and Technology, 1994

Samenvatting

Methoden voor code generatie voor digitale signaal processoren (DSP) worden in toenemende mate belemmerd door de combinatie van stringente tijdsbeperkingen opgelegd door DSP applicaties, en de resource beperkingen opgelegd door de processor architectuur. Indien loop pipelining wordt toegepast vormt de beperkte beschikbaarheid van resources een probleem voor 'greedy' scheduling heuristieken. De beperkte beschikbaarheid van registers, gedistribueerd over de processor, vormt een probleem voor oplossingesmethoden die de taken van scheduling en register toewijzing in opeenvolgende fasen uitvoeren. Door deze scheiding kunnen vaak suboptimale of zelfs geen oplossingen gegenereerd worden, omdat het probleem van fase koppeling genegeerd wordt; doordat de levens intervallen van variabelen worden bepaald door het schedule, is scheduling mede bepalend voor de zoekruimte voor register toewijzing. Als gevolg hiervan 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 en randvoorwaarden en met het probleem van fase koppeling.

De benadering voorgesteld in dit proefschrift is gebaseerd op analyse van de randvoorwaarden om de schedule zoekruimte in te perken. Op deze manier wordt vaak voorkomen dat de scheduler een beslissing neemt die onherroepelijk leidt tot schending van de randvoorwaarden. Het belangrijkste aspect van ons model van de schedule zoekruimte is de afstandsmatrix, waar de minimum and maximum tijdsrelaties worden bijgehouden tussen elk paar operaties in een Basic Block. Algoritmen met een lage complexiteit worden gebruikt om extra volgorde relaties te identificeren die genoodzaakt zijn door de combinatie van tijdsrelaties in the afstandsmatrix en de resource beperkingen in de processor architectuur. De resultaten van de analyses worden gecombineerd in the afstandsmatrix door het berekenen van de langste paden geinduceerd door de gegenereerde volgorde relaties. Interactie tussen de scheduler en Constraint Analyse wordt bewerkstelligd door schedule beslissingen uit te drukken in extra volgorde relaties en de afstandsmatrix aan te passen.

Teneinde de register benodigdheden te minimaliseren of de beperkingen op de register capaciteit te respecteren, wordt de vrijheid in het schedule domein benut om de levensintervallen van variabelen te serializeren. Variabelen die een potentieel knelpunt vormen voor register toewijzing worden geidentificeerd, en hun levensintervallen geserializeerd. Deze serializaties worden geevalueerd in the context van de randvoor-

waarden en de afstandsmatrix wordt aangepast. Na het serializatieproces komt elke completering van het schedule gegarandeerd overeen met een geldige register toewijzing. Op soortgelijke wijze is het mogelijk de lees- en schrijfacties op een register file te ordenen zodat de gecommuniceerde variabelen zich gedragen volgens een stroomgebaseerd schema. Deze variabelen kunnen dan worden opgeslagen in een FIFO. In termen van instructie bitten hebben FIFO's dezelfde adresseringskosten als registers, maar bieden een veel grotere opslagcapaciteit. Dit is interessant doordat register adressering bij VLIW processoren ongeveer 60% van de programmacode bepaalt. Op soortgelijke wijze worden lees- en schrijfacties geordend teneinde variabelen te kunnen opslaan in een stack of een FILIFO.

Curriculum Vitae

Bart Mesman received the Electrical Engineering Degree (with honors) from the Eindhoven University of Technology, the Netherlands, in 1995. From 1995 to 1999 he was a member of the digital VLSI group at Philips Research in Eindhoven and the Information and Communication Systems group of the electrical engineering department at the University of Technology in Eindhoven. Since 1999 he is a member of the group Embedded Systems Architectures on Silicon at Philips Research in Eindhoven and the Eindhoven Embedded Systems Institute. Bart Mesman is currently working in the COCOON project with the explicit goal of co-designing a processor architecture and a code generation methodology. His research interests include High-Level Synthesis, ASIP-architectures and code generation for embedded DSPs.

Stellingen

behorende bij het proefschrift

Constraint Analysis for DSP Code Generation

van Bart Mesman

Technische Universiteit Eindhoven, 23 Mei 2001

1. Bij het ontwerp van een instructie set dient een afweging te worden gemaakt tussen de uitdrukkingskracht van de instructie set en de efficientie waarmee de compiler om kan gaan met de beperkingen die daaruit voortvloeien.

2. Tijdens het genereren van een schedule kunnen beslissingen genomen worden die inconsistent zijn met de randvoorwaarden. Dit wordt dan vaak pas laat opgemerkt, waardoor veel beslissingen teniet gedaan moeten worden, en lange rekentijden ontstaat. Om dit te voorkomen is het voor sommige randvoorwaarden zinnig om een analyse te doen, terwijl voor andere randvoorwaarden een expliciete zoek strategie met identificatie van bottlenecks een betere oplossing vormt (H.6).

3. Hoewel de afstands matrix een complexere representatie is van de schedule zoekruimte dan de zogenaamde executie interval representatie [Timmer], leent de eerste zich beter voor het respecteren van de register file capaciteit. Dit komt doordat precedentie relaties meer zeggen over de registerbezetting dan tijdsintervallen van operaties (H.3 & 4).

   A.H. Timmer, "From Design Space Exploration to Code Generation", Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 1995

4. De constraint analyse technieken uitgelegd in dit proefschrift werken efficienter bij de scheduling problematiek dan meer algemeen toepasbare "constraint satisfaction" technieken [Kuchcinski]. Bij de eerstgenoemde zijn namelijk zowel de data structuren als de daarop opererende algoritmes toegesneden op de scheduling problematiek (H.3).

   K. Kuchcinski, "Embedded system synthesis by timing constraints solving", Proc. Int. Symp. on System Synthesis, pp. 50-57, 1999

5. Door de sterk toenemende maskerkosten bij nieuwe generaties van de fabricage technologie voor ICs neemt het belang toe van herconfigureerbare IC architecturen.

6. Bij het besteden van computer tijd aan het oplossen van moeilijke problemen dient een balans gevonden te worden tussen het aantal beslissingen (oplossingen) dat geevalueerd wordt en het aantal foute beslissingen (oplossingen) dat vermeden wordt door analyse van de zoekruimte.

7. De werking van de constraint analyse technieken uitgelegd in dit proefschrift laat zich vergelijken met de werking van ons brein. Daar kunnen complexe gedachten ontstaan door interactie van een grote hoveelheid zeer primitieve elementen (neuronen). Zo kunnen, door veelvuldig toepassen van elementaire regels van constraint analyse, complexe redenaties ontstaan over de oplossingsruimte van een scheduling probleem.

8. Het korte termijn denken van productgroepen remt de ontwikkeling tot het ontwerpen op een hoger niveau van abstractie, omdat een hogere prioriteit verleend wordt aan het halen van een deadline dan aan training in dit niveau van ontwerpen en de daarbij behorende ontwerpgereedschappen.

9. Dat de automatisering heeft toegeslagen in alle denkbare bezigheden op het kantoor blijkt wel uit de observatie dat zelfs de traditionele term "paperware" is verdrongen door de term "powerpointware".

10. Wanneer de waarde van een strategie doorgedrongen is tot de hogere management lagen in een groot bedrijf, ontstaat veelvuldig de situatie dat de strategy concepten een doel op zich worden. Daarbij wordt voorbijgegaan aan de mogelijkheid dat het nastreven van zo'n strategie soms meer werk kost dan het originele doel. Zo kan de dwang tot hergebruik van hardware of software blokken teneinde de ontwerpinspanning te verminderen, juist een enorme ontwerpinspaning met zich meebrengen.

11. Dat een Nederlandse vertaling van Engels geschreven technische tekst niet altijd begrijpelijker is voor niet-Engelstaligen, blijkt wel uit dit proefschrift.

12. Vooral mannen hebben nogal eens moeite te herkennen wanneer ze gevoelsmatig gemanipuleerd worden. De volgende uitspraak is een handige vuistregel voor deze mannen: Als je je schuldig voelt en je kan niet goed uitleggen waarom, dan ben je gemanipuleerd.

13. Volgens de evolutietheorie ontstaan nieuwe rassen doordat de natuurlijke selectiecriteria die wezens bevoordelen die zich meer hebben aangepast aan de omstandigheden waarin ze leven. De zingevingsvraag van veel mensen is in deze context dan meer een vraag om onzin dan om zin.