Exploiting instruction-level parallelism : a constructive approach

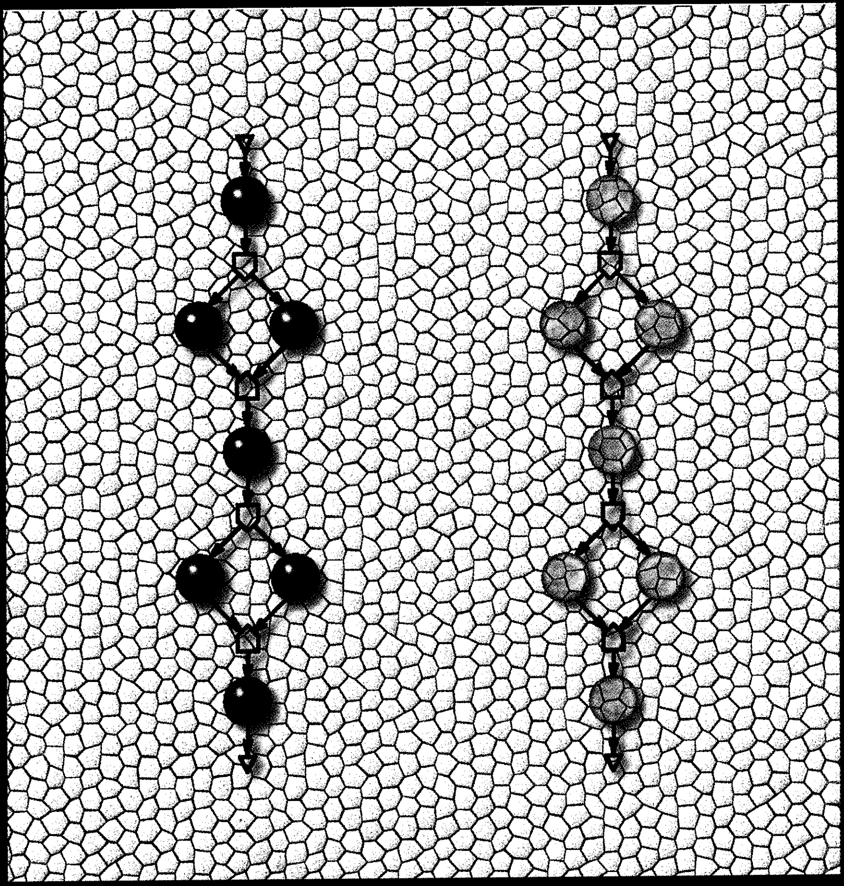
DOI:
10.6100/IR516269

Document status and date:
Published: 01/01/1998

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

Exploiting instruction-level parallelism

a constructive approach



Luiz Cláudio Villar dos Santos

Exploiting instruction–level parallelism

a constructive approach

Exploiting instruction–level parallelism

a constructive approach

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 maandag 23 november 1998 om 16.00 uur

door

Luiz Cláudio Villar dos Santos

geboren te Arapongas, Brazilië

Dit proefschrift is goedgekeurd door de promotoren:

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

Copromotor:

dr.ir. C.A.J. van Eijk

Druk: Universiteitsdrukkerij Technische Universiteit Eindhoven

CIP–DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Villar dos Santos, Luiz C.

Exploiting instruction–level parallelism : a constructive approach / by Luiz C. Villar dos Santos. – Eindhoven : Technische Universiteit Eindhoven, 1998.
Proefschrift. – ISBN 90-386-0490-4
NUGI 832
Trefw.: grote geintegreerde schakelingen ; CAD / digitale systemen ; CAD / combinatorische optimalisering / scheduling.
Subject headings: VLSI / digital systems / high level synthesis / scheduling.

Summary

The increasing complexity of integrated circuits provided by VLSI technology requires design automation at higher levels of abstraction. In such context, high–level synthesis translates a behavioral–level specification of the digital system into an architecture consisting of a data path and a control unit. Emerging design problems are prompting the utilization of instruction–level parallelism (ILP), traditionally an object of parallelizing compilers, for the *synthesis* of digital systems. In this thesis, techniques like code motion, speculation and loop pipelining are employed to expose ILP and their application is oriented to digital systems designed to operate under a global time–constraint. A resource–constrained optimization problem is formulated as a starting point. From a given specification and a set of resource constraints, the goal is to obtain a symbolic finite state machine (FSM) for the control unit of the digital system such as to minimize the schedule length of the critical execution path. An approach is proposed in which several alternative solutions are generated and explored by means of a local search algorithm.

For the construction of a FSM, our approach combines both graph algorithms and Boolean techniques. The flow of control is represented in the form of a graph consisting of branch and merge junctions and so–called basic blocks. During a top–down traversal of this graph, the operations scheduled in each state may come from different basic blocks and possibly from different iterations of a loop. Equivalent states are detected and merged on–the–fly. A code–motion pruning technique is proposed to prevent inefficient code motions caused by the typical unbalance between the potential parallelism *exposed* and the parallelism that can actually be *exploited* within the available resources.

Experimental results show, on the one hand, that the growth of the number of states caused by ILP techniques can be restrained efficiently by the on–the–fly merging of equivalent states. On the other hand, they provide evidence that the pruning technique increases the density of good–quality solutions in the search space, thereby paving the way to a reduction of average search time.

Samenvatting

De toenemende complexiteit van de hedendaagse chips creëert een behoefte om op een hoger niveau van abstractie te ontwerpen. Vanuit een functionele beschrijving van een digitale systeem, genereert hoog–niveau synthese een architectuur die bestaat uit een datapad en een "control unit". Hedendaagse ontwerpproblemen suggereren het gebruik van instructie–niveau parallellisme (ILP), welbekend uit het vakgebied van compilers, voor de *synthese* van digitale systemen. In dit proefschrift wordt het gebruik van ILP technieken als "code motion", speculatieve executie en "loop pipelining" gericht op digitale systemen onder tijdsbeperkingen. Een optimaliseringsprobleem wordt geformuleerd waarin het aantal en soort bouwblokken wordt beperkt. Het doel is, om vanuit de beschrijving en het aantal bouwblokken, een symbolische FSM te genereren voor de "control unit" met minimaal kritiek executiepad. Een methode wordt gepresenteerd waarbij verschillende oplossingen gecreëerd en onderzocht worden met behulp van een lokale zoekmethode.

Om een FSM op te bouwen, gebruikt de methode graaf algoritmen en Boolse technieken. De controle–stroom wordt gerepresenteerd als een graaf die bestaat uit "branch" en "merge" knooppunten en zogenaamde "basic blocks". Tijdens het doorlopen van deze graaf, kunnen de operaties die geselecteerd zijn voor een toestand komen vanuit verschillende "basic blocks" en van verschillende loop iteraties. Gelijke toestanden worden geïdentificeerd en samengevat tijdens het opbouwen van de FSM. Door het gebrek aan evenwicht tussen het *potentieel* parallellisme en het echt *bruikbaar* parallellisme binnen de beschikbaar bouwblokken, wordt een techniek genaamd "code–motion pruning" (CMP) voorgesteld om inefficiënt "code motions" te voorkomen.

Resultaten uit experimenten tonen aan dat de toename van het aantal toestanden, door het gebruik van ILP technieken, kan worden beperkt door het samenvatting van gelijke toestanden. Andere resultaten tonen aan dat het percentage oplossingen van goede kwaliteit in de zoekruimte toeneemt door de CMP techniek. Dit baant de weg voor een reductie van de gemiddelde zoektijd.

Contents

Acknowledgements

This Ph.D. thesis reports the research performed in the Design Automation Section of the Department of Electrical Engineering of the Eindhoven University of Technology in the Netherlands, under the supervision of prof.Dr.–Ing. J. A. G. Jess.

I am thankful to prof. Jess for the opportunity to work in his research group. Our first meeting occurred on March 1992, after his lectures during the 2nd Brazilian Microelectronics School, held in the town of Gramado. Since then, many fruitful discussions have followed, influencing my research activities.

I would like to thank Marc Heijligers and Koen van Eijk for their cooperation in different phases of my research work. Marc has introduced me to important issues in high–level synthesis, thereby pointing the first stepping–stones. Koen found the time for various discussions and gave me many suggestions. Through a careful and fast proofreading of the first draft of this thesis, he has indicated various parts of the text requiring elaboration. I am also grateful to Jos van Eijndhoven for his suggestions in the course of my research. I appreciate his welcoming attitude towards me and my family in our early days in the Netherlands.

I would like to express my gratitude to the people whose software I reused or combined within my own programs. Marc Heijligers, Ric Hilderink, Adwin Timmer, Wim Philipsen and Harm Aarts have developed the NEAT System, upon which I have built my implementation. Geert Janssen has developed the BDD package that I have used for Boolean manipulation. The C^{++} interface written by Koen van Eijk paved the way to bridging the BDD package with the NEAT System.

Furthermore, I am grateful to my roommate Michel Berkelaar and to Jeroen Rutten. I appreciate their help and our pleasant conversations. Also, I would like to thank all other members of the Information and Communication Systems Group, where I found a well–balanced combination of skills, sociability and good humor.

I am also grateful to the members of the reading committee for their remarks and suggestions.

I owe a lot to my compatriots living in the Netherlands. Special thanks to Carla and Reginaldo, Viviane and Cícero, Sônia and Egbert, Isabel and César, Sílvia, Javier, Luís Barbosa, Carlos, Lincoln, and many others.

Last, but not least, I would like to thank the patience and comprehension of my wife Joice and my son Rafael, who have made of this long stay in the Netherlands, a pleasant experience in family.

I was partially supported by the Federal University of Santa Catarina, my employer in Brazil, and by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), one of the Brazilian agencies supporting research, under fellowship award n. 200283/94–4.

Chapter 1 Introduction

The term *instruction–level parallelism* (ILP) stands for the fine–grain parallelism observable among the elementary operations used to build up a program. Traditionally, exploitation of ILP is the object of parallelizing compilers. However, emerging design problems are prompting the utilization of ILP for the *synthesis* of digital systems. This thesis addresses the application of ILP techniques during the synthesis of synchronous digital systems. It focuses on systems which have to operate under a *time constraint*.

Several techniques to deal with ILP are proposed in the compiler–technology arena. Although most of the techniques to *expose* parallelism are quite general, the way of *exploiting* the exposed parallelism varies depending on the target application domain. As a consequence, the techniques conceived in the compiler domain can not be directly applied to the synthesis of time–constrained systems. This is due to the fact that the goal of parallelizing compilers is to optimize average program runtime, while time–constrained systems need support for worst–case runtime.

This chapter presents a brief overview of the available techniques and of the issues involved in the synthesis of digital systems for some emerging applications leading to time–constrained design problems.

1.1 High–level synthesis

The increasing complexity of integrated circuits (ICs) provided by very large scale integration (VLSI) technology requires design automation at higher levels of abstraction, such as the behavioral level and the register–transfer level (RTL). At the behavioral level, the function of a digital system is described in the form of an algorithm which computes the output values of the system from its input values, abstracting from the way the system is actually implemented. At the register–transfer level, the structure of the digital system is described as a netlist of functional units (adders, ALUs, multipliers, etc.), memory elements and interconnect elements (buses, multiplexers, etc.).

High–level synthesis (HLS) has been defined as a translation "from an algorithmic level specification of the behavior of a digital system to a register–transfer level structure that implements that behavior" [45]. HLS is a field of intensive research and comprehensive surveys can be found in the literature [16] [23].

HLS results in the architecture of a digital system, consisting of a data path and a control unit, as shown in Figure 1.1. The data path is described as a network at the register–transfer level and the control unit is usually described in the form of a symbolic finite–state machine. In HLS, the design has to comply with a set of *constraints*, like completion time, throughput rate and execution order and it is driven by a set of *objectives*, like the minimization of IC area, power consumption or number of states.

HLS is usually decomposed in several subproblems, like module selection, allocation, scheduling and binding. *Module selection* determines the kind of resources needed in the data path, *allocation* evaluates how many of such resources are necessary, *scheduling* determines when the operations are executed, and *binding* assigns operations to specific resources. Although these subproblems are interdependent, they are solved separately in most cases, because a completely unified approach seems unpractical. Due to this interdependence, the order of solving these subproblems may lead to different final results. The most suitable order is dictated by the target application domain [63].

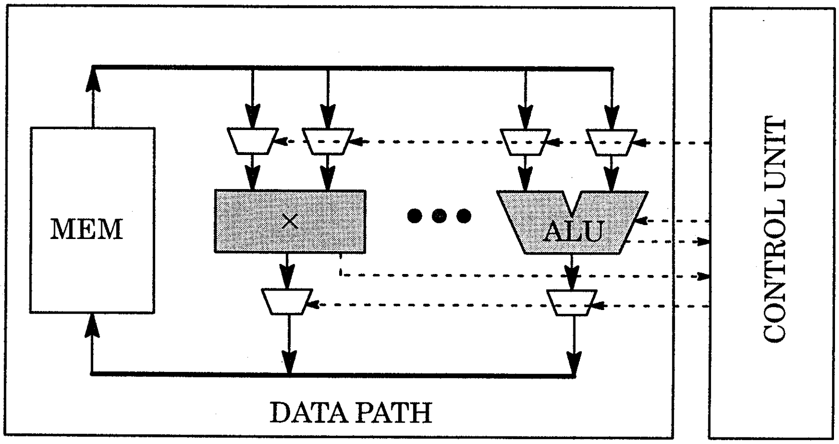


FIGURE 1.1. IC architecture at the register–transfer level

Traditionally, HLS tools are oriented to the synthesis of hard–wired VLSI circuits, which are known as *application specific integrated circuits* (ASICs). A recent trend broadens the scope of HLS to include the design of *application specific instruction set processors* (ASIPs), which are programmable circuits tailored to an application domain [25] [68]. This broader scope has its roots in the adoption of an *architecture template*, as a starting point for HLS. An example of such a template is the MISTRAL architecture [64], which is oriented towards audio applications. This template is the result of early research on silicon compilation, in particular the CATHEDRAL silicon compilers [15], which are successfully used for digital signal processing (DSP) applications.

From a HLS perspective, the design of an ASIP core can be viewed as if module selection and allocation had already been performed, such that a complete data path suitable for the whole application domain is determined. As a consequence, the remaining tasks, scheduling and binding, are responsible for the programmability. After they are performed, the resulting symbolic finite–state machine is then mapped to a microcoded controller. Flexibility is obtained by providing on–chip RAM as control store, allowing microcode downloading. Within this scenario, the role of the remaining tasks is to accomplish the (micro) *code generation* for the application domain specific processor.

1.2 ILP techniques

Modern architectures, such as superscalars and very long instruction word (VLIW) machines, have multiple functional units. They rely on the overlapped execution of independent instructions. Superscalar machines require specific hardware for run–time scheduling and dynamic dependence analysis, while VLIW machines perform compile–time scheduling and static dependence analysis. Although the ILP techniques described here are used by both superscalar and VLIW processors, this thesis focuses on *compile–time scheduling* techniques. A comprehensive overview of specific techniques for superscalar processors can be found in [30].

The traditional scope for exploiting parallelism in early compilers was the *basic block* (BB), a straight–line code sequence without branches, except at the entry and exit points. Since the amount of parallelism available in a basic block is limited, the ample resources present in modern architectures would be poorly utilized. As a consequence, techniques are required to expose parallelism beyond basic–block boundaries. This is performed by allowing code to move from one BB to another, which is called *code motion*. Some code motions place instructions ahead of conditional branches, a technique known as *speculation*.

One of the first proposed ILP techniques is Trace Scheduling [22], which takes the most likely execution path and optimizes it as if it were a single basic block. Extra code is inserted to compensate for the side effects of some of the optimizations. However, this technique is not general enough since it assumes a highly predictable control–flow. A more general technique is Percolation Scheduling [48], which defines a set of semantics–preserving code motions. Percolation Scheduling provides a way of exposing parallelism iteratively, by the successive application of primitive code motions.

Nevertheless, these first ILP methods are essentially resource–unconstrained parallelization techniques. Their drawback is that some of the greedily performed code motions might have to be undone, since they can not be accommodated within the available resources. This lack of global management of resources motivated the development of resource–constrained parallelization techniques [19][46][47].

ILP can also be uncovered beyond loop boundaries, by means of a straightforward method called *loop unrolling*. It consists in replicating the loop body several times such that the resulting loop body contains multiple iterations of the original loop and, as a consequence, more parallelism is exposed. A more elaborate technique is *software pipelining*, which is also known as *loop pipelining* or *loop folding*. The main idea of software pipelining is to allow the exploitation of ILP across loop boundaries, by overlapping the execution of instructions belonging to different iterations of a loop, but without unrolling the loop.

A simple and widely used software pipelining technique is Modulo Scheduling [39]. It is a very efficient approach suitable for single basic–block loops, but it does not address properly the software pipelining of loops containing conditional constructs. This more general problem is tackled by other methods like Perfect Pipelining [2][3], which combines loop unrolling and the detection of a repeating pattern to form the pipelined schedule. Another suitable method is Enhanced Pipeline Scheduling [17], which relies on the motion of instructions around the loop.

1.3 ILP in high–level synthesis

Most HLS methods are oriented to data–flow dominated designs. Although loop pipelining is commonly supported [13][24], code motion is rarely addressed [54][70]. Nevertheless, some methods have been proposed to cope with behavioral descriptions containing conditional constructs, such as “if–then–else”. Path–based Scheduling [12] and Tree–based Scheduling [34] aim at optimizing the execution time of each path. Others, like conditional vector list scheduling [71][72] and the hierarchical reduction approach [37], are oriented to average execution time. The combination of speculation and loop pipelining is addressed in [32]. Recently, an approach has been proposed to tackle applications with combined data and control–flow [9]. However, neither code motion nor loop pipelining is supported. Although powerful techniques [44] [67] are available for handling time–constraints, they can not cope properly with optimization under complex control flow, since they are oriented to data–flow dominated applications.

HLS for high–throughput applications, like real–time video encoding and decoding, is addressed in the PHIDEO system [43]. In this application domain, a typical behavioral description is structured as a hierarchy of nested loops. The basic concept of PHIDEO is the assumption that executions of a same operation are periodical in time. The loop hierarchy is translated to periodic operations and scheduling consists of selecting the start times and the periods of the operations [43]. Such specialized techniques for high–throughput applications do not fall within the scope of this thesis.

Since the starting point of HLS is a behavioral description and not a sequence of instructions, the techniques used in this thesis are actually applied to *operations*, instead of instructions. Although the term *operation–level parallelism* would be appropriate, the more usual term instruction–level parallelism is adopted and used throughout this thesis.

1.4 Emerging time–constrained problems

Complex modern digital systems perform several tasks. Their design is typically approached by partitioning the tasks into a data–flow dominated and a control–flow dominated part, such that two different HLS tool–suites can be applied, each one specialized on one of these domains. An instance of this strategy can be found in [11], where the design of a videophone coder–decoder motion estimator is described.

Another example is reported in the design of an IC known as "I.McIC" chip [38][73]. This IC performs MPEG2 video encoding for applications in digital video recorders and cameras. It contains a PHIDEO processor for the high–throughput tasks and a MISTRAL core for the control tasks. The MISTRAL core is designed as an ASIP to provide the necessary programmability. It is conceived to accommodate different standards and all tasks that are subject to change in future products [38]. The behavioral description for the tasks assigned to the ASIP exhibits a structure of nested conditional constructs enclosed by an outer loop [65]. Those tasks must be completed within a limited interval of time. This interval is dictated by the time to process the information in a so–called macroblock of a video picture, which is a composition of one 16 × 16 luminance block and two 8 × 8 chrominance blocks. In practice, this represents a tight time–constraint, which makes it difficult to add extra functionality to the ASIP, unless a better exploitation of parallelism is envisaged.

A somewhat similar behavioral description structure, also subject to a global cycle budget, is reported for applications in the area of Asynchronous Transfer Mode (ATM) [61]. A sketch of such a structure is given in Figure 1.2. In the figure, conditional constructs are explicitly shown and t_i designates the test associated with the i^{th} conditional construct. Braces represent the basic blocks enclosing operations such as additions, subtractions and multiplications. The dashed arrow represents the outer loop, while the bidirectional arrow depicts the time constraint (e.g. latency, data introduction interval), meaning that an upper bound of T_C clock cycles must be satisfied for the worst–case execution of the loop body.

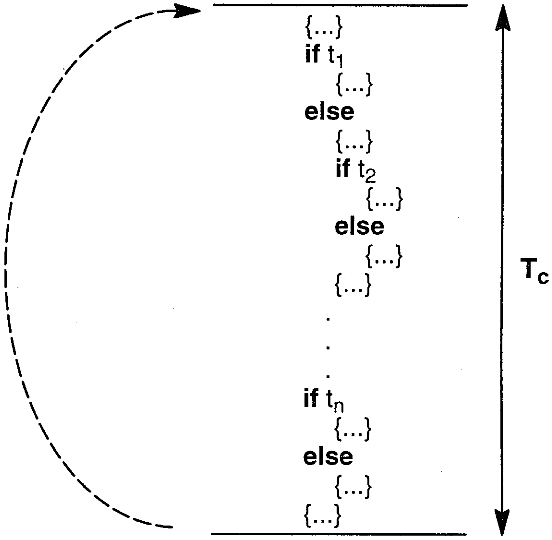


FIGURE 1.2. Structure of a behavioral description under a time constraint

For these emerging problems, whose structure is sketched in Figure 1.2, if optimization is restricted to the scope of basic blocks, it might be impossible to meet a tight time–constraint. In this scenario, the application of ILP techniques, such as code motion, speculation and loop pipelining may reduce the schedule length of the critical execution path and grant time–constraint feasibility. Besides, the combination of intensive data–flow, complex control–flow and time constraints creates a challenging problem to which few solutions have been proposed, as shown in the previous section.

These reasons motivate the research described in this thesis, where the application of ILP techniques is oriented to the satisfaction of time constraints, instead of execution speeding up. As a consequence, speeding–up techniques such as those based on branch prediction are not addressed. More information on the application of prediction–based techniques in the domain of HLS can be found in [32].

1.5 Outline of this thesis

As opposed to classical compilers, synthesis tools for embedded system design can often afford to spend more time on optimization. The effort in saving a few clock cycles to meet a tight constraint tends to outdo a redesign from scratch. For this reason, a good exploration of the design space is very important in HLS.

This thesis proposes a *constructive* approach oriented to the *synthesis* of time–constrained digital systems. The approach provides a way of*generating* and *exploring* several alternative solutions to a given optimization problem.

The proposed approach assumes that module selection and the allocation of functional units have been performed beforehand. In other words, the number and kind of functional units, henceforth referred to as *resource constraints*, are fixed prior to scheduling. Therefore, a *resource–constrained optimization problem* is formulated as a starting point. In the construction of each solution, ILP techniques are used in such a way that the parallelism exposed is constrained by the available resources. In this thesis, we assume that binding, along with the allocation of registers and interconnect elements, will be performed after scheduling.

An advantage of adopting a resource–constrained optimization problem as a starting point is that it paves the way to an unified approach comprising not only the synthesis of ASICs, but also the code generation for ASIPs, if registers and interconnect elements are also modeled as resource constraints.

The topics addressed in this thesis are organized as follows.

Chapter 2 describes the design representation and the terminology used throughout the thesis and defines the optimization problem to be tackled.

Chapter 3 summarizes the proposed constructive approach. It explains the interaction between the several engines which co–operate in the construction of a solution to the optimization problem.

Chapter 4 addresses code motion and related issues. It explains how code motion and speculation are modeled and induced in the frame of our approach. Also, an expedient called code compensation is introduced for coping with some side effects of code motion. This broadens the range of legal code motions supported in the approach. In addition, a technique is proposed to control the growth of the number of states when code motion and speculation are applied. This technique exploits the notion of state equivalence in the course of scheduling. The experimental results in this chapter show that when a HLS tool is required to make use of flexible code motions in order to face a tight time–constrained problem, the resulting number of states would be unpractical if the notion of state equivalence is overlooked during scheduling.

Chapter 5 focuses on code–motion pruning, a technique proposed to prevent inefficient code motions. It shows that, due to the typical unbalance between the potential parallelism and the constrained amount of resources to accommodate it, some code motions do not contribute to shortening schedules and, as a consequence, they are not worth doing from a worst–case execution perspective. The experimental results reported in this chapter show that the application of the pruning technique increases the density of promising solutions observable during the search, paving the way to a reduction of average search time.

Chapter 6 addresses ILP techniques for dealing with loops. First, it explains the deficiencies of modulo scheduling in properly handling loop bodies containing conditional constructs. Then, it describes how our approach can be extended for inducing loop pipelining. The chapter illustrates the close relationship between code motion and loop pipelining.

Chapter 7 contains concluding remarks and suggestions for further work. It places the application of ILP techniques in perspective with some on–going work on related research topics.

Chapter 2 Modeling the problem

2.1 Basic terminology

Similarly to a high–level programming language, the core of a behavioral description consists of statements, conditional constructs and loop constructs. Statements define the *operations*, whereas conditional and loop constructs specify *conditional execution.*

As a consequence of conditional execution, a *flow of control* is introduced, creating junctions among different sequences of statements. A *junction* in the control flow is either a *fork*, where the flow diverges from a point, or a *join*, where the flow converges to a point. When associated with a conditional construct, a divergent junction is called a *branch* and a convergent junction is called a *merge*. Junctions split the control flow into basic blocks. A *basic block* (BB) is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end, without junctions in between.

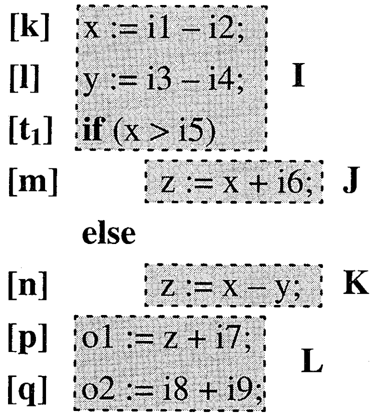
Consider, for instance, the description in Figure 2.1, where i_1 to i_9 represent inputs, o_1 and o_2 designate outputs and x, y and z are local variables. Operations are labeled with small letters at the left of their respective statements. In this description, a branch junction occurs after t_1 and a merge junction before p. As a result, the control flow exhibits four BBs, which are depicted by the shadowed boxes and are labeled with capital letters.

Among the operations in a behavioral description, some perform relational *tests* (e.g. $>$, $==$, $\leq$) and produce a Boolean value as outcome. Depending on the result of the test, a conditional branch will occur in the control flow. The operations which do not perform a test are called *ordinary operations* (e.g. +, –, ×) or simply *operations*, whenever clear from the context.

The exploitation of parallelism in a behavioral description is constrained by the available resources, the so–called *resource constraints*. For the description in Figure 2.1, we assume, throughout this chapter, that an adder, a subtracter and a comparator are available.

Let us now analyze how the operations in the example of Figure 2.1 can be scheduled. We could think of scheduling each BB independently. Such straightforward approach would not be efficient, because the amount of parallelism within a BB is limited. For example, in BB I the adder would remain idle during two cycles, even though operation q in BB L could be scheduled at the same time step as either k or l. The example suggests that we should exploit parallelism across BB boundaries, by allowing operations to move from one BB to another. This is called *code motion.*

Two operations can not always be executed in parallel, even if there are free resources. For example, operation m can not be executed in parallel with operation k, because the value of variable x consumed by operation m depends on the value produced by operation k. We say that operation m is *data dependent* on operation k. On the other hand, since there is no data dependence between operations q and k, they could be executed simultaneously. Note that, if operation q is allowed to move from BB L into BB I, a cycle will be saved in BB L. Since data dependences limit the order in which operations can be executed, they are sometimes called*precedence constraints*.



```
[k]  x := i1 – i2;
[l]  y := i3 – i4;
[t1] if (x > i5)
[m]        z := x + i6;
     else
[n]        z := x – y;
[p]  o1 := z + i7;
[q]  o2 := i8 + i9;
```

FIGURE 2.1. A behavioral description with conditional constructs

As a consequence of conditional execution, another kind of dependence arises. For instance, while operation q is always executed, operations m and n are conditionally executed, depending on the result of test t_1. We say that operations m and n are *control dependent* on test t_1.

Code motion may occur across branch junctions, because control dependences can be disregarded under certain circumstances, as opposed to data dependencies. For example, although m is control dependent on t_1, they are not data dependent on each other and can be executed simultaneously. For the same reason, m and l could be executed in parallel. In both scenarios, the motion of m from BB J to BB I causes the violation of a control dependence. However, this kind of motion is legal, provided that some mechanism be used to preserve semantics. This is obtained by either committing or discarding the result of the moved operation, as soon as the outcome of the test is known. The technique of moving an operation ahead of a conditional branch is called *speculation* or *speculative execution*. Note that, if operation m is speculatively executed and the result of test t_1 turns out to be true, a cycle will be saved, otherwise there will be no savings.

In the general case, it may be necessary to insert extra code in order to "clean" the outcome of the speculatively executed operation, the so–called *compensation code*. For this example, however, no compensation code is needed, since variable z (assigned by operation m) will be overwritten by operation n, if the result of test t_1 turns out to be false.

When operations move across merge junctions, they do not violate control dependences. For instance, we can consider moving operation q into BB K, as it can be executed in parallel with operation n. However, as operation q must always be executed and the operations in BB K are only executed when the result of t_1 is false, a copy of q has to be placed at the end of BB J. As a result, we say that *duplication* takes place and the copy can be seen as a form of compensation code. For this example, duplication saves a cycle if the result of t_1 is false, but there are no savings otherwise.

To accomplish a code motion, the moved operation has to be deleted from its original position, copies might be inserted as compensation code and sometimes the control flow itself has to be changed. This whole procedure is known as *bookkeeping* in compiler–technology terminology and it is addressed in Chapter 4.

2.2 Design representation

This section describes the design representation used throughout this thesis. It introduces the main notions used to model the system behavior, the data path structure, the symbolic description of the control unit and the execution timing.

Since most of our modeling relies on graph representations, it is convenient to briefly recollect some graph concepts [14] and associate them with the notation adopted throughout this text.

A graph $G = (V, E)$ consists of a set of nodes V and a set of edges E, where $E \subseteq V \times V$. Most of the graphs in this thesis are directed graphs. In a directed graph, the set E consists of ordered pairs of nodes and its elements are directed edges. If (u, v) is a directed edge, we say that (u, v) *leaves* node u and *enters* node v. Also, u is called a *predecessor* of v and v is said to be a *successor* of u. Given a node v, the set of all predecessors of v is denoted by PRED(v) and the set of all successors is designated by SUCC(v). The *out–degree* of a node v, written outdegree(v), is the number of edges leaving it. Conversely, the *in–degree* of a node v, written indegree(v), is the number of edges entering it. A fundamental notion throughout this thesis is the concept of a path, as defined below.

DEFINITION 2.1
A *path* in a graph $G = (V, E)$, from node v_0 to node v_k, is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of nodes such that $(v_{i-1}, v_i) \in E$, for $i = 1, 2, \ldots, k$.

A path is *simple* if all nodes in the path are different. A *subpath* is a contiguous subsequence of the nodes of a path. A path $\langle v_0, v_1, \ldots, v_k \rangle$ in a directed graph is said to form a *cycle* if $v_0 = v_k$ and it contains at least one edge. A cycle is called *simple* when $v_1, v_2, \ldots, v_k$ are distinct. A graph with no cycles is *acyclic*. In this thesis, since only simple cycles are of interest, the term cycle is used to mean a simple cycle. Two other central notions throughout this thesis, reachability and topological ordering, are highlighted below, where v_i and v_j designate arbitrary nodes of a directed graph $G = (V, E)$.

DEFINITION 2.2
Given a directed graph $G = (V, E)$ and two arbitrary nodes $v_i, v_j \in V$, node v_i *reaches* node v_j via p, written $v_i \xrightarrow{p} v_j$, if there is a path p from v_i to v_j.

Sometimes, it is unnecessary to name the path from v_i to v_j. We write $v_i \xrightarrow{*} v_j$ to mean that there is a path from v_i to v_j. Note that this path may be trivial if $v_i = v_j$.

DEFINITION 2.3
Given a directed acyclic graph $G = (V, E)$, a *topological ordering* of G is a linear ordering $<_T$ of all its nodes such that if an edge $(v_i, v_j) \in E$, then v_i precedes v_j in the ordering, written $v_i <_T v_j$.

2.2.1 Modeling behavior

In HLS, a behavioral description is usually compiled to an intermediate representation. In this thesis, behavior is modeled in the form of a data flow graph, as defined below.

DEFINITION 2.4
A *data flow graph* $DFG = (V, E)$ is a directed graph, where V is the set of nodes, representing operations, and $E \subseteq V \times V$ is the set of edges, representing dependences between operations.

Several DFG representations are available in the literature and the choice of a particular representation does not limit the application of ILP techniques. In this thesis, we adhere as much as possible to the so–called ASCIS model [21].

We assume that the behavioral description contains conditional constructs and that the respective DFG has special nodes to represent them. In the literature, a DFG supporting conditional constructs is also known as a *control data flow graph* (CDFG). An example is shown in Figure 2.2b for the description in Figure 2.2a. Circles represent either ordinary operations or tests. Triangles denote inputs or outputs. Pentagonal nodes are associated with control–flow decisions. A *branch* node (B) distributes a single value to different operations and a *merge* node (M) selects a single value among different ones.

Branch and merge nodes are controlled by a dummy node called a *conditional*, which is represented by a diamond in Figure 2.2b. A conditional transfers the outcome of a test to branch and merge nodes. It provides a simple way of modeling the fact that the result of a test is not necessarily used immediately after its evaluation. For instance, the outcome of a test can be stored to be recovered later on or it might not be available instantly (see Section 2.2.4).

The flow of data between operations is represented by *data edges*. The outcome of a test is carried by *control edges* (dashed edge in Figure 2.2b) from the conditional to branch and merge nodes. A detailed explanation of those symbols and their semantics can be found in [20].

Our DFG model also has special nodes to represent loop constructs. However, for simplicity, we postpone the discussion of loops to Chapter 6.

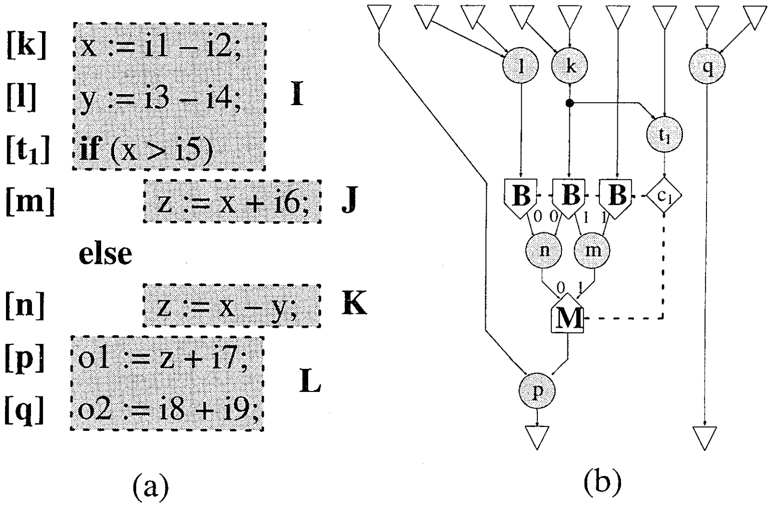
The operations in a DFG can be classified in different *operation types*, such as addition, subtraction, merge, branch, conditional, etc., as formalized below.

DEFINITION 2.5

Let T_V be the set of operation types such that $T_V = T_O \cup T_X$, $T_X = \{\text{branch, merge, conditional}\}$ and $T_O = T_V \setminus T_X$. The function $\omega : V \mapsto T_V$ represents the mapping of an operation v to an operation type t_V.

Note that the set T_O contains the operation types associated with tests and ordinary operations, whereas the set T_X contains the types of special nodes representing conditional constructs.

In general, we refer to some node of the DFG as an operation $v_k \in V$. However, we sometimes need to emphasize, in the notation, the distinction between a test, a conditional and an ordinary operation. When necessary, a test is designated by t_k, a conditional by c_k and an ordinary operation by o_k. Most of the time, however, we use the terms *test* and *conditional* interchangeably, since they refer to closely related notions.



```
[k]  x := i1 – i2;
[l]  y := i3 – i4;       I
[t1] if (x > i5)
[m]        z := x + i6;  J
     else
[n]        z := x – y;   K
[p]  o1 := z + i7;
                         L
[q]  o2 := i8 + i9;
```

FIGURE 2.2. A behavioral description and its DFG

In order to help us to keep track of code motion, an auxiliary graph is defined below. This graph is a condensation of the DFG.

DEFINITION 2.6

A *basic–block control flow graph* BBCG = (U, F) is a directed graph, where U is the set of nodes, representing basic blocks or junctions, and $F \subseteq U \times U$ is the set of edges, representing the flow of control.

The BBCG is derived during a depth–first traversal of the DFG and it is built as follows. All operations in the DFG which are enclosed between a pair of branch and merge nodes controlled by the same conditional are condensed in the form of a basic block in the BBCG. All branch (merge) nodes in the DFG controlled by the same conditional are condensed into a single branch (merge) node in the BBCG domain. Similarly, all input nodes are contracted to a single *source* node and all output nodes to a *sink* node.

Figure 2.3 shows both the DFG and the BBCG for the description in Figure 2.1. In the BBCG, circles represent basic blocks and each BB is associated with a set of operations in the DFG. Branch and merge junctions in the control flow are explicitly represented, in the BBCG domain, by branch (B) and merge

(M) nodes, which are drawn as pentagons. Triangles denote the source and the sink nodes.

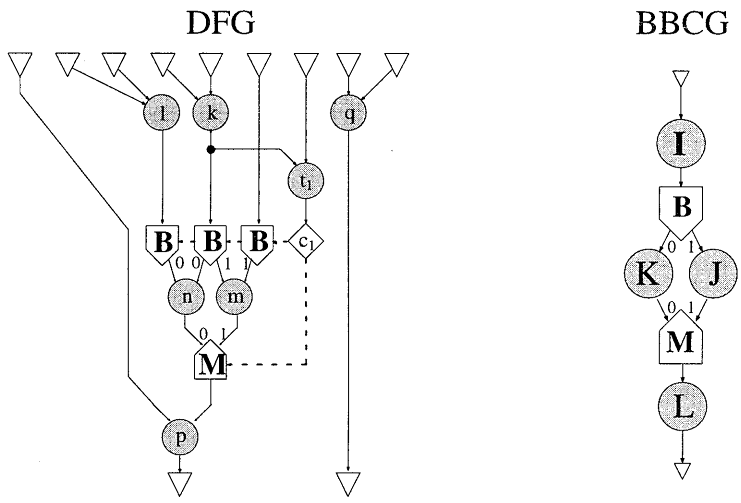


FIGURE 2.3. A DFG and its associated BBCG

In general, we designate by u_i an arbitrary node of the BBCG, although we sometimes distinguish a BB from branch and merge junctions in the notation. A basic block is denoted as BB_i, whereas B_k and M_k respectively represent branch and merge junctions associated with some conditional c_k.

Note that, for a given branch node B_k, the flow of control reaches a different BB, depending on the outcome of conditional c_k. In Figure 2.3b, this is marked by labeling the edges leaving the branch node with “1” and “0”, depending on whether the outcome of c_1 is true or false, respectively. We sometimes make a distinction on which BB is reached, as follows. Given a branch node B_k, we denote as succ(B_k, true) the successor of B_k when the outcome of conditional c_k is true. Similarly, succ(B_k, false) designates the successor when the result of c_k is false.

A path $\langle u_0, u_1, \ldots, u_k \rangle$ in the BBCG=(U, F) with u_0 = source and u_k = sink is called a *control path*. As the outcome of the tests is dependent on some setting of the data, the taken control path can be determined at execution time only. The set of operations enclosed by the BBs contained in a given control path is here called a *trace* or *execution instance*. In other words, each path in the BBCG corresponds to exactly one trace in the DFG. We say that an *operation* o_n *executes on a control path* p if o_n is in the trace associated with p.

The relationship between the DFG and the BBCG is kept by means of so–called *links*. A link connects one node v in the DFG with a node u in the

BBCG. Links play an important role in modeling code motion and for this reason we will especially be interested in the links connecting ordinary operations and tests to BBs. From now on, we will use the notation $o_n \overset{\lambda}{\twoheadrightarrow} BB_i$ to designate that an operation o_n is connected to a basic block BB_i by means of link λ. This notion is illustrated in Figure 2.4, where each arrow represents a link. Only links from operations and tests are shown. Note that, in Figure 2.4, each operation is linked to the BB where it was initially described. We say that each operation is linked to its *initial* BB, or equivalently, that a set of *initial links* is defined. The fact that an operation o_n is *not* linked to a basic block BB_i is denoted as $o_n \not\twoheadrightarrow BB_i$ throughout this thesis.

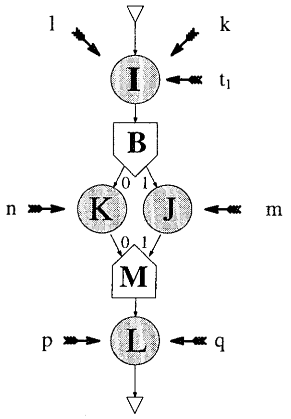


FIGURE 2.4. The initial links for the example in Figure 2.2

Now we formalize some classical notions from the compiler–technology domain [7], by casting them into our own representation. The notions of domination, postdomination and control equivalence are useful during bookkeeping. The definitions below assume an acyclic BBCG=(U, F) with single–source and single–sink nodes and in which $u_i, u_j \in U$.

DEFINITION 2.7

Node u_i *dominates* node u_j, written u_i dom u_j, if every path from the source to u_j includes u_i.

DEFINITION 2.8

Node u_j *postdominates* node u_i, written u_j pdom u_i, if every path from u_i to the sink includes u_j.

DEFINITION 2.9

Nodes u_i and u_j are *control equivalent*, written u_i equiv u_j, if and only if u_i dominates u_j and u_j postdominates u_i or vice–versa.

Note that, in the BBCG shown in Figure 2.3, BB I dominates BB K, but BB K does not postdominate BB I, while BBs I and L are control equivalent.

2.2.2 Modeling the data path

The data path is modeled as a set of interconnected components, such as functional units, multiplexers and memory elements, which are here called *modules*. The data path is represented in the form of a graph, as defined below. Although this graph is a very abstract model for the RTL structure, it is sufficient for the purposes of this thesis.

DEFINITION 2.10
A *network graph* NWG=(M, W) is an undirected graph, where M is the set of nodes, representing modules, and $W \subseteq M \times M$ is the set of edges, representing the connectivity of the modules.

The modules are classified according to their function, such as adder, multiplier, register, etc. This leads to the notion of *module type*. The mapping between modules and types is formalized as follows.

DEFINITION 2.11
Let M be a set of modules and T_m be a set of module types. The function $\mu : M \mapsto T_m$ maps a module m to a module type t_m.

In HLS, the relation between operation types and module types is usually described in the form of a *library.* For example, given a set of module types T_m = {adder_subtracter, ALU, ripple_carry_adder, multiplier}, we could instantiate modules by choosing from the set T_m. A plus–operation could be performed on any module of the three first types and a multiplication, on the last one only. This notion is formalized below.

DEFINITION 2.12
The *operation mapping* function $\beta : T_o \mapsto P(T_m)$ maps each operation type $t_o \in T_o$ to the subset of the available module types on which it can be executed.

For the example given above, we could thus write: $\beta(\times)$ = {multiplier} and $\beta(+)$ = {adder_subtracter, ALU, ripple_carry_adder}.

Since an operation can be executed on various module types, it is the task of module selection to choose a subset of suitable module types, say $T_s \subseteq T_m$, and to assign each operation to a single module type $t_s \in T_s$, according to the formulation below.

DEFINITION 2.13

Let $V_o = \{v \in V \mid \omega(v) \in T_o\}$. The *operation assignment* function $\tau : V_o \mapsto T_s$ maps each operation v to a selected module type t_s such that $t_s \in \beta(\omega(v))$.

2.2.3 Modeling the control unit

As a consequence of the application of ILP techniques during scheduling, the operations from different parts of the DFG are packed together for simultaneous execution. In this thesis, the operations executing in parallel are represented in the form of a *state*, and an execution sequence is represented by *transitions* between states. When a given state s_k is reached, all operations associated with it are executed simultaneously. These notions are also cast into a graph representation, as follows.

DEFINITION 2.14

A *state machine graph* SMG = (S, T) is a directed graph, where S is set of nodes, representing states, and $T \subseteq S \times S$ is the set of edges, representing transitions.

The SMG can be seen as a prototype for the state transition diagram of the underlying finite state machine (FSM), whose formal definition can be found, for instance, in [16].

We assume that the SMG has dummy nodes called *source* and *sink* states. The duration of every state is one clock cycle, except for the sink and the source, whose execution is assumed to take no time.

Recall from Section 1.4 that our goal is to focus on design problems subject to a global time–constraint T_c. From a DFG perspective, the time constraint can be interpreted as follows. The difference between the time when the input values are available and the time when the output values are determined can not exceed T_c clock cycles. The availability of input values in the DFG can be interpreted in the SMG domain as a transition from the source to an initial state s_0, whereas the availability of output values can be interpreted as a transition from a state s_{k-1} to the sink. As a consequence, the time to execute all the operations in every path from s_0 to s_{k-1} must not exceed T_c clock cycles. This notion is formalized as follows.

DEFINITION 2.15

Given a simple path $p = \langle source, s_0, s_1, \ldots, s_{k-1}, sink \rangle$ in a SMG, the *schedule length of path* p, written L_p, is the number k of states included in path p.

Our main goal is to find SMGs for which the inequality $\max(L_p) \leq T_c$ holds for every path p. For simplicity, we sometimes refer to the path with greatest schedule length as the “longest” path in the SMG.

Given DFG and a set of resource constraints, several different SMGs can be synthesized, depending on how the operations are packed together. Figure 2.5 illustrates alternative SMGs, synthesized from the DFG in Figure 2.2b.

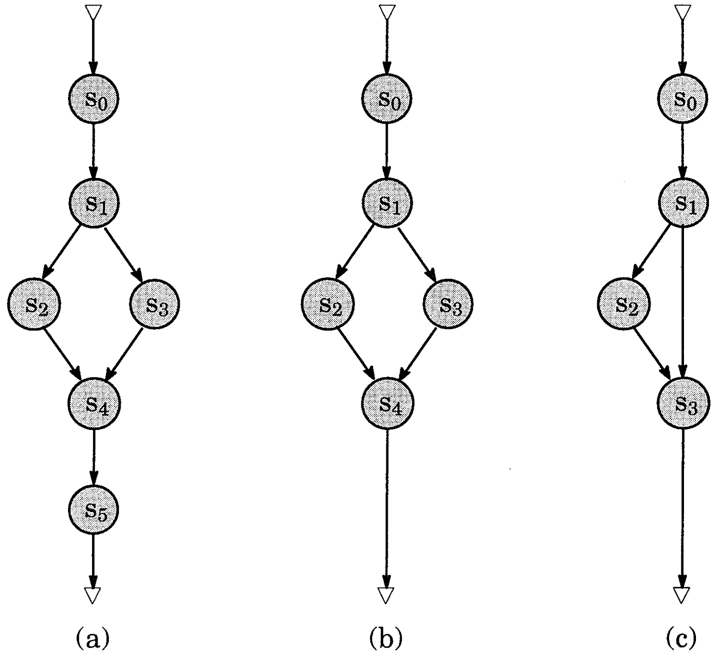


FIGURE 2.5. Alternative SMGs for the DFG in Figure 2.2b

To construct the first SMG in Figure 2.5a, exploitation of parallelism is restricted within BB boundaries. The remaining alternatives, however, are constructed using code motion. The operations executed in each of the states in those SMGs are given in Table 2.1. Although all alternatives comply with resource and precedence constraints, they have different properties. The choice of the most convenient alternative is based on the design objectives. For example, if we want to minimize completion time, the SMG in Figure 2.5a is of inferior quality when compared to the others. If we also want to reduce the number of states as a second objective, the SMG in Figure 2.5c should be preferred.

TABLE 2.1 Operations executed in each state for the SMGs in Figure 2.5

SMG	s_0	s_1	s_2	s_3	s_4	s_5
(a)	k	l, t_1	n	m	p	q
(b)	k, q	l, t_1	n	m	p	–
(c)	k, q	l, m, t_1	n	p	–	–

In the context of code generation, the number of states in the SMG correlates with code size. The application of code motion may lead to SMGs with a larger number of states. In the compiler–technology arena, this is known as *code expansion* or *code explosion*. In this thesis, we sometimes use these customary terms informally to mean an increase in the number of states.

In summary, the DFG is the starting point for synthesis. The NWG is obtained by performing allocation and module selection. Our goal is to obtain a SMG complying with both design constraints and objectives. The BBCG is a useful intermediate representation which assists on the construction of a SMG from the DFG, under the resource constraints imposed by an already defined NWG.

2.2.4 Modeling timing

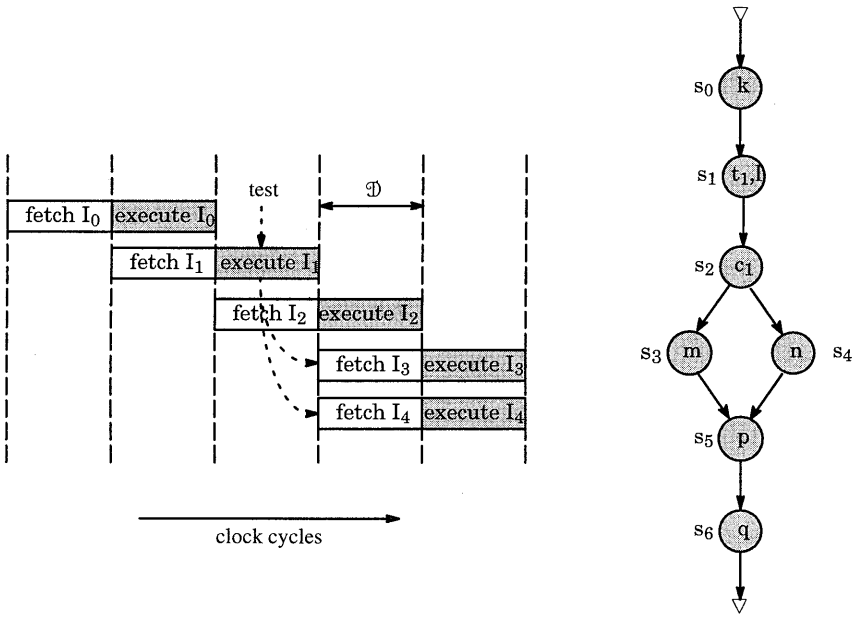
The amount of time spent to execute an operation of the DFG on a module of the data path is usually modeled with the notion of *delay*. Different operations might have different delays. As a consequence, delays are usually normalized to the clock cycle and are represented by positive real numbers.

In general, the execution delay depends on both the operation type and on the module type (e.g. a plus–operation executes faster on a carry look–ahead adder than on a ripple–carry adder). This notion is formalized below.

DEFINITION 2.16
The *execution delay* is a function $\delta : \mathrm{T_o} \times \mathrm{T_m} \mapsto \mathbb{R}^+$, such that $\delta(\mathrm{t_o}, \mathrm{t_m})$ represents the number of clock cycles taken by module type $\mathrm{t_m}$ to complete the execution of an operation of type $\mathrm{t_o}$.

Timing can also be affected by the advance choice of a controller. Although the synthesis of the controller is performed at a later phase of the design flow, the fact that a controller architecture is chosen beforehand imposes extra constraints for HLS. Although this problem can be postponed by performing a re–scheduling during the synthesis of the controller [42], a more elaborate approach is proposed in [36] for addressing this issue *during* HLS. This is performed by extending the DFG representation so as to capture the effects of the advance choice of a controller. In the following, we illustrate how one of this effects can be captured in our DFG representation. A detailed discussion of this issue can be found in [36].

Assume that the symbolic FSM generated by HLS is mapped to the microcoded controller of an ASIP. The operations assigned to a state of the symbolic FSM are mapped to a *micro–instruction* format. A micro–instruction consists of a number of operation fields. Each field is associated with the control signals steering the modules involved in the execution of a given operation. The micro–instructions are kept in a control–store memory, from where they are fetched prior to execution. It might be necessary to overlap the micro–instruction fetch in the control unit and the execution in the data path, for reducing the overall critical path. As a result, the whole digital system may be organized as a pipeline, by the proper insertion of registers to isolate pipeline stages. As a result of the latency of the pipeline, a delay occurs between the time in which the outcome of a test operation is made available to the control unit and the time when it actually influences the data path. This time interval is known as *delay slot,* which we denote as $\mathfrak{D}$. Given a pipeline with n stages, the delay slot is equal to n − 1 clock cycles [36].



(a) (b)

FIGURE 2.6. The notion of delay slot

If the effect of the latency of the pipeline is overlooked during HLS, the generated FSM may turn out *not* to satisfy the time constraint during microcode generation. To overcome this problem the effect of the latency of a pipeline can be modeled during HLS by associating a delay $\mathfrak{D}$ to every conditional c_k in the DFG. This notion is illustrated in Figure 2.6. A simple pipeline with 2 stages is sketched in Figure 2.6a, where the k^{th} micro–instruction is denoted by I_k and the delay slot is indicated by $\mathfrak{D}$. In the first stage, a micro–instruction is fetched and in the second, it is executed. In Figure 2.6b, a SMG is constructed such that the delay slot is taken into account. Note that the operations executing in state s_k (Figure 2.6b) correspond to micro–instruction I_k (Figure 2.6a). Assume that a test is executed in micro–instruction I_1. Depending on the outcome of test t_1, either I_3 or I_4 will be executed. Since one of them has to be fetched, the operations associated with the fetched micro–instruction will be executed one cycle later, after a delay $\mathfrak{D}$.

Recall that, after module selection is performed, the operation assignment τ is determined. This fact allows us to combine the effects of execution delay and delay slot into the concept of operation delay, as defined below:

DEFINITION 2.17

The *operation delay* is a function $d : V \mapsto \mathbb{R}^+$, such that:

$$d(v) = \begin{cases} \delta(\omega(v), \tau(v)), \text{if}\, \omega(v) \in T_0 \\ \mathfrak{D}, \text{if}\, \omega(v) = \text{conditional} \\ 0, \text{if}\, \omega(v) = \text{branch} \vee \omega(v) = \text{merge} \end{cases} .$$

Note that an operation delay is a fraction of a clock cycle. If $d(v) > 1$, an operation v is said to be a *multicycle* operation, which means that its execution takes longer than a single cycle. On the contrary, when $d(v) < 1$, more than one operation could be executed within a clock cycle. This leads to the notion of *chaining*, which is the execution of data–dependent operations within a clock cycle. The occurrence of multicycling or chaining depends on the choice of value for the clock cycle. For instance, assuming that the execution delay of a multiplication is twice the execution delay of an addition, the multiplication in Figure 2.7a is a multicycle operation, while the additions in Figure 2.7b are chained.

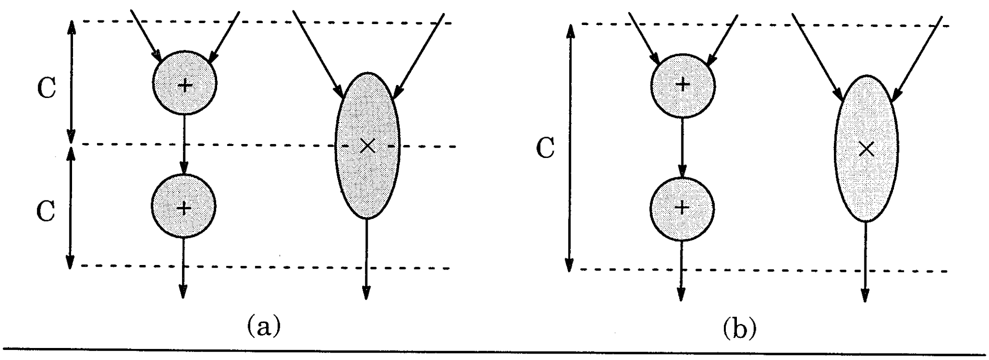


FIGURE 2.7. The notions of multicycling and chaining

A module can be designed to operate as a *pipeline*, which means that it can process new input data while the execution for some old input data is still unfinished, in such a way that the processing of different data can be overlapped.

DEFINITION 2.18

The *data introduction interval* is a function dii : $T_m \mapsto \mathbb{N}^+$, such that dii(t_m) represents the minimal number of clock cycles required between different executions of a same module of type t_m.

Although each operation takes d(v) cycles for completion, successive execution of operations can be issued each dii(τ(v)) cycles. If dii(τ(v)) $<$ d(v) then the execution is pipelined, otherwise the module does not operate as a pipeline, since it just starts the execution of the next operation after the completion of the previous one.

Part of the formulation in this section summarizes some of the major achievements of years of research in design representation for HLS, which are borrowed from many sources such as [16][21][28][63][66]. Other notions, however, are especially introduced as a consequence of our focus on ILP techniques applied to synthesis. The described design representations are implemented using the so–called NEAT System [29], an object–oriented framework for HLS.

2.3 Formulation of the optimization problem

Throughout this thesis, it is assumed that allocation and module selection have already been performed. As a result, the selected module types, the number of modules per type and the assignment of operations to module types are considered to be known and fixed. This is formalized as follows.

Recall that T_m designates the set of module types in the library, and that $T_s \subseteq T_m$ is the set of *selected* module types. Also, remember that the function $\tau : V \mapsto T_s$ maps each operation on a selected module type. Finally, let n(t_s) be the number of selected modules of type t_s. From now onwards, it is assumed that T_s is determined, that n(t_s) is known for each $t_s \in T_s$ and that the mapping τ has been accomplished. These parameters represent *resource constraints* for further synthesis steps. Without loss of generality, the application of resource constraints is restricted, in this thesis, to functional units. Extensions are suggested in Chapter 7.

Although the relation between a DFG and a SMG is described more formally in Chapters 3 and 4, we need to anticipate some notions thereof. Our goal is to generate a SMG such that each operation of the DFG is associated with some state. Assume that operations o_m and o_n are associated with states s_i and s_j, respectively. Essentially, operations are assigned to states such that:

- If o_n is data dependent on o_m, then there must be a simple path from the source to s_j that includes state s_i, such that the execution of operation o_n starts only after the execution of operation o_m is completed. If this condition holds for all data dependent operations, we say that the precedence constraints are obeyed.
- The resources needed for the execution of the operations associated with a given state can not exceed the available number of resources. When this holds for every state, we say that the resource constraints are satisfied.

The subproblems stemming from HLS can usually be formulated as combinatorial optimization problems [50]. An *instance* of an optimization problem is a pair ($\mathcal{F}$, c), where the *solution space* $\mathcal{F}$ is the set of all feasible solutions, and the cost function c is a mapping $c : \mathcal{F} \mapsto \mathbb{R}$. An *optimization problem* (OP) is a collection of problem instances. An OP can be formulated either as a minimization or as a maximization problem. The goal is to find a solution for which the cost function reaches either its minimal or its maximal value, depending on whether a minimization or a maximization problem is envisaged.

One way of tackling the synthesis of time–constrained designs is to formulate an OP whose goal is to minimize completion time and to search for a solution satisfying the time constraint. The completion time of a path in the SMG depends on how the operations are scheduled in the states included in that path. As a result, cost is evaluated in this thesis as a function of schedule lengths of paths. In the following formulation, c is a monotonically increasing function and L_p denotes the schedule length of a path p in the SMG such that source $\xrightarrow{p}$ sink.

OPTIMIZATION PROBLEM

Given a DFG, a set T_s of selected module types, a number of available modules $n(t_s)$ for each module type t_s and an operation mapping τ, find a SMG in which the precedence constraints of the DFG are obeyed and the resource constraints are satisfied for each module $t_s \in T_s$, such that the cost function $c(L_1, L_2, \ldots, L_n)$ is minimized.

The solution of this OP implies the solution of resource–constrained scheduling, a well–known intractable problem. In this thesis, instead of relying on an exact algorithm to solve the OP, we assume that promising solutions are explored via *local search* [50]. Local search is based on the notion of *neighborhood*. Given an instance ($\mathcal{F}$, c), a neighborhood is a mapping $\mathcal{N} : \mathcal{F} \mapsto P(\mathcal{F})$. This leads to the notion of *local optimum* with respect to a given neighborhood, in contrast to the notion of *global optimum*, which refers to the whole scope of the solution space. Essentially, the principle of a local search algorithm is to explore a given neighborhood of the current solution (or of a group of solutions) and search for a better solution, until some stopping criterion is satisfied. Examples of local search algorithms are iterative improvement, tabu search, simulated annealing and genetic algorithms (for an overview, see [69]).

Some solutions, although feasible, are deliberately not generated, for reasons of efficiency. Reconsider the solutions in Figure 2.5, for instance. Since code motion in general leads to shorter schedule lengths, it would be unnecessary to generate a solution like the one in Figure 2.5a, where code motion is prohibited, as far as minimization of schedule lengths is the primary objective. Therefore, a given synthesis method can be conceived such that some feasible solutions of poor quality are never constructed. As a result, not all feasible solutions in the solution space can be explored. In order to mark the distinction, we call *search space* the set of feasible solutions that can actually be explored during the search.

It should be noted that, unlike most HLS approaches, where greedy heuristic algorithms are used to come up with a single solution, our choice of a local search approach is more flexible, since it provides a way of *exploring* alternative solutions. This is especially important because some solutions to the OP might not satisfy a given time constraint and have to be ruled out.

Figure 2.8 summarizes how the solution of the OP is placed in the context of a synthesis methodology. If the inequality $\max(L_p) \leq T_c$ is not satisfied for some path p, previous decisions in the HLS design flow should be revoked and new decisions should be taken, through an iterative process, until the time constraint is met.

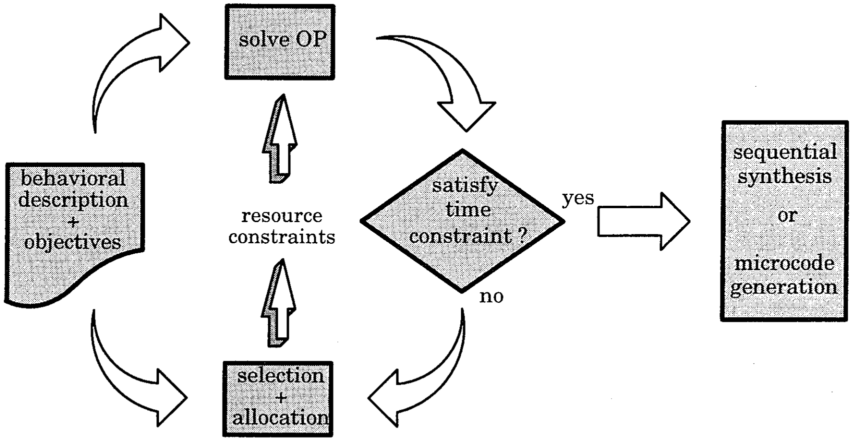

solve OP
behavioral description + objectives
resource constraints
satisfy time constraint ?
yes
no
sequential synthesis or microcode generation
selection + allocation

FIGURE 2.8. A design methodology for synthesis

Chapter 3 A constructive approach

3.1 Motivation

To tackle the optimization problem defined in the previous chapter, the following main difficulties have to be faced when conditionals and loops are present in the behavioral description:

- the NP–completeness of resource–constrained scheduling;
- the limited parallelism of operations enclosed by basic blocks, such that available resources are poorly utilized;
- the possibility of state explosion because the number of control paths may explode in the presence of conditionals;
- the limited resource sharing of mutually exclusive operations, due to the late availability of test results.

Most methods address these issues as separate subproblems (BB scheduling, code motion, code size reduction, conditional resource sharing). Since the optimization goals of different subproblems may conflict, the result typically depends on the order in which the subproblems are solved. Besides, most methods apply *different heuristics* to each subproblem, as if they were independent. An heuristic is used to determine the order of the operations during scheduling (like the many flavors of priority functions), another to decide whether a particular code motion is worth doing [22][48][54], yet another for reducing the number of states [71]. The application of an aggregate of unrelated heuristics makes it difficult to control the quality of the final result. Therefore, these approaches might miss optimal solutions.

This chapter proposes a formulation [56] to encode potential solutions to the entire optimization problem, instead of addressing the interdependent subproblems separately. The formulation abstracts from the linear–time model commonly used in HLS and allows us to concentrate on the order of operations and on the availability of resources. Different priority encodings are used to induce alternative solutions and many solutions are generated and explored. The basic idea is to keep high–quality solutions in the search space when ILP techniques like code motion, speculative execution and loop pipelining are applied.

Before presenting our own approach to tackle the OP, later referred as the *constructive approach*, we review related approaches to similar problems.

3.2 Related high–level synthesis approaches

In path–based scheduling (PBS) [8][12] a so–called as–fast–as–possible (AFAP) schedule is found for each path independently, provided that a fixed order of operations be chosen in advance. Due to the fixed order and to the fact that scheduling is cast as a clique covering problem on an interval graph, code motions resulting in speculative execution are not allowed. The original method has been recently extended to relieve the fixed order [9], but reordering of operations is performed inside BBs only. Reordering is not allowed across branch junctions, because this would destruct the notion of interval, which is the very foundation of the PBS technique. Consequently, although reordering improves the capability of efficiently handling more complex data–flow, the method cannot support speculative execution, which limits the exploitation of parallelism with complex control flow [41]. This limitation is relieved in tree–based scheduling (TBS) [34], where speculative execution is allowed and the AFAP approach is conserved by keeping all paths on a tree. However, since the notion of interval is lost, an heuristic list scheduler is used to fill states with operations.

Condition vector list scheduling (CVLS) [71][72] allows code motion across branch and merge junctions, and supports some forms of speculative execution. Although it is shown in [53] that the underlying mutual exclusion representation is limited, the approach would possibly remain valid with some extension of the original condition vector or with some other alternative, such as the representations suggested in [6] and [53].

A hierarchical reduction approach (HRA) is presented in [37]. A DFG with conditionals is transformed into an "equivalent" DFG without conditionals, which is scheduled by a conventional scheduling algorithm. Code motion across merge points is allowed, but speculative execution is not supported.

In [54] an approach is presented where code–motions are exploited. At first, BBs are scheduled using a list scheduler and, subsequently, code motions are allowed. One priority function is used in the BB scheduler and another for code motion. Code motion is allowed only inside windows containing a few BBs to keep the runtime low, but then iterative improvement is needed to avoid restricting too much the kind of code motions allowed.

Among those methods, only PBS is exact, but it solves a partial problem where speculative execution is not allowed. TBS and CVLS address BB scheduling and code motion simultaneously, but use classical list scheduler heuristics. In [54] a different heuristic is applied to each subproblem. All those methods may exclude optimal solutions from the search space.

In [52], an exact method is presented for solving a resource–constrained scheduling problem, which is entirely modeled in Boolean form. Code motion is largely supported. However, some traces are scheduled in such a way that they would lead to infeasible solutions, which requires a backtracking procedure for trace validation. As a consequence of being an exact method and due to the lack of efficient pruning, the reported runtime is large. Therefore, the use of this technique in the early (more iterative) phases of a design flow is unlikely.

3.3 Related approaches in the compiler arena

In Trace–scheduling (TS) [22] a main trace is chosen to be scheduled first and independently of the others, then another trace is chosen and scheduled, and so on. First, resource unconstrained schedules are produced and then heuristically mapped to the available resources. TS does not allow code motion between traces. The downside of TS is that its main–trace–first heuristic works well only in applications whose profiling shows a highly–predictable control flow (e.g. in numerical applications).

Percolation Scheduling (PS) [48] defines a set of semantics–preserving transformations which convert a program into a more parallel one. Each primitive transformation induces a local code motion. PS is an iterative neighborhood scheduling algorithm in which the atomic transformations (code motions) can be combined to permit the exploration of a wider neighborhood. Heuristics are used to decide when and where code motions are worth doing (priorities are assigned to the transformations and their application is directed first to the “important” part of the code).

The most important aspect of PS is that the defined primitive transformations are potentially able to expose all the available instruction–level parallelism. Another system of transformations is presented in [27] and it is based on the notion of regions (control–equivalent BBs). Operations are moved from one region to another by the application of a series of primitive transformations. Since the original PS is essentially not a resource constrained parallelization technique, it is extended with heuristic mapping of the idealized schedule into the available resources [47][51].

The drawback of the heuristic mapping to resources performed by both TS and PS is that some of the greedy code motions have to be undone [19][46], since they can not be accommodated within the available resources.

More efficient global resource–constrained parallelization techniques are reported [19][46][62], whose key idea is a two–phase scheduling scheme. First, a set of operations available for scheduling is computed globally and then heuristics are used to select the best one among them.

In [19], a *global* resource–constrained percolation scheduling (GRC–PS) technique is described. After the best operation is selected, the actual scheduling takes place through a sequence of PS primitive transformations, which allow the operation to migrate iteratively from its original location to its final position.

A global resource–constrained selective scheduling (GSS) technique is presented in [46]. As opposed to GRC–PS, the *global* code motion of the selected operation is performed *in one step*, instead of applying a sequence of local code motions. The results presented in [46] give some experimental evidence that, although PS and GSS achieve essentially the same results, GSS seems to lead to smaller compilation times.

3.4 How our contribution relates to previous work

On the one hand, we keep in our approach some of the major achievements on resource–constrained scheduling in recent years, as follows.

- Like most global scheduling methods [3][19][46][62], our approach also adopts a global computation of available operations. However, our implementation is different [59], since it is based on a DFG, unlike the above mentioned approaches.
- We perform global code motions in one step, in a way similar to [46], but different from [19] and [27], which apply a sequence of primitive transformations.

On the other hand, our approach distinguishes itself from related work, as follows:

- Our formulation is different from all the methods above in the way it uses the notion of priority for generating schedules. Instead of using one heuristically established priority function [16], *many* priority encodings are generated by an external (and therefore tunable) search engine (see Section 3.5), thereby allowing the exploration of alternative solutions.
- Unlike most global resource–constrained approaches [3][19][46][62], we include support for exploiting downward code motion (see Section 5.1).
- We propose a technique to exploit the notion of state equivalence during scheduling by merging equivalent states on the fly. The technique relies on

information usually overlooked by traditional scheduling methods (see Section 4.5).

- We present a new code–motion pruning technique, which exploits precedence and resource constraints to prevent inefficient code motions (see Section 5.2).

3.5 An overview of the constructive approach

An approach could be envisaged where no restriction is imposed beforehand neither on the kind of code motion, nor on the order the operations are selected to be scheduled. In this section, we will introduce an approach, which is indeed largely free from such restrictions, henceforth referred to as the *constructive approach.*

An outline of our approach is shown in Figure 3.1. Solutions are encoded by a priority encoding Π, which is essentially a more or less arbitrary permutation of the operations in the DFG. A solution *explorer* creates the priority encodings. A solution *constructor* builds a solution for each priority encoding and evaluates its cost. The explorer looks for the solution with lowest cost by means of a local search algorithm and decides about time–constraint satisfiability.

While building a solution, the constructor needs to check many times whether a given operation is available for scheduling in a given state or whether a certain code motion needs compensation code. These tests are modeled as Boolean queries and are directed to a so–called *Boolean oracle* (the term was coined in [10]) which allows us to abstract from the way the queries are implemented.

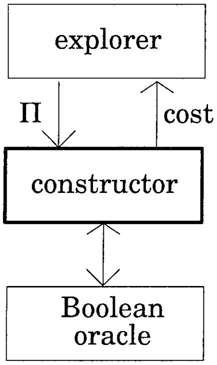


FIGURE 3.1. An outline of the approach

Note that the approach consists of co–operating but *orthogonal* engines. This has the advantage of allowing further tuning, since one engine can be modified or replaced without the need to change the other. For instance, an explorer based on genetic algorithms could be replaced by another based on simulated annealing; the Boolean oracle could be based on binary decision diagrams [16] or on simpler one–hot encoding techniques like condition vectors [71]. The constructor can be modified to include various kinds of pruning techniques.

Another advantage is that the approach allows to trade search time against solution quality, since the number of explored solutions can be, in general, controlled by the parameters of the search method. In the early (more iterative) phases of a design flow a quick local search can be used, while broader neighborhoods can be explored in the final optimization phase.

The approach is designed such that the following properties hold:

- **The priority encoding is not determined by greedy heuristics:** The explorer determines a priority encoding depending on the criteria of a given local search algorithm, instead of relying on greedy priority heuristics, like "critical–path–first", "main–trace–first", etc.
- **The constructor manages code motion:** Although constrained by the available resource and by the dependences, code motions are induced by the priority encoding and no limitation on type, scope or amount of code motions is imposed beforehand.
- **Decisions are exclusively made in the explorer:** The constructor simply generates a solution for each priority encoding. All the decisions are made by the explorer based on the cost evaluated by the constructor.
- **Pruning is used to discard low–quality solutions:** The constructor exploits precedence and resource constraints in order to avoid the generation of solutions which certainly do not lead to lower cost. This is performed by preventing inefficient code motions.
- **Only feasible solutions to the OP are generated:** The construction of a solution is determined by the priority encoding in combination with precedence and resource constraints, such that only solutions satisfying all these constraints are constructed.
- **The notion of state prevails over the notion of time step:** Unlike most HLS approaches, our constructor assigns operations directly to states and defines state transitions, instead of scheduling operations on a linear sequence of time steps.

In the sequel, we refine the description of our approach by focusing on each of the co–operating engines.

3.6 The priority encoding

Most of the issues described in this thesis are performed in the constructor, such as the application of ILP techniques and the handling of resource and precedence constraints.

Precedence constraints impose a restriction on the set of operations to be scheduled in a given state, since the values used by some operations may not yet be computed. The operations which can potentially be scheduled in a given state, without violating the semantics of the behavioral description, are called *available operations* [3]. They are also known as unifiable operations [19] or ready operations [62]. The *set of operations available for scheduling at a given state* s_k is denoted as A_k. The evaluation of A_k depends, not only on the precedence constraints, but also on operation delays (see Section 4.3). However, not all operations in A_k can be scheduled in state s_k simultaneously, due to resource constraints. Therefore, a subset of the operations in A_k has to be selected.

The assignment of operations to each state s_k in the SMG depends on the order governing the selection of operations from the set A_k. As a consequence, a potential solution to the OP can be induced by assigning a *priority* to each operation. For this reason, the interaction between the explorer and the constructor relies on encoding distinct solutions in the form of distinct priority encodings.

Given a DFG=(V, E), a priority encoding is essentially a permutation Π of operations from V. The notion of priority is associated with the relative position of operations in Π. Let $\Pi(v_i)$ denote the position of some operation v_i in permutation Π. Operation v_i has the priority over v_j, written $v_i <_\Pi v_j$, if $\Pi(v_i) < \Pi(v_j)$.

3.7 The constructor

The solution constructor consists of two main engines, a *scheduler* and a so–called *parallelizer*, as shown in Figure 3.2. The role of the parallelizer is to manage code motion and loop pipelining. It traverses the BBCG and appoints a *current state* s_k to be scheduled. After the assignments implied by the scheduling of that state are completed, a set of states called *next states* is determined. As a result, the parallelizer packs operations into states and defines the state transitions, thereby generating the SMG on the fly. The parallelizer maintains, within the set A_k, the operations available for scheduling in the current state.

The task of the scheduler is to select, from the set A_k, one operation v_i to be executed in state s_k. The scheduler then returns the selected operation and the parallelizer updates the set A_k accordingly. This interaction proceeds until all resources available are occupied at state s_k or the set A_k is empty.

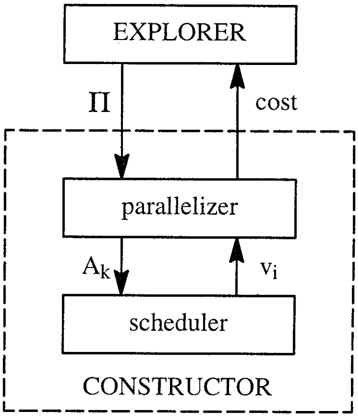


FIGURE 3.2. An outline for the solution constructor

Although most global resource–constrained scheduling methods rely on the notion of available operations, they differ on how this set is used to accomplish parallelization. In [46] greedy scheduling heuristics are used to prioritize the set A_k. In [3] the handling of resource constraints is orthogonal to the software pipelining algorithm. This separation allows scheduling heuristics to be modified for an efficient management of resources. In our approach, not only scheduling is orthogonal to the parallelization, but also heuristics are *removed from the scheduler* and placed in the explorer. This allows proper exploration of alternative solutions.

3.7.1 The scheduler

Operation selection

In our approach, each set A_k is *ordered* according to some priority encoding Π. The scheduler selects operations from the set A_k, whose ordering is induced by $<_\Pi$. Given a state s_k and an ordered set A_k, the scheduler selects the first operation $v_i \in A_k$ that satisfies resource constraints.

Since heuristics are pushed to the explorer engine, our selection mechanism depends only on the linear order $<_\Pi$ and on the free resources, but does not depend on built–in scheduler heuristics (as opposed to classical methods [3][19][46][62]). This makes it easy to compare the operations to be scheduled in distinct states without actually scheduling all of them. Such predictability can be used for further optimization, as will be shown in the next chapter (see Section 4.5.2).

Management of resources

Occupation of resources is modeled by resource utilization functions. The *resource utilization function* associated with a basic block BB_j, is a mapping $r_j : \mathbb{N} \times T_S \mapsto \mathbb{N}$ such that $r_j(k, t_S)$ represents the number of resources of a given type t_S which are occupied in some state $s_{j,k}$ within a basic block BB_j. The *joint resource utilization* function associated with basic block BB_j is a mapping $R_j : \mathbb{N} \mapsto \mathbb{N}^{|T_s|}$ such that:

$$R_j(k) = (\ r_j(k, t_1), r_j(k, t_2), \dots, r_j(k, t_S), \dots, r_j(k, t_{|T_s|})\ ). \qquad (3.1)$$

We say that there exists some *free* resource when $r_j(k, t_S) < n(t_S)$. Each time an operation v_i is scheduled, $r_j(k, \tau(v_i))$ is incremented by one. Since v_i can be a multicycle operation and possibly executed on a pipelined resource, the occupation of a module, starting at some state $s_{j,k}$, is modeled by incrementing $r_j(n, \tau(v_i))$ by one, for each $n \in \mathbb{N}$, such that $k \leq n < k + dii(\tau(v_i))$. The joint utilization function is fundamental for the detection of equivalent states in the SMG, as will be shown in Chapter 4.

3.7.2 The parallelizer

State handling

One of the main tasks of the parallelizer is to perform a top–down traversal of the BBCG, while following the flow of tokens in the DFG. We say that the parallelizer *visits* basic blocks and that the BB being visited is the *current* BB. Given two distinct basic blocks, say BB_i and BB_j, if BB_i reaches BB_j, then BB_i is visited before BB_j and is never revisited.

The BBCG is used as a frame of reference for the construction of a SMG. Each basic block BB_j is modeled as a sequence of successive states $\langle s_{j,0}, s_{j,1}, \dots, s_{j,N}\rangle$ such that $indegree(s_{j,k}) = outdegree(s_{j,k-1}) = 1$, for $k = 1, 2, \dots, N$ and such that $s_{j,0} \neq$ source and $s_{j,N} \neq$ sink. We say that the parallelizer *appoints a state* within a BB when it creates a new state to which operations are not yet assigned. Scheduling assigns operations v_i to states $s_{j,k}$ within basic block BB_j.

Assume that a current state $s_{j,k}$ is appointed within the current BB, say BB_j. For assigning operations to state $s_{j,k}$, the parallelizer interacts with the scheduler, which selects them from the set $A_{j,k}$. The parallelizer assigns each scheduled operation, one after another, to state $s_{j,k}$ until no more available operations remain or no more resources are free. In this case, we say that the *state is scheduled.* When the current state is scheduled, the next states are determined. A next state may fall within BB_j itself or within some other BB still to be visited. In the first scenario, the next state $s_{j,k+1}$ becomes the current state and the process is repeated. In the second scenario, the visiting of BB_j is completed and the traversal of the BBCG resumes at another basic block, say BB_x. Then, the state $s_{x,0}$ becomes the current state, restarting the process.

Bookkeeping

The parallelizer is also in charge of detecting if a code motion requires compensation code. If this is the case, it evaluates in which BB the needed code has to be inserted. The detailed description of this procedure is given in the more appropriate scope of Chapter 4.

Availability analysis

The evaluation of which operations are available for scheduling, from now on referred to as *availability analysis,* is also performed in the parallelizer and is formalized in the next chapter. Assume that BB_i is the current BB. The parallelizer searches for available operations among those linked to BBs *reachable* from BB_i. The available operations discovered during this search are kept in the so–called *set of available operations at basic block* BB_i, which we denote as $\mathcal{A}_i$. Note that $\mathcal{A}_i$ contains operations that should execute on some path from BB_i to the sink. Besides, due to the effect of different operation delays, not all operations in $\mathcal{A}_i$ are available for scheduling at the same state. Some operations might be available at the current state $s_{i,k}$, whereas others may be available at some state $s_{i,k+x}$, reachable from $s_{i,k}$. For this reason, the parallelizer splits the set $\mathcal{A}_i$ into several subsets, say $A_{i,k}, A_{i,k+1}, \cdots, A_{i,k+x}$, each of them containing the operations available for scheduling at states $s_{i,k}, s_{i,k+1}, \cdots, s_{i,k+x}$, respectively. For simplicity, in the remainder of this chapter, we assume provisionally that all operations in the DFG have unit delay and, as a consequence, that the sets $\mathcal{A}_i$ and $A_{i,k}$ can be used interchangeably. This assumption is relaxed in Section 4.3.

As a result of availability analysis, the set $\mathcal{A}_i$ may contain operations from different BBs (possibly inducing code motion) or instances of operations belonging to different iterations of a loop (possibly inducing loop pipelining). Efficient ways of evaluating $\mathcal{A}_i$ are proposed in the literature [4][19][46], but they are based on a so–called control flow graph representation. In Chapter 4, we propose algorithms for availability analysis that are suitable for a DFG representation.

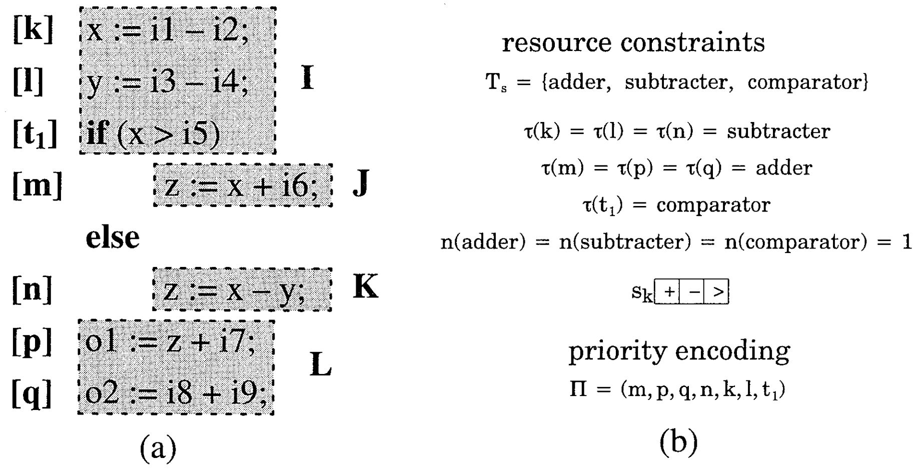
Code–motion pruning

On the one hand, it is convenient to *expose* as much parallelism as possible, by keeping a *global* view of the operations available for scheduling in set A_k. On the other hand, the more global the scope, the larger the search space becomes. As a consequence, we should also prune inferior solutions from the search space. Code–motion pruning is a technique included in the parallelizer for this purpose, and it will be explained in Chapter 4.

In early versions of our approach [56][57][60], we neither used the notion of available operations at some state s_k explicitly, nor we stressed the notion of current and next states. These notions were later incorporated, after the work presented in [4], since they allow us to extend the original approach with loop pipelining, as will be shown in Chapter 6.

3.7.3 An example

The process of construction is now illustrated with a simple example. The starting point is the behavioral description of Figure 2.2, which is repeated in Figure 3.3a for convenience. The resource constraints are given in Figure 3.3b, along with a priority encoding Π. The rectangle in Figure 3.3b is a pattern used throughout the example to represent resource utilization schematically at a given state s_k. Each field within the rectangle is associated with a different module type. Suppose that $d(t_1) = 1$ and that $d(v) = 1$ for every ordinary operation v.



```
[k]  x := i1 – i2;
[l]  y := i3 – i4;        I
[t1] if (x > i5)
[m]       z := x + i6;    J
     else
[n]       z := x – y;     K
[p]  o1 := z + i7;
[q]  o2 := i8 + i9;       L
```

(a)

resource constraints

T_s = {adder, subtracter, comparator}

$\tau(k) = \tau(l) = \tau(n)$ = subtracter

$\tau(m) = \tau(p) = \tau(q)$ = adder

$\tau(t_1)$ = comparator

n(adder) = n(subtracter) = n(comparator) = 1

s_k | + | – | > |

priority encoding

$\Pi = (m, p, q, n, k, l, t_1)$

(b)

FIGURE 3.3. Behavioral description and resource constraints for the example in Figure 3.4

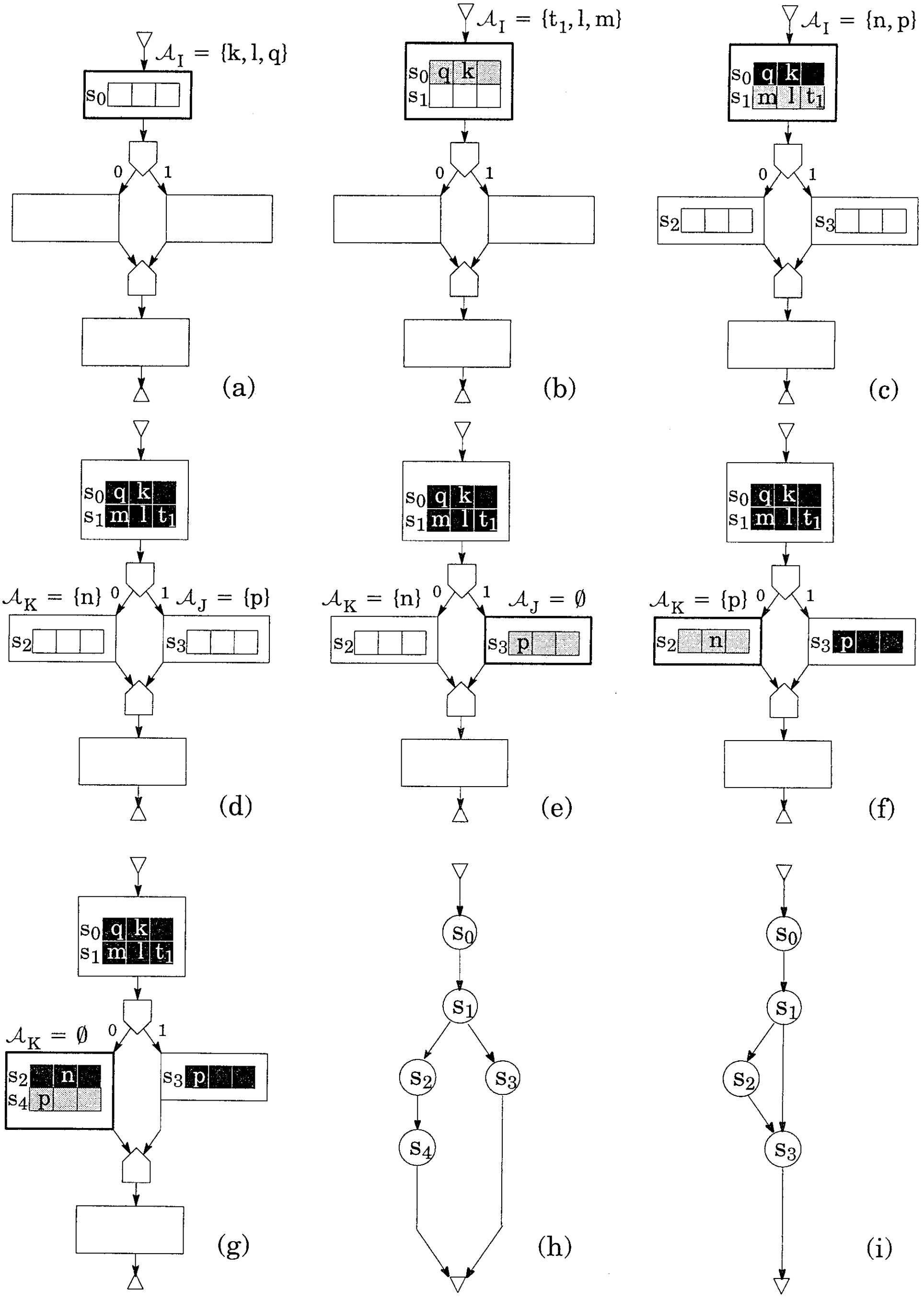


FIGURE 3.4. An example of the process of solution construction

The process is depicted in Figure 3.4, showing the scheduling of state after state. The permutation Π induces the construction of the final solution in Figure 3.4h. The intermediate steps to obtain this solution are given from Figure 3.4a to 3.4g. In the figure, boxes represent BBs and small rectangles denote the states forming a BB. Each rectangle is subdivided in fields representing the different module types available, in correspondence with the pattern in Figure 3.3b. In order to stress the handling of states, we have marked the states with different colors. A black rectangle represents a state already scheduled, a gray rectangle depicts the current state being scheduled and a white rectangle denotes unscheduled states. Note also that the current BB being visited in the BBCG traversal is emphasized with a heavy outline.

In addition, the sets of available operations at a given BB are shown. They represent the operations available on entry to a state about to be scheduled, or on exit from a state just scheduled. Since the example does not have multicycle operations, sets $\mathcal{A}_i$ and A_k are used interchangeably.

The process starts with the creation of a first state s_0 in BB I. Note that, in Figure 3.4a, operations k, l and q are available on entry to s_0. Since $k <_\Pi l$, k is the first subtraction selected for scheduling in s_0. Operation l, however, turns out not to be scheduled in s_0, due to resource constraints. Besides, operation q is eventually scheduled in s_0, since it is the only addition available on entry to s_0. Observe that the scheduling of k in s_0 makes t_1 and m available on exit from s_0, that is to say, on entry to s_1, as depicted in Figure 3.4b, where the next state s_1 is created.

Figure 3.4c shows that all the operations available on entry to s_1 can actually be scheduled in that state. As l is scheduled in s_1 and k was scheduled in s_0, operation n becomes available. Notice that, after m is scheduled in s_1, operation p also becomes available, because its input values are determined on control path $\langle I, J, L \rangle$, although not on the other path. Since a test t_1 is executed in state s_1, two alternative next states will be created, depending on the outcome of t_1, which is illustrated in Figure 3.4d. Note also that not all operations available on exit from state s_1 are made available for next states s_2 and s_3. On the one hand, operation n does not belong to the execution instance associated with the control path through BB J. On the other hand, operation p belongs to the execution instance associated with the control path through BB K, but it depends on n for this path and operation n, although available, has not yet been scheduled.

Observe that, given a sequence of states $\langle s_0, ..., s_k, ..., s_N \rangle$ of a BB, the set A_k on entry to s_k equals the set A_{k-1} on exit from s_{k-1}, for $k = 1, 2, ..., N$. However, if state s_N contains a test, the set A_{N+1} on entry to s_{N+1} is a subset of A_N on exit from s_N.

Figure 3.4e shows that, after state s_3 is scheduled, the set A_J is empty, meaning that the operations executing on the control path through BB J have all been scheduled. The same happens for the other control path after the steps in Figures 3.4f and 3.4g. Note that BB L turns out to contain no states, since all its operations are moved upwards. The resulting SMG is drawn in Figure 3.4h. It should be noted that operation p was duplicated in states s_3 and s_4. Since those states are redundant, it would be convenient to merge them into a single state, as suggested by Figure 3.4i. Our approach supports this feature, as will be explained in Chapter 4.

3.7.4 Discussion

Since our constructor is designed such that the scheduler engine is largely independent of the parallelizer, distinct schedulers can be used in our approach. We summarize in this section some of the issues involved in the choice of a scheduling mechanism.

The priority encoding Π can be used in distinct ways to generate a schedule, giving rise to distinct classes of schedulers. In the following, we discuss two examples of scheduling mechanisms based on a priority encoding:

- **List scheduling (LS):**
  This is one of the most popular mechanisms employed in HLS [16]. It relies on the notion of "filling" a *current* time step with operations. Given a current time step, an available operation is allowed to be scheduled in that step if there is a free resource for executing the operation. If more than one operation can be executed on a same resource at the current step, the operation with highest priority in the permutation is selected. When no more operations can be scheduled in that step, a successive time step is allocated, it becomes the current step and the process of "filling" restarts.
- **Topological permutation scheduling (TPS):**
  This mechanism [28] selects the first operation in the permutation that is available for scheduling. The selected operation is scheduled at the earliest time step in which a resource is free. There is no notion of current time step. The "filling" of a time step can start even if the previous step is not completely "filled".

The mechanism of selection of operations in LS is such that in some cases the optimal schedule can not be generated, regardless of the ordering chosen in the priority encoding. This is illustrated in Figure 3.5, which is borrowed from [28] and where the operation delay is 2 cycles for multiplications and 1 cycle for additions. Observe that, since v_1 and v_5 are the only initially available operations and $\tau(v_1) \neq \tau(v_5)$, they are eventually scheduled at the first time step, regardless of the priority encoding, as shown in Figure 3.5a. As a consequence, the optimal schedule shown in Figure 3.5b can not be induced by the LS mechanism. This problem is overcome by TPS. It is proven in [28] that the optimal schedule is always among those created by the TPS mechanism. This is the main advantage of TPS when compared with LS. The proof, however, is restricted to DFGs with a single BB. For the example in Figure 3.5, TPS can generate both schedules, depending on the relative order between v_2 and v_5 in the permutation. Note, however, that the optimum is missed by LS only when multicycle operations occur. When $d(v_i) = 1$, for every available operation v_i, the problem disappears and both TPS and LS lead to the same schedule.

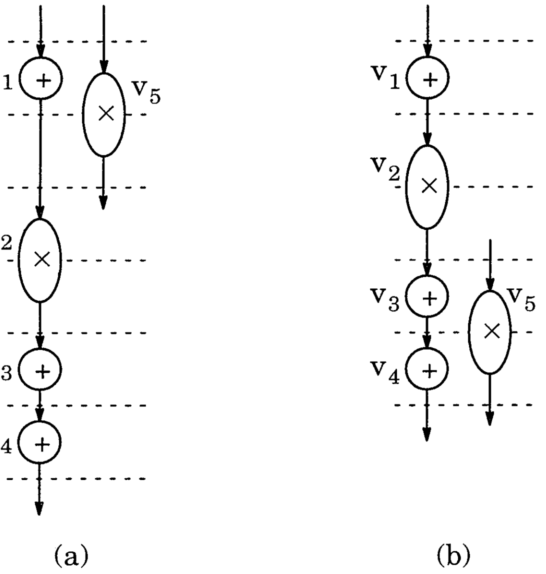


FIGURE 3.5. A list schedule and an optimal schedule of a same DFG

A disadvantage of TPS is that many distinct permutations lead to a same schedule. Consider two operations in A_k, say v_i and v_j, such that $\tau(v_i) \neq \tau(v_j)$. Assume that they have the highest priorities among the operations in set A_k. Since v_i and v_j will both be scheduled in s_k, their relative order in Π is irrelevant. As a consequence of irrelevant ordering, several permutations may induce a same schedule. In other words, the priority encoding Π is redundant with respect to operations that map to different resource types.

A way of alleviating such redundancy is obtained by restricting the permutation to each module type separately. As the scheduler does not have to arbitrate the occupation of a given resource by operations mapping to different module types, we can induce a schedule by using a set containing one permutation per module type. This notion is formalized below, where Π_{t_s} denotes a permutation consisting of all operations mapping to module type $t_s \in T_s$ and $K = |T_s|$:

DEFINITION 3.1

A *condensation* of a priority encoding Π, written $\varepsilon = \left\{\Pi_{t_1}, \Pi_{t_2}, ..., \Pi_{t_s}, ..., \Pi_{t_K}\right\}$, is a set of permutations Π_{t_s} such that:

- $\forall v_i \in \Pi : (\tau(v_i) = t_s) \Rightarrow v_i \in \Pi_{t_s}$,
- $\forall v_i, v_j \in \Pi : ((\tau(v_i) = \tau(v_j) = t_s) \wedge (v_i <_{\Pi} v_j)) \Rightarrow v_i <_{\Pi_{t_s}} v_j$.

Therefore, only the priority among operations of a same module type is taken into account. If the scheduler relies on the condensation ε, instead of on the priority encoding Π, the size of the search space is reduced substantially. Reconsider, for instance, the DFG in Figure 3.3a. For this example, a possible permutation is $\Pi = (k, l, t_1, n, m, p, q)$, whose condensation ε consists of $\Pi_{adder} = (k, l, n)$, $\Pi_{subtracter} = (m, p, q)$ and $\Pi_{comparator} = (t_1)$. Note that, for this example, the search spaces induced by Π and ε would be $7! = 840$ and $3!\ 3!\ 1! = 36$, respectively.

After using the TPS in early implementations of our approach, we have observed that too many identical schedules are generated and, as a consequence, that much time is spent on exploring identical solutions. For this reason, despite the theoretical advantage of TPS, our practical experience has guided us towards the use of a mechanism similar to LS for selecting operations to be scheduled in a given state. This mechanism is used for the experiments reported in this thesis. An efficient way of overcoming the generation of redundant schedules during the exploration of alternative solutions is presented in Section 3.9 and it is based on the condensation ε of the priority encoding Π.

Although other scheduler classes can be supported in our constructor, the study of different scheduler classes does not fall within the scope of this thesis, where the main focus is on the parallelizer and not on the scheduler. A comprehensive overview of scheduler classes can be found in [28].

3.8 The Boolean oracle

In order to provide extra support for availability analysis and bookkeeping, we propose a Boolean encoding for control path information. Classical notions like reachability, domination, postdomination and control equivalence are cast into Boolean form. As a result of this encoding, it is unnecessary to frequently traverse the BBCG in order to check control path information. Traversals are replaced by Boolean queries, which are submitted to the Boolean oracle. The purpose of our Boolean encoding is thus twofold. On the one hand, it supports major notions borrowed from the compiler–technology domain. On the other hand, it can rely on efficient Boolean representation techniques developed as part of the design automation technology.

The key to encoding path information in Boolean form is to associate a Boolean variable with each conditional and to define expressions on this set of variables. As the number of control paths may grow exponentially with the number of conditionals, paths are not represented explicitly. Instead, the execution condition of the operations associated with a BB is encoded. We formalize below the notions required for such an encoding.

The output of a conditional is associated with a Boolean variable, whose value is determined by the outcome of the respective test. For simplicity of notation, let us give this Boolean variable the same name as its respective conditional. Therefore, given a conditional c_k, its output value is denoted either as c_k or as $\bar{c}_k$, depending on whether the outcome of the conditional is true or false, respectively. In the literature, this Boolean variable is sometimes called a *guard* [52], a term that we adopt throughout this thesis.

Consider a BBCG generated for a behavioral description containing N tests. Let B_k and M_k be, respectively, the branch and merge nodes associated with conditional c_k. Recall that succ(B_k, true) is the successor of B_k that is reached when the outcome of c_k is true. The predicate representing the execution condition of the operations enclosed by basic block BB_i is defined as follows.

DEFINITION 3.2

The *predicate of a basic block* BB_i, written $G(BB_i)$, is a Boolean function defined on the set of guards $\{c_1, c_2, \ldots, c_N\}$ according to Algorithm 3.1.

ALGORITHM 3.1. Algorithm for obtaining the predicate of a basic block

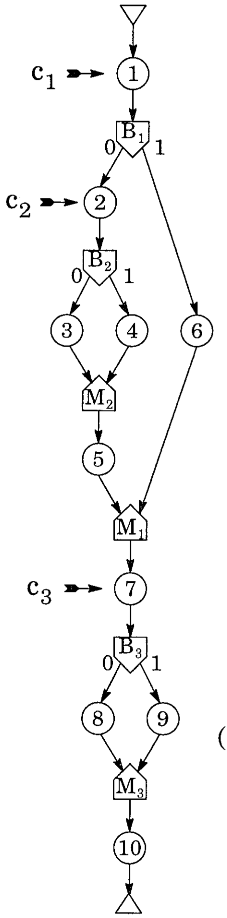
```
  $G(BB_i) := 1$;
  for ($k := 1$ to $N$)
     if ($B_k$ dom $BB_i$ $\wedge$ $M_k$ pdom $BB_i$)
        if (succ($B_k$, true) $\overset{*}{\rightarrow}$ $BB_i$)
           $G(BB_i) := G(BB_i) \cdot c_k$;
        else
           $G(BB_i) := G(BB_i) \cdot \bar{c}_k$;
```

Algorithm 3.1 can be interpreted as follows. In a topological traversal of the BBCG, if the branch junction B_k is reached prior to BB_i and the latter before the junction M_k, the guard c_k contributes to the Boolean product. The polarity is determined by which of the successors of B_k reaches BB_i. Informally, we can say that a guard c_k contributes to the predicate if BB_i is “enclosed” in between a pair of junctions B_k and M_k in the BBCG. Predicates can be efficiently evaluated during a topological traversal of the BBCG from source to sink.

An example is shown in Figure 3.6. A sketch of a behavioral description is given in Figure 3.6a for the BBCG in Figure 3.6b, where the arrows indicate links from conditionals in the DFG. The predicates associated with BBs are given in Figure 3.6c.

```
{...}
if t1
   {...}
else
   {...}
   if t2
      {...}
   else
      {...}
   {...}
{...}
if t3
   {...}
else
   {...}
{...}
```

(a)



(b)

$$G(BB_1) = G(BB_7) = G(BB_{10}) = 1$$
$$G(BB_2) = G(BB_5) = \overline{c}_1$$
$$G(BB_6) = c_1$$
$$G(BB_3) = \overline{c}_1 \cdot \overline{c}_2$$
$$G(BB_4) = \overline{c}_1 \cdot c_2$$
$$G(BB_8) = \overline{c}_3$$
$$G(BB_9) = c_3$$

(c)

FIGURE 3.6. An example of Boolean encoding in terms of predicates

Having defined the encoding, we are able to reason about paths in Boolean form. Let BB_i and BB_j denote arbitrary basic blocks and assume that BB_i is the current BB being visited. Let $BB_i <_T BB_j$ denote that BB_i precedes BB_j in some topological ordering $<_T$ of the nodes of the BBCG. Table 3.1 summarizes the most frequently queries submitted to the Boolean oracle.

Note that the first three queries require $BB_i <_T BB_j$ as a pre–condition. This pre–condition is automatically satisfied by the traversal of the BBCG adopted in the parallelizer. Consequently, only the Boolean queries within the shaded area are actually handled by the Boolean oracle. Therefore, a path query is split into two parts: one tackled in Boolean form by the Boolean oracle and another performed as graph manipulation by the parallelizer.

TABLE 3.1 Most frequently used queries

NOTION	PATH QUERY	BOOLEAN QUERY
REACHABILITY	$BB_i \xrightarrow{*} BB_j$	$BB_i <_T BB_j \wedge G(BB_i) \cdot G(BB_j)$
DOMINATION	BB_i dom BB_j	$BB_i <_T BB_j \wedge G(BB_j) \Rightarrow G(BB_i)$
POSTDOMINATION	BB_j pdom BB_i	$BB_i <_T BB_j \wedge G(BB_i) \Rightarrow G(BB_j)$
CONTROL EQUIVALENCE	BB_i equiv BB_j	$G(BB_i) = G(BB_j)$

Let us apply some queries to the example in Figure 3.6, where BBs are numbered in a topological order for convenience. For instance, proposition $BB_3 \xrightarrow{*} BB_4$ is false, since the predicate $c_2 \cdot \bar{c}_1 \cdot \bar{c}_2$ is not satisfiable. Note that proposition BB_2 dom BB_3 is true, since $BB_2 <_T BB_3$ and $\bar{c}_1 \cdot c_2 \Rightarrow \bar{c}_1$ is a tautology, which agrees with the fact that all paths from the source to BB_3 include BB_2.

Note that the encoding leads to predicates which are always Boolean products of guards. Since the Boolean queries shaded in Table 3.1 involve two predicates and assuming that m and n are their respective number of guards, the worst–case complexity of such queries is O(m.n). Moreover, the number of guards in a predicate is bounded by the maximal depth of conditional nesting, which in practice tends to be a fraction of the total number of conditionals. As a consequence, the path queries can be efficiently handled by a Boolean oracle.

For the experiments presented in this thesis, we use a Boolean oracle based on binary decision diagrams (BDDs) [16]. Although other Boolean representations could be utilized, as suggested in [10], we have used a BDD package for convenience, but without loss of generality.

The utilization of a Boolean encoding has been proposed before in HLS, but for different uses and purposes. In [52], the whole scheduling problem is represented in Boolean form. In other words, data and control dependences, resource constraints and timing are *all* cast into Boolean expressions. The downside of such a monolithic approach is that it leads to a huge number of Boolean variables, making memory size a critical issue when handling large behavioral descriptions.

Our approach differs from [52] in three main aspects. First, *only* conditional execution is represented in Boolean form. As a result, the guards are the only Boolean variables, whose total number is bounded by the number of conditionals. Second, we utilize the Boolean encoding to reason about *paths*. Unlike the work in [52], we cast into Boolean form important notions from the compiler–technology domain, such as domination, postdomination and control equivalence. These notions are useful, for instance, in bookkeeping code motions (Section 4.4). Third, we do not encode solutions in Boolean form. Instead, solutions are represented in the form of graphs and predicates assist on the *construction* of solutions, for instance during availability analysis (Section 4.3) and state equivalence checking (Section 4.5).

3.9 The explorer

A detailed discussion of the explorer is beyond the scope of this thesis. The way in which priority encodings are generated according to the criteria of a given local search algorithm has already been object of previous research [28].

For the experiments in this thesis, we want to evaluate the techniques used in the constructor by obtaining statistics about their impact on the search space. In order to keep the figures largely independent of the choice of a particular search method, random search will be used in our experiments. Experimental results showing the impact of a genetic algorithm in combination with scheduling methods can be found in [28].

The generation of identical solutions during the exploration of alternative SMGs can be largely avoided by using the condensation ε of the priority encoding Π, as suggested by the following idea: since a list–scheduling selection mechanism is used in the constructor, if two priority encodings have the same condensation, then the solutions induced by these encodings are identical.

Assume that a so–called *condensation table* is provided to the explorer engine. Each row of this table contains the condensation ε_i of some priority encoding Π_i. The table contains exactly one condensation for every priority encoding sent to the constructor hitherto. Assume that the explorer generates a priority encoding Π_j. Before sending Π_j to the constructor, the explorer checks if there is some ε_i in the table such that $\varepsilon_j = \varepsilon_i$. In this case, we say that a *hit* occurs in the condensation table.

A hit in the condensation table means that a solution f_i has already been explored and that it is *not* worth applying Π_j to the constructor, because it would induce a solution f_j identical to f_i. Instead, Π_j is discarded, a new priority encoding Π_{j+1} is generated and the process is repeated. However, if no hit occurs, this means that no solution with condensation ε_j is explored so far. Therefore, the priority encoding Π_j is applied to the constructor and the generated solution f_j is explored. The checking on hit can be efficiently implemented by means of a hash table.

In summary, methods to implement the explorer and the Boolean oracle are borrowed from other research areas. Since neither the explorer nor the Boolean oracle handle exploitation of parallelism, from now on, this thesis will focus on the solution constructor, the engine which actually deals with ILP techniques.

Chapter 4 Code motion

This chapter describes the support for code motion and speculation in the frame of the constructive approach presented in the previous chapter. It addresses the following issues:

- **How to model code motion:**
  It is shown that the effect of moving operations can be captured by reorganizing their respective links to BBs, instead of actually moving the operations in the DFG. A mechanism is proposed for maintaining information on conditional execution by updating predicates dynamically.
- **How to induce code motion:**
  The notion of available operation is the key concept for inducing code motion. Basically, an operation is available at a given BB if its scheduling in that BB satisfies data dependences and preserves the semantics of conditional execution, although the operation may initially be linked to some other BB. A global availability analysis technique is presented for finding out operations whose scheduling induces legal code motions.
- **How to compensate for side effects of code motion:**
  Some code motions require the insertion of copies of operations in order to preserve the semantics of conditional execution. A technique based on Boolean queries is proposed to tackle this issue, thereby broadening the range of legal code motions.
- **How to control the growth of the number of states:**
  As a result of the insertion of copies, code motion may lead to a growth of the number of states. On the one hand, such a growth may represent the price to pay for a shorter schedule, but on the other hand, it may be caused by the introduction of redundant states. A technique exploiting the notion of state equivalence is presented to eliminate redundant states in the course of the scheduling process.

4.1 Fundamental notions

In our approach, the parallelizer employs code motion in the context of global scheduling. Recall that the parallelizer visits BBs during a top–down traversal of the BBCG. Given the currently visited basic block, say BB_i, operations are selected for scheduling in the current state appointed within BB_i, say $s_{i,k}$. Not all operations to be scheduled in $s_{i,k}$ are necessarily linked to BB_i. Some operation o_n may be linked to a basic block BB_j reachable from BB_i. When operation o_n is scheduled in state $s_{i,k}$, it is as if o_n were moved from BB_j to BB_i. We assume that each time an operation o_n is scheduled in some state within BB_i, the link λ such that $o_n \xrightarrow{\lambda} BB_i$ is marked as scheduled. Such a link is said to be a *scheduled* link, written scheduled(λ). As a consequence, during the construction of a solution, some links are scheduled and others, unscheduled. When the construction of a solution is completed, the links from operations to BBs are all scheduled.

In order to determine if a code motion is legal, we rely on the notion of scheduled links, along with the notion of data dependence, which was introduced in Chapter 2 and is formalized in the following. The formulation assumes a DFG=(V, E) and operations $o_n, o_m, v_i \in V$.

Data dependence

Since our parallelizer performs resource–constrained scheduling, not all nodes in the DFG are actually relevant, since some of them do not imply occupation of resources. For instance, branch and merge nodes are not mapped to any module of the network graph (NWG). To simplify further formulations, we introduce a function $\theta : V \mapsto B$, defined as follows:

$$\theta(v_i) = \begin{cases} \text{true, if } (\omega(v_i) \neq \text{branch}) \wedge (\omega(v_i) \neq \text{merge}) \\ \text{false, if } (\omega(v_i) = \text{branch}) \vee (\omega(v_i) = \text{merge}) \end{cases}. \tag{4.1}$$

After an operation o_m is scheduled, some operations which are data dependent on o_m may become available for scheduling. In order to identify these operations, we first find the data–dependent operations immediately reachable from o_m. During this procedure, we discard the nodes of the DFG which are irrelevant from the point of view of resource occupation. A node v_i such that $\theta(v_i)$ = false can thus be overlooked. Control dependences are disregarded by discarding all operations reachable through control edges. Theses notions are formalized below. Let N(p) be the set of nodes included in some path p and E(p) denote the set of edges contained in path p.

DEFINITION 4.1

A path p with $o_m \xrightarrow{p} o_n$ and $o_n, o_m \in V$ is a *cleared path* iff it satisfies the following conditions:

- $\theta(o_n) = \theta(o_m) = \text{true}$,
- $\forall v_i \in N(p) \backslash \{o_m, o_n\} : \theta(v_i) = \text{false}$,
- $\forall e \in E(p) : e$ is a data edge.

Definition 4.2

The set of *data consumers immediately reachable from* o_m, written CONS(o_m), is the set of all operations $o_n \in V$ with $o_m \xrightarrow{p} o_n$ such that p is a cleared path.

Conversely, due to the need to check if all data values consumed by a given operation are actually produced, we define the concept of data producers.

Definition 4.3

The set of *data producers immediately reaching* o_n, written PROD(o_n), is the set of all operations $o_m \in V$ with $o_m \xrightarrow{p} o_n$ such that p is a cleared path.

Notice that CONS(o_m) represents the set of operations which are data dependent on o_m, whereas PROD(o_n) represents the set of operations on which o_n is data dependent.

Having introduced these essential notions, we show how code motion is supported in our parallelizer.

4.2 Modeling code motion

Given a DFG=(V, E) and a BBCG=(U, F), we assume that every node $v \in V$ is initially linked to exactly one node $u \in U$. Each of such links is called an *initial link*. An initial link points to the initial position of an operation in the control flow prior to any code motion. In the next chapter, a more elaborate initialization scheme is proposed in which the links are established differently. For a given initialization, the effect of code motion is modeled as follows. Assume that an operation o_n is linked to some basic block BB_j by means of link λ_b. The motion of operation o_n from BB_j to some basic block BB_i is modeled by creating a *new* link λ_a to BB_i and by deleting the *old* link λ_b. Code motions through not more than one pair of branch or merge nodes are illustrated in the BBCGs of Figure 4.1. These are referred as *local* code motions. They are similar to some of the primitive code motions defined in Percolation Scheduling [48]. In the figure, a black circle denotes the BB currently visited by the parallelizer. Arrows represent links emanating from operation o_n. Each link denotes the position of o_n in the control flow. Links λ_b and λ'_b point to locations of operation o_n *before* code motion. Conversely, links λ_a and λ'_a point to positions *after* code motion.

Figure 4.1a depicts the so–called *useful* code motion [62], which occurs when an operation moves between control–equivalent BBs. A code motion to a basic block BB_i that does not dominate the original basic block BB_j is illustrated in Figure 4.1b. To preserve the semantics of conditional execution (as will be discussed in Section 4.4), this motion requires the insertion of a "copy" of o_n, which is modeled by the link λ'_a. For this reason, such code motion is sometimes called *duplication–up* [54]. Figure 4.1c illustrates a code motion from a basic block BB_j that does not postdominate the new basic block BB_i. Since it represents the motion of a conditionally executed operation ahead of the branch junction, it is known as *pre–execution*. Finally, Figure 4.1d illustrates the *unification* [48] of two "copies" of o_n into a single link.

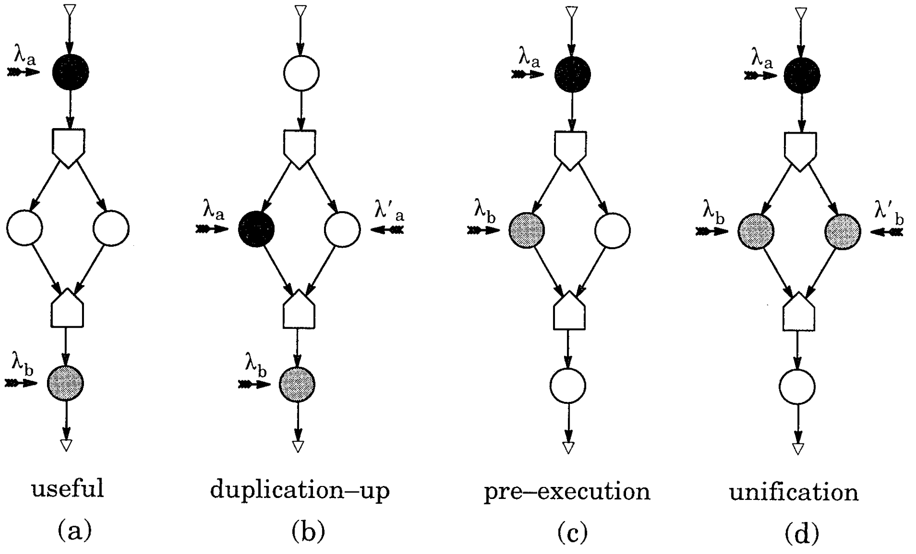


FIGURE 4.1. Examples of local code motions

The scope of code motion can be broadened to comprise several BBs, leading to the so–called *global* code motions. The idea of global code motion is illustrated in the BBCGs of Figure 4.2. Dashed arrows indicate possible upward code motions. Motions between gray circles are local, whereas motions between black circles are global. Note that these global code motions could be obtained by the successive application of local code motions, like in Percolation Scheduling [19][27] and similar methods [54]. As opposed to Percolation Scheduling, our approach performs global code motion between two basic blocks *in one step*, without having to rely on the iterative application of local code motions between intermediate BBs. In other words, we do support local code motions, but we do *not* use them as *primitive* transformations to induce global code motions, as will be shown in Section 4.3.

Although the modeling in this chapter assumes upward code motions, our method supports downward code motion by means of an initialization procedure, as will be explained in Chapter 5.

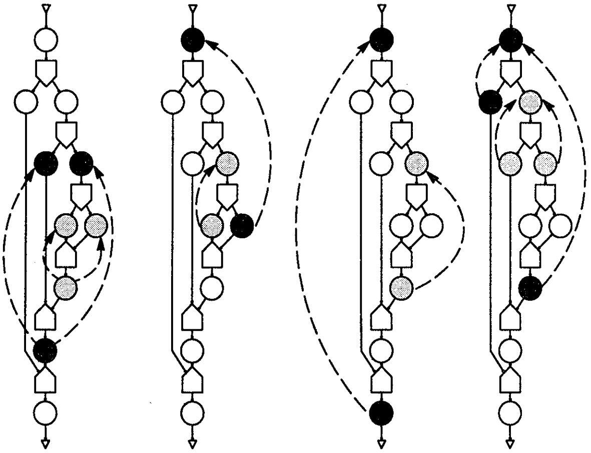


FIGURE 4.2. Local code motion *versus* global code motion

Let us introduce some extended path queries, derived from the basic path queries in Table 3.1, which allows us to distinguish between different kinds of code motion. The purpose of introducing such extended queries is twofold. On the one hand, they are helpful in describing and explaining several issues addressed in this chapter. On the other hand, one of them is used to handle code compensation (see Section 4.4.3) and others can be used to include new features in the constructor (see Section 4.6).

Code motions can be distinguished by using the path queries introduced in Table 3.1. Let BB_i designate the BB currently visited by the parallelizer. Suppose that o_n is linked to some BB_j such that $BB_i \xrightarrow{*} BB_j$. Assume that o_n is about to move to BB_i. Table 4.1 expresses the code motions in terms of path queries. Unification is omitted in the table, since it can be identified by testing if more than one link is pre–executed.

TABLE 4.1 Extended path queries for distinguishing code motions

CASE	CODE MOTION	PATH QUERY
(a)	useful	BB_i equiv BB_j
(b)	duplication–up	$\neg(BB_i$ dom $BB_j) \wedge (BB_j$ pdom $BB_i)$
(c)	pre–execution	$(BB_i$ dom $BB_j) \wedge \neg(BB_j$ pdom $BB_i)$
(d)	duplication + pre–execution	$\neg(BB_i$ dom $BB_j) \wedge \neg(BB_j$ pdom $BB_i)$

Examples of code motions corresponding to the cases distinguished in Table 4.1 are given in Figure 4.3. In each case, BB_i is denoted by a black circle, whereas BB_j is denoted by a gray circle.

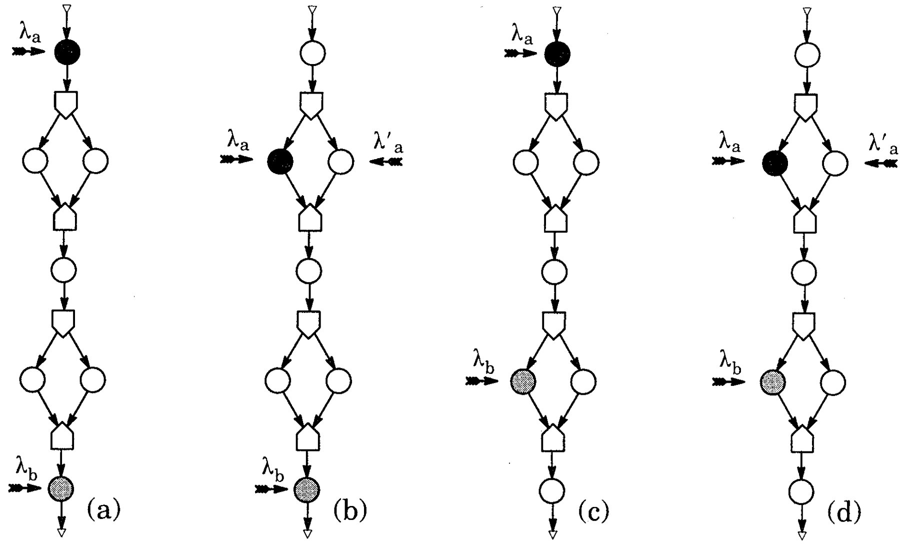
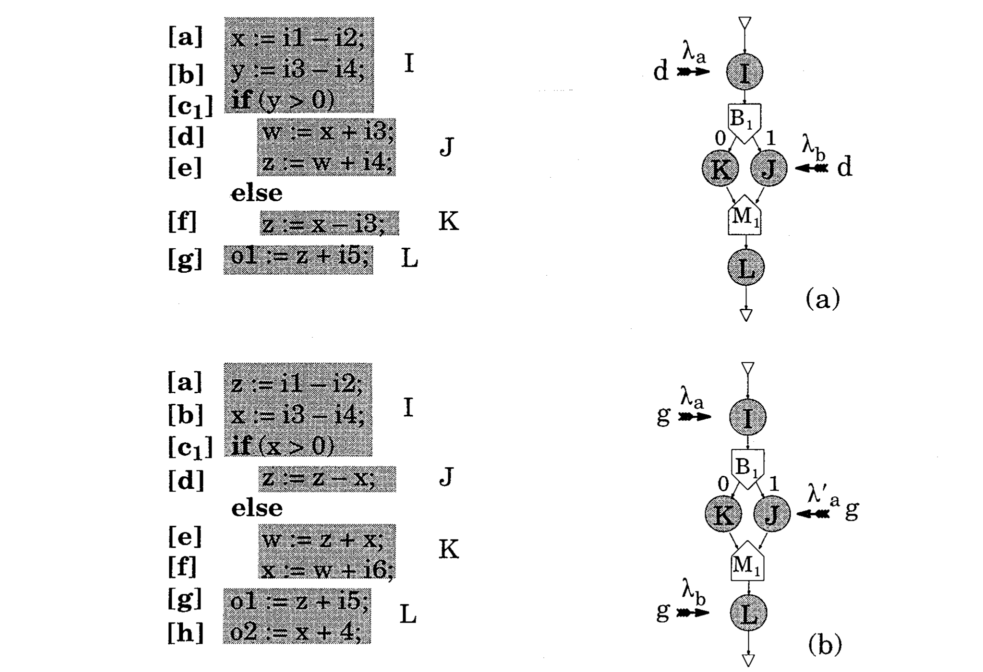


FIGURE 4.3. Examples of code motions for the cases in Table 4.1

Speculative execution

The examples in Figure 4.4 illustrate code motions leading to different forms of speculation. In Figure 4.4a, after operation d is moved from BB J to BB I, this operation is executed unconditionally, although d should be executed only if the outcome of conditional c_1 turns out to be true. Consequently, the result of operation d has to be *discarded* on path $\langle I, B_1, K, M_1, L\rangle$, whereas it is *committed* on path $\langle I, B_1, J, M_1, L\rangle$. Another situation is illustrated in Figure 4.4b. Assume that operation a is scheduled in BB I and operation d is unscheduled. Note that operation g can be moved from BB L to BB I on path $\langle I, B_1, K, M_1, L\rangle$, since the data dependence between operations a and g is satisfied. However, the data dependence between operations d and g obstructs the motion on path $\langle I, B_1, J, M_1, L\rangle$. To make the motion legal, a replica of g is inserted in BB J. This means that, after code motion, the execution of operation g is speculative regarding the path $\langle I, B_1, J, M_1, L\rangle$. If this path is taken during execution, the operation is executed twice. The result of the first execution is discarded, whereas the second is committed. Essentially, speculation arises from code motion under two circumstances:

- When, after code motion, the operation executes on control paths on which it does not execute before code motion (Figure 4.4a).
- When the operation executes on exactly the same control paths before and after code motion, but the motion is blocked on some path due to a data dependence on an unscheduled operation (Figure 4.4b).



```
[a]  x := i1 – i2;
[b]  y := i3 – i4;     I
[c1] if (y > 0)
[d]      w := x + i3;
[e]      z := w + i4;     J
     else
[f]      z := x – i3;     K
[g]  o1 := z + i5;     L
```

```
[a]  z := i1 – i2;
[b]  x := i3 – i4;     I
[c1] if (x > 0)
[d]      z := z – x;      J
     else
[e]      w := z + x;
[f]      x := w + i6;     K
[g]  o1 := z + i5;
[h]  o2 := x + 4;      L
```

FIGURE 4.4. Examples of speculation under different circumstances

Let us now analyze the types of code motion in Table 4.1 with respect to the first form of speculation. Cases (c) and (d) lead to *pre–execution*, that is to say, the moved operation o_n executes ahead of at least one branch junction B_k such that the operation is control dependent on conditional c_k. Therefore, these two types of code motion always lead to speculation. However, the first form of speculation does not occur for cases (a) and (b). On the other hand, each of the cases of code motion in Table 4.1 may give rise to the second form of speculation, since the moved operation might depend on an unscheduled operation linked to some basic block BB_k such that $BB_i \xrightarrow{p_1} BB_k$ and $BB_k \xrightarrow{p_2} BB_j$, where p_1 and p_2 are subpaths of some path p. Clearly, this kind of speculation can be excluded beforehand if BB_i and BB_j are "adjacent", i.e., if there is no BB_k in between BB_i and BB_j on path p.

It is important to notice that, as a result of the possible insertion of copies, there may be *more than one link emanating from a given operation*. From now on, let $\Lambda(o_n)$ denote the set of all links emanating from operation o_n.

Recall that, given a BB_i, the predicate $G(BB_i)$ represents the execution condition of the group of operations condensed into BB_i. Assume that an operation o_n is moved from BB_j to BB_i. To model speculation properly, a distinction should be made between the condition under which o_n is *executed* and the condition under which the result is actually *committed*. The first condition is represented by $G(BB_i)$ and the second by $G(BB_j)$. Since the information regarding the second predicate would be lost after code motion (because the old link λ_b and possible copies λ'_b are deleted), it is captured in the new link λ_a by associating a predicate with the link itself, as explained in the following.

It is assumed that, before the construction of a solution starts, the predicate of every initial link λ, written $G(\lambda)$, is initialized with the predicate of the BB to which it points, as shown in Algorithm 4.1. The predicate $G(\lambda)$ can be interpreted as the execution condition of operation o_n when no code motions *at all* are performed.

ALGORITHM 4.1. Algorithm for initializing the predicates of all links

```
foreach $BB_j \in U$
    foreach $\lambda$ such that $o_n \overset{\lambda}{\dashrightarrow} BB_j, o_n \in V_o$
        $G(\lambda) := G(BB_j)$;
```

Assume that we want to model the motion of operation o_n to basic block BB_i. A *new link* λ_a is created and its predicate $G(\lambda_a)$ is obtained from the predicates of the old links. In determining this predicate, the following two properties of conditional execution are observed:

- The values required for executing an operation o_n may be computed by data producers executing on distinct control paths.
- The value computed by some operation o_n may have no consumer on some control path.

The procedure to determine the predicate $G(\lambda_a)$ is performed in two phases. In the first phase, we determine the *predicate under which data values are produced for an operation* o_n *executing in basic block* BB_i, written $G_P(o_n, BB_i)$. To illustrate this notion, we reconsider the example in Figure 4.4b. Assume that operation a is already scheduled in BB I and that operation d is unscheduled. Assume that operation g is about to move from BB L to BB I. Note that PROD(g) = {a, d}. Since d is unscheduled, the motion of operation g is blocked by the data dependence on operation d, which is linked to BB J. As a consequence, the required data value is not yet produced under predicate $G(J) = c_1$. As a consequence, the data value is only produced under predicate $G_P(g, I) = \bar{c}_1$.

Algorithm 4.2 shows how the predicate $G_P(o_n, BB_i)$ is determined. The underlying idea is that a data value is *not* produced under some predicate

$G(\lambda)$ if λ is unscheduled. The nested loops check all the links emanating from every data producer o_m that are pointing to some BB reachable from BB_i. The Boolean product of the complements of the predicates of unscheduled links gives the condition under which data values are produced.

ALGORITHM 4.2. Algorithm for determining the predicate $G_P(o_n, BB_i)$

```
$G_P := G(BB_i);$
foreach $o_m \in \mathrm{PROD}(o_n)$
    foreach $\lambda \in \Lambda(o_m)$ with $o_m \xrightarrow{\lambda} BB_k \wedge BB_i \xrightarrow{*} BB_k$
        if $(\neg \mathrm{scheduled}(\lambda))$
            $G_P := G_P \cdot \overline{G(\lambda)};$
```

In the second phase, we determine the *predicate under which the data value computed by an operation* o_n *executing in basic block* BB_i *is actually consumed*, written $G_C(o_n, BB_i)$. This condition is captured by a predicate that takes into account the fact that operation o_n may not be executed on all paths starting from BB_i. Algorithm 4.3 shows how to obtain the predicate $G_C(o_n, BB_i)$, where the loop performs the Boolean sum of all predicates of the old links λ_b to be replaced by the new link λ_a after the motion, that is to say, the links emanating from o_n pointing to some basic block reachable from BB_i.

ALGORITHM 4.3. Algorithm for determining the predicate $G_C(o_n, BB_i)$

```
$G_C := 0;$
foreach $\lambda_b \in \Lambda(o_n)$ with $o_n \xrightarrow{\lambda_b} BB_j \wedge BB_i \xrightarrow{*} BB_j$
    $G_C := G_C + G(\lambda_b);$
```

Finally, the predicate of a new link λ_a modeling the code motion of an operation o_n to basic block BB_i is given by the following equation:

$$G(\lambda_a) = G_P(o_n, BB_i) \cdot G_C(o_n, BB_i). \tag{4.2}$$

The concept of predicate of a link represents the condition under which the result of operation o_n is committed in BBs reachable from BB_i. Therefore, an operation o_n is speculatively executed when its *committing condition* differs from its actual *execution condition* within BB_i, as formalized below:

DEFINITION 4.4

An operation o_n is *speculatively executed* within some basic block BB_i iff:

$\exists \lambda \in \Lambda(o_n) : o_n \xrightarrow{\lambda} BB_i \wedge (\ G(\lambda) \neq G(BB_i)\ ).$

An implicit assumption throughout this thesis is that there exists a sufficient number of registers for storing the results of speculatively executed operations. This is a particular aspect of the more general assumption that register allocation occurs after scheduling. In Chapter 7, an extension is suggested for taking into account the effect of a fixed number of registers. However, we do not assume special hardware support for committing speculative results in execution time [62]. Essentially, this mechanism consists in keeping speculative results in a dedicated buffer until the outcome of the required tests is known. Then, those results are committed by allowing them to update the register file. On the one hand, such hardware support is unnecessary in our approach, since semantics is preserved via bookkeeping and code compensation. On the other hand, as pointed out in [30], the advantages of such hardware–based speculation relies on its combination with dynamic scheduling and dynamic branch prediction. Although efficient when aiming at average execution time, these techniques can not improve worst–case execution and, consequently, are not suitable under time–constraints.

Another aspect of hardware support concerns the handling of exceptions. It seems reasonable to expect that operations prone to raise exceptions should not occur very often in the tasks of a digital system that are subject to time–constraints. Therefore, we assume for simplicity that when an operation (if any) might raise an exception, its speculative execution is inhibited. However, this assumption can be relaxed with special hardware support, as suggested in [62] for instance. A comprehensive overview of hardware support for speculation can be found in [30]. In particular, the use of hardware support for speculation in HLS is addressed, for instance, in [32].

4.3 Availability analysis

In DFGs containing no conditionals, availability analysis is very simple. Essentially, an operation is available if it is not yet scheduled and all its predecessors are scheduled. However, availability analysis for DFGs containing conditionals is a more elaborate task, since it has to take into account *conditional execution*.

Each time a BB is visited, say BB_i, the set $\mathcal{A}_i$ of available operations at BB_i is evaluated according to the dependences in the DFG and subject to the delays of the modules in the NWG. In this section, we show how to evaluate which operations are elements of the set $\mathcal{A}_i$.

Since our goal is to induce *global* code motions, we search for available operations beyond the current BB. As a consequence, the set $\mathcal{A}_i$ may contain operations which are linked to different BBs reachable from BB_i. In determining $\mathcal{A}_i$, only data dependences are considered. Control dependences are disregarded, since they would unnecessarily restrict code motion beforehand. The fact that we are not enforcing control dependences does not mean that we are ignoring the effects of conditionals. It should be noted that, despite the possible violation of control dependences, semantics is preserved via code compensation (see Section 4.4).

The remaining of this section is organized as follows. We first formalize the notion of available operation and then we show how to determine the initial set of available operations. Afterwards, it is shown how available operations can be incrementally computed after some operation is scheduled in the current state and how to evaluate the available operations at the next states. The section concludes with comments on how to take account of multicycling effects.

As opposed to the simpler analysis for DFGs free of conditionals, the availability analysis for more general graphs relies on the following notions:

- An operation o_n may be available on a given control path but unavailable on another when o_n is conditionally executed.
- An operation o_n may be available on a given control path, but unavailable on another when o_n is blocked by a data dependence on some operation which is conditionally executed.

Availability of an operation

An operation o_n is considered available at a basic block BB_i if it executes on some path from BB_i to the sink and if o_n can be legally moved to BB_i. This happens when there is some link from operation o_n to a basic block BB_j reachable from BB_i and there exits at least one path from the source to BB_i on which the input data values of o_n are already produced. This notion is formalized below:

DEFINITION 4.5

An operation o_n is *available at* BB_i, written available(o_n, BB_i), iff:

$o_n \xrightarrow{\lambda} BB_j \wedge BB_i \xrightarrow{*} BB_j \wedge G_P(o_n, BB_i)$ is satisfiable.

Initial and incremental availability

The inputs of the DFG are the primary data producers. Let the set of inputs of the DFG be $\text{INP}(V) = \{ v_i \in V \mid \omega(v_i) = \text{input} \}$. Recall that the parallelizer performs a top–down traversal of BBs. Let BB_0 be the first BB visited during the traversal of the BBCG. The set of operations initially available, written

$\mathcal{A}_0$, is obtained by finding all the operations o_n that are consumers of input values and by selecting those whose producers o_m are *all* inputs of the DFG, as follows:

$$\mathcal{A}_0 = \{\, o_n \in \bigcup_{v_i \in \text{INP}(V)} \text{CONS}(v_i) \,|\, \forall o_m \in \text{PROD}(o_n) : o_m \in \text{INP}(V) \,\}. \tag{4.3}$$

Let us now consider how to update some generic availability set, say $\mathcal{A}_i$, after some operation o_m is scheduled within basic block BB_i. First, the already scheduled operation o_m is excluded from the set $\mathcal{A}_i$. Then, the operations which become available due to the scheduling of o_m are added to $\mathcal{A}_i$ if they satisfy the criterion formulated in Definition 4.5. These operations are a subset of $\text{CONS}(o_m)$, because the scheduling of o_m is a necessary condition for the scheduling of its consumers. This procedure is summarized in Algorithm 4.4.

ALGORITHM 4.4. Incremental availability analysis algorithm

```
$\mathcal{A}_i := \mathcal{A}_i \setminus \{o_m\}$;
foreach $o_n \in \text{CONS}(o_m)$
  if (available($o_n$, $BB_i$))
    $\mathcal{A}_i := \mathcal{A}_i \cup \{o_n\}$;
```

Availability at the next state

Assume that BB_i is the currently visited BB. After the current state, say $s_{i,k}$, is scheduled, one or more next states are appointed by the parallelizer. Recall that, if no conditional is scheduled in $s_{i,k}$, a next state is appointed within the same basic block BB_i. However, if at least one conditional is scheduled in $s_{i,k}$, next states are appointed in other BBs, since the conditional causes a branch in the control flow.

When the next state falls within BB_i, all operations available on exit from $s_{i,k}$ are available on entry to $s_{i,k+1}$. However, when the next states fall within other BBs, say BB_j and BB_x, an operation available on exit to $s_{i,k}$ is *not* necessarily available on entry to *both* $s_{j,0}$ and $s_{x,0}$, since the operation might be conditionally executed. Thus, the set of operations available on entry to one of the next states is a subset of the operations available on exit from the current state. This notion is formalized in the following.

Given two states $s_{i,k}$ and $s_{j,0}$ which fall within basic blocks BB_i and BB_j, respectively, with $s_{j,0} \in \text{SUCC}(s_{i,k})$ and $BB_i \xrightarrow{*} BB_j$, the set of *available operations at* BB_j is obtained as follows:

$$\mathcal{A}_j = \{\, o_n \in \mathcal{A}_i \,|\, \exists \lambda \in \Lambda(o_n) : o_n \overset{\lambda}{\twoheadrightarrow} BB_k \wedge BB_j \xrightarrow{*} BB_k \,\}. \tag{4.4}$$

Modeling the effect of multicycling

Although several operations may be available for scheduling at a given BB, they may *not* be available at the *same* state. To illustrate the effect of multicycling on the availability set, consider the DFG sketched in Figure 4.5a. The DFG has a single BB for simplicity. Assume that one adder and one multiplier represent the resource constraints and that an addition takes one clock cycle, whereas a multiplication takes two cycles. In Figures 4.5b to 4.5g, the evolution of scheduling is shown, state after state. Notice that the current state is indicated in gray. Observe that operations a and d are available at state s_0. Since $\tau(a) \neq \tau(d)$, they are both scheduled in state s_0, as shown in Figure 4.5c. Note that the scheduling of operation a makes b available, whereas the scheduling of d makes operation e available. However, b and e are not available at the same state. Since a multiplication takes two cycles, the value computed by d can be consumed only at state s_2 or later. Consequently, operation e is available at state s_2, whereas operation b is available at state s_1.

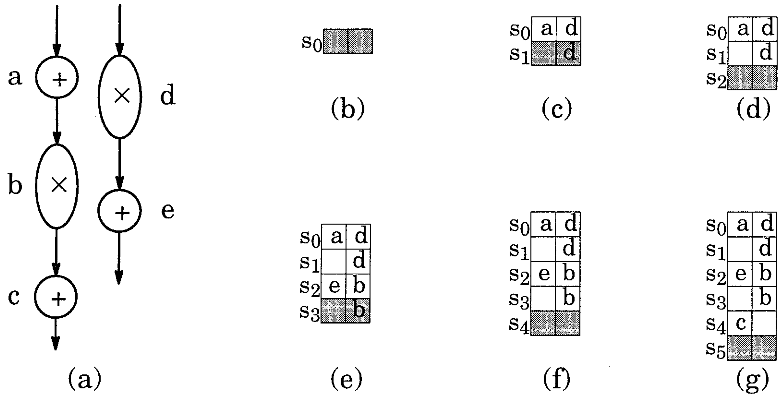


FIGURE 4.5. The effect of multicycle operations on availability

In summary, given a basic block BB_i, the operations in $\mathcal{A}_i$ may be available at distinct states, as a consequence of possibly different delays of their immediate data producers. For this reason, given the current state $s_{i,k}$ within BB_i, the set $\mathcal{A}_i$ is split into a collection of availability sets $A_{i,k+x}$ associated with states $s_{i,k+x}$. However, there is no need for maintaining availability sets $A_{i,k+x}$ for more than D successive states starting from $s_{i,k}$, where D denote the ceiling of the maximal operation delay. Every set $A_{i,k+x}$ is obtained from the set $\mathcal{A}_i$ by selecting the operations whose data producers have been scheduled long enough before the state $s_{i,k+x}$ is reached. A function denoted as max_displacement($o_n, s_{i,k}$), which is described in Appendix B, performs the selection as follows:

$$A_{i,k+x} = \{ o_n \in \mathcal{A}_i \,|\, \text{max\_displacement}(o_n, s_{i,k}) = x \}. \qquad (4.5)$$

For the example in Figure 4.5, the status of the availability sets at every state is given in Table 4.2. Since the maximal delay is 2, the set $\mathcal{A}$ is formed by two subsets A_k and A_{k+1} (the first index is dropped for simplicity). Note that, on entry to state s_1, the availability set is formed by subsets $A_1 = \{b\}$ and $A_2 = \{e\}$, meaning that only b is available at s_1, since the predecessor of e is a multicycle operation. Observe that, since b can not be scheduled in state s_1 due to resource constraints, it remains available on entry to state s_2.

If multicycle operations do not occur, there is no distinction between $\mathcal{A}_i$ and $A_{i,k}$. This is, by the way, the form adopted by most methods developed in the compiler domain. These methods do not provide support for the effects of multicycle operations at the level of availability analysis. Instead, they treat a multicycle operation, say v_i, as a chain of $\lceil d(v_i) \rceil$ one–cycle operations, like in [3] for instance. However, artificial constraints have to be added to avoid the illegal preemptive scheduling of operations in the chain.

TABLE 4.2 Availability sets for the example in Figure 4.5

CASE	CURRENT STATE	$\mathcal{A}$	A_k	A_{k+1}
(b)	s_0	{a, d}	A_0 = {a, d}	$A_1 = \emptyset$
(c)	s_1	{b, e}	A_1 = {b}	A_2 = {e}
(d)	s_2	{b, e}	A_2 = {b, e}	$A_3 = \emptyset$
(e)	s_3	{c}	$A_3 = \emptyset$	A_4 = {c}
(f)	s_4	{c}	A_4 = {c}	$A_5 = \emptyset$
(g)	s_5	$\emptyset$	$A_5 = \emptyset$	$A_6 = \emptyset$

4.4 Code compensation

To support unrestricted code motion, it is not enough to provide mechanisms for inducing it. Code motions may have side effects which must be compensated with extra code. However, the greedy insertion of compensation code may prevent satisfaction of time–constraints. In this section, we address code compensation mechanisms oriented towards the synthesis of time–constrained systems. After introducing the notion of compensation code and briefly reviewing related work, we show how the Boolean queries defined in the previous chapter can be used to replace the frequent depth–first search traversals used in classical handling of code motion, which are the main source of bookkeeping overhead. Then we show a code compensation technique that does not increase schedule lengths during global scheduling, given an arbitrary priority encoding, a desirable feature for handling tight time–constraints.

```
[a]  x := i1 – i2;
[b]  y := i3 – i4;       I
[c1] if (y > 0)
[d]      z := x + i3;        J
     else
[e]      w := x – y;         K
[f]      z := w + i4;
[g]  o1 := z + i5;       L
[h]  o2 := y + i3;
```

```
[a]  z := i1 – i2;
[b]  x := i3 – i4;       I
[c1] if (x > 0)
[d]      z := z – x;         J
     else
[e]      w := z + x;         K
[f]      x := w + i6;
[g]  o1 := z + i5;       L
[h]  o2 := x + 4;
```

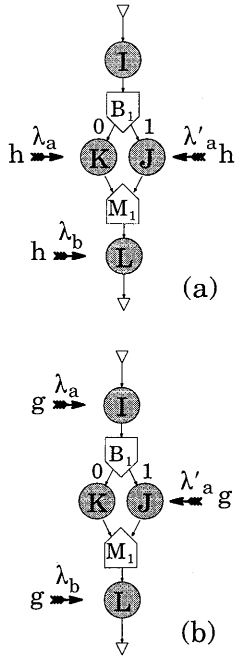


FIGURE 4.6. Examples in which code compensation is obligatory

To introduce the notion of code compensation, the examples in Figure 4.6 are used. Assume that an adder and a subtracter are available and that the comparison incurs no delay (its outcome is a flag set by the subtracter). A situation where compensation code is needed is shown in Figure 4.6a. Suppose that BB K is currently visited and that operation b is already scheduled. Consequently, operation h is available at the first state within BB K, thereby inducing the code motion of operation h from BB L to BB K, which saves a cycle. Since h has to be executed on both paths, a copy of h must be inserted in BB J. Another situation is shown in Figure 4.4b, repeated in Figure 4.6b for convenience, where speculation occurs when g moves from BB L to BB I. Note that g can move up on path $\langle I, B_1, K, M_1, L\rangle$ only. However, the operation is blocked on path $\langle I, B_1, J, M_1, L\rangle$ due to the data dependence on d. For correctness, a replica of g must be inserted in BB J to compensate for the effect of the motion when the outcome of c_1 is true. These examples are deliberately chosen such that the depicted code motion is vital for decreasing the schedule length of the longest control path.

It may be important to note that not all code motions lead to speculation, that non–speculative code motions may also require compensation code (e.g. Figure 4.6a) and that not all speculative code motions need compensation code (e.g. Figure 4.4a).For each code motion, a bookkeeping procedure checks where compensation code is needed (if any) and inserts the necessary copies. In the next subsection, different bookkeeping methods are analyzed.

4.4.1 Related work

Several techniques for inserting compensation code are reported in the literature. They vary depending on how ILP is exploited. For instance, in Trace Scheduling (TS) [22] code compensation is performed after the main trace is completely scheduled. Copies of operations are inserted off–trace during a bookkeeping phase. However, the code compensation mechanism may insert redundant copies. Despite some improvements [26][62], the main–trace–first approach limits the control over off–trace penalties.

Percolation Scheduling (PS) [48] performs code compensation locally, within the scope of a primitive code motion. For this reason, it is difficult to control the global impact of compensation code on schedule lengths.

Global Selective Scheduling (GSS) [46] is an approach more conscious of penalties imposed by compensation code. After each code motion, bookkeeping is performed by a depth–first traversal of the so–called control flow graph from the operation's new position to the original location. Compensation code is inserted before merge junctions during the traversal.

As we aim at time–constrained applications, we must have control over the impact of code motion on *all* paths. For this reason, our constructive approach exploits global code motion. However, global code motion leads to a more complex compensation mechanism than the local code motions in PS, because operations move over a longer distance in one step, possibly crossing several merge junctions and perhaps being blocked by data dependences in some paths (recall Figure 4.6). As opposed to GSS, our bookkeeping does not rely on one depth–first traversal per code motion. Instead, our idea is to encode the mechanism in Boolean form [58]. This allows us to replace the traversal by a few efficient queries (recall Tables 3.1 and 4.1).

4.4.2 Bookkeeping code motions

In this section, we show how the queries in Table 3.1 are used to identify *when* and *where* code compensation has to be inserted. We address the code motion of an operation $o_n \in V_o$ or of a conditional c_k and we assume that either o_n or c_k is moved from BB_j to BB_i such that $BB_i \xrightarrow{*} BB_j$. Code compensation is necessary under two circumstances:

- When the motion of the operation is such that there is some path from the source to BB_j not including BB_i (recall Figure 4.6a). This leads to the insertion of compensation code into some BB unreachable from BB_i, but reaching BB_j. This is called *duplication* and we say that *copies* are inserted.
- When the motion succeeds on one path but is blocked on some other path due to a data dependence (recall Figure 4.6b). This leads to the insertion of compensation code into some BB reachable from BB_i. This scheme is called *replication* and we say that *replicas* are inserted. (Note that this is the necessary support for the second form of speculation in Section 4.2)

In the following, we explain how to perform bookkeeping under these circumstances, without jeopardizing schedule lengths.

When the proposition BB_i dom BB_j holds, there is no need for duplication, because the execution of o_n in its new BB guarantees that o_n is executed on all control paths including its original BB. As a consequence, a necessary and sufficient condition for duplication is $\neg(BB_i \text{ dom } BB_j)$. Algorithm 4.5 shows how to determine the BBs in which an operation has to be duplicated. In the algorithm, the function support(G) denotes the set of all Boolean variables occurring in G. The outer loop iterates over all merge junctions "crossed" during the motion. Next, every BB_x being a predecessor of such a junction is checked. If BB_x reaches the original position of the operation, but it is not reached from the new position, BB_x is inserted in the set Where. In the end, this set contains all BBs in which copies have to be inserted.

ALGORITHM 4.5. Algorithm for finding out where to duplicate an operation

```
procedure whereDuplicate($BB_i$, $BB_j$)
Where := $\emptyset$;
foreach $M_k$ with $c_k \in$ support($G(BB_i)$)
   foreach $BB_x \in$ PRED($M_k$)
      if ($BB_x \xrightarrow{*} BB_j \wedge \neg(BB_i \xrightarrow{*} BB_x)$)
         Where := Where $\cup$ {$BB_x$};
return(Where);
```

Algorithm 4.6 shows how to determine the set of BBs where the motion of o_n is blocked by a data dependence. The outer loop enumerates every operation o_m such that there is a data dependence between o_m and o_n. In the inner loop, every link from o_m pointing to some BB reachable from BB_i is checked. Every basic block BB_k pointed by an unscheduled link is inserted in the set Where. In the end, this set contains all the BBs in which replicas must be inserted.

ALGORITHM 4.6. Algorithm for finding out where to replicate an operation

```
procedure whereReplicate($o_n$, $BB_i$)
Where := $\emptyset$;
foreach $o_m \in$ PROD($o_n$)
   foreach $\lambda \in \Lambda(o_m)$ with $o_m \xrightarrow{\lambda} BB_k \wedge BB_i \xrightarrow{*} BB_k$
      if ($\neg$scheduled($\lambda$))
         Where := Where $\cup$ {$BB_k$};
return(Where);
```

The compound effect of duplication and replication is captured by Algorithm 4.7. First, the necessary and sufficient condition for duplication is checked. If it turns out to be true, the sets returned by Algorithms 4.5 and 4.6 are merged; otherwise, only the second algorithm is performed. The returned set contains all the BBs in which either copies or replicas of o_n have to be inserted.

ALGORITHM 4.7. Algorithm for the insertion of compensation code

```
procedure whereCompensate($o_n$, $BB_i$, $BB_j$)
Where := $\emptyset$;
if ($\neg$($BB_i$ dom $BB_j$))
   Where := Where $\cup$ whereDuplicate($BB_i$, $BB_j$);
Where := Where $\cup$ whereReplicate($o_n$, $BB_i$);
return (Where);
```

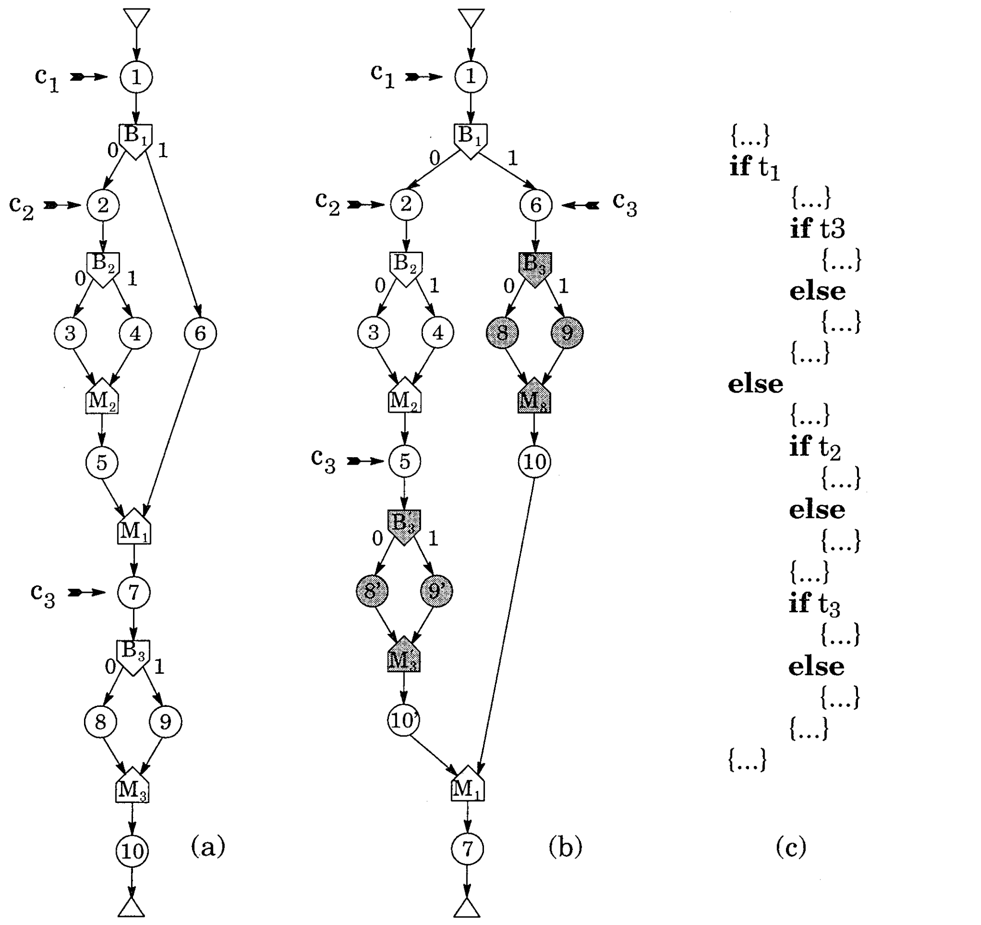
Now the code motion of a conditional is addressed. This motion is similar to the motion of an ordinary operation, except for the fact that it also induces changes in the control flow. This leads to a more elaborate mechanism, because some BBs might be created, some deleted and the order of conditionals might be changed. To keep track of changes in the control flow induced by the motion of a conditional c_k, we rely on the notion of a *conditional subgraph induced by* c_k, as formalized below.

DEFINITION 4.6

Given a BBCG=(U, F), the junctions B_k, $M_k \in U$ and a conditional $c_k \in V$, the *conditional subgraph induced by* c_k is a graph $CS_k = (U', F')$ such that:

- $U' = \{ u_i \in U \mid B_k \text{ dom } u_i \wedge M_k \text{ pdom } u_i \}$,
- $F' = \{ (u_i, u_j) \in F \mid u_i, u_j \in U' \}$.

Reconsider the example of Figure 3.6, which is partially copied in Figure 4.7a for convenience. Assume that conditional c_3 moves from BB_7 to BB_6. A branch must occur just after the new position of c_3. One way of modeling this is by moving the entire conditional subgraph induced by c_3. The result is shown in Figure 4.7b, where the moved subgraph is shaded. Notice that duplication is required in this example, because the proposition $\neg(BB_6 \text{ dom } BB_7)$ holds in Figure 4.7a. As a consequence, conditional c_3 is duplicated into BB_5, which induces the duplication of the whole subgraph, along with the links pointing to its nodes. Compare Figures 3.6a and 4.7c to observe the effect of the duplication of a conditional as a reorganization of the behavioral description.



```
{...}
if t1
          {...}
          if t3
               {...}
          else
               {...}
          {...}
else
          {...}
          if t2
               {...}
          else
               {...}
          {...}
          if t3
               {...}
          else
               {...}
          {...}
{...}
```

(c)

FIGURE 4.7. The result of moving a conditional for the example in Figure 3.6

Casting the motion of a conditional c_k as the motion of the conditional subgraph induced by c_k allows us to share the bookkeeping functions designed for ordinary operations, instead of relying on special duplication mechanisms for conditionals. The difference does not lie in *where* copies are placed, but in *what* is copied, whether a single operation or a whole subgraph.

As code motion of conditionals is likely to lead to code expansion, most methods inhibit it to a certain extent [4][46]. However, for the class of applications at which we are aiming, it might be mandatory to fully support the motion of conditionals for the sake of time–constraint satisfiability. We address this issue in the sequel.

4.4.3 Overcoming the effects of greedy choices

In this section, we show that code compensation may increase schedule lengths, as a result of greedy choices. However, this effect can be analyzed by using one of the extended queries in Table 4.1 and it can be controlled by allowing changes in the original control flow during the scheduling process.

Duplication

Assume for simplicity the duplication of an ordinary operation $o_n \in V_o$, although the reasoning below is also valid for a conditional. Recall that duplication is needed when the proposition $\neg$(BB_i dom BB_j) holds. This can occur in two different scenarios, as follows.

When the proposition BB_j pdom BB_i holds, the duplication of o_n into BBs on different control paths does not increase the number of operations to be executed on any path, since operation o_n has to be eventually executed on every control path including BB_j (recall Figure 4.3b). However, when the proposition does not hold, the number of operations to be executed *does* increase in at least one control path (recall Figure 4.3d), thereby possibly increasing its schedule length.

Consider the example in Figure 4.8a, where BBs are numbered in topological order. Suppose that BB_5 is currently being visited. Assume that operation o_n moves from BB_8 to BB_5. Note that a copy of o_n has to be inserted in BB_6 for compensation. Notice, however, that o_n should originally not execute on the highlighted path in Figure 4.8a, but it will be executed on this path after the "insertion" of the copy in BB_6. As it can not be evaluated *a priori* whether an idle resource will be available in BB_6 to accommodate the operation, this code motion might lead to a longer schedule, possibly jeopardizing time–constraint satisfiability.

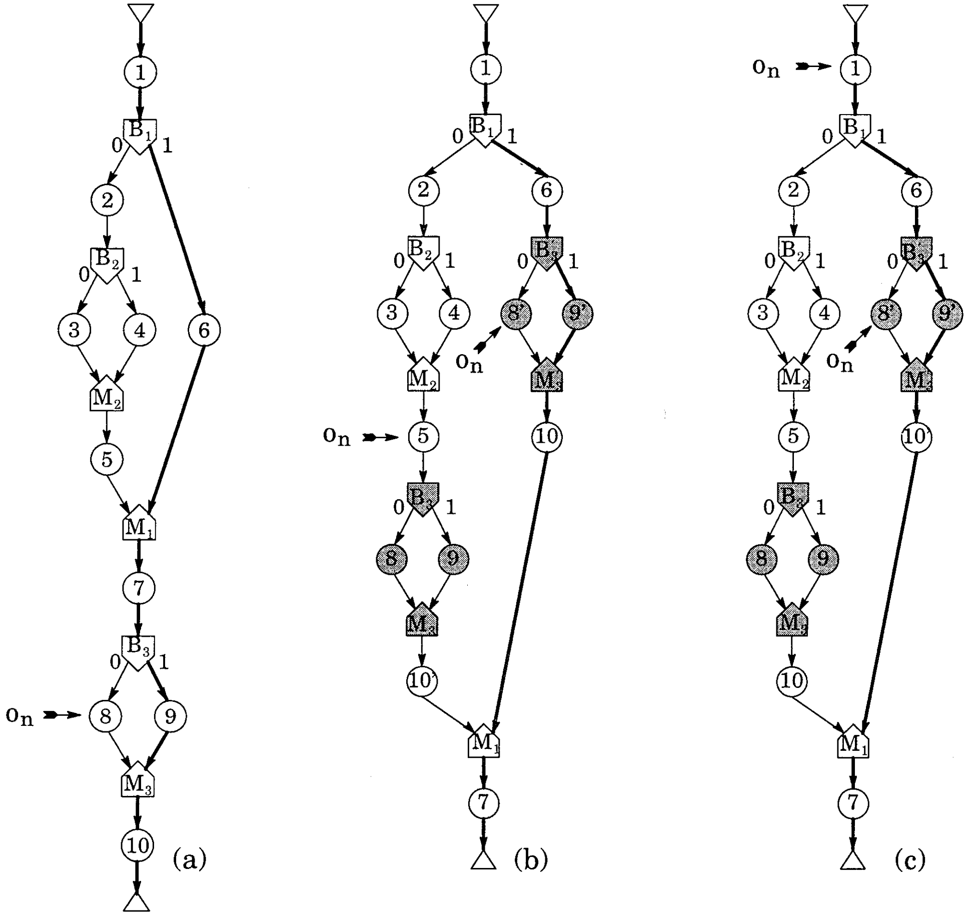


FIGURE 4.8. Example of how to avoid that the insertion of compensation code might increase schedule lengths

In many cases, this issue is alleviated by the fact that a code motion with jeopardizing side–effects is simply not induced when another permutation Π is used to construct an alternative solution. However, this may not always be possible. Since most scheduling mechanisms always "fill" a free resource when there is some available operation capable of occupying it, there might not exist an alternative priority encoding Π able to prevent a code motion whose required compensation code jeopardizes time–constraint satisfiability. To avoid that the insertion of compensation code might hamper the proper exploration of alternative solutions, we propose a technique that changes the control flow and overcomes the problem, as follows.

Let us come back to the example in Figure 4.8, where the motion of operation o_n from BB_8 to BB_5 implies the decision of inserting a copy in BB_6. The key idea to overcome the problem is to postpone the decision until BB_6 is visited. We determine the path requiring compensation code and restructure its basic blocks, thereby conserving the number of operations to be executed. Figure 4.8b illustrates the idea for the above mentioned motion. Note that the subgraph CS_3 was duplicated in a way somewhat similar to the motion of a conditional, thereby avoiding the "inclusion" of operation o_n on the path highlighted in Figure 4.8b. (For simplicity, the duplication of conditional c_3 is omitted in the figure). It is important to note that the extended query (d) in Table 4.1 is used to detect *when* this mechanism must be performed.

Replication

In order to control the impact of replication, a similar expedient can be used. Suppose that BB_1 is the currently visited BB for the example in Figure 4.8a. Assume that operation o_n is moved from BB_8 to BB_1 and suppose that the motion is blocked at BB_6, due to a data dependence. The simple replication of o_n in BB_6 would cause the execution of this operation on the path highlighted in Figure 4.8a. Similarly to the previous example, the solution is to restructure the affected paths, as shown in Figure 4.8c. On the other hand, the fact that o_n is executed twice in path $\langle BB_1, BB_6, BB'_8, BB'_{10}, BB_7 \rangle$ does not increase the length of this path, because the execution of o_n in BB_1 uses an otherwise idle resource.

It should be noted that the above described technique can be disabled if the time–constraint is not tight and it can be enabled *on demand*, as a last resort, if the time–constraint can not be met otherwise.

4.5 Exploiting state equivalence

In the previous section, we have shown that code compensation can be used to broaden the range of legal code motions. Although code motion and speculation can shorten schedules and thereby grant time–constraint satisfiability, the insertion of compensation code may increase the number of states or the microcode size, which might not be affordable for embedded systems.

To overcome this side–effect, we propose in this section a technique for constraining code expansion by eliminating redundant states while applying code motion and speculation. This is performed in such a way that the remaining code expansion closely represents the price to pay for a better schedule. Equivalent states are detected and merged while the SMG is constructed on the fly, during global scheduling. In spite of the fact that these redundant states would be eliminated later on during sequential synthesis, their presence not only slows down scheduling, but also impairs the exploration of alternative solutions. The proposed technique is based on availability analysis and on our Boolean encoding for conditional execution.

A survey of techniques based on state equivalence for logic–level synthesis can be found in [16]. To our knowledge, no method provides a strict checking of state equivalence, while simultaneously applying code motions and speculation during global scheduling. One reason is that built–in scheduler heuristics used in classical approaches make it expensive to predict future scheduler decisions. Another reason is that information on conditional execution is not properly maintained such that it could efficiently be recovered on the fly. In our method, this information is explicitly available, since it is encoded in Boolean form in the very beginning, it is updated after each code motion, and it is efficiently recovered any time a query is answered.

This section is organized as follows. First, the concept of state equivalence in the SMG is formalized. Then, we formulate the criterion for on–the–fly detection of state equivalence and we comment on some aspects of the implementation. To conclude the section, experimental results are summarized and discussed.

4.5.1 Equivalent states in the SMG

To illustrate the notions introduced in this section, we refer to the example in Figure 3.6, which is copied as part of Figure 4.9 for convenience. The BBCG in Figure 4.9b is derived from the description sketched in Figure 4.9a. Assume that the SMGs in Figures 4.9c and 4.9d are obtained from the description in Figure 4.9a. The arrows pointing to a state in the SMGs indicate the conditionals scheduled in that state.

The notion of truth assignment to the conditionals in a state

Among the operations scheduled in a given state s_i, some may be conditionals. Let $\{c_1, c_2, \dots, c_n\}$ represent the set of conditionals scheduled in state s_i. During execution, a *truth assignment* to these conditionals determines their Boolean–valued outcome and can be represented by a Boolean product $G = l_1 \cdot l_2 \cdot \dots \cdot l_k \cdot \dots \cdot l_n$, where each literal l_k represents the guard c_k or its complement. For instance, in the example of Figure 4.9d, the predicate $G = \overline{c}_1 \cdot c_2$ represents the effect of a setting of data for which the outcome of conditional c_1 turns out to be false, and the result of conditional c_2 becomes true.

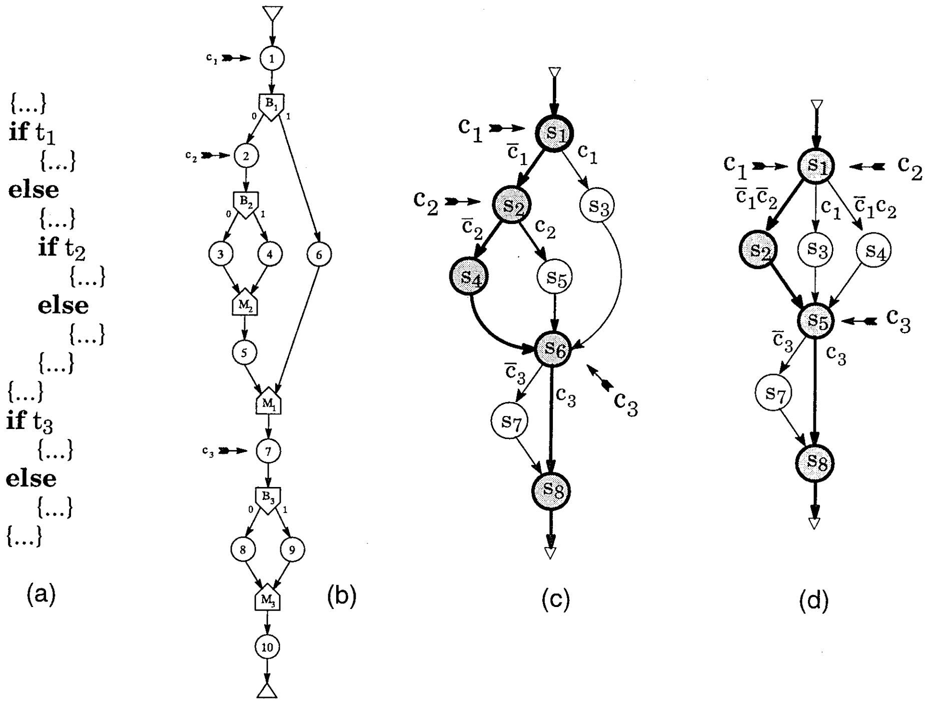


FIGURE 4.9. The relationship between a BBCG and derived SMGs

The notion of enabling predicate of a transition

Our parallelizer schedules states in a top–down manner. After scheduling a state s_i, the next states appointed by the parallelizer are unscheduled. For this reason, all possible transitions from state s_i are *provisionally* constructed, although some of them may be merged later on, as a result of merging equivalent next states. Suppose that $\{c_1, c_2, \ldots, c_n\}$ is the set of conditionals scheduled in state s_i. The parallelizer constructs 2^n edges leaving s_i, each of them representing the effect of a different truth assignment to the conditionals scheduled in s_i. Every transition (s_i, s_j) owns a predicate $G((s_i, s_j)) = l_1 \cdot l_2 \cdot \ldots \cdot l_k \cdot \ldots \cdot l_n$. Later on, each time two transitions are merged, the predicate owned by the resulting transition is obtained by the Boolean sum of the predicates of the merging transitions. If no conditionals are scheduled in state s_i, there is a single transition (s_i, s_j) leaving s_i and $G((s_i, s_j)) = 1$. Consider the examples in Figures 4.9c and 4.9d, where constant predicates are omitted for simplicity. Note, for instance, that the predicate owned by transition (s_4, s_6) in Figure 4.9c is $G((s_4, s_6)) = 1$, since no conditional is scheduled in state s_4. Observe also that the predicate owned by transition (s_1, s_2) is $G((s_1, s_2)) = \bar{c}_1$ for the SMG in Figure 4.9c and

$G((s_1, s_2)) = \overline{c}_1 \cdot \overline{c}_2$ for the SMG in Figure 4.9d. Notice that the predicate for the second case contains two guards, because two conditionals are scheduled in s_1. Although four transitions would be expected to leave state s_1 in Figure 4.9d, only three of them remain in the final SMG, because the transitions with predicates $c_1 \cdot c_2$ and $c_1 \cdot \overline{c}_2$ were merged into the transition (s_1, s_3), whose predicate can be written as $G((s_1, s_3)) = c_1 \cdot c_2 + c_1 \cdot \overline{c}_2 = c_1$.

In summary, given a SMG=(S, T), every edge $(s_i, s_j) \in T$ owns an *enabling predicate* $G((s_i, s_j))$. The value of $G((s_i, s_j))$ is determined at execution time by a truth assignment to the conditionals scheduled in state s_i. Among the edges leaving s_i, the transition actually taken during execution is the one whose predicate evaluates to true. It might be worthwhile to notice that the enabling predicate of a transition is in general a sum of Boolean products, that the predicates of outgoing transitions of a state are mutually exclusive for a deterministic FSM and also that the sum of all those predicates must be a tautology.

These notions are used in the following to formulate the concept of state equivalence in the SMG.

A reformulation for the notion of state equivalence

The classical notion of state equivalence has its roots in the synthesis and optimization of sequential circuits. It relies on a FSM model, consisting of an automaton which consumes a sequence of inputs and generates a sequence of outputs. Two states, say s_n and s_m, are equivalent if the output sequences of two instances of the FSM, one of them initialized in state s_n and the other in state s_m, match for any input sequence [16][31].

Since HLS precedes sequential synthesis in the design flow, the HLS representation for the control unit is more abstract and typically based on a symbolic description of a FSM. In this model, an output pattern of the symbolic FSM is associated with the set of *operations executing in a given state,* which is sometimes called a *bundle* in compiler–technology terminology [22]. An input pattern of the symbolic FSM is associated with the *predicate* representing a truth assignment to the conditionals scheduled in the previously executed state.

Since we can not use the classical notion of state equivalence, we rely on the following concept of equivalent states in the SMG. Assume that OP_n denotes the bundle containing the operations scheduled in some state s_n. Let $\langle s_n, s_{n+1}, \dots, s_{n+k} \rangle$ be a path in the SMG and let $\langle OP_n, OP_{n+1}, \dots, OP_{n+k} \rangle$ be the sequence of bundles associated with each state on that path.

DEFINITION 4.7

Let s_n be a state and let $\langle(s_n, s_{n+1}), (s_{n+1}, s_{n+2}), \ldots, (s_{n+k-1}, s_{n+k})\rangle$ be a sequence of k transitions starting at s_n. Given a sequence of predicates $\mathcal{G} = \langle G_1, G_2, \ldots, G_k\rangle$ such that $G_i = G((s_n, s_{n+i}))$ with $1 \le i \le k$, the *sequence of bundles induced by* $\mathcal{G}$, written $OP(s_n, G_1, G_2, \ldots, G_k)$, is the sequence $\langle OP_n, OP_{n+1}, \ldots OP_{n+k}\rangle$.

For instance, for the example in Figure 4.9c, the sequence of predicates $\langle \bar{c}_1, \bar{c}_2, 1, c_3\rangle$ induces the sequence of bundles $\langle OP_1, OP_2, OP_4, OP_6, OP_8\rangle$. Similarly, for the example in Figure 4.9d,the sequence $\langle \bar{c}_1 \cdot \bar{c}_2, 1, c_3\rangle$ induces the sequence of bundles $\langle OP_1, OP_2, OP_5, OP_8\rangle$. Note that these sequences of bundles represent the operations executed on the highlighted paths in those figures.

Now we are able to introduce the notion of state equivalence in the SMG, as follows:

DEFINITION 4.8

States s_n and s_m are *schedule equivalent*, written $s_n \overset{\Phi}{=} s_m$, if and only if $OP(s_n, G_1, G_2, \ldots, G_k) = OP(s_m, G_1, G_2, \ldots, G_k)$, for every possible sequence $\langle G_1, G_2, \ldots, G_k\rangle$.

This definition is the reformulation, from a HLS perspective, of the classical concept for sequential synthesis defined in [31] and it can be interpreted as follows. In order to be equivalent, not only the bundles of operations scheduled in states s_n and s_m must coincide, but also the bundles of every state reachable from them under a same sequence of enabling predicates. The notion of schedule–equivalent states is illustrated in Figure 4.10. In the SMG of Figure 4.10a, the operations scheduled inside states are shown explicitly and enabling predicates with constant value are omitted.

Note that duplication of conditionals has occurred: conditional c_2 is scheduled in states s_2 and s_4 and c_3 is scheduled in states s_6 and s_{13}. Note, for instance, that states s_2 and s_4 are not equivalent, since $OP_2 \neq OP_4$. On the other hand, several states in the SMG of Figure 4.10a are equivalent, namely $s_5 \overset{\Phi}{=} s_{12}$, $s_6 \overset{\Phi}{=} s_{13}$, $s_7 \overset{\Phi}{=} s_{14}$, $s_8 \overset{\Phi}{=} s_{15}$, $s_9 \overset{\Phi}{=} s_{16}$ and $s_{10} \overset{\Phi}{=} s_{17}$. Let us consider states s_6 and s_{13}, for instance. Observe that not only the equality $OP(s_6, \bar{c}_3, 1, 1, 1) = OP(s_{13}, \bar{c}_3, 1, 1, 1)$ holds, but also the equality $OP(s_6, c_3, 1, 1, 1) = OP(s_{13}, c_3, 1, 1, 1)$.

Therefore, all the shaded states in Figure 4.10a can be considered redundant and can be merged with their respective equivalent states, as shown in Figure 4.10b. Our goal is to avoid the construction of a solution like the one in Figure 4.10a. However, the concept of state equivalence assumes that the SMG is completely constructed. Since during scheduling, some states and transitions are not yet defined, the formulation above can not be applied directly. In the sequel, we show how to exploit the notion of state equivalence on the fly in order to prevent the scheduling of redundant states.

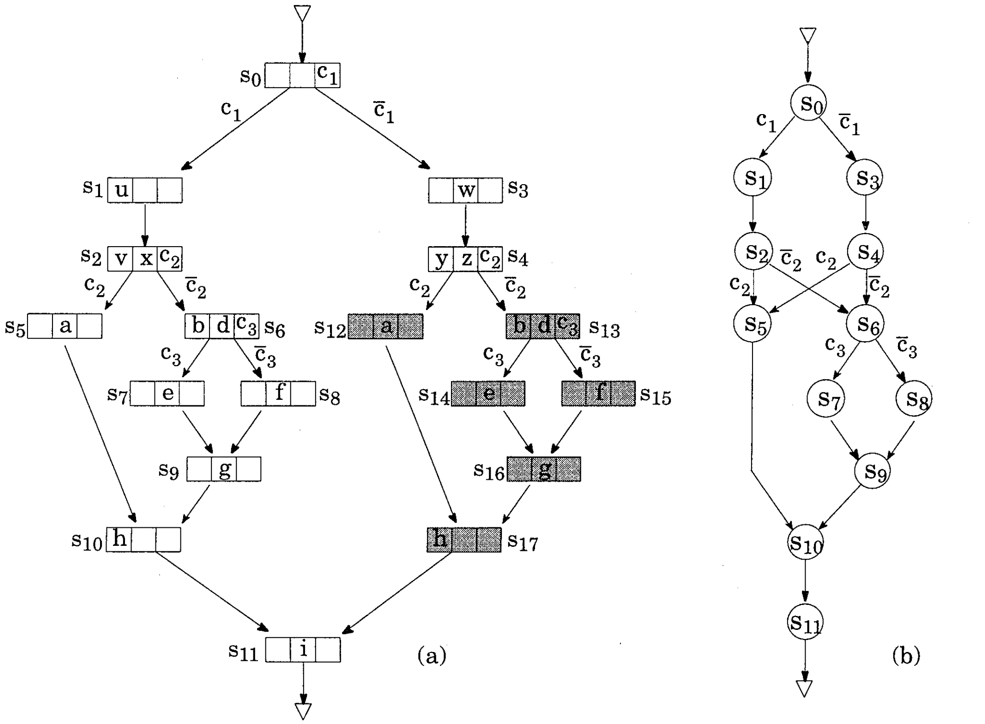


FIGURE 4.10. Illustrative example for state equivalence

4.5.2 On–the–fly detection of state equivalence

Before presenting our criterion for on–the–fly detection of state equivalence, we formulate some essential notions. The first of them is the concept of a *sequence of transitions*. Although a sequence of transitions may be not completely defined in the course of the scheduling process, it can be captured by a predicate, as illustrate below.

Consider the example in Figure 4.11. Let s_m be a state within a basic block BB_i. Suppose that an operation o_z is to be executed in some state, say s_x, reachable from s_m. As a consequence, this operation must be linked to some BB_j such that there is a path p with $BB_i \xrightarrow{p} BB_j$. Assume, for instance, that $G(BB_i) = \bar{c}_1 \cdot c_2$ and that $G(BB_j) = \bar{c}_1 \cdot c_2 \cdot c_3 \cdot \bar{c}_4$, as illustrated in the fragment of BBCG given in Figure 4.11a. Since BB_i precedes BB_j on path p, the guards c_3 and c_4 must be associated with branches occurring after BB_i on path p. This illustrates that a predicate determining a sequence of transitions starting from s_m to s_x can be obtained by eliminating from predicate $G(BB_j)$ the guards in common with $G(BB_i)$. This is implemented by a Boolean operator called smoothing. The *smoothing* of a predicate G with respect to guard c, written $\mathcal{S}_c(G)$, is obtained by dropping from consideration all the occurrences of guard c in G [16]. For instance, for the example in Figure 4.11a, the predicate $\Gamma = c_3 \cdot \overline{c}_4$ is obtained by smoothing in the predicate $G(BB_j)$ the guards which appear in $\text{support}(G(BB_i))$. Note that the predicate Γ determines the sequence of transitions highlighted in Figure 4.11b. This notion is formalized below.

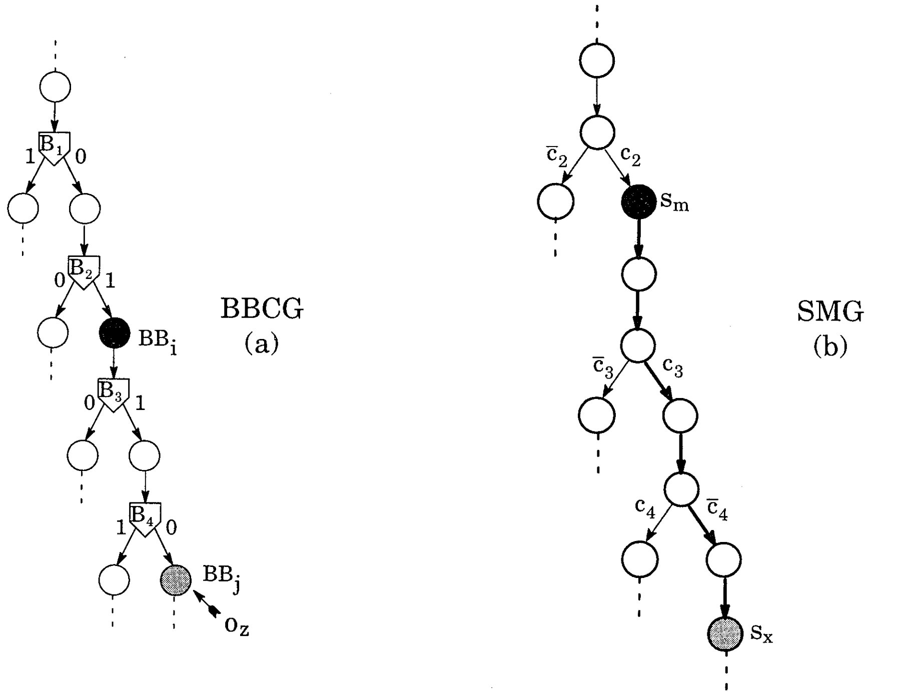


FIGURE 4.11. An illustration of how to interpret the smoothing with respect to the predicate of a basic–block

DEFINITION 4.9

Let $\mathcal{T}_i \in T$ denote some transition of the SMG. We say that a sequence of transitions $\langle \mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_k \rangle$ is *induced by a predicate* Γ, if and only if $\Gamma \cdot G(\mathcal{T}_i)$ is satisfiable for every transition $\mathcal{T}_i$ in the sequence.

For instance, the predicate $\Gamma = \overline{c}_1 \cdot \overline{c}_2 \cdot c_3$ induces the sequence of transitions highlighted in Figure 4.9c, but not the sequence $\langle(s_1, s_3), (s_3, s_6), (s_6, s_8)\rangle$. Note that a given predicate may induce different sequences of transitions, depending on how the operations are scheduled in the states. For example, assuming that the SMGs in Figures 4.9c and 4.9d are alternative solutions, the predicate $\Gamma = \overline{c}_1 \cdot \overline{c}_2 \cdot c_3$ induces the distinct sequences of transitions highlighted in each of those figures.

Having introduced these basic notions, we can now formalize the relationship between the DFG and the SMG by utilizing the BBCG. Let $\{c_1, c_2, \dots, c_N\}$ be the set of all conditionals in the DFG and let Γ be a predicate. If an operation o_n is linked to some BB_i such that $G(BB_i) \cdot \Gamma$ holds, then the operation must be executed in some state reachable through the sequence of transitions induced by predicate Γ.

A criterion for detecting state equivalence

Our goal is to detect equivalent states, not after scheduling is completed, but *during* the construction of the SMG itself. Given a basic block BB_i, recall that $R_i(n)$ represents the resource occupation in some state $s_{i,n}$ and that $A_{i,n}$ denotes the respective set of available operations. For simplicity, let us drop the first index from now on, so that they are denoted as A_n and R_n, respectively. Given a state s_n, already scheduled, a state s_m still to be scheduled, and the pairs (A_n, R_n) and (A_m, R_m), we want to check if s_n and s_m will turn out to be equivalent. Based on the initial status on entry to states s_n and s_m, we have to *predict* if the process of scheduling, starting at those states, results on equivalent sequences of bundles.

In our approach, the choices of the scheduler can be predicted at low cost, since heuristics are removed from the scheduler and placed in the external explorer. Besides, our Boolean encoding for conditional execution provides an efficient way of checking if the operations to be scheduled in states reachable from s_n and s_m are executed under exactly the same predicates. The on–the–fly detection of state equivalence relies on the three following key properties:

- Given the pairs (A_n, R_n) and (A_m, R_m), it is straightforward to predict if the bundles associated with states s_n and s_m will coincide.
- It is possible to tell in advance if some operation, say o_n, will eventually be scheduled in states reachable from s_n and s_m.
- An operation o_n may be executed, possibly on different paths in the SMG, but under the same sequence of enabling predicates.

Each of the properties listed above is formulated in the following, where we assume a state s_n within some basic block BB_i and a state s_m within BB_j.

Scheduler predictability

With respect to the first of these properties, recall that, for our scheduler, the set of operations selected to be executed in a given state s_n depends only on the resource occupation and on the available operations on entry to s_n, as well as on a given priority encoding Π. As a consequence, if the equality $(A_n, R_n) = (A_m, R_m)$ holds, then $OP_n = OP_m$ also holds.

Reachability from available operations

Let $\mathcal{R}(o_y)$ denote the *set of operations reachable from operation* o_y in the DFG=(V, E) excluding branch and merge nodes, which is obtained as follows:

$$\mathcal{R}(o_y) = \{\, o_z \in V \,|\, o_y \xrightarrow{*} o_z \wedge (\theta(o_z) = \text{true})\}. \tag{4.6}$$

Now, given a state s_n within BB_i, let us find the set of all operations that are eventually executed on some path from BB_i to the sink. This set can be found by applying the concept of reachability above to each operation available at state s_n. This set, written as $\mathcal{R}_i(A_n)$, is obtained as below:

$$\mathcal{R}_i(A_n) = \{\, o_z \in \bigcup_{o_y \in A_n} \mathcal{R}(o_y) \,|\, o_z \xrightarrow{\lambda} BB_k \wedge BB_i \xrightarrow{*} BB_k \,\}. \tag{4.7}$$

As a consequence, if it is known that $\mathcal{R}_i(A_n) = \mathcal{R}_j(A_m)$, we can conclude that the same set of operations is bound to be executed in states reachable either from s_n or from s_m. However, this does not guarantee that a given operation is executed on different paths under exactly the same predicate, which motivates the ensuing analysis.

Execution under a same sequence of enabling predicates

Remember that an operation o_z may be linked to several BBs reachable from some BB_i, as a result of code motion. To capture the joint effect of all “copies” of o_z, we first find the set of all links emanating from o_z that are linked to some BB reachable from BB_i. This set, written as $\Lambda_i(o_z)$, is obtained as follows:

$$\Lambda_i(o_z) = \{\, \lambda \in \Lambda(o_z) \,|\, o_z \xrightarrow{\lambda} BB_k \wedge BB_i \xrightarrow{*} BB_k \,\} \tag{4.8}$$

The *joint execution predicate* of an operation o_z on all control paths including a given BB_i, written as $G_i(o_z)$, can be expressed as follows:

$$G_i(o_z) = \sum_{\lambda \in \Lambda_i(o_z)} G(\lambda). \tag{4.9}$$

Assume that operation o_z will be scheduled in some state s_x reachable from s_m (reconsider Figure 4.11). Let us now find a predicate able to induce a sequence of transitions leading to s_x. Since such a predicate should take into account only the effect of conditionals *still to be scheduled* from s_m to s_x, we have to drop all the guards of conditionals scheduled prior to s_m, i.e., all the guards in $G(BB_i)$. Therefore, given a state s_m within a basic block BB_i, the predicate inducing a sequence of transitions leading to some state s_x in which operation o_z is executed, written $\Gamma_i(o_z)$, is obtained by Algorithm 4.8.

ALGORITHM 4.8. Algorithm for determining the predicate $\Gamma_i(o_z)$

```
$\Gamma_i(o_z) := G_i(o_z)$;
foreach $c \in \text{support}(G(BB_i))$
    $\Gamma_i(o_z) := \mathscr{S}_c(\Gamma_i(o_z))$;
```

With this notion in mind, given states s_n and s_m within basic blocks BB_i and BB_j, respectively, if we find out that $\Gamma_i(o_z) = \Gamma_j(o_z) = \Gamma$, we can conclude that operation o_z will be executed, on *different* paths starting from s_n and s_m, but under sequences of transitions induced by a *same* predicate Γ.

Now, we are able to formalize the criterion for on–the–fly detection of state equivalence.

THEOREM 4.1

Let s_n and s_m denote states within basic blocks BB_i and BB_j, respectively. Assume that all availability sets are ordered according to a given priority encoding Π. The equivalence $s_n \overset{\Phi}{=} s_m$ holds for a given Π, if and only if all the following conditions hold:

- $A_n = A_m$ and $R_n = R_m$,
- $\mathfrak{R}_i(A_n) = \mathfrak{R}_j(A_n) = \mathfrak{R}$,
- $\forall o_z \in \mathfrak{R} : \Gamma_i(o_z) = \Gamma_j(o_z)$.

PROOF OUTLINE

To prove this theorem, assume a path p from s_n to the sink and a path q from s_m to the sink, such that both p and q are determined by sequences of transitions induced by the same sequence of enabling predicates, say $\mathfrak{G}$. Let s_{n+x} and s_{m+x} represent states reachable via paths p and q, respectively, through a number x of transitions. Assume by hypothesis that $A_{n+x-1} = A_{m+x-1}$ and $R_{n+x-1} = R_{m+x-1}$. Since both availability sets are ordered according to a same priority encoding Π, the scheduler will select exactly the same operations in both cases, i.e., $OP_{n+x-1} = OP_{m+x-1}$. As a consequence, resource occupation on exit to both states will be the same, i.e., $R_{n+x} = R_{m+x}$. Note that the set A_{n+x} can be expressed as the union of two components. The first is constructed by taking the *remaining* (unscheduled) elements of set A_{n+x-1}, that is to say, $A_{n+x-1} \backslash OP_{n+x-1}$. The second consists of *new* elements made available by the operations just scheduled.

Note that $A_{n+x-1} \backslash OP_{n+x-1} = A_{m+x-1} \backslash OP_{m+x-1}$, for the first component. For the second, recall that A_{n+x} and A_{m+x} are subsets of $\mathcal{R}_i(A_n)$ and $\mathcal{R}_j(A_n)$. Since the theorem's second condition ensures that the supersets coincide and since the third condition guarantees that the new operations to be included in both A_{n+x} and A_{m+x} will be the same for an arbitrary sequence of predicates $\mathcal{G}$ enabling the transitions on paths p and q, we conclude that $A_{n+x} = A_{m+x}$. Since the theorem's first condition enforces the proven hypothesis to hold for $x = 1$, we conclude by induction that $OP_{n+x} = OP_{m+x}$ for an arbitrary x. ❑

Similarly to the classical concept of state equivalence, the states of the SMG can be organized as a unique partition in terms of equivalence classes [16]. As a result, if a new SMG is constructed with a single representative state from each class, no other equivalent SMG has fewer states [31]. However, since Theorem 4.1 assumes a fixed but arbitrary priority encoding Π, we conclude that our technique guarantees a minimal state equivalent SMG, *given an arbitrary priority encoding*. As our approach allows the exploration of alternative SMGs induced by different priority encodings, it provides a clean solution to the optimization of the number of states.

4.5.3 Comments on the implementation

An efficient implementation for Theorem 4.1 is obtained as follows. The pair (A_n, R_n) is stored in a table for every scheduled state s_n. For a given "empty" state s_m about to be scheduled, we first check if the *necessary* condition $(A_m, R_m) = (A_n, R_n)$ is satisfied. This test can efficiently be performed by means of a hash table. Only if a hit occurs, state s_m has to go through the whole test on equivalence, as summarized in Algorithm 4.9.

Algorithm 4.9. An algorithm for exploiting state equivalence

```
procedure equivalent_state($s_m$)
if ($\exists s_n \in S \mid (A_n, R_n) = (A_m, R_m)$ ) /* necessary condition */
   if ($s_n \overset{\Phi}{=} s_m$)                    /* Theorem 4.1          */
      return ($s_n$);
return(none);

procedure handle_current_state($s_m$)
$s_n$ = equivalent_state($s_m$);
if ($s_n \neq$ none)
   merge $s_m$ with $s_n$;
else
   schedule $s_m$;
```

The test on the second condition of Theorem 4.1 is performed efficiently by keeping the sets $\mathfrak{R}_i(A_n)$ and $\mathfrak{R}_j(A_m)$ ordered by the priority encoding. Checking the third condition is fast, since it relies on queries involving predicates whose number of guards is bounded by the depth of conditional nesting, typically a small fraction of the total number of tests.

The use of a table to check the initial status on entry to states s_n and s_m is similar to the implementation suggested in [4] for the sake of loop pipelining. However, that method does not handle state equivalence. Although it is shown in [4] that the test $(A_n, R_n) = (A_m, R_m)$ is a necessary and sufficient condition for detecting the boundaries of a pipelined loop body, it represents just a necessary condition for state equivalence.

4.5.4 Experimental results

To evaluate the impact of the described technique, we have performed a series of experiments, organized under the following set–up. Speculation can be performed through an unrestricted number of branches. Global code motions leading to duplication of ordinary operations or duplication of conditionals are enabled (the latter changes the control flow). A randomly generated sequence of priority encodings is used to construct a large number of solutions, from which statistics are derived. As a result, we can evaluate the average impact of our technique for an *arbitrary priority encoding*. The examples used for the experiments are listed in the first column of Table 4.3. Each example is submitted to distinct resource constraints, which are labeled in the fourth column, leading to different cases for each example. The resource constraints corresponding to these cases, along with the adopted operation delays, are described in detail in Appendix A.

In a first experiment, we compare the number of states *with* and *without* the exploitation of state equivalence. Results are shown in Table 4.3. The mean value for the schedule length of the longest path in the SMG is denoted by L_i in the table. Both the mean value and the standard deviation are given for the number of states. The last column quantifies the relative increase of the number of states which would occur, had we not exploited state equivalence. It was observed that the values of L_i coincide in both cases and for every example. This is an evidence that, without exploiting equivalence, we are looking at too many "equivalent" solutions and thus paying too high a price for the same schedule quality.

The mean value for the number of states is given in the shaded columns. Observe that, without exploiting equivalence, the size of the SMG seems unpractical for DFGs of reasonable size. Besides, for DFGs with complex control flow, as it is indeed the case for the last example, the expansion is probably not affordable for embedded systems. To overcome expansion, most methods either restrict the types of code motion allowed [4][46][62] (e.g. by disallowing the duplication of conditionals), or have to rely on smart heuristics to alleviate the problem [4]. The results in the shaded columns give some evidence that, when state equivalence is exploited, some restrictions usually imposed on code motions can be relaxed, since the state expansion will be controlled by the on–the–fly merging of equivalent states.

TABLE 4.3 The impact of on–the–fly exploitation of state equivalence

example	nodes	BBs	case	L_i	without		with		
					#states		#states		
					mean	σ[%]	mean	σ[%]	
waka	46	10	A	7.8	14.5	4.5	11.6	7.6	1.2
[72]			B	7.9	14.7	5.7	11.8	8.9	1.2
kim	48	10	B	8.8	21.8	4.4	19.8	6.2	1.1
[37]			C	6.9	15.7	5.8	14.9	5.2	1.1
rotor	66	10	A	11.0	35.0	0.0	21.4	8.5	1.6
			B	8.0	21.3	2.1	17.3	2.6	1.2
			C	7.0	20.0	0.0	14.6	3.3	1.4
[52]			D	7.0	20.0	0.0	14.6	3.3	1.4
			E	9.8	29.8	1.2	19.5	4.9	1.5
			F	8.0	24.0	0.0	15.6	3.2	1.5
			G	8.0	24.0	0.0	15.6	3.2	1.5
s2r	122	22	A	14.7	127.5	3.2	71.9	14	1.8
			B	9.5	77.0	4.4	59.2	6.9	1.3
			C	8.9	73.2	4.3	54.2	8.0	1.3
[52]			D	8.9	73.2	4.3	53.9	7.7	1.4
			E	13.1	95.5	3.8	73.0	6.9	1.3
			F	10.0	78.9	5.1	58.0	8.9	1.4
			G	9.4	75.8	5.1	56.9	7.1	1.3
			H	9.4	75.8	5.1	56.9	7.1	1.3
kim_big	464	52	A	59	2406	16	495	17	4.9
[37]			C	59	2395	15	476	15	5.0
			D	58	2317	16	437	20	5.3

The standard deviation for the number of states is denoted by σ in the table. Notice that the value of σ tends to grow when state equivalence is exploited. This shows that the size of the SMG is actually more sensitive to the priority encoding than we could tell if the technique had not been applied. This means that the solutions, apparently with about the same number of states during exploration, may end up in very different SMG sizes. As a consequence, not merging equivalent states on the fly is likely to impair exploration. Therefore, further phases of a design flow, like sequential synthesis, might not be able to compensate for the overlooking of superior solutions in earlier phases.

To quantify the impact on runtime, we have measured the average time to build up a SMG for an arbitrary priority encoding. Average runtime with and without exploiting state equivalence are expressed in seconds for a HP9000/735 workstation. The results in Table 4.4 show that our technique accelerates the construction of solutions. The reason is that the time actually spent on state equivalence checking is less than the time to schedule all redundant states. The speed gain is shown in the last column of Table 4.4.

TABLE 4.4 Average time to construct a solution

example	case	without	with	gain
rotor	A	0.10	0.08	1.2
	E	0.09	0.08	1.1
s2r	A	0.58	0.45	1.3
	E	0.52	0.48	1.1
kim_big	A	26.7	11.9	2.2
	C	27.6	11.6	2.4
	D	26.0	10.3	2.5

4.6 Discussion

In this chapter, we have shown how code motion is supported within our constructive approach. Our modeling of code motion relies on two basic concepts: link and predicate. These concepts allows us to combine graph algorithms and Boolean techniques in the implementation of the HLS tool.

Some HLS methods for control–flow dominated applications do not support speculation at all [9][12][37]. Our support for speculation is more general than most related HLS methods: although pre–execution is allowed in [34][52][54][72], the second form of speculation presented in Section 4.2 is either not supported or neglected. In particular, the method reported in [52], which also relies on Boolean manipulation, is limited by the model adopted for scheduling. In that approach, a linear–time sequence of time steps is employed in such a way that it would be cumbersome to support the scheduling of an operation more than once per trace. As a consequence, the support for speculation in [52] is limited by the absence of powerful code compensation techniques. In our approach, however, where the notion of *state* is predominant over the notion of *time step* and where states are clustered within BBs, this support is naturally provided via code compensation.

This chapter has also shown that global code motion can be induced by determining the operations available for scheduling in a given state, regardless of their original position in the control flow. Despite the fact that the main notions for availability analysis are not new [4][19][46][62], those classical analyses from the compiler arena rely on a so–called control flow graph representation. We have shown how to cast those notions into a different representation, which is based on a DFG, a BBCG and Boolean queries. Although as general as the above mentioned techniques, our availability analysis is relatively simpler for two main reasons. On the one hand, the so–called anti–dependences and output–dependences [30][46] occurring in classical compiler representations are eliminated by the DFG representation. On the other hand, our Boolean queries replace the traversals traditionally employed to search control paths for new available operations. Since most HLS approaches are based on some form of DFG representation and since public–domain programs for Boolean manipulation (such as BDD packages) are widely available, our formulation for availability analysis and code compensation can be incorporated by other HLS approaches at the expense of a few extensions.

Although the restriction to certain types of code motion is common practice in the compiler–technology domain [4][46][62], such an expedient is not suitable for time–constrained systems, because some "forbidden" code motions might be essential for time–constraint feasibility. The use of Boolean queries to identify different types of code motion, as summarized in Table 4.1, allows the implementation of a flexible HLS tool for exposing parallelism. Although unrestricted code motion is supported by the approach, some types of code motion could be disabled on demand depending on how tight the time constraint is. For instance, the cases (b) and (d) in Table 4.1 tend to increase the number of states. They could be disabled by an option of the tool if the time constraint is not tight. Therefore, a HLS tool with unrestricted code motion capability and with facilities for inhibiting code motion *on–demand* provides more control on the quality of the final solution than a tool limited beforehand by built–in heuristics. The price to pay for such flexibility seems affordable for the following reasons. First, the queries in Table 4.1 involve predicates that are always Boolean products, leading to a worst–case complexity that is polynomial with the number of guards of a predicate. Second, since the number of guards in the predicate is bounded by the depth of conditional nesting (typically a fraction of the total number of conditionals), the average–case complexity is likely to be small in most practical cases.

Even though unrestricted code motion (especially duplication of conditionals) may increase the number of states, we have shown that it can be supported without inserting redundant states, by detecting and merging equivalent states on–the–fly. Experimental results give evidence that if a HLS tool is required to make use of flexible code motions in order to face a tight time–constrained problem, the size of the SMG would be unpractical without on–the–fly exploitation of state equivalence. Besides, our technique speeds up the construction of solutions, since a large reduction on the number of states can be obtained by the use of an efficient state equivalence checking. Since the flow of control forks in two or more states each time conditionals are scheduled, top–down scheduling would result in a tree structure. Actually, some approaches are reported [33][34] where top–down scheduling is performed on a tree structure and states may be merged later on, during a bottom–up rescheduling phase [34]. Unlike those methods, our technique finds *join* junctions in the SMG dynamically.

Now, let us focus on some aspects of the exploration of alternative solutions. It should be noted that, since the priority encoding Π induces an ordering on each set A_k and different orderings typically induce distinct code motions, our approach provides a mechanism to seek for the code motions leading to high–quality results. Different priority encodings typically lead to different number of states (see standard deviations in Table 4.3). Since our method guarantees a minimal state equivalent SMG *for a given* Π, the optimization of both schedule length and number of states can be achieved by the exploration of solutions induced by different priority encodings.

Chapter 5 Code–motion pruning

In this chapter, two techniques for improving the exploitation of ILP are proposed. The first technique is a method able to examine data–flow information in such a way that potential downward code motions are captured by the links. The second technique is a method to prune inefficient code motions. It relies on the idea that, given an arbitrary priority encoding, code motion is worth doing as far as it does not jeopardize the schedule length of any control path. The main topics of this chapter are:

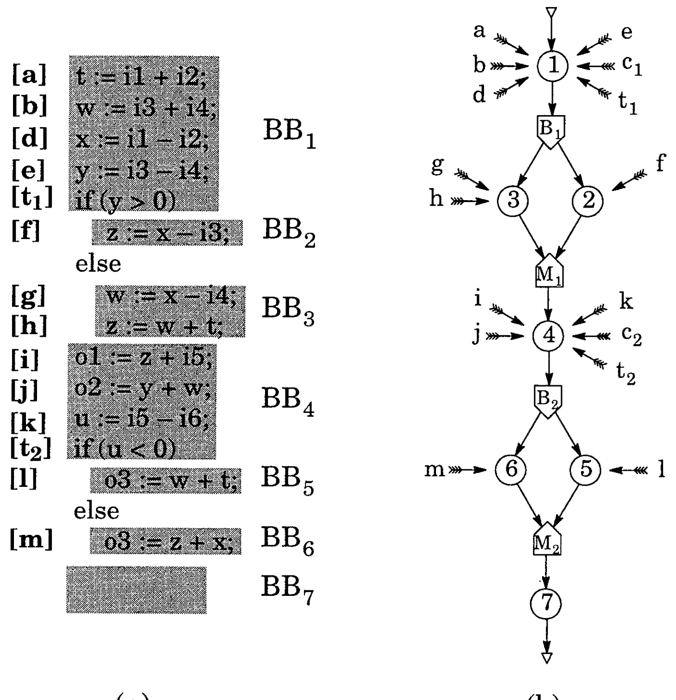
- **How to capture data–flow information in the links:**
  We propose a data–flow analysis technique based on freedom for downward code motion. The analysis is efficiently performed starting from a DFG.
- **How to prune inefficient code motions:**
  Given some priority encoding, the result of the above mentioned analysis is used to induce a precedence relation, which is employed to modify the original linear ordering determined by the priority encoding. This gives rise to a reordering of the sets of available operations. Such a reordering prevents code motions that are not worth doing, since (as will be shown) the operations moved can not be accommodated within the available resources.
- **Experimental evidence of impact on search space:**
  A set of experimental results gives evidence that the pruning technique improves the quality of global scheduling. It is shown that the application of the technique increases the density of promising solutions in the search space, paving the way to a faster exploration of alternative solutions.

5.1 A data–flow analysis technique

In this section, we introduce a data–flow analysis technique that finds a new set of links from the set of initial links. Given some operation o_n and its initial link, each new link is determined such that o_n being linked to some basic block BB_j implies o_n to be scheduled in some BB reaching BB_j or in BB_j itself, but not any further down in the flow of control. We henceforth refer to this new set of links as the *lowest links*. We first introduce some fundamental notions and then we formulate our analysis. To illustrate the notions addressed throughout this section, we refer to the example shown in Figure 5.1, where a behavioral description, its BBCG and the initial links are shown. The respective DFG is illustrated in Figure 5.2.

5.1.1 Fundamental notions

The traditional representation for global data–flow analysis [1] is a so–called control flow graph, whereas we rely on a DFG. For this reason, the notions introduced in this section are the counterparts of classical concepts of global data–flow analysis in the DFG domain.



```
[a]   t := i1 + i2;
[b]   w := i3 + i4;
[d]   x := i1 - i2;      BB1
[e]   y := i3 - i4;
[t1]  if (y > 0)
[f]      z := x - i3;    BB2
      else
[g]      w := x - i4;    BB3
[h]      z := w + t;
[i]   o1 := z + i5;
[j]   o2 := y + w;       BB4
[k]   u := i5 - i6;
[t2]  if (u < 0)
[l]      o3 := w + t;    BB5
      else
[m]      o3 := z + x;    BB6
                         BB7
```

(a) (b)

FIGURE 5.1. A behavioral description, its BBCG and initial links

We say that a value x is *used* in some basic block BB_i if x is consumed by some operation linked to BB_i. A value is used on some path p if it is used in some BB included in p. One of the keys of our analysis is the notion of live value. A value x is *live* at a basic block BB_i if x is produced by an operation *not* linked to BB_i and x can be used on a path from BB_i to the sink, as formalized below.

DEFINITION 5.1

The value computed by an operation o_n is *live* at basic block BB_i, written live(o_n, BB_i), if and only if both following conditions hold:

- $o_n \not\twoheadrightarrow BB_i$,
- $\exists o_m \in \textsc{cons}(o_n) : \exists BB_j \in U : o_m \overset{\wedge}{\twoheadrightarrow} BB_j \wedge BB_i \xrightarrow{*} BB_j$.

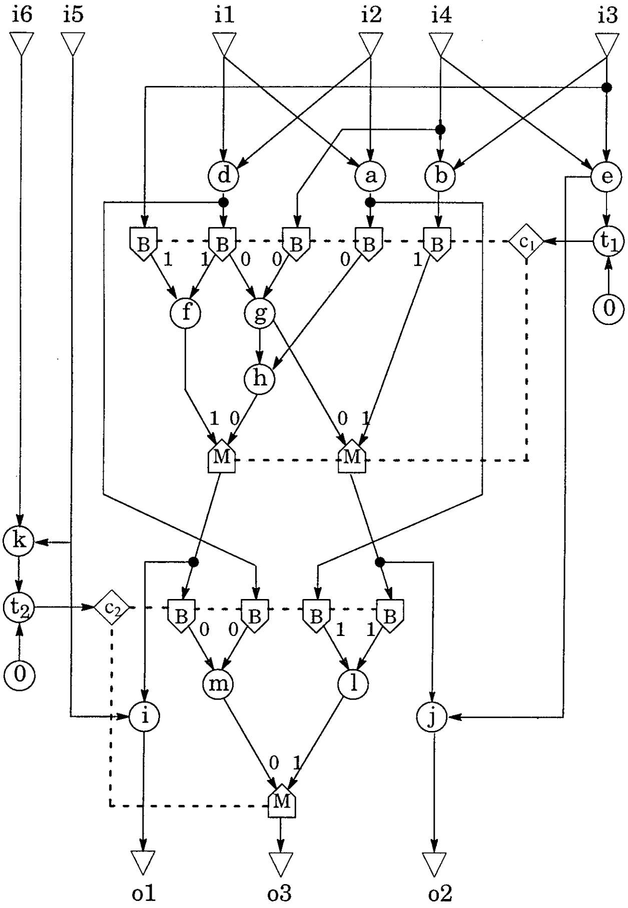


FIGURE 5.2. The DFG for the example in Figure 5.1

The *set of operations whose output values are live* at a given basic block BB_i, written $\text{LIVE}(BB_i)$, is obtained as follows:

$$\text{LIVE}(BB_i) = \{ o_n \in V_o \mid \text{live}(o_n, BB_i) \}. \tag{5.1}$$

The sets $\text{LIVE}(BB_i)$ can be efficiently computed for every BB_i during a depth–first traversal of the DFG. Table 5.1 shows the sets of operations whose output values are live at a given BB for the example of Figure 5.1. Although inputs are live at some BBs, we omit them in the table because they are not relevant for our coming analysis.

TABLE 5.1 Sets $\text{LIVE}(BB_i)$ for the example in Figure 5.1.

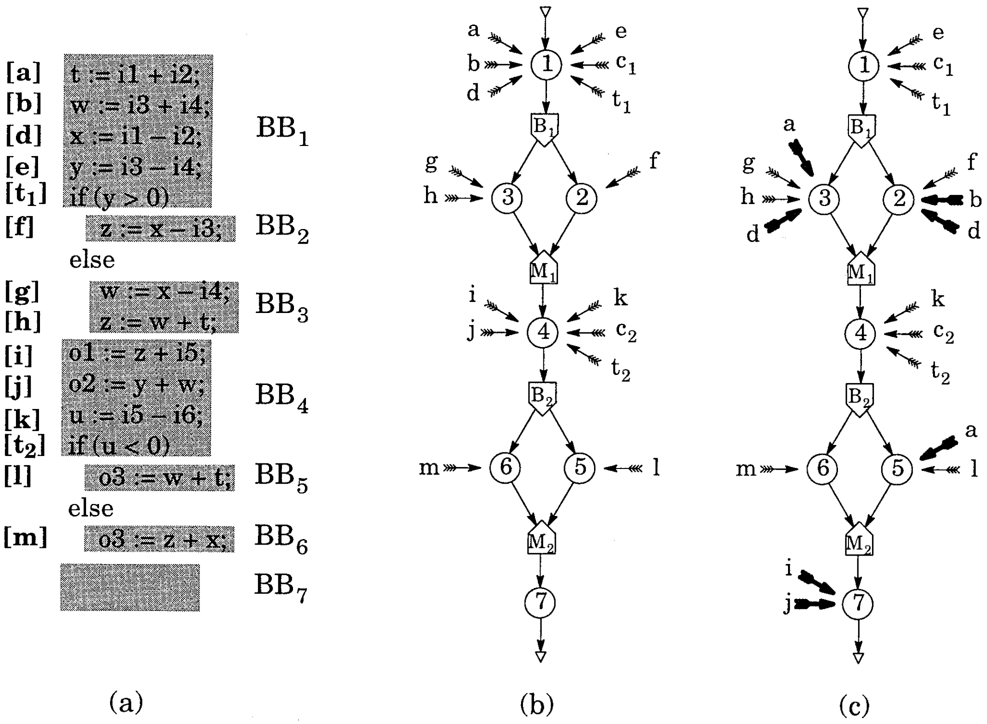
BB_1	BB_2	BB_3	BB_4	BB_5	BB_6	BB_7
∅	{a,b,d,e}	{a,d,e}	{a,b,d,e,f,g,h}	{a,b,g}	{d,f,h}	∅

For simplicity, we henceforth refer to the *output value* of an operation o_n as "the value o_n", whenever clear from the context. Therefore, we can simply say that o_n is used or o_n is live at a given BB.

5.1.2 Formulation of our analysis technique

In this section, we show some properties of the flow of data that are observed when we try to perform downward code motion. We illustrate these properties with the example in Figure 5.1, which is repeated in Figures 5.3a and 5.3b for convenience.

Given some operation, assume that we want to move it towards the sink as far as possible. The ability to move is constrained by the DFG. Below, we explain, by means of examples, how the data flow restricts moving operations downwards.



```
[a]  t := i1 + i2;
[b]  w := i3 + i4;
[d]  x := i1 – i2;       BB1
[e]  y := i3 – i4;
[t1] if (y > 0)
[f]      z := x – i3;    BB2
     else
[g]      w := x – i4;    BB3
[h]      z := w + t;
[i]  o1 := z + i5;
[j]  o2 := y + w;        BB4
[k]  u := i5 – i6;
[t2] if (u < 0)
[l]      o3 := w + t;    BB5
     else
[m]      o3 := z + x;    BB6
                         BB7
```

FIGURE 5.3. Illustration of how to find the set of lowest links

Consider the links emanating from operations h and f. Note that, in both Figures 5.3b and 5.3c they point to BB_3 and BB_2, respectively, which are the latest BBs prior to the selection of a value for variable z. If, for instance, h was moved to BB_4, the value f, which is live at BB_4, would be destroyed. The same argument is valid for operations m and l, whose links in Figure 5.3c are kept pointing to the same BBs as in Figure 5.3b. Moreover, as h is bound to precede M_1 and g is used by h, operation g is also kept linked to BB_3. Note that those links point to the latest BB in which the operations can be executed, because any motion further down would be illegal.

Consider now the links from the conditionals. Observe that the links from each conditional c_k are kept pointing to the BB which immediately precedes B_k. This captures the property that the outcome of a conditional must be available prior to a branch in the control flow. The links from operations e and k, which are used by tests t_1 and t_2, respectively, are pointing to the same BBs as the conditionals c_1 and c_2. This captures the property that e and t_1 must precede c_1; k and t_2 must precede c_2, which in turn are bound to the BBs preceding the branches B_1 and B_2, respectively.

Let us now focus on the links moved, which are highlighted in Figure 5.3c. The link from operation b has moved down to BB_2. Note that, since b and g assign distinct values to variable w, neither the value g nor the value b could move further down without destroying the other. On the other hand, note that the link from operation a not only has moved to BB_3, but is also replicated to point to BB_5. Operation a assigns a value to variable t, which is used by operation h in BB_3. However, if we move operation a to BB_3, the value of variable t will be undefined if a control path through BB_2 is taken, which is incorrect, because the value a is live at BB_2 (recall Table 5.1). However, as a replica of operation a is inserted in BB_5, where the value a is actually used, no value is undefined. Observe that operation d was duplicated such that its links point to BB_3 and BB_2. This operation could not move further down, because the value d is used by operations f and g in BB_2 and BB_3, respectively. Although the value d is used by operation m in BB_6, there is no use to restore this value locally, because it is already defined in every control path including BB_6. Finally, note that the links from operations i and j have moved down to BB_7, since the values i and j are neither used in BB_5 nor in BB_6.

In summary, given an operation o_m, we want to re–link o_m to the latest BB down on a given control path where the execution of o_m would neither destroy a live value on some other path, nor render o_m undefined. In the following, we formalize the mechanisms illustrated so far.

For the formal analysis, it is essential to check the destruction of some value by downward code motion. This test can be performed as follows.

Given two operations o_n and o_m, assume that o_m moves from a basic block BB_i down to a basic block BB_j. If both values o_n and o_m are live at BB_j and both operations reach the same merge node in the DFG, then the value o_n is destroyed by the code motion. This is formalized below.

DEFINITION 5.2

Given the basic blocks BB_i and BB_j with $BB_i \overset{*}{\rightarrow} BB_j$ and an operation o_m linked to BB_i, the motion of o_m from BB_i to BB_j *kills* a value computed by some other operation, written kill(o_m, BB_i, BB_j), iff:

$$\exists o_n, v \in V : (o_n, o_m \in \text{LIVE}(BB_j)) \wedge o_n \overset{*}{\rightarrow} v \wedge o_m \overset{*}{\rightarrow} v \wedge (\omega(v) = \text{merge}).$$

For instance, the motion of operation f from BB_2 to BB_4 would kill the value h, because both values f and h are live at BB_4 (see Table 5.1) and both operations reach the same merge node (see Figure 5.2).

When performing downward code motion, it might be necessary to insert copies or replicas of operations, as illustrated in Figure 5.3 for operations a and d. During our data–flow analysis, we want to make sure that replicas are inserted only where strictly necessary. Suppose that u_i and u_j are arbitrary nodes of a BBCG=(U, F) and let X denote a set of arbitrary nodes of the BBCG.

DEFINITION 5.3

X *co–dominates* u_j, written X cdom u_j, if every path from the source to u_j includes at least one node $u_i \in X$.

In the example of Figure 5.3, for instance, the set $\{BB_2, BB_3\}$ co–dominates BB_6.

Assume that, given the links from the same operation to distinct BBs, we want to check if semantics is preserved when one of them is removed. Suppose that operation o_m is linked to basic block BB_j. Suppose that o_m is also linked to some other basic block BB_z where the value o_m is used, and that BB_z is reachable from BB_j. We want to check if the replica of o_m in BB_z is redundant. In other words, we want to check if the value o_m is kept defined correctly on all control paths including BB_z if the replica is removed.

This procedure can be performed as shown in Algorithm 5.1. The set X contains all BBs linked to operation o_m. For a given BB_z in set X, if every path from the source to BB_z includes an element of $X \setminus \{BB_z\}$, a value o_m is defined whichever the path taken from the source. As a consequence the link λ_z represents a redundant replica and is removed.

Notice that Algorithm 5.1 can be used to prevent the unnecessary linking of operations to every BB where the value o_m is used. In Figure 5.3c, for example, although the value d is used in BB_6, operation d is not linked to it, because d is already linked to the elements of set $\{BB_2, BB_3\}$, which co–dominates BB_6. On the other hand, the value a is used in BB_5 and, even though already linked to BB_3, operation a is also linked to BB_5 because $\{BB_3\}$ does not co–dominate BB_5.

ALGORITHM 5.1. Algorithm for removing redundant links

```
procedure remove_redundant($o_m$)
$X := \{BB_k \in U \mid o_m \overset{\lambda_k}{\rightarrowtail} BB_k\}$;
foreach $BB_z \in X$ with $o_m \overset{\lambda_z}{\rightarrowtail} BB_z$
    if ( $X \backslash \{BB_z\}$ cdom $BB_z$ )
        remove $\lambda_z$;
```

Fundamentally, we want to check the legality of moving an operation o_m from BB_i down to BB_j. This means that o_m should not move if it kills a value, no conditional should move past the point where the control–flow decision is due and no operation should move through the sink. This is formalized below.

DEFINITION 5.4
Given the basic blocks BB_i and BB_j with $BB_i \xrightarrow{*} BB_j$, the operation o_m is *free* to move from BB_i to BB_j, written free(o_m, BB_i, BB_j), iff:

$$\neg \mathrm{kill}(o_m, BB_i, BB_j) \wedge (\omega(o_m) \neq \mathrm{conditional}) \wedge (BB_j \neq \mathrm{sink}).$$

Our central idea is to find the boundaries for downward code motions. Assume that we move an operation o_m downwards on some path p from a basic block BB_i towards some basic block BB_k. A boundary for downward code motion can be found by checking iteratively if the operation is free to move between "adjacent" BBs on path p. The idea is illustrated in Algorithm 5.2, where the last basic block on path p satisfying that property is returned, namely BB_j.

Algorithm 5.2 is deliberately written to explicate the main idea of our analysis. However, a more efficient version is used in our implementation in which depth–first search is used to avoid enumeration of paths.

ALGORITHM 5.2. Algorithm for checking Definition 5.4 iteratively

```
procedure find_boundary($o_m$, $BB_i$, $BB_k$)
foreach $p = \langle BB_i, BB_{i+1}, \cdots, BB_j, BB_{j+1}, \cdots, BB_k \rangle$ with $BB_i \xrightarrow{p} BB_k$
    for ($j :=$ i to $k - 1$)
        if ($\neg$ free($o_m$, $BB_j$, $BB_{j+1}$))
            return($BB_j$);
return($BB_k$);
```

In the following, we show how to obtain a new set of links such that every operation o_m is linked to its boundary for downward code motion on all control paths where o_m executes. Algorithm 5.3 describes how to derive the *set of lowest links* from the *set of initial links*. In this algorithm, the operations in the DFG are visited using depth–first search and it assumes that each operation o_m owns a Boolean–valued attribute for marking its visiting status.

ALGORITHM **5.3.** Algorithm for finding the lowest links

```
procedure find_lowest_links()
mark initial links;
foreach $v_i \in$ INP(V)
   foreach $o_m \in$ CONS($v_i$)
      visit($o_m$);
delete initial links;


procedure visit($o_m$)
if ($o_m$ is visited)
   return;
mark $o_m$ as visited;
foreach $o_n \in$ CONS($o_m$)
   visit($o_n$);
   foreach $BB_i$ with $o_m \xrightarrow{\lambda_i} BB_i$;
      foreach $BB_k$ with $o_n \xrightarrow{\lambda_k} BB_k$;
         $BB_j$ := find_boundary($o_m$, $BB_i$, $BB_k$);
         create a new link $\lambda$ such that $o_m \xrightarrow{\lambda} BB_j$;
         remove_redundant($o_m$);
```

In the first procedure of Algorithm 5.3, the initial links are marked so as to distinguish themselves from the new links to be created. Then, the operations are visited recursively (via the second procedure), starting from every operation being an immediate consumer of an input value. After the traversal is completed, the initial links are deleted.

In procedure visit(o_m), the idea is to propagate a given operation o_m from its initial BB towards the initial BB of some consumer o_n. As the consumer o_n itself might have been propagated from its initial BB to some other basic block, links are moved down as much as possible in the BBCG. Clearly, if the BB which immediately precedes the sink node is reached, there is no way to move further down. Due to the recursive call, the operations close to the outputs are re–linked first, whereas the operations nearby the inputs are re–linked last, according to an inverse topological ordering. In this procedure, o_m denotes the operation currently visited, which is linked to its initial basic block BB_i. This operation produces a value that is consumed by operation o_n, which is linked to some basic block BB_k such that there is a path p with $BB_i \xrightarrow{p} BB_k$. Ideally, the procedure tries to link o_m to BB_k. This is performed by first finding the lowest links of operation o_n (recursive call). However, the downward motion might be blocked somewhere on path p, in between BB_i and BB_k if the operation is not free to move further down. Consequently, BB_j represents the latest basic block downwards on path p where the operation can be legally moved. As a consequence of the recursive call, when every consumer o_n is visited and linked, the producer o_m itself is linked. Then, all the links that become redundant after the linking of o_m are removed. Observe that the initial links are all removed by Algorithm 5.3. For this reason, when we refer to a link from now on, we mean a link determined by that algorithm, unless otherwise specified.

In summary, the new set of links obtained by Algorithm 5.3 capture the *maximal freedom* for moving operations *downwards*. As a consequence, the following theorem holds.

THEOREM 5.1

If there is a path p from the source to a basic block BB_j such that operation o_m was not scheduled within any BB included in path p, and if there exists a lowest link connecting operation o_m with BB_j, then o_m must be scheduled within BB_j.

PROOF

There are two cases in which a lowest link is obtained, as follows:

Case 1: $BB_j \neq BB_k$ in Algorithm 5.3

Under this assumption, we conclude from Algorithm 5.2 that procedure free(o_m, BB_j, BB_{j+1}) must have returned false. As a consequence, the following predicate holds:

$$\text{kill}(o_m, BB_j, BB_{j+1}) \vee (\omega(o_m) = \text{conditional}) \vee (BB_j = \text{sink}).$$

This means that postponing the scheduling of o_m to BB_{j+1} either kills a value, or a conditional moves beyond a decision point of the control flow, or the operation illegally moves through the sink. Since from the hypothesis, o_m was not scheduled prior to BB_j and can not be scheduled any later, o_m must be scheduled in BB_j.

Case 2: $BB_j = BB_k$ in Algorithm 5.3

Under this assumption, there must be some operation o_p reachable from o_m and linked to BB_k for which Case 1 holds. Since o_p is reachable through data edges in the DFG, o_p is data dependent on o_m. Assume that the scheduling of o_m is postponed to BB_{j+1}. Since from the hypothesis, o_m was not scheduled before BB_j and o_p can not be moved to BB_{j+1} (Case 1), the data dependence between o_m and o_p will be violated if the scheduling of o_m is postponed to BB_{j+1}. Hence, o_m must be scheduled within BB_j. ❑

In other words, Algorithm 5.3 identifies the constraints imposed on downward code motion by the flow of data.

5.2 Pruning inefficient code motions

Recall that the sets of available operations are ordered according to some priority encoding Π. In this section, we show how the lowest links, obtained by the data–flow analysis described in the previous section, can be used to *modify the ordering* of the sets of available operations in such a way that ineffective code motions are prevented. First, we introduce the basic notions with a couple of examples. Second, we show how a precedence relation can be obtained from the lowest links. Then, we show how the pruning of inefficient code motions can be formulated as a dynamic reordering of the sets of available operations.

5.2.1 Motivation

In the compiler arena, most of the early ILP techniques are oriented to architectures where resources are abundant, like VLIW machines [22][48]. Some approaches reorient such techniques to architectures with scarce resources [62]. In both scenarios, the techniques aim at speeding up average program runtime.

In the context of HLS, due to the requirements dictated by embedded systems, the architectures of ASICs and ASIPs are designed with *as few resources as possible*. The amount of resources is just enough to comply with the requirements of a single application or of a bounded application domain. As a consequence, the direct application of ILP techniques developed in the compiler domain may create an unbalance between the parallelism *exposed* and the parallelism that can actually be *exploited.*

To illustrate such unbalance, we refer to the simple examples in Figures 5.4 and 5.5. In each example, a behavioral description and its respective BBCG are shown, along with two alternative solutions induced by different priority encodings Π_1 and Π_2. Only the links relevant for our analysis are shown. In addition, suppose that one adder, one subtracter and one comparator are the available resources.

Let us first consider code motion ahead of a branch junction, as illustrated in Figure 5.4. Solution SMG_1, shown in Figure 5.4c, is induced by an encoding

Π_1 such that b $<_{\Pi_1}$ a. Solution SMG_2, shown in Figure 5.4d, is induced by Π_2 such that a $<_{\Pi_2}$ b. Note that the operations a and b are available at state s_0 within BB I. We conclude that the schedule length of the left path is increased by the execution of b in the first state of SMG_1, as compared to SMG_2. The reason is that a and b can not be scheduled in a same state because $\tau(a) = \tau(b)$ and a single subtracter is available. Note that, although operation a must be executed before the branch junction, the execution of b can be postponed until after the branch, since it does not necessarily have to execute on the left path. Consequently, solution SMG1 is inferior and it is convenient to prevent its construction. For this purpose, we should be able to detect beforehand that the pre–execution of b before a may lead to an inferior result. Actually, the links emanating from a and b hint the criterion for such detection, as follows.

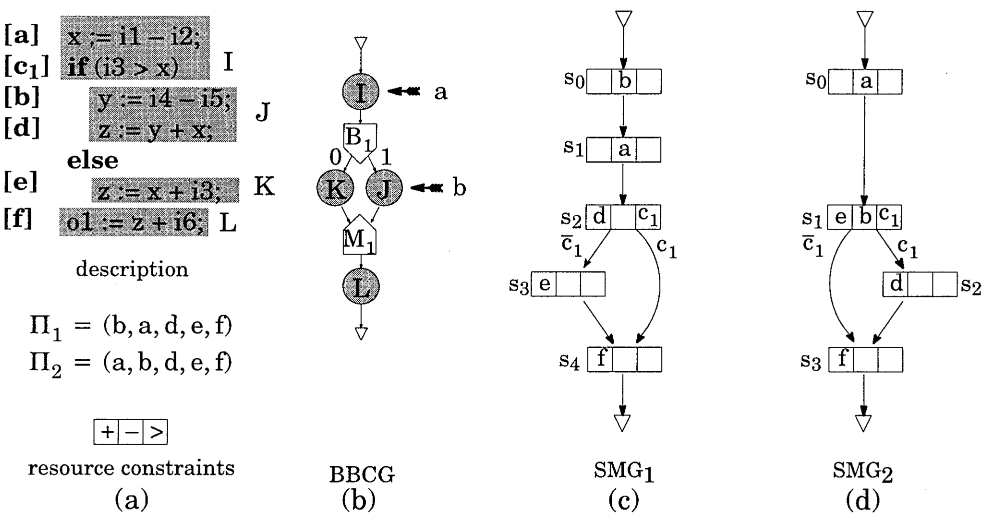


FIGURE 5.4. Example of code motion ahead of branch junction

On the one hand, a is linked to I and must be scheduled within BB I, because the result of a must be available prior to branch B_1 in the control flow. On the other hand, operation b does not necessarily have to be executed in BB I, the only requirement being its execution prior to operation d. Therefore, it seems that when the precedence in the control flow between the BBs pointed to by the links from a and b coincides with the precedence in the priority encoding, a better solution is obtained. This suggests that, in the case such a coincidence does not hold, we should *reorder* available operations to avoid the construction of an inferior solution. For instance, given the encoding Π_1 in Figure 5.4c, if we reorder a and b in the availability set A_0, the code motion of b that places it in state s_0 is prevented and solution SMG1 is not constructed. This is the key for the pruning technique to be described in this section.

Let us now consider code motion ahead of a merge junction, as illustrated in Figure 5.5. On the one hand, solution SMG_1, shown in Figure 5.5c, is induced by priority encoding Π_1 such that $g <_{\Pi_1} d \wedge g <_{\Pi_1} f$. On the other hand, solution SMG_2, depicted in Figure 5.5d, is induced by Π_2 such that $d <_{\Pi_2} g \wedge f <_{\Pi_2} g$. Assuming that operations a, b and e are already scheduled, operations d and g are simultaneously available at BB J and operations f and g are simultaneously available at BB K. Observe that the duplication of g into BBs J and K does not increase the schedule lengths of any path since g has to be executed on both control paths anyhow. However, it might increase the number of states. For instance, note that states s_1 and s_4 in SMG_1 can not be merged by the technique described in the previous chapter, since they are not equivalent. If we compare solutions SMG_1 and SMG_2, we conclude that the larger number of states in SMG_1 occurs because g is moved across the merge junction, but it can not be accommodated in a same state with either d or f, because $\tau(d) = \tau(g) = \tau(f)$ and only one subtracter is available. Note that such code motion is actually unnecessary. Although the results of d and f must be available before the merge junction, the execution of g can be postponed until after the merge. The code motion of g might be beneficial if the involved operations happened to map to different module types or if more resources were available.

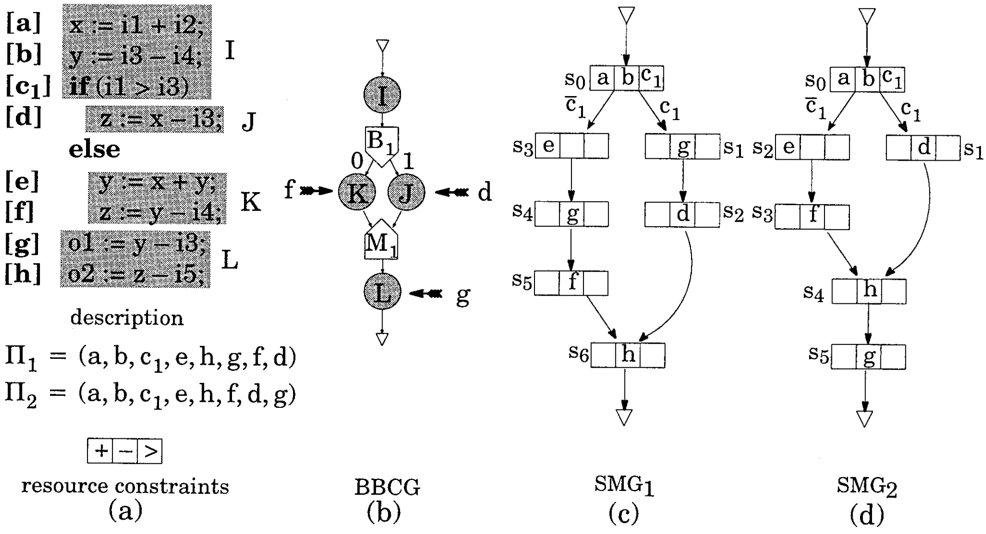


FIGURE 5.5. Example of code motion ahead of merge junction

Similarly to the previous example, the links emanating from the operations mapping to a same module type hint us some criterion to reorder operations. For the second example, such a reordering will prevent the construction of SMG_1, which is inferior with respect to the number of states.

The examples analyzed above illustrate that availability analysis may expose too much parallelism. Since we can not simply restrict availability analysis beforehand without limiting the design space exploration, we have to find another mechanism. Actually, the parallelism exploitable can be hinted by a convenient interpretation of the links: given two available operations mapping to a same module type, the execution of the operation that is linked to the currently visited BB should have the priority over the other. This suggests that a *reordering* of the availability sets has the effect of pruning inferior solutions from the search space. Henceforth, we refer to this kind of pruning as *code–motion pruning* (CMP).

Since the set of links emanating from the operations in the DFG changes after each code motion and is also affected by code compensation, the reordering must be determined on the fly, depending on the decisions taken dynamically by the scheduler as it proceeds. As illustrated in Figures 5.4 and 5.5, such a reordering is likely to have an impact not only on schedule length (for code motion ahead of branch junctions), but also on the number of states (for code motion ahead of merge junctions).

5.2.2 A precedence relation based on the links

The examples in the previous section suggest that, if some operation a *precedes* some operation b in terms of lowest links, a should be scheduled before b, even if b precedes a in the priority encoding Π. Such precedence, hinted by the lowest links, can be modeled by a precedence relation, as follows.

To begin with, precedence should be defined only between operations mapping to a same module type, because only in that case their parallel execution might be impaired due to the lack resources.

According to Theorem 5.1, if an operation a is linked to the currently visited basic block, say BB_i, and there is an operation b linked to some other basic block BB_j such that $BB_i \xrightarrow{*} BB_j$, operation a *must* be scheduled in BB_i, where operation b does *not necessarily* have to be scheduled, since b has the freedom to move down to BB_j. This suggests that operation a should have the priority over operation b. These notions are formalized below.

Definition 5.5

Given the set of lowest links Λ and some $\lambda \in \Lambda$, let $<_{\Lambda,i}$ denote the *precedence relation induced by the set of links* Λ when the basic block BB_i is visited. We say that a precedes b at basic block BB_i, written $a <_{\Lambda,i} b$, iff:

$$(\tau(a) = \tau(b)) \wedge ((a = b) \vee (a \overset{\lambda}{\rightarrowtail} BB_i \wedge b \not\rightarrowtail BB_i)).$$

Note that, in the term (a = b) ensures that the relation is reflexive. Notice also that two operations are not ordered by the precedence relation if they are linked to the same BB. In addition, observe that the relation is anti–symmetric, but it is not transitive.

Below, we show how this precedence relation can be combined with the priority encoding to prevent the induction of inefficient code motions.

5.2.3 Reordering the sets of available operations

Recall that the priority encoding Π induces a linear order $<_{\Pi}$ such that $a <_{\Pi} b$ denotes that a has priority over b. Recollect that $<_{\Pi}$ is used to keep the sets of available operations ordered. To influence the induction of code motion, we modify the linear ordering implied by $<_{\Pi}$. Given two operations a and b, our idea is to make the precedence relation $<_{\Lambda,i}$ prevail over the priority encoding either when $a <_{\Lambda,i} b$ or when $b <_{\Lambda,i} a$. Otherwise, the ordering induced by the priority encoding is maintained. This notion is formalized by defining a new linear ordering $<_{\Pi,i}$ for each BB_i, as follows:

$$a <_{\Pi,i} b \Leftrightarrow (a <_{\Lambda,i} b) \vee (\neg(b <_{\Lambda,i} a) \wedge (a <_{\Pi} b)). \qquad (5.2)$$

The linear order $<_{\Pi,i}$ represents the modification of the original linear order defined by the priority encoding such that the information captured by our data–flow analysis technique is taken into account. Assume that we order every set $A_{i,k}$ according to the relation $<_{\Pi,i}$. Suppose that two operations $a, b \in A_{i,k}$, which map to a same module type t_m, are such that $a <_{\Pi,i} b$, although b precedes a in the priority encoding Π. We conclude that a is linked to basic block BB_i and that b is linked to some other basic block BB_j. If the scheduler selects a to be scheduled in state $s_{i,k}$, the code motion of b from BB_j to BB_i is prevented if no more resources of type t_m are free within BB_i. In other words, *reordering* the set $A_{i,k}$ effectuates the pruning of code moves.

The reordering does not move an operation to a basic block, say BB_i, on some control path p, under two circumstances, namely:

- If the code motion may increase the schedule length of some other control path q that also includes BB_i (recall Figure 5.4).
- If the code motion may unnecessarily increase the number of states (recall Figure 5.5).

However, the method is heuristic: it may exclude beneficial moves. Yet, our criterion is more conscious of scheduler decisions and of data–flow properties than most of the built–in heuristics found in the literature. For instance, the heuristic reported in [62] gives priority to the operations whose *initial* BB is being visited, but no support is provided to check if the operations could move further down in the flow of control.

Another example is the method reported in [46], where available operations are prioritized according to two main heuristic criteria, namely the topological ordering of the operations in the control–flow graph and the so–called degree of speculativeness. Although that method relies on a quite different representation, we could approximately capture both criteria in our context as follows: available operations are ordered according to a topological ordering of their initial BBs. These criteria make three main greedy assumptions. First, the topological ordering relies on the *initial* position of the operations in the control flow. It overlooks the fact that the initial position may be arbitrary and, since only upward code motions are supported by that method, the effect of possible downward code motions is not captured. Second, the criteria enforce unnecessarily the order of operations linked to mutually unreachable BBs. Third, the criteria enforce the precedence of operations linked to BBs on a same control path. Assume, for instance, that operations p, q and r are linked to BBs P, Q and R, respectively. If these BBs are such that $P \overset{*}{\rightarrow} Q \overset{*}{\rightarrow} R$, the order (p, q, r) is enforced, *regardless of which BB is currently visited*. In other words, the criteria overlook the fact that the scheduling of some operations can be postponed. In our technique, however, we not only define a precedence relation for each visited BB depending on the *current* set of links, but also our relation is *not* transitive to avoid enforcing an order of available operations induced by the precedence of their associated BBs on a given path.

Note that the methods [46][62] rely on built–in heuristics, i.e., an ordering criterion determines a single priority–list leading to the construction of a single solution, as opposed to our notion of several priority encodings inducing the exploration of alternative solutions.

5.3 Experimental results

In this section, we show the effect of applying CMP during the exploration of alternative solutions. First, we look at the overall impact of the technique on the search space. Afterwards, we report experimental results showing the impact of that technique on *both* the schedule length and the number of states. The experiments are performed for several examples extracted from the HLS literature under the following set–up. Essentially, all the types of code motion discussed in the previous chapter are allowed, along with both forms of speculation. In addition, our technique for exploiting state equivalence on the fly is enabled for all experiments. For every experiment, we perform two runs: one *without* and one *with* CMP.

Our objective is to evaluate the impact of the technique for an arbitrary priority encoding. For this purpose, several priority encodings are generated such that the position of each operation in a permutation is determined at random. The solutions induced by each priority encoding are computed. Then, the schedule length of the longest path and the total number of states are measured for each solution. Some statistics on the measured values are examined.

5.3.1 The impact on the search space

In order to ease the interpretation of the results in this section, assume that the i^{th} solution in the search space is identified by the pair (L_i, S_i), where L_i denotes the schedule length of the longest path in the respective SMG and S_i denotes its number of states. In addition, since different values of L_i are obtained for different solutions, we say that the *range of schedule lengths*, written ΔL, is the difference between the maximal and the minimal values observed for L_i among a set of generated solutions. Similarly, the *range of number of states*, written ΔS, is the difference between maximal and minimal values observed for S_i.

To illustrate the overall effect of the technique, we select the example “s2r” [52] and, given a set of resource constraints, we apply a sequence of 100 randomly–generated priority encodings, thereby constructing 100 solutions. Different runs of the experiment are performed for different resource constraints corresponding to cases A, B and C (see Appendix A). For each of these cases, we provide plots for the search space induced by the randomly–generated priority encodings.

Let us first illustrate the effect observed in the search space when the experiments are performed *without* CMP, as shown by the plot in Figure 5.6. The number of available resources decreases for cases C, B and A in this order (the number of instances of “alu” is 3, 2 and 1, respectively). In the plot, each point represents a different solution, the vertical axis indicates the number of states and the horizontal axis represents the schedule length of the longest path in terms of number of clock cycles. Compare cases A, B and C with respect to the observed ranges of schedule lengths. Note that $\Delta L = 3$, for case C, $\Delta L = 4$, for case B and $\Delta L = 7$, for case A. We observe that ΔL increases when the number of resources decreases. Similarly, if we compare the number of states for cases A, B and C, we can say that ΔS also increases when the number of resources decreases. Informally, we can say that the tighter the resource constraints are, the more spread out the search space becomes. This is a first experimental evidence that not all ILP exposed is beneficial. On the contrary, parallelism that is uncovered but is not accommodated leads to the generation of many inferior solutions in the course of the optimization process.

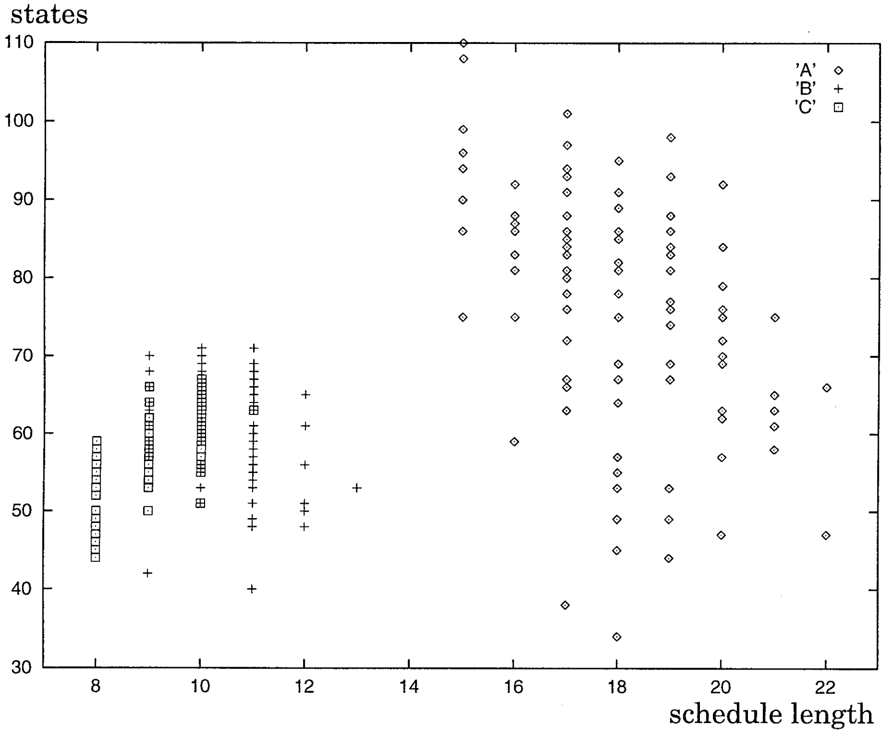


FIGURE 5.6. The search space without code–motion pruning

Let us now observe what happens when the same experiments summarized in Figure 5.6 are performed such that solutions are computed by exactly the same priority encodings, but then *with* the application of our CMP technique. The new results are plotted in Figure 5.7. By comparing Figures 5.6 and 5.7, we conclude that ΔL is reduced from 3 to 2, for case C; from 4 to 3, for case B and from 7 to 2, for case A. Similarly, ΔS also exhibits a reduction, although in a smaller scale. Note that inferior solutions with respect to schedule lengths are pruned from the search space and that CMP causes the solutions to "shift" towards the optimal schedule length. In case A, for instance, assume that the optimal schedule length is $L_i = 14$. Note that, unlike Figure 5.7, no solution with optimal schedule length is generated in Figure 5.6 for case A, although exactly the same set of priority encodings is used in both runs. This means that the original priorities of some operations in several permutations were modified by CMP such as to generate solutions with shorter schedule length. Without CMP, we would need to generate more permutations in order to increase the probability of an optimal solution being induced.

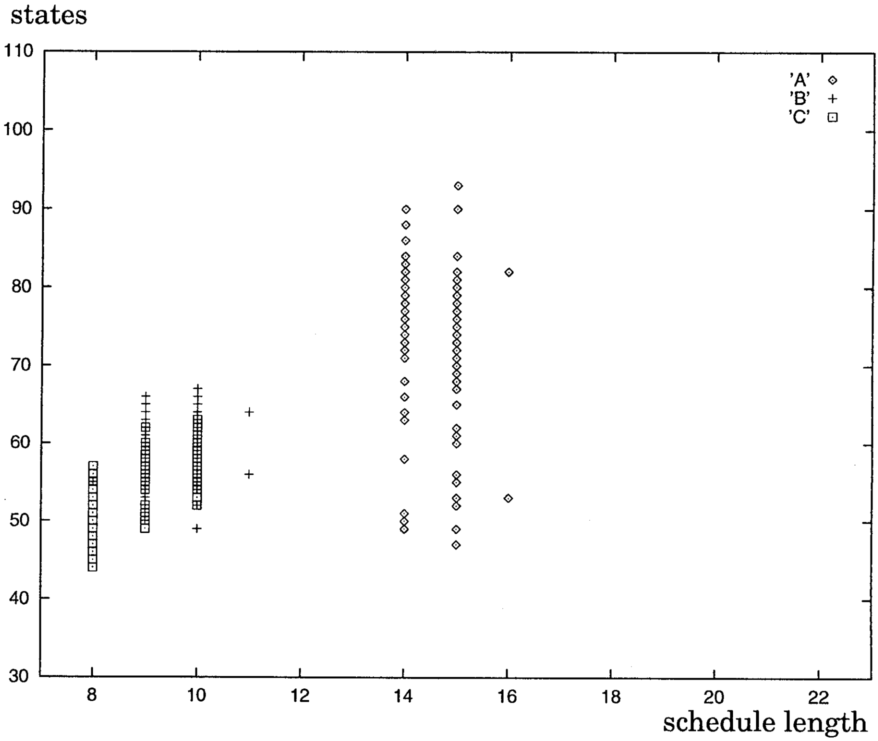


FIGURE 5.7. The search space with code–motion pruning

However, note that several solutions with a small number of states are pruned, especially when their schedule lengths are far from the optimum. In Figure 5.6, for instance, the solutions (18, 34) and (17, 38) are generated. These solutions are not generated when CMP is applied. This observation suggests that the application of CMP may prevent the construction of cheap solutions when the global time–constraint is not tight. On the one hand, this disadvantage could obviously be overcome by disabling pruning if the global time–constraint is not too tight, although possibly at the expense of larger search times. On the other hand, it is unlikely that code motion would have an essential impact under loose time–constraints. Therefore, in such a context, a HLS tool with on–demand code motion capabilities is likely to achieve a reduction of the number of potential solutions to be explored and still keep acceptable solutions in the search space, simply by inhibiting some code motions (like duplication of conditionals). Yet, CMP seems to perform well with respect to the number of states under tight time–constraints. For instance, note that in Figure 5.7, case A, more solutions with low number of states are generated with $L_i = 15$ than in Figure 5.6. A similar result can be observed for case B with $L_i = 9$. For case C, the quality in terms of states is essentially the same for $L_i = 8$ in both figures.

In the next two sections, we refine our evaluation of the impact of CMP by analyzing detailed statistics for several examples.

5.3.2 The impact on schedule length

Let us first observe what happens with the schedule lengths when we apply CMP. Table 5.2 reports statistics for the schedule length of the longest path for several examples shown in the first column. Each different case in the second column corresponds to distinct resource constraints, as described in Appendix A. The third column shows the best schedule length known for each case. Two sets of results are shown: one *without* and one *with* CMP.

Each set of results is organized in three columns. The first two columns show the mean value an the standard deviation for the schedule length of the longest path. The last column reports the percentage of solutions with the best L_i, i.e., it represents the density of observed solutions with minimal schedule length, which is referred to as the *density of promising solutions*.

Note that the mean value either decreases or remains the same for all examples when pruning is applied. Notice also that, for a given example, the fewer resources are available, the more reduction is obtained on the mean value. This observation can be interpreted as follows. The fewer resources are available, the less code motions are effective and, consequently, the more code–motions are pruned. Therefore, our technique has the effect of compensating for the unbalance between the exposed ILP and the exploitable ILP. In addition, notice that the standard deviation decreases for most examples under CMP. This means that the technique not only shortens schedule lengths on average for a same set of priority encodings, but also spreads them over a smaller range of possible schedule lengths.

Observe that CMP increases the density of promising solutions in the search space for all the examples, except for a few of them where the density remains the same. It is important to note that the fewer resources are available, the more impact CMP has on the density. For data paths with few resources, such as in cases A, B, E and F for the example “s2r” (see Appendix A), the percentages obtained without pruning are very low. As a consequence, we conclude that many more solutions should be explored under tight time–constraints if no mechanism is employed to compensate for the unbalance between exposed ILP and exploitable ILP. In this context, CMP seems to be an efficient mechanism.

Table 5.2 Results showing the impact of CMP on schedule length

example	case	best L_i	without			with		
			mean	σ[%]	density	mean	σ[%]	density
waka	A	7	7.8	8.5	33%	7.8	8.5	33%
	B	7	8.0	7.8	21%	7.9	8.3	24%
kim	B	8	9.0	5.3	10%	8.8	4.6	23%
	C	6	7.0	6.6	7%	6.9	5.7	12%
rotor	A	11	13.6	12.8	12%	11.0	0.0	100%
	B	8	8.3	5.5	67%	8.0	0.0	100%
	C	7	7.0	0.0	100%	7.0	0.0	100%
	E	9	10.6	5.4	2%	9.8	3.6	15%
	F	8	8.0	0.0	100%	8.0	0.0	100%
	G	8	8.0	0.0	100%	8.0	0.0	100%
s2r	A	14	18.0	9.5	1%	14.7	3.5	41%
	B	8	10.4	7.8	1%	9.5	6.1	16%
	C	8	9.1	9.0	41%	8.9	8.7	62%
	E	12	15.0	8.4	1%	13.1	5.7	25%
	F	8	10.3	8.2	2%	10.0	7.6	16%
	G	8	9.7	11.4	31%	9.4	11.0	45%
kim_big	A	49	64	6.2	0%	59	4.5	0.1%
	C	49	63	5.1	0%	59	4.3	0.1%
	D	47	61	5.9	0%	58	5.0	0.1%

It is interesting to observe that for the example "rotor" in cases A and B the pruning is such that all the solutions generated have minimal schedule lengths. Note that this situation can occur without CMP only if the number of resources is increased as in case C, where all the exposed parallelism is accommodated within the available resources.

Let us now focus on the example "kim_big". Table 5.3 reports densities of solutions representing various potential local–optima in the range from

$L_i = 49$ to $L_i = 59$. For each case, the percentages *without* and *with* CMP are presented. Note that, when CMP is applied, the densities increase for every potential local–optimum and for all reported cases. It is interesting to notice that for case A no solution with $L_i < 53$ is observed within the set of generated solutions when CMP is disabled, whereas solutions with L_i in the interval $49 \leq L_i < 53$ are observed when CMP is applied. This means that, without CMP, more solutions have to be explored on average in order to reach local optima of good quality.

In order to ease the interpretation of the results in Table 5.3, we organize the results in the chart of Figure 5.8 for the example "kim_big", case D. In the figure, the horizontal axis represents different time constraints T_c. The vertical axis represents the percentages obtained by adding the densities of all solutions with $L_i \leq T_c$, for a given T_c. In other words, it represents the density of solutions satisfying a given time constraint T_c. Results are shown *with* and *without* CMP and are distinguished by the areas shaded in black and gray, respectively.

TABLE 5.3 Densities of potential local–optima for the example "kim_big"

case		densities [%]										
	Li	49	50	51	52	53	54	55	56	57	58	59
A	without	0.0	0.0	0.0	0.0	0.1	0.0	0.4	0.6	1.9	3.8	5.3
	with	0.1	0.2	0.0	0.6	1.3	2.4	3.6	6.3	9.8	12.6	14.2
C	without	0.0	0.0	0.0	0.1	0.1	0.5	0.2	1.0	2.3	4.8	5.3
	with	0.1	0.0	0.2	0.7	1.0	2.1	4.8	6.4	9.8	13.4	17.3
D	without	0.0	0.0	0.0	0.3	0.5	1.1	1.4	1.8	6.0	7.0	10.8
	with	0.2	0.6	1.0	1.4	2.6	4.6	6.3	8.9	10.2	16.6	16.2

Note, for instance, that when $T_c = 56$, then 5% of the solutions observed without CMP satisfy the time constraint, against about 25% when CMP is applied. Similarly, when $T_c = 58$ those percentages are about 18% and 52% without and with CMP, respectively. Observe also that approximately 70% of the solutions observed without CMP have $L_i > 59$, whereas only 30% of such low–quality solutions are observed with CMP.

Those results provide evidence that CMP increases the number of solutions satisfying a given time constraint. Since the reported statistics are obtained with randomly–generated priority encodings, it is reasonable to expect that the technique should reduce search times in a way largely independent of the choice of a local search algorithm for the explorer.

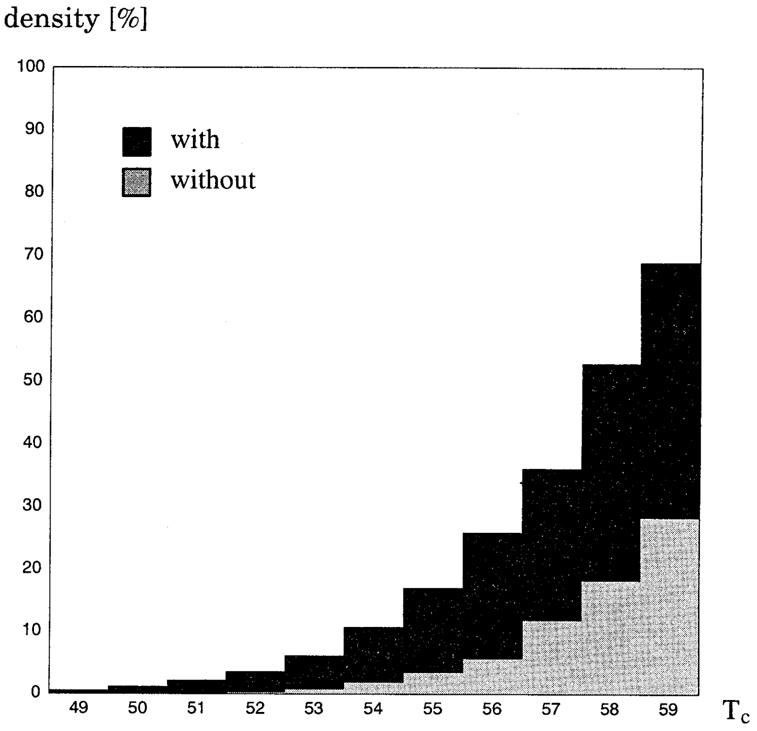


FIGURE 5.8. Density of solutions satisfying a given time constraint

5.3.3 The impact on the number of states

Although the minimization of schedule lengths is the primary issue for time–constrained applications, it is also convenient to observe the effect of CMP with respect to the number of states.

Table 5.4 shows the mean value and the standard deviation of the total number of states for the same examples in Table 5.2. Note that, when pruning is applied, the mean value decreases or remains the same in most examples, except for the case E of examples “rotor” and “s2r”, and for cases A and C of example “kim_big”. However, in these cases the increase on the mean value is less than 8%. With respect to the standard deviation, it decreases in several cases and in others a slight growth is observed.

Therefore, the results suggest that the technique can be applied to prune inferior solutions in terms of schedule lengths, whereas the side effect of possibly increasing the number of states is negligible on average.

TABLE 5.4 Results showing the impact of CMP on the number of states

example	case	without		with	
		mean	σ[%]	mean	σ[%]
waka	A	11.6	7.6	11.6	7.6
	B	11.9	8.3	11.8	8.9
kim	B	20.8	5.8	19.8	6.2
	C	15.3	5.1	14.9	5.2
rotor	A	25.7	7.0	21.4	8.5
	B	17.3	6.0	17.3	2.6
	C	14.6	3.3	14.6	3.3
	E	19.0	4.7	19.5	4.9
	F	15.6	3.2	15.6	3.2
	G	15.6	3.2	15.6	3.2
s2r	A	77.0	20.3	71.9	14
	B	61.2	10.1	59.2	6.9
	C	57.5	9.0	54.2	8.0
	E	67.6	18.1	73.0	6.9
	F	60.9	7.8	58.0	8.9
	G	59.7	6.3	56.9	7.1
kim_big	A	472	27	495	17
	C	451	29	476	15
	D	462	29	437	20

5.3.4 Comparison with other methods

In this section, our goal is to provide extra evidence that our approach can reach high–quality solutions, by comparing our schedule lengths with those obtained by other methods.

In Table 5.5, we compare our results with heuristic methods (TBS, CVLS and HRA) and one exact method (ST) in terms of schedule lengths of the longest path. The resource constraints are shown at the top of the table. Our results are shown in the shadowed row and results from other methods are assembled at the bottom. Note that our method reaches the best published results.

In Table 5.6, we capture the effect of the pipeline latency imposed by the advance choice of a controller. A pipeline of two stages is adopted such that each conditional has a delay slot of 1 cycle. Note that our approach can reach the same schedule lengths obtained by the exact method described in [52]. This agrees with the fact that we support essentially the same types of code motion as the method in [52], which provides one of the most general sets of code motions reported in the HLS literature, despite its large runtime.

TABLE 5.5 Comparison of schedule lengths with other methods

	waka	kim	maha[49]		parker[49]	
add	1	2	1	2	1	2
sub	1	1	1	3	1	3
cmp	1	1	–	–	–	–
ours	7	6	5	4	5	4
ST [52]	7	6	5	4	–	4
TBS [34]	7	–	5	–	–	–
CVLS [72]	7	6	5	4	5	4
HRA [37]	7	7	8	–	–	–

TABLE 5.6 Comparison of schedule lengths with the method in [52]

	rotor								s2r
alu	1	2	3	4	1	2	3	4	3
mul	0	0	0	0	2	2	2	2	2
ours	12	7	7	6	10	8	8	8	8
ST	12	7	7	6	10	8	8	8	8

For the example "kim_big", we are unfortunately not able to make an accurate comparison. This example is obtained from a DFG in [37], but we were obliged to adapt the original graph to our DFG model and also to comply with some restrictions of format imposed by our current implementation. Therefore, the example "kim_big" refers to a DFG slightly *different* from the original DFG, which prevents a precise comparison with the method in [37]. However, let us illustrate the difference with a few values. The best schedule lengths obtained in [37] are 47, 46 and 46, for cases A, C and D, respectively, whereas the best schedule lengths for the example "kim_big" using our approach are 49, for cases A and C and 47, for case D. The difference in schedule lengths is small enough to suggest that the same results in [37] would be reached, should our implementation restrictions be overcome.

5.4 Discussion

In this chapter, a data–flow analysis technique was proposed for extending our approach with the capability of handling downward code motion, as opposed to most global scheduling methods [19][46][62]. The underlying idea is to propagate operations downwards in the flow of control as much as possible. The resulting set of links, the so–called lowest links, can be used instead of the initial links. As a consequence, the analysis is executed only once, as a pre–processing step, before the construction and the exploration of alternative solutions. Since downward code motions are captured by the initialization, only upward code motions are performed during global scheduling, without loss of generality. Note that the proposed analysis is fundamental to capture conditional execution properly. Since the lowest links are such that operations are linked only to the BBs where their computed values are used, we can ensure that the execution condition and the committing condition are equal for each operation. This is the underlying hypothesis of our modeling for speculation (recall Definition 4.4).

An example of scheduling method which supports downward code motion to a certain extent is Tree–Based Scheduling (TBS) [34]. First, the control flow graph is converted into several trees, each tree representing a structure of nested conditional constructs. The conversion is performed by duplicating all the operations after join junctions. Then, operations are propagated towards the leaves of each tree. However, downward code motion is performed only within the scope of a *single* tree. Unlike TBS, our method is *global*: operations can move down possibly across various subgraphs representing nested conditionals. Besides, in TBS, the goal of downward propagation is merely to guarantee that operations execute only on control paths where their values are actually used, instead of our more general capturing of maximal freedom for downward code motion. Moreover, TBS does not handle the replication of operations on a same control path, which limits the range of legal downward code motions.

Our data–flow analysis suggests, as a by–product, a criterion for pruning inefficient code motions. Code–motion pruning is formulated as a reordering of the availability sets such that the code motions possibly increasing the schedule lengths are not effectuated. The price to pay for including the dynamic reordering expressed in Equation 5.2 seems affordable for the following reasons. The test $a <_{\Pi} b$ can be implemented in constant time. The test $a <_{\Lambda,i} b$ may require the enumeration of all links from a and b. Therefore, the test $a <_{\Pi,i} b$ takes $O(|\Lambda(a)| + |\Lambda(b)|)$, i.e., it is bounded by the total number of copies of a and b, typically a small fraction of the total number of operations.

Even though the basic properties of the data–flow captured by our techniques are also observed by other methods, classical approaches exploit them in the form of built–in schedule heuristics. In other words, the scheduling order of some operations is unnecessarily enforced, thereby preventing the exploration of possibly superior alternative solutions. Although our method is carefully designed to avoid the greed of classical built–in heuristics, it still has the theoretical drawback that optimal solutions might be excluded. In spite of this fact, experimental results indicate that the pruning increases the density of promising solutions, especially under *tight resource constraints*. This is typically the case for HLS. Although the technique might prune cheap solutions with respect to the number of states, this effect is smaller for solutions whose schedule lengths are close to the optimum, i.e., this side effect can be considered marginal under *tight time constraints*. Since code–motion pruning improves the quality of global scheduling and since the inherent dynamic–reordering mechanism relies on an efficient test on precedence, we conclude that the technique represents a viable option for a HLS tool.

Moreover, code–motion pruning could be used *on demand*. For instance, in the early (more iterative) phases of a design flow, quick exploration is imperative and the CMP should be enabled. On the other hand, in the late phases of a design flow, more time can be spent with optimization. For this reason, the CMP could be switched off in order to explore possibly cheaper solutions overlooked by the early quick exploration.

Chapter 6 Towards loop pipelining

In the previous chapters, techniques are proposed to overcome the limited ILP within BBs by exposing and exploiting parallelism beyond basic–block boundaries. Similarly, to surmount the limited ILP within a single iteration of a loop, this chapter discusses techniques to uncover parallelism beyond iteration boundaries. It shows how our approach can be extended to include the pipelining of loops containing conditional constructs.

6.1 Motivation

Let us illustrate some basic notions with the simple example in Figure 6.1. Assume that the fragment of DFG in Figure 6.1a represents the *body* of a loop, i.e., the operations to be executed in each iteration of a loop. Suppose that one adder and one subtracter are available and assume that execution takes a single cycle for both module types. In the figure, different superscripts denote operations executed in different iterations of a loop. For simplicity, the test governing the exit condition of the loop is omitted.

Assume that the loop body is scheduled such that the available parallelism is uncovered only within the scope of a single iteration of the loop. In each valid schedule, each iteration of the loop takes at least three cycles to execute and new output values are generated after every three cycles. An example of such a schedule is given in Figure 6.1b.

We can consider uncovering parallelism by constructing a new loop body comprising the operations associated with successive iterations of the original loop, as shown in Figure 6.1c. In other words, the original loop is turned into a *multi–iteration loop*. This is called *loop unrolling*. Note that if this scheme is used, two sets of output values are generated (one for each iteration in the loop body) in each interval of 5 cycles.

Alternatively, if we consider that another iteration of the loop can start before the previous iteration has finished, extra parallelism is uncovered without the need for unrolling the loop. This technique is known as *loop pipelining.* For instance, the execution of operation a in the second iteration can be performed simultaneously with the execution of operation d in the first iteration, as shown in Figure 6.1d. As a consequence, although each iteration still takes three cycles to complete, new results are produced after every two cycles. We say that the loop body is operating as an execution *pipeline*. In order to start the pipeline, some operations are executed in a so–called *pre–amble*. Conversely, in order to drain the pipeline, some operations are executed in a so–called *post–amble*. We henceforth refer to the pipelined loop body as the *kernel*. In the example of Figure 6.1d, the operation in state s_0 forms the pre–amble, whereas the operations in states s_3 and s_4 form the post–amble. The kernel consists of the operations in states s_1 and s_2. If it is required that the algorithm should process new data every two cycles, we conclude that the first two solutions do not satisfy the time constraint, whereas the last solution does.

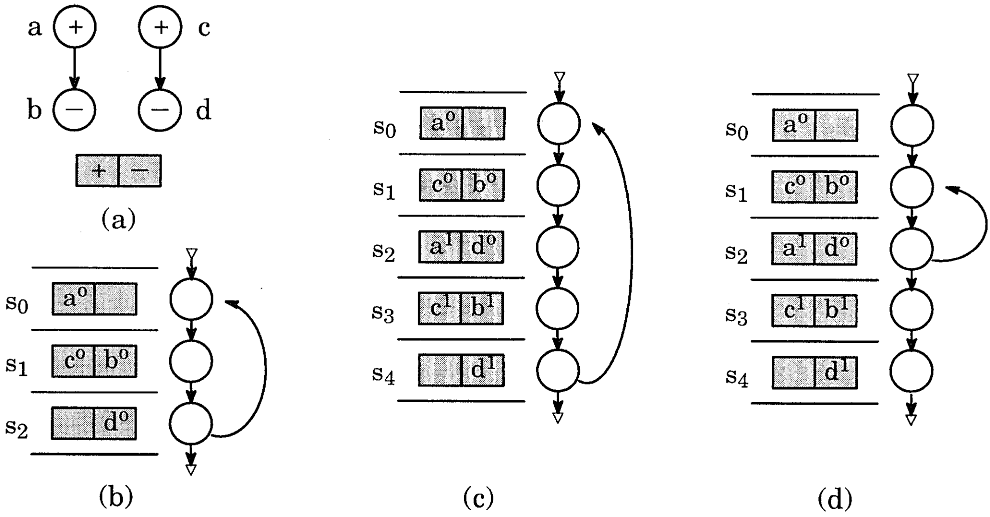


FIGURE 6.1. Illustrative example for loop unrolling and pipelining

Since the loop body operates as a pipeline, the notions of data–introduction interval and latency can be associated with the execution of the loop. The *data–introduction interval* of a loop, written DII, is the minimal number of clock cycles required between the execution of successive iterations of the loop. The *latency* of the pipelined loop is the length of the interval between the arrival time of input data and the time at which the computation of the respective output values is completed. For instance, for the example in Figure 6.1d, the DII is 2 cycles, whereas the latency is 3 cycles. On the other hand, for the example of Figure 6.1b both DII and latency are 3 cycles, since the loop body is not pipelined.

Essentially, the idea of loop unrolling is to increase the number of operations within the loop body in order to increase ILP. On the other hand, the underlying idea of loop pipelining is to execute operations from different iterations of the original loop body, but without increasing the number of operations in the kernel. Loop pipelining achieves the effect of scheduling with full unrolling [4], while producing less code than loop unrolling [30]. For these reasons, in the remainder of this chapter, we focus on loop pipelining. A comprehensive study of loop unrolling can be found in [30].

A common requirement when dealing with time–constrained applications is that the total number of iterations of a loop should not depend on the setting of the data [36][70]. Therefore, we address the pipelining of loops whose total number of iterations can be determined at compile time (e.g. "for" loops).

In the examples in this chapter, we assume a time constraint in the form of a specified upper bound for the DII of a loop. We also assume, for simplicity, that only data dependences within the scope of a same iteration of a loop are present. Section 6.5 discusses ways of relaxing these assumptions.

6.2 Fundamental notions

Let us now introduce some essential notions used throughout this chapter. A graph representing a loop is a directed *cyclic* graph (DCG). A DCG can be seen as a directed *acyclic* graph (DAG) that is made cyclic by the insertion of edges with special properties. To stress some properties of edges, we adopt the classification of edges proposed in [14], as summarized below.

Let $G = (V, E)$ be a directed graph and let $G_T = (V, X)$ be a rooted tree whose edges X connect all the nodes of G. An edge $(u, v) \in E$ is classified as follows:

- (u, v) is a *tree* edge if $(u, v) \in X$;
- (u, v) is a *forward* edge if u is an ancestor of v in G_T.
- (u, v) is a *back* edge if v is an ancestor of u in G_T.
- (u, v) is a *cross* edge if neither v nor u is an ancestor of the other in G_T.

Recall that, after scheduling a current state s_i, our parallelizer appoints next states and constructs the transitions from s_i to the next states. Since the flow of control forks when conditionals are scheduled, if our technique of merging equivalent states is not applied, the constructed transitions will form a tree rooted at the source, whose edges connect all the states created by the parallelizer. This is the tree used as the reference to classify the edges throughout this chapter. As a consequence, the edges inserted by merging equivalent states are either forward or cross edges and their insertion turns the tree into a DAG. As will be seen, the pipelining of loop bodies causes the insertion of back edges, turning the DAG into a DCG.

When operations are enclosed in the body of a loop, their execution is repeated several times, giving rise to distinct iterations. We refer to the operation executed at a given iteration as an *instance* of the operation at that iteration. Given an operation o_n, the instance of o_n at iteration j is written as o^j_n. We refer to the superscript j as the *iteration number*. Throughout this chapter, the "instance of an operation" at some iteration is simply called "instance", whenever that is clear from the context.

6.3 Related work

Modulo scheduling (MS) [39] is one of the most popular pipelining techniques. It assumes a fixed DII, which is calculated beforehand. Given a value for DII, if an operation is scheduled at cycle step c, the instance of this operation in iteration j is scheduled at cycle step $c + j \cdot \text{DII}$. Resource constraints must be satisfied for all the operations scheduled within the same cycle *modulo* DII. The main disadvantage of MS is that it does *not* handle loop bodies with conditional constructs directly. MS requires the prior application of auxiliary mechanisms, addressed as *hierarchical reduction* or *if–conversion*, in order to transform a multi–BB flow graph into a single–BB flow graph, as explained below.

Given the innermost conditional of a structure of nested conditional constructs, *hierarchical reduction* [40] consists of three main tasks. First, the BBs associated with the "then" and the "else" branches are scheduled independently. Second, the BB with shorter schedule length is padded with null operations ("nops") until it has the same length of the BB with longer schedule length. Third, the entire conditional construct is encapsulated as a single entity. The process is repeated for the next levels in the hierarchy of conditional nesting from the innermost to the outermost conditional. However, since hierarchical reduction preserves the original control structure, it restricts code motions. In *if–conversion* [5], each operation is guarded with a predicate associated with the tests on which the operation is control dependent. An operation is executed only when all the guards in the predicate evaluate to true. In other words, control dependences are expressed as data dependences. Consequently, the operations in the loop body are scheduled as if they were within a single BB, thereby overlooking conditional resource sharing, because mutually exclusive operations may compete for a same resource. Since the mechanism enforces control dependences, if–conversion restricts speculative execution. In summary, despite its efficiency, MS requires mechanisms to handle conditionals that restrict code motion and speculation.

An alternative for producing software pipelining is to use loop unrolling as the underlying mechanism. An example is *Perfect Pipelining* (PP) [2], which unrolls the loop a certain number of times and schedules the operations until a repetitive pattern is detected for the pipelined kernel. Some rules are necessary during the construction of a schedule to ensure that a pattern will eventually emerge. In [3][4], PP was extended to handle conditionals, such that no restriction is imposed on code motion within the unrolled loop body. One disadvantage of PP is that the convergence to a pattern may be slow in some cases [4]. The main advantages of the method in [3][4] is the graceful handling of conditional constructs that does not restrict code motion, along with the orthogonality between parallelization and scheduler heuristics. This eases the control of the quality of the generated solutions.

Another way of generating a pipelined loop body is by performing code motion across back edges of a cyclic flow graph. This is the underlying mechanism of Enhanced Pipeline Scheduling (EPS) [17][18]. Initially, an empty state is appointed just after the entry point of the loop. This state is "filled" with available instances of operations belonging to the first iteration. Then, the operations in the "filled" state are moved from the loop body to the pre–amble and new instances of those operations are added to the loop body. In general, each time an instance o^i is moved to the pre–amble, another instance o^{i+1} is inserted into the loop body. After "filling" a given state, next states are appointed and available instances are determined among the instances currently within the loop body. The process is repeated until all the operations in the original loop body have been moved once across a back edge. Since operations are moved from one iteration to another, but the number of operations in the loop body is conserved, this mechanism produces loop pipelining without the need for detecting a pattern. The main advantage of EPS is its ability to gracefully and efficiently handle conditionals within the loop body. The main disadvantage is that availability analysis is limited to the instances within the loop body, even if the available resources are poorly utilized. This restricts the exploitation of ILP to a certain extent. Also, the stop criterion for moving operations across back edges is completely arbitrary, which may prevent the generation of superior solutions. Moreover, since the criterion is not affected by the application of different priority encodings, design space exploration may be impaired.

In the context of HLS, there are approaches that treat loops, but do not allow loop pipelining, such as Path–Based Scheduling (PBS) [12] and Tree–Based Scheduling (TBS) [34]. In [12], for example, back edges are "broken" to obtain an acyclic control–flow graph prior to the application of the PBS algorithm. On the other hand, most of the loop pipelining techniques in the HLS literature restrict the handling of conditionals constructs [13][24][36]. Rotation Scheduling (RS) [13], for instance, has a mechanism of moving operations around the loop that is similar to EPS, but it relies on a formulation based on a DFG without conditionals. The method in [52] deals with acyclic DFGs containing conditionals, but only performs loop pipelining for cyclic DFGs without conditionals. Although the method in [32] combines speculative execution and loop pipelining, it is based on speeding up execution via branch prediction and, consequently, it is not suitable to applications oriented to worst–case execution. A recent approach [55] extends RS to combine both loop pipelining and the handling of conditional constructs. However, it employs greedy heuristics to maximize conditional resource sharing prior to loop pipelining.

6.4 Our approach for loop pipelining

Among the classes of loop pipelining algorithms summarized above, the one based on detection of a pattern [4] seems the most suitable to our approach, for the following reasons:

- **Parallelization is orthogonal to scheduler heuristics.** This allows us to borrow loop pipelining techniques from [4], without the need to introduce scheduler heuristics in the constructor engine, thereby keeping heuristics in the explorer engine, for the sake of proper design space exploration.
- **Detection of patterns and of equivalent states can be combined.** The mechanism for inducing loop pipelining suggested in [4] relies on an equivalence relation defined in terms of available instances of operations. This relation is used to detect a repeating pattern for the kernel of a pipelined loop. Given the states s_m and s_n, if the process of scheduling starting in each of them produces the same pattern and s_n reaches s_m, state s_m can be merged with s_n, closing a cycle in the SMG. Recall that our detection of equivalent states also merges states in the SMG. Although state equivalence is not addressed in [4], our parallelizer can be extended to perform loop pipelining by building upon this previous work.
- **Loop pipelining can be seen as code motion between iterations.** When loop unrolling is used as the underlying mechanism for inducing loop pipelining, the BBCG associated with the unrolled loop body is a DAG to which our modeling of code motion can be directly applied, along with code–motion pruning. In other words, code motion beyond iteration boundaries is modeled as code motion across basic–block boundaries.

In the remainder of this section, we describe how our constructor can be extended to induce loop pipelining. ILP is uncovered by implicitly unrolling the loop for the sake of availability analysis and by scheduling states forming an acyclic SMG until the merging of states adds the back edges, thereby forming the pipelined kernel and distinguishing it from pre–amble and post–amble.

6.4.1 Required extensions

Since the SMG generated by loop pipelining is cyclic, the evaluation of the cost of a solution can not rely on measuring the schedule length of some path p (L_p) from source to sink, as it is the case for acyclic SMGs. For instance, if a time constraint T_c is imposed as an upper bound on the DII, the number of steps to execute a *cycle* should be measured instead, in order to check if T_c is satisfied. Given a back edge (s_j, s_i), the schedule length of the respective cycle is the schedule length of the path from s_i to s_j, as formalized below.

DEFINITION 6.1
Given a simple cycle $c = \langle s_0, s_1, \dots, s_{k-1}, s_0 \rangle$ in a SMG, the *schedule length of cycle* c, written L_c, is the number k of states included in the path $\langle s_0, s_1, \dots, s_{k-1} \rangle$.

In the context of loop pipelining, L_c should replace L_p in the cost function used in the optimization problem, since L_c represents the DII.

To support loop pipelining, the sets of available operations should contain instances of operations belonging to distinct iterations of a loop, i.e., $A_k = \{\dots, o^j_n, o^i_m, \dots\}$, where $i \neq j$ possibly holds. In the loop pipelining technique described in [4], a constraint has to be applied in order to guarantee termination: the instances of operations in every set A_k should contain instances of at most W successive iterations, where W is an arbitrary natural number. It is as if available instances of operations were confined to a *window* comprising W successive iterations. The value of W is a parameter of the loop pipelining algorithm and it does not need to be the same for every loop. The algorithm is such that, when all the instances with minimal iteration number are scheduled, this iteration is shifted out of the window and a new iteration is shifted into it, thereby sliding the window one iteration further. For this reason, we say that available instances of operations are confined to a *sliding window* of size W. As a consequence, the following extensions should be made:

- The set of available operations A_k may contain instances of operations belonging to at most W successive iterations of a loop. In other words, if the minimal iteration number of the instances in A_k is i, then only available instances o^j such that $j < i + W$ are included in A_k.
- A priority has to be associated with each possible instance inside the sliding window. Therefore, given a loop body with N operations, the priority encoding becomes a permutation Π consisting of W distinct instances of each operation, i.e., $|\Pi| = W \cdot N$.

Although the sliding window constrains the scope of globality to at most W successive iterations of a loop, the value W can be changed, since it is a parameter of the loop pipelining algorithm. Fortunately, this constraint imposed on availability analysis is acceptable and even desirable in practice for two reasons. First, it provides a way of controlling the size of the search space, since a reduction in W decreases the number of distinct permutations. Second, it contributes to controlling the growth of the number of states. The reason is that, when a loop body containing conditional constructs is unrolled, the number of control paths increases with the size of the window. A similar concept of window is used in [47] to make loop parallelization affordable in terms of code expansion.

6.4.2 Basic principle

Let us illustrate the basic idea of the detection of a pipelined kernel. Reconsider the DFG in Figure 6.1a, assuming that it represents the body of a loop. Suppose that we choose a sliding window of two iterations ($W = 2$). This means that we are allowed to look for available instances among those belonging to two successive iterations of the loop.

Figures 6.2a to 6.2e show the scheduling process, state after state, where the set of available instances is shown for each state. Since operations a and c depend only on the inputs, their instances in iterations 0 and 1 are available at state s_0 (Figure 6.2a). After a^0 is scheduled in state s_0, b^0 becomes available at state s_1 (Figure 6.2b). When c^0 and b^0 are scheduled in state s_1, d^0 becomes available (Figure 6.2c). Note that d^0 is the last unscheduled instance from iteration 0. When d^0 is finally scheduled in state s_2 (Figure 6.2d), all instances from iteration 0 are scheduled and, for keeping the size of the sliding window, we are allowed to seek for available operations among the instances from iteration 2. Note that a^1 is also scheduled in state s_2, which contains instances from different iterations, paving the way to loop pipelining. Note that the set A_3 contains c^1 (which could not be scheduled in state s_2), b^1 (which becomes available after a^1 is scheduled) and the instances a^2 and c^2 (available in the new iteration "added" to the window). Note that the sets A_1 and A_3 are similar, except for the superscripts. As a consequence, the process of scheduling starting at state s_3 generates a sequence of states whose operations are identical to those in the sequence of states starting at state s_1, except that the operations execute in different iterations, as indicated in Figure 6.2e. This suggests that the repetitive pattern should become the kernel of a pipelined loop body, by merging s_3 with state s_1, as shown in Figure 6.2f. Consequently, the detection of a pattern can rely on the set of available operations without the need for scheduling any further. In addition, the merging of the states with same available instances causes the insertion of back edges.

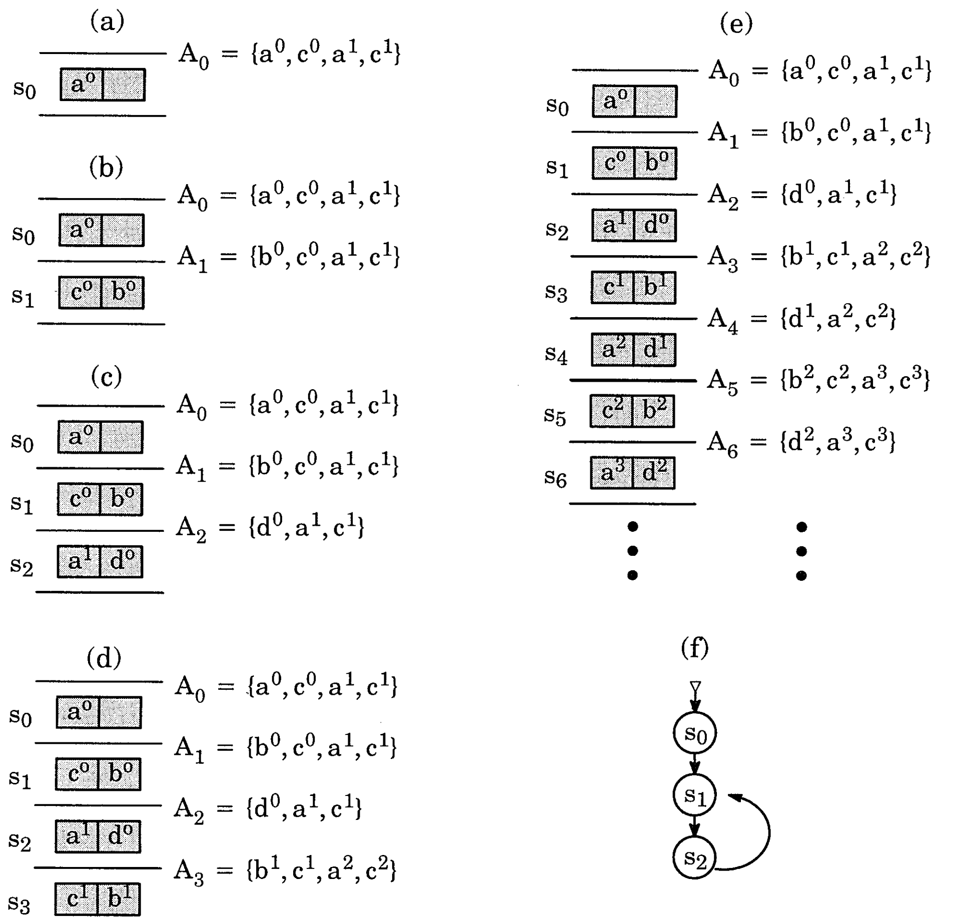


FIGURE 6.2. An illustration for the detection of a repetitive pattern

The idea illustrated in this example is formalized in [4] and it is based on an equivalence relation on the sets of available operations, as summarized below. Let $X = \{ \dots, o_n^j, \dots \}$ be a set of instances of operations. Given an integer c, the set X^c is the set $\{ \dots, o_n^{j+c}, \dots \}$, i.e., X^c and X contain instances of exactly the same operations, but the iteration numbers differ by a constant c. In the following definition, let **Z** be the set of integers.

DEFINITION 6.2

Given two sets A and B of instances of operations, A is *equivalent* to B, written $A \equiv B$, iff:

$$\exists c \in \mathbf{Z} : A = B^c.$$

Now we are able to introduce the basic principle for detection of a pipelined kernel. Let s_n be an already scheduled state and let s_m be an "empty" state about to be scheduled. Consider a state $s_i \in$ PRED(s_m). Recall that R_n represents the resource occupation in state s_n. In the loop pipelining algorithm in [4], when $A_n \equiv A_m$, $R_n = R_m$ and $s_n \overset{*}{\rightarrow} s_m$, the state s_m is merged with state s_n, such that each transition (s_i, s_n) becomes a back edge.

Both our scheduling mechanism and our availability analysis satisfy the requirements for the correct detection of a pipelined kernel, as described in [4]. Termination is guaranteed by providing a sliding window of size W. Proofs of correction and termination for the loop pipelining algorithm are given in [4]. Algorithm 6.1 illustrates how the detection of a pipelined kernel is combined with the detection of equivalent states, by modifying the procedures in Algorithm 4.9.

ALGORITHM 6.1. Combining detection of kernel and of state equivalence

```
procedure merging_state($s_m$)
if ($\exists s_n \in S \,|\, A_n \equiv A_m \wedge R_n = R_m$) /* criterion in [4] */
  if ($s_n \overset{*}{\rightarrow} s_m$)
    return ($s_n$);
  else
    if ($s_n \overset{\phi}{=} s_m$)                /* criterion in Theorem 4.1 */
      return ($s_n$);
return(none);

procedure handle_current_state($s_m$)
$s_n$ := merging_state($s_m$);
if ($s_n \neq$ none)
  merge $s_m$ with $s_n$;
else
  schedule $s_m$;
```

In the first procedure, the condition for detecting a pipelined kernel proposed in [4] is tested beforehand. If that condition holds and the outcome of the test on reachability is true, the merging of s_m with s_n will insert a back edge. However, if the result of the test on reachability is false, either a cross edge or a forward edge will be inserted when states s_m and s_n are detected to be schedule equivalent.

6.4.3 An example

In this section, we show a simple example of how our constructor can manage loop pipelining. In Figure 6.3a, the behavioral description of a loop body is given, along with the resource constraints. We assume that the test takes zero cycles (its outcome is a flag set by the subtracter) and that the conditional has zero delay. To simplify the example, we assume that the loop is executed an infinite number of times (i.e. there is no exit test).

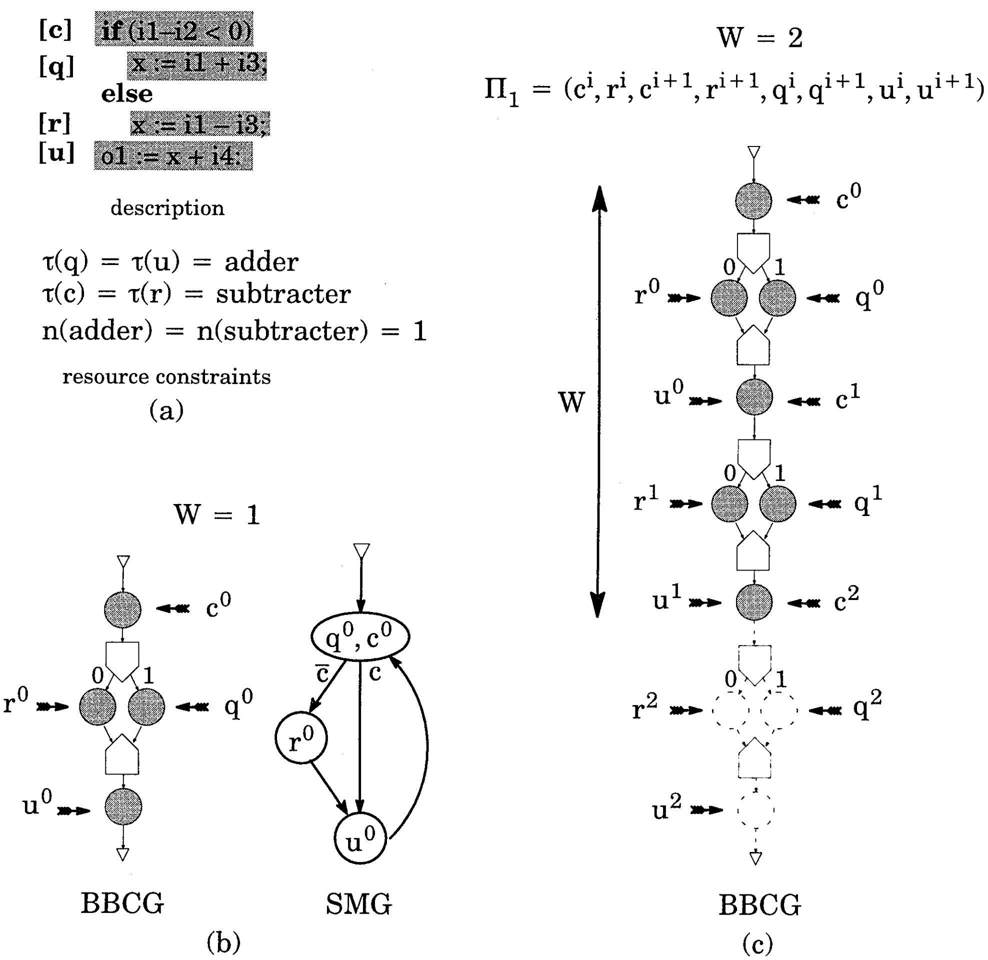


FIGURE 6.3. An example of loop body to be pipelined

In Figure 6.3b, a BBCG and a SMG are given for the loop whose body is in Figure 6.3a. This SMG is obtained if the sliding window comprises a single iteration (W = 1, i.e. no loop pipelining). Figure 6.3c contains a priority encoding, which is a permutation Π of instances of the operations within a sliding window of two successive iterations (W = 2). In the permutation, i denotes the minimal iteration number currently within the sliding window.

This figure also contains a BBCG for the loop body unrolled W times, although we also indicate the dashed subgraph that will emerge when the window is slid one iteration further. Our constructor appends one instance of such a subgraph to the BBCG each time a new iteration is added to the sliding window. Note that the links emanate from instances of operations in distinct iterations. In particular, note that distinct instances of the conditional (c^0, c^1, c^2) are linked to the BBs preceding the branch nodes.

In the following, we describe the process of constructing a SMG, state after state, for the example in Figure 6.3 when $W = 2$. The process is illustrated in Figures 6.4 and 6.5, where unscheduled states are shaded, the current state has a heavier outline and edges inserted via merging of states are highlighted. Since we have already illustrated in Chapter 3 how a SMG is derived from the BBCG, here we show the SMG directly, for simplicity of presentation. The sets of available instances at each state are shown in Table 6.1. For instance, when s_0 is appointed, the set A_0 contains instances of operations that depend only on the inputs.

Current state: s_0 (Figure 6.4a)
Since conditional c^0 is scheduled in s_0, two next states are appointed, s_1 and s_2. As q^0 is also scheduled in s_0, instance u^0 becomes available at state s_2, but not at state s_1, because u^0 depends on r^0 if the outcome of c^0 is false. On the other hand, r^0 remains available at state s_1, but not at state s_2, because r^0 does not execute on the path taken when the outcome of c^0 is true.

Current state: s_1 (Figure 6.4b)
When r^0 is scheduled in state s_1, instance u^0 becomes available at state s_3. Since q^1 is scheduled in state s_1 too, instance u^1 also becomes available at state s_3.

Current state: s_2 (Figure 6.4c)
Instances c^1 and q^1 are scheduled. Since c^1 is an instance of a conditional, two next states are appointed, s_4 and s_5. Since q^1 is scheduled in state s_2, instance u^1 becomes available at s_5, but not at s_4, because it depends on r^1 if the outcome of c^1 is false. Instance r^1 is available at s_4, but not at s_5, because it is not executed when the outcome of c^1 is true.

Current state: s_3 (Figure 6.4d)
Instances c^1 and u^0 are scheduled. As a conditional is scheduled, two next states are appointed, s_6 and s_7. Since u^0, which was the last unscheduled instance with minimal iteration number, is scheduled in state s_3, the sliding window is shifted one iteration further and we can now search for available instances from iteration 2. As a consequence, instances c^2, r^2, q^2 become available at both s_6 and s_7. Note that r^1 remains available at s_6 only, whereas u^1 remains available only at state s_7.

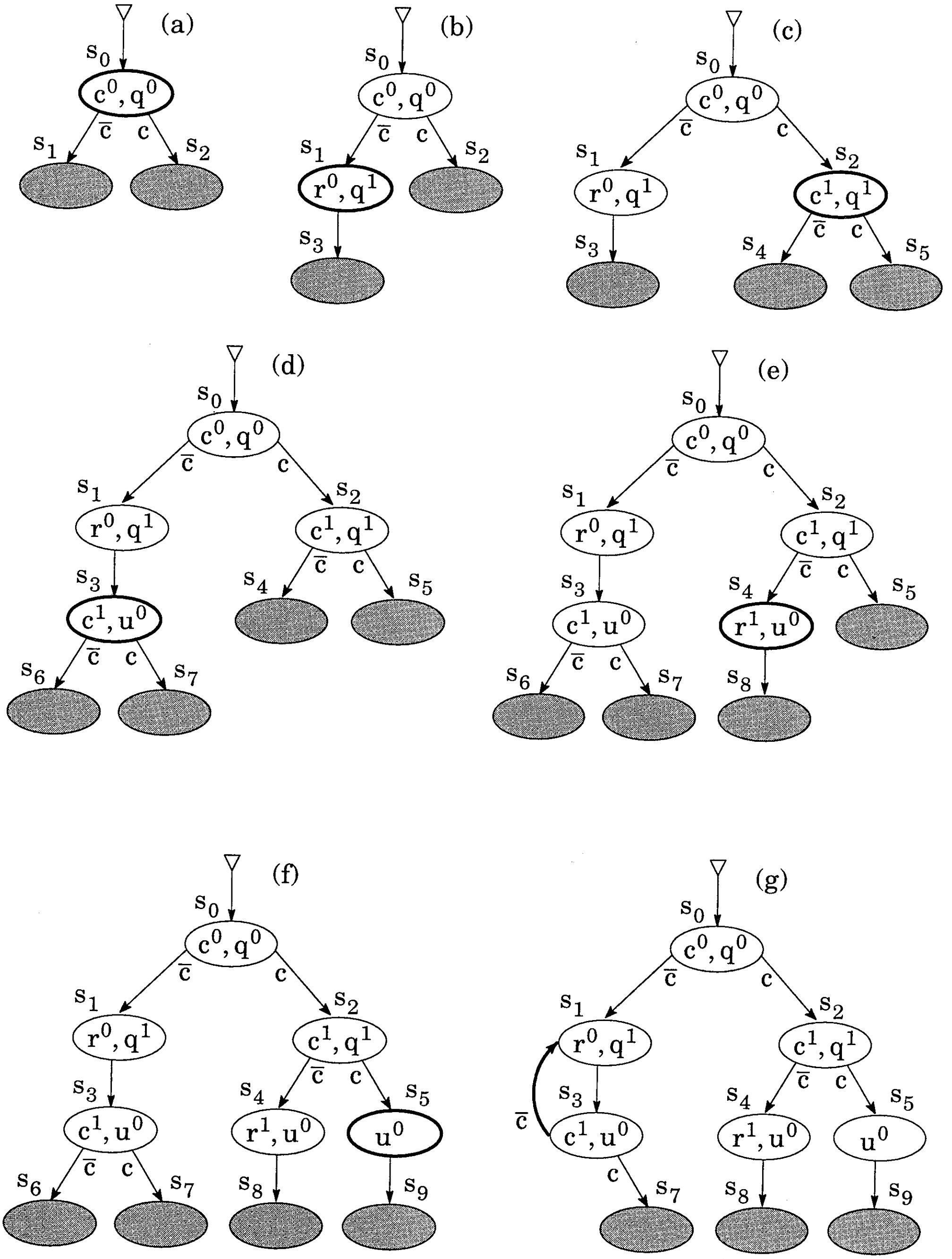


FIGURE 6.4. Pipelining the loop body in Figure 6.3 (Part I)

Current state: $\mathbf{s_4}$ (Figure 6.4e)
Since u^0, the last instance from iteration 0, is scheduled in state s_4, available instances from iteration 2 (c^2, r^2, q^2) become available at state s_8. The scheduling of r^1 makes u^1 available at state s_8.

Current state: $\mathbf{s_5}$ (Figure 6.4f)
Only u^0 can be scheduled at s_5. Consequently, iteration 0 is completed and instances c^2, r^2, q^2 become available at state s_9.

Current state: $\mathbf{s_6}$ (Figure 6.4g)
Note that both $A_6 \equiv A_1$ and $s_1 \xrightarrow{*} s_6$ hold. For this reason, s_6 is merged with s_1 and the back edge (s_3, s_1) is inserted.

Current state: $\mathbf{s_7}$ (Figure 6.5a)
Note that $A_7 \equiv A_2$, $\neg(s_2 \xrightarrow{*} s_7)$ and $s_2 \overset{\phi}{\equiv} s_7$ is detected. For this reason, s_7 is merged with s_2 and the cross edge (s_3, s_2) is inserted.

Current state: $\mathbf{s_8}$ (Figure 6.5b)
Note that $A_8 \equiv A_2$ and $s_2 \xrightarrow{*} s_8$ hold. When s_8 is merged with s_2, a back edge (s_4, s_2) is inserted.

Current state: $\mathbf{s_9}$ (Figure 6.5c)
Note that $A_9 \equiv A_2$ and $s_2 \xrightarrow{*} s_9$ hold. When s_9 is merged with s_2, a back edge (s_5, s_2) is inserted.

TABLE 6.1 The sets A_k for the examples in Figures 6.4 and 6.5

state	available instances
s_0	$A_0 = \{c^0, r^0, c^1, r^1, q^0, q^1\}$
s_1	$A_1 = \{r^0, c^1, r^1, q^1\}$
s_2	$A_2 = \{c^1, r^1, q^1, u^0\}$
s_3	$A_3 = \{c^1, r^1, u^0, u^1\}$
s_4	$A_4 = \{r^1, u^0\}$
s_5	$A_5 = \{u^0, u^1\}$
s_6	$A_6 = \{r^1, c^2, r^2, q^2\}$
s_7	$A_7 = \{c^2, r^2, q^2, u^1\}$
s_8	$A_8 = \{c^2, r^2, q^2, u^1\}$
s_9	$A_9 = \{c^2, r^2, q^2, u^1\}$

Note that the final SMG in Figure 6.5c contains three cycles. Each cycle has instances of operations from distinct iterations of the loop, which is therefore pipelined. In this figure, the pre–amble is automatically separated from the kernel as a result of the insertion of back edges (the same would be valid for the post–amble, which does not occur in this example, because we are assuming an infinite number of iterations for simplicity).

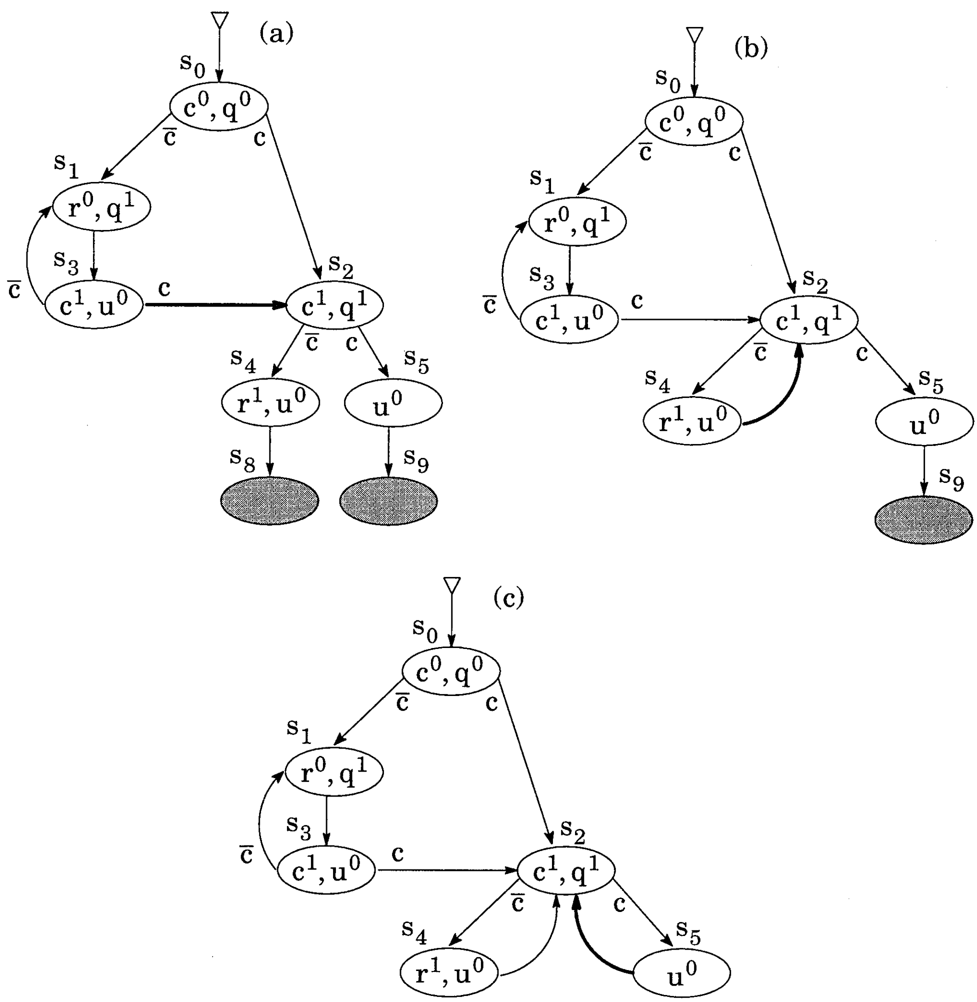


FIGURE 6.5. Pipelining the loop body in Figure 6.3 (Part II)

It may be interesting to observe some sequences of bundles induced by sequences of enabling predicates. For instance, assume a sequence of predicates where the outcome of the conditionals alternate between true and false, i.e., $\mathcal{G}_1 = \langle c, \bar{c}, 1, c, 1, \bar{c}, 1, c, 1, \bar{c}, 1, \dots \rangle$. Figure 6.6a shows the sequence of bundles induced by $\mathcal{G}_1$. Figure 6.6b illustrates a sequence of bundles induced by a sequence of predicates in which the outcome of the first two tests is false and the outcome of all other tests is true, i.e., $\mathcal{G}_2 = \langle \bar{c}, 1, \bar{c}, 1, c, c, 1, c, 1, \ldots \rangle$. The arrows indicate the arrival time of input data for each new iteration of the original loop body (from iteration 1 on), along with the time where output data is available. Note that in both cases DII = 2.

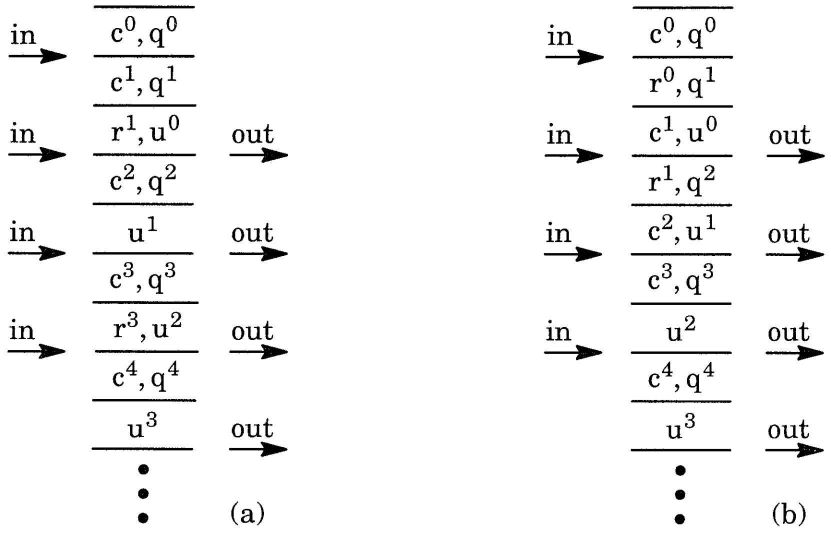


FIGURE 6.6. Sequences of bundles for the example in Figure 6.5c

Compare Figure 6.5c with Figure 6.3b, assuming a time constraint of two cycles for the DII. The SMG in Figure 6.3b does not satisfy this time constraint, whereas the one in Figure 6.5c does. Note also that the SMG in Figure 6.5c has a larger number of states. The growth of the number of states can be seen as the price to pay for time–constraint satisfiability.

6.5 Discussion

This chapter has shown that the pipelining of loops with conditional constructs can be interpreted as code motion across iteration boundaries. The chosen loop–pipelining algorithm relies on loop unrolling as an implicit mechanism for exposing parallelism. States are scheduled until a pipelined kernel is detected. Such choice turns the problem of finding a pipelined kernel into the problem of moving operations across basic blocks. This allows us to reuse the methods proposed in the previous chapters, thereby integrating loop pipelining in our constructive approach. Consequently, our support for upward and downward code motions, along with techniques like code compensation, availability analysis, on–the–fly detection of equivalent states, and code–motion pruning can all be employed for the pipelining of loops containing conditional constructs.

Although we focussed on the impact of schedule lengths on the DII, latency can also be evaluated by measuring the schedule lengths of paths from a state in which input data arrives to a state in which the corresponding output data is available. Since different permutations typically lead to distinct pipelined kernels, it is possible to explore alternative SMGs with different DII, latency and number of states.

For simplicity, we have assumed throughout this chapter only data dependences within the scope of a same iteration. However, some loops give rise to dependences between instances of operations in different iterations, the so–called *loop–carried dependences* [30]. Availability analysis can be extended to cope with this kind of dependences. The modeling of loop–carried dependences and its relationship with a DFG is addressed, for instance, in [13][28]. Besides, we have focussed on pipelining a single loop, even though the method can be further extended to cope with nested loop structures by applying loop pipelining in a hierarchical way. First, the innermost loop is pipelined. Then, loop pipelining is applied to the outer loop in the next level of the hierarchy of nesting. In the outer loop, the kernel of the inner loop is encapsulated as a single entity, which is considered as a "complex" operation. The operations in the pre–amble and post–amble are not encapsulated and can therefore be scheduled simultaneously with the operations in the outer loop. The process is repeated from the inner to the outermost loop.

Since the pipelining of loop bodies containing conditionals tends to increase the number of states, the flexibility granted by the sliding window should be employed to avoid paying too high a price for time–constraint satisfiability. In practice, it may be convenient to start the design space exploration by checking if the time constraint can be satisfied by simply applying code motion to the loop body. This can be done by setting W = 1. If the time constraint can not be satisfied, then loop pipelining should be enabled by setting W > 1.

Although the pipelining of loops containing conditional constructs represents a topic for further research, a rough implementation of the method in [4] already indicates some of the issues that have to be solved to make this technique affordable within our approach. For instance, one of the difficulties preliminarily detected when using the algorithm in [4] is that it may converge faster to a pattern on some control paths than others. As a consequence, the final SMG sometimes contains not only loops with pipelined kernels, but also loops with multi–iteration bodies. In other words, we have detected in a preliminary implementation that the algorithm in [4] has the disadvantage of sometimes generating a kernel that is just a scheduled version of the unrolled loop body, instead of a pipelined version. As a result, if a SMG contains various cycles, the ones which comprise actual pipelined kernels are compact, while the ones with unrolled bodies exhibit a larger number of states. The cause of this side effect seems to be the generality of the criterion of kernel detection used in [4]. This criterion, which is summarized in Algorithm 6.1, seems too weak for avoiding unrolled bodies. A more elaborate criterion should be investigated as a way to avoid a large number of inferior solutions. One possible line of research is to exploit the time constraint itself to enforce precedence of operations. This extra precedence can be used to restrict or reorder the set of available operations and thereby overcoming undesirable loop unrolling. A recent technique [44] that exploits time–constraints is applied successfully to DFGs without conditional constructs. We believe that the combination of some ideas from that related work with some concepts described in this chapter may lead to promising results.

In summary, by choosing an algorithm that gracefully handles conditional constructs and is largely free of restrictions, our approach seems able to combine loop pipelining with code motion and speculative execution, although more research is needed to make the pipelining of loops containing conditional constructs mature for the HLS of time–constrained systems.

Chapter 7 Conclusions

7.1 Concluding remarks

In this thesis, we have proposed an approach for exploiting ILP during the high–level synthesis of synchronous digital systems. The approach supports ILP techniques like code motion, speculative execution and can be extended to include loop pipelining. It is oriented towards applications in which the operation of the digital system should comply with a global time–constraint. Given a behavioral description of the digital system and a set of resource constraints, the goal of the approach is to find a symbolic FSM whose critical execution path complies with a pre–specified time constraint. For this reason, we have directed the application of ILP techniques towards *worst–case execution* time, as opposed to traditional approaches, which aim at speeding up average execution time.

We have shown that various classical approaches rely on an aggregate of heuristics for generating a single solution, hopefully satisfying the time constraint. Since it is difficult to control the quality of the final result, those approaches can jeopardize time–constraint satisfaction. Our approach distinguishes itself from others, because it allows the *exploration* of several alternative solutions. In order to provide proper design space exploration, our constructive approach relies on three engines which co–operate in the generation of alternative FSMs with different schedule lengths and number of states. The *constructor* manages the application of ILP techniques, the *explorer* is responsible for searching for the best solution and the *Boolean oracle* answers queries about conditional execution. An important feature of the approach is that it allows us to trade search time against solution quality. For instance, in the early (more iterative) phases of a design flow a quick search can be used, while broader neighborhoods can be explored in the final optimization phase.

Support for upward and downward code motion

We have shown that code motion can be modeled by reorganizing the links between operations and basic blocks. In our approach, downward code motion is performed during an initialization phase and upward code motion takes place in the course of global scheduling.

Support for speculative execution

It was pointed out that traditional approaches efficiently cope with the side effects of code motion by means of hardware support for committing or discarding speculative results at execution time. Although efficient for applications aiming at speeding up average execution, such a technique is not suitable when we have to guarantee compliance with the time constraint. We have proposed a method for keeping track of speculative execution based on Boolean predicates. An important aspect of this method is that, when combined with code compensation, there is no need for specialized hardware support, since the side effects of speculative execution are handled in software.

Availability analysis

The mechanism of inducing code motion presented in this thesis relies on the *global* computation of the operations available for scheduling at a given state. The proposed technique captures major achievements from the compiler domain while keeping one of the most popular HLS representations, namely, the DFG. The key is to combine graph manipulation for keeping track of data dependences with Boolean techniques for dealing with conditional execution.

Code compensation

The technique of inserting compensation code to preserve the semantics of conditional execution was addressed as a way of broadening the range of legal code motions. We have shown that this classical compiler technique can be incorporated in a HLS tool by reformulating the traditional path queries as a combination of Boolean queries and simple graph handling. Since our approach is directed towards worst–case execution (as opposed to parallelizing compilers), we have proposed a technique to avoid that code compensation might jeopardize time–constraint satisfiability.

Boolean encoding for conditional execution

The backbone for most of the techniques we proposed is a simple and efficient Boolean encoding for conditional execution. Boolean predicates are defined in terms of the outcome of conditionals and used to implement queries. Most of the employed queries involve two predicates which are Boolean products. In the worst case, these queries can be performed in low–order polynomial time. Although other queries may have higher complexity, the encoding guarantees that the number of Boolean variables is bounded by the depth of conditional nesting, which in practice is typically a small fraction of the total number of conditionals. Therefore, all the queries can be efficiently handled by the Boolean oracle.

An important practical aspect of our Boolean modeling of conditional execution is that public–domain packages for Boolean manipulation are widely available in the design automation community. Therefore, our formulation for availability analysis and code compensation holds the promise of allowing existent HLS tools to benefit from *global* scheduling at the expense of a few extensions.

On–the–fly detection of equivalent states

One of the main advantages of removing built–in scheduler heuristics is that it makes the detection of equivalent states affordable, due to the predictability of scheduling selection. This fact, combined with an efficient way of encoding and recovering information on conditional execution, allows our approach to guarantee a minimal state equivalent SMG for a given priority encoding. As a consequence, the optimization of the number of states can be achieved via exploration of alternative solutions. To our knowledge, no other scheduling method supports on–the–fly detection of equivalent states while simultaneously applying code motion and speculative execution.

As opposed to most traditional approaches, which disregard the use of some code motions for the sake of avoiding code expansion, we have shown some experimental evidence that the merging of equivalent states allows us to make more flexible code motions affordable. The technique works not only as a mechanism for restraining code expansion (because a smaller number of states is obtained), but it also speeds up the process of scheduling (because less states are actually scheduled).

Code–motion pruning

One of the major results reported in this thesis refers to the unbalance between the parallelism *exposed* by the application of ILP techniques and the parallelism that can actually be *exploited*. Experimental results indicate that not all ILP exposed is beneficial. Parallelism that is exposed but is not accommodated within the available resources leads to the generation of many inferior solutions in the course of the optimization process. We have proposed a technique to cope with such an unbalance, the so–called code–motion pruning (CMP). This technique first captures the constraints imposed to downward code motion. Then, these constraints are used as a criterion to select the most efficient code motions. We have shown experimental evidence that the solution space has higher density of good–quality solutions when CMP is applied. As a consequence, for a given local–search method and for a same number of explored solutions, the application of CMP is likely to lead to a superior local optimum. Conversely, a smaller number of solutions has to be explored to reach a given schedule length, what correlates to a reduction of search time.

Extension for loop pipelining

The close relationship between code motion and loop pipelining was pointed out as a last topic in this thesis. Uncovering ILP across iteration boundaries was modeled as exposing ILP across basic block boundaries. This formulation allows loop pipelining to benefit from all the techniques mentioned above. A very important consequence of this modeling is that we do not have to restrict code motion for supporting the pipelining of loop bodies containing conditional constructs, as opposed to the most popular methods. Moreover, we have shown that the detection of a pipelined kernel and the detection of equivalent states can be integrated in the same mechanism of merging states.

Flexibility

An important practical aspect of our approach is its flexibility of use. For instance, although largely unrestricted code motions are supported, some kinds of code motion could be enabled or disabled on demand. Code–motion pruning can be switched on and off, depending on how much time can be afforded during exploration. The size of the sliding window can be adjusted to restrain code expansion, depending on how tight the constraints are. The parameters of a given search method can be modified for trading–off solution quality against search time. Due to its flexibility, our constructive approach represents an alternative method in between monolithic exact methods such as [52] and approaches based on an aggregate of unrelated heuristics such as [54].

7.2 Topics for further research

Even though promising results are obtained with the application of our techniques, some issues still have to be addressed to bring the approach closer to practical application. The most important extensions and improvements are discussed below.

Incorporating register allocation

We have focussed on scheduling with the assumption that the allocation of registers is performed afterwards. This is a common situation in HLS, where the number of registers is not necessarily fixed beforehand. It may be interesting to perform register allocation for each generated solution in order to take the number of implied registers into account during the exploration of alternative solutions.

The main notion guiding register allocation is the concept of *lifetime* of a value [16]. This is the interval between the time at which a value is computed and the latest time when it is consumed. The analysis of lifetimes can be extended to deal with SMGs by using our Boolean modeling for conditional execution, as follows. Assume that an operation x is linked to some basic block my means of a link λ. Assume that operation x is scheduled in some state s_m. The register storing the value x can be reused to store another value, say y, in some state s_n when one of the following circumstances holds:

- The value x is not consumed by any operation scheduled in any other state reachable from s_n.
- State s_n and s_m are mutually unreachable (the execution of operations x and y is mutually exclusive).
- State s_n is reachable from s_m through a sequence of transitions induced by some predicate Γ and $\Gamma \cdot G(\lambda)$ is not satisfiable (this means that operation x was speculatively executed on the taken control path and its output value is discarded).

In summary, classical lifetime analysis [16] can be extended with our Boolean queries to take the effects of mutually exclusive execution and speculative execution into account.

Fixed number of registers as extra resource constraints

We have addressed only the resource constraints imposed by the allocated functional units. However, the approach can be further extended to include a fixed number of registers as extra resource constraints. If no free register is available to store the result of a moved operation, this code motion must be prevented. The support for viewing a fixed number of registers as resource constraints is essential to the code generation for programmable ASIPs. It is also useful in the HLS of ASICs, because the number of registers implied by a given schedule might turn out to be excessive and a more practical number should then be used as a constraint during a new iteration through the design flow.

Extra pruning by exploiting the time constraint

In this thesis, we have focussed on techniques that avoid jeopardizing time–constraint satisfiability: unrestricted code motions are largely supported, code compensation mechanisms are carefully designed, no built–in scheduler heuristics are used and design space exploration is emphasized. In addition, our code–motion pruning utilizes *precedence and resource constraints* for improving the quality of the generated solutions. However, we do not exploit the *time constraint* itself for the sake of pruning. Recently, powerful techniques [44][67] are proposed for taking advantage of time constraints. The time constraint dictates a reduction of scheduling freedom, thereby discarding inferior solutions. The tighter the time constraint, the larger the reduction becomes. Experimental results illustrate the efficiency of those techniques when applied to DFGs *without* conditional constructs. Since the efficiency of those techniques is not yet demonstrated when applied to DFGs with complex control–flow and under a general speculative execution model, we believe that the integration of those techniques with our work should be investigated. As our techniques and those described in [44][67] address issues that are essentially complementary, their combination seems promising.

As a final remark, it may be worth mentioning that speculative execution and code motion are recently receiving more attention in HLS [32] [36] [52] [54], although most of the work focuses on speeding up average execution. We believe that this trend is likely to be accentuated also in favor of time–constrained systems. The reason is that simple expedients as allocating more resources for the data path in order to grant time–constraint satisfiability can not be conceived as permanent solutions in the absence of support for exploiting ILP. In face of complex emerging applications combining intensive data–flow, complex control–flow and time constraints, the exploitation of parallelism available within basic blocks is likely to be insufficient to ensure the satisfaction of a tight time–constraint.

References

[1] A. Aho, R. Sethi and J. Ullmann, "Compilers – Principles, Techniques, and Tools", Addison–Wesley, 1986, pp. 631–632.

[2] A. Aiken and A. Nicolau,"Perfect pipelining: A new loop parallelization technique", Proc. European Symposium on Programming, pp. 221–235, 1988.

[3] A. Aiken and A. Nicolau,"A Realistic Resource–Constrained Software Pipelining Algorithm", Advances in Languages and Compilers for Parallel Processing, MIT Press, Cambridge, Mass., pp. 274–290, 1991.

[4] A. Aiken et al.,"Resource–Constrained Software Pipelining", IEEE Transactions on Parallel and Distributed Systems, vol. 6, no. 12, pp. 1248–1270, December 1995.

[5] J. Allen et al., "Conversion of control dependence to data dependence", Proc. Symposium on Principles of Programming Languages, pp. 177–189, January 1983.

[6] S. Amellal and B. Kaminska,"Functional Synthesis of Digital Systems with TASS",IEEE Trans. on Computer Aided Design, vol. 13, no. 5, pp. 537–552, May 1994.

[7] U. Banerjee et al.,"Automatic Program Parallelization", Proceedings of the IEEE, vol. 81, no. 2, pp. 211–243, February 1993.

[8] R. Bergamaschi et al.,"Area and Performance Optimizations in Path Based Scheduling", Proc. European Conference on Design Automation, pp. 304–310, 1991.

[9] R. Bergamaschi et. al.,"Control–Flow Versus Data–Flow Based Scheduling: Combinining Both Approaches in an Adaptive Scheduling System", IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 5, no.1, pp.82–100, March 1997.

[10] M. Berkelaar and L. van Ginneken, "Efficient Orthonormality Testing for Synthesis with Pass–Transistor Selectors", Proc. ACM/IEEE Int. Conference on Computer Aided Design, pp. 256–263, 1995.

[11] E. Berrebi et. al.,"Combined Control Flow Dominated and Data Flow Dominated High–Level Synthesis", Proc. 33rd ACM/IEEE Design Automation Conference, pp. 573–578, 1996.

[12] R. Camposano,“Path–based scheduling for synthesis”, IEEE Trans. on Computer–Aided Design, vol. 10, no.1, pp. 85–93, January 1991.

[13] L.–F. Chao et. al.,“Rotation Scheduling: A Loop Pipelining Algorithm”, Proc. 30th Design Automation Conference, pp. 566–572, 1993.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest,“Introduction to Algorithms”, McGraw–Hill Book Company, 1991.

[15] De Man, H. et al.,“Architecture Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms”, Proc. of the IEEE, pp. 319–335, 1990.

[16] G. De Micheli,“Synthesis and Optimization of Digital Circuits”, Mc Graw–Hill, 1994.

[17] K. Ebcioglu,“A compilation technique for software pipelining of loops with conditional jumps”, Proc. 20th Annual Workshop on Microprogramming, pp. 69–79, December 1987.

[18] K. Ebcioglu and R. Nakatani, “A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture”, in Languages and Compilers for Parallel Computing, edited by D. Gelernter et al., MIT Press, 1989, pp. 213–229.

[19] K. Ebcioglu and A. Nicolau,“A global resource–constrained parallelization technique”, Proc. of the ACM SIGARCH Int. Conference on Supercomputing, pp. 154–163, 1989.

[20] J. Eijndhoven, G. G. de Jong and L. Stok,“The ASCIS Data Flow Graph – semantics and textual format”, EUT Report 91–E–251, Eindhoven University of Technology, 1991.

[21] J. Eijndhoven and L. Stok,“A Data Flow Exchange Standard”, Proc. European Conference on Design Automation, pp. 193–199, 1992.

[22] J. A. Fisher,“Trace Scheduling: A technique for global microcode compaction”, IEEE Trans. on Computers, vol. C–30, no. 7, pp.478–490, July 1981.

[23] D. D. Gajski et al.,“High–Level Synthesis: Introduction to Chip and System Design”, Kluwer Academic Publishers, 1991.

[24] G. Goossens et al.,“Loop optimization in register–transfer scheduling for DSP–systems”, Proc. 26th Design Automation Conference, pp. 826–831, 1989.

[25] G. Goossens et al.,“Integration of Medium–Throughput Signal Processing Algorithms on Flexible Intruction–Set Architectures”, Journal of VLSI Signal Processing, vol. 9, no. 1, pp. 49–65, 1995.

[26] T. Gross and M. Ward,“The suppression of Compensation Code”, Advances in Languages and Compilers for Parallel Processing, MIT Press, pp. 260–273, 1991.

[27] R. Gupta and M.L. Soffa,“Region Scheduling: An Approach for Detecting and Redistributing Parallelism”, in IEEE Trans. on Software Engineering, vol. 16, no. 4, pp. 421–431, April 1990.

[28] M.J.M. Heijligers,“The Application of Genetic Algorithms to High–Level Synthesis”, PhD. Thesis, Eindhoven University of Technology, 1996.

[29] M. J. M. Heijligers et al.,“NEAT: an Object Oriented High–Level Synthesis Interface”, Proc. of the IEEE International Symposium on Circuits and Systems, pp. 1.233–1.236, 1994.

[30] J. L. Hennessy and D. A. Patterson,“Computer Architecture – A Quantitative Approach”, 2nd edition, Morgan Kaufmann Publishers Inc., 1996.

[31] F. J. Hill and G. R. Peterson,“Computer Aided Logical Desigh with Emphasis on VLSI”, John Wiley & Sons, Inc., 4th edition, pp. 267–274, 1993.

[32] U. Holtmann and R. Ernst,“Combining MBP–Speculative Computation and Loop Pipelining in High–Level Synthesis”, Proc. European Design and Test Conference, pp. 550–555, 1995.

[33] P. Y. T. Hsu and E. S. Davidson, “Highly Concurrent Scalar Processing”, Proc. of the 13th Annual International Symposium on Computer Architecture, pp. 386–395, June 1986.

[34] S.Huang et al.,“A tree–based scheduling algorithm for control dominated circuits”, Proc. ACM/IEEE Design Automation Conference, pp. 578–58, 1993.

[35] K. Kennedy, “A Survey of Data Flow Analysis Techniques”, in “Program Flow Analysis: Theory and Applications”, edited by S. Muchnick and N. Jones, Prentice–Hall, 1981, pp.17–18.

[36] A. Kifli,“Global Scheduling in High–Level Synthesis and Code Generation for Embedded Processors”, PhD. Thesis, Catholic University of Leuven, Belgium, 1996.

[37] T. Kim et al.,“A Scheduling Algorithm for Conditional Resource Sharing – A Hierarchical Reduction Approach”, IEEE Trans. on Computer Aided Design, vol. 13, no. 4, pp. 425–438, April 1994.

[38] R. P. Kleihorst et al.,"MPEG2 Video Encoding in Consumer Electronics", Journal of VLSI Signal Processing, vol. 17, pp. 241–253, 1997.

[39] M. Lam,"Software pipelining: An effective scheduling technique for VLIW machines", Proc. SIGPLAN'88 Conf. on Programming Language Design and Implementation, pp. 318–328, June 1988.

[40] M. Lam, "A Systolic Array Optimizing Compiler", Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[41] M. S. Lam and R. P. Wilson,"Limits of Control Flow on Parallelism", Proc. ACM/IEEE International Symposium on Computer Architecture, pp. 46–57, 1992.

[42] S.–Z. Lin, C.–T. Hwang and Y.–C. Hsu,"Efficient Microcode Arrangement and Controller Synthesis for Application Specific Integrated Circuits", Digest of Technical Papers of the International Conference on Computer–Aided Design, pp. 38–41, 1991.

[43] J. L. van Meerbergen et al.,"PHIDEO: High Level Synthesis for High–Throughput Applications", Journal of VLSI Signal Processing, vol. 9, pp. 89–104, 1995.

[44] B. Mesman et al.,"Constraint Analysis for DSP Code Generation", Proc. 10th International Symposium on System Synthesis", pp. 33–40, 1997.

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

[46] S.–M. Moon and K. Ebcioglu,"An Efficient Resource–Constrained Global Scheduling Technique for Superscalar and VLIW processors", Proc. Int. Symposium and Workshop on Microarchitecture *(MICRO–25)*, pp. 55–71,December 1992.

[47] T. Nakatani and K. Ebcioglu,"Making Compaction–Based Parallelization Affordable", IEEE Trans. on Parallel and Distributed Systems, vol. 4, no. 9, pp. 1014–1029, September 1993.

[48] A. Nicolau,"Uniform Parallelism Exploitation in Ordinary Programs", Proc. International Conference on Parallel Processing, pp. 614–618, 1985.

[49] A. C. Parker et al., "MAHA: A program for datapath synthesis", Proc. ACM/IEEE Design Automation Conference, pp. 461–465, 1986.

[50] C. Papadimitriou and K. Steiglitz,“Combinatorial optimization: algorithms and complexity”, Prentice Hall, pp. 3–8, 454–455, 1982.

[51] R. Potasman et al., “Percolation Based Synthesis”, Proc. ACM/IEEE Design Automation Conference, pp. 444–449, 1990.

[52] I.Radivojevic and F.Brewer,“A New Symbolic Technique for Control Dependent Scheduling”, IEEE Transactions on Computer Aided Design, vol.15, no. 1, pp. 45–57, 1996.

[53] M. Rim and R. Jain,“Representing conditional branches for high–level synthesis applications”, Proc. ACM/IEEE Design Automation Conference, pp. 106–111, 1992.

[54] M. Rim, et al.,“Global Scheduling with Code–Motions for High–Level Synthesis Applications”, IEEE Trans. on VLSI Systems, vol. 3, no. 3, Sept. 1995, pp. 379–392.

[55] J. Siddhiwala et al., “Scheduling Conditional Data–Flow Graphs with Resource Sharing”, Proc. Great Lakes Symposium on VLSI, 1995, pp. 94–97.

[56] L.C.V. dos Santos et al.,“A Constructive Method for Exploiting Code Motion”, Proc. ACM/IEEE International Symposium on System Synthesis, pp. 51–56, 1996.

[57] L.C.V. dos Santos et al.,“Combining code motion and scheduling”, Proc. ProRISC/IEEE–Benelux Workshop on Circuits, Systems and Signal Processing, pp. 279–284, 1996.

[58] L.C.V. dos Santos, “A method to control compensation code during global scheduling”, Proc. ProRISC/IEEE–Benelux Workshop on Circuits, Systems and Signal Processing, pp. 527–534, 1997.

[59] L.C.V. dos Santos, “Modeling speculative execution and availability analysis with Boolean expressions”, to appear in: Proc. ProRISC/IEEE–Benelux Workshop on Circuits, Systems and Signal Processing, 1998.

[60] L.C.V. dos Santos et al., “A Code–Motion Pruning Technique for Global Scheduling”, to appear in: ACM Transactions on Design Automation of Electronic Systems, vol. 5, n. 2.

[61] P. Slock et al.,“Fast and extensive system–level memory exploration for ATM applications”, Proc. 10th ACM/IEEE Int. Symposium on Systems Synthesis, pp. 74–81, September 1997.

[62] M. Smith, M. Horowitz, and M. Lam.,“Efficient Superscalar Performance Through Boosting,” Proc. International Conference

on Architectural Support for Programming Languages and Operating Systems (ASPLOS–5), pp. 248–259, 1992.

[63] L. Stok,"Architectural Synthesis and Optimization of Digital Systems", PhD. Thesis, Eindhoven University of Technology, 1991.

[64] M. T. J. Strik et al.,"Efficient Code Generation for In–House DSP–Cores", Proc. European Design and Test Conference, pp. 244–249, 1995.

[65] R.M.H. Takken,"Controller Design for a Single–chip MPEG–2 Encoder", Master thesis, Eindhoven University of Technology, August 1995.

[66] A. H. Timmer,"From Design Space Exploration to Code Generation – a constraint satisfaction approach for the architectural synthesis of digital VLSI circuits", PhD. Thesis, Eindhoven University of Technology, 1996.

[67] A. H. Timmer and J. A. G. Jess,"Execution Interval Analysis under Resource Constraints", Digest of technical papers of the International Conference on Computer–Aided Design, pp. 454–459, 1993.

[68] A. H. Timmer et al.,"Conflict Modelling and Instruction Scheduling in Code Generation for In–House DSP Cores", Proc. 32nd Design Automation Conference, pp. 593–598, 1995.

[69] R. Vaessens, E. Aarts, J. Lenstra, "A Local Search Template", Computers and Operations Research, vol. 25, no. 11, pp. 969–979, 1998.

[70] J. Vanhoof et al., "High–Level Synthesis for Real–Time Digital Signal Processing", Kluwer Academic Publishers, pp. 12–26, 149–166, 1993.

[71] K. Wakabayashi and T. Yoshimura,"A resource sharing and control synthesis method for conditional branches", Proc. ACM/IEEE Int. Conference on Computer Aided Design, pp.62–65, 1989.

[72] K. Wakabayashi and H. Tanaka,"Global scheduling independent of control dependencies based on condition vectors", Proc. ACM/IEEE Design Automation Conference, pp.112–115, 1992.

[73] A. van der Werf et al.,"I.McIC: A single Chip MPEG2 Video Encoder for Storage", Proc. International Solid State Circuits Conference, pp. 254–255, 1997.

Appendix

A Experimental set–up

In this appendix, we summarize the resource constraints and delay values for the examples used in the experiments reported in this thesis.

In Table A.1, the examples are listed in the first column, along with the references to the papers where they are introduced. The second and third columns give an idea about DFG and BBCG sizes, respectively. In the fourth column, different labels are associated with the distinct sets of resource constraints reported in the last five columns. The labels are used to refer to the respective constraints in the tables of results included in this thesis.

TABLE A.1 The resource constraints for the employed examples

example	nodes	BBs	case	resource constraints				
				alu	add	sub	mul	cmp
waka	46	10	A	0	1	1	0	1
[72]			B	2	0	0	0	1
kim	48	10	B	0	1	1	0	1
[37]			C	0	2	1	0	1
rotor	66	10	A	1	0	0	0	0
			B	2	0	0	0	0
			C	3	0	0	0	0
[52]			D	4	0	0	0	0
			E	1	0	0	2	0
			F	2	0	0	2	0
			G	3	0	0	2	0
s2r	122	22	A	1	0	0	0	0
			B	2	0	0	0	0
			C	3	0	0	0	0
[52]			D	4	0	0	0	0
			E	1	0	0	2	0
			F	2	0	0	2	0
			G	3	0	0	2	0
			H	4	0	0	2	0
kim_big	464	52	A	0	1	1	1	1
			C	0	1	2	1	1
[37]			D	0	1	1	2	1

In Table A.2, we illustrate the operation assignment τ and the operation delay d(v) for each of the cases described in Table A.1. This information is organized in two distinct groups of columns, where each column corresponds to an operation type $t_O \in T_O$ with $T_O = \{ +, -, >, <, \times \}$. The entries in the first group show the selected module type t_S for each operation v with $\omega(v) \in T_O$. Acronyms are used to identify module types such as adder (add), subtracter (sub), arithmetic–logic unit (alu), comparator (cmp) and multiplier (mul). The entries in the second group represent the delay d(v) for each operation v with $\omega(v) \in T_O$. In addition to the information in this table, we assume $dii(t_S) = 1$ for each module type t_S. Also, we assume that every conditional c is such that $d(c) = 0$, i.e., the delay slot is zero and we suppose 2–way branch capability.

TABLE A.2 Operation assignment and operation delay information

example	case	operation assignment				delay			
		+	−	>,<	×	+	−	>,<	×
waka [72]	A	add	sub	cmp	–	1	1	1	–
	B	alu	alu	cmp	–	1	1	1	–
kim [37]	B,C	add	sub	cmp	–	1	1	1	–
rotor [52]	A,B,C,D	alu	alu	alu	alu	1	1	0	1
	E,F,G,H	alu	alu	alu	mul	1	1	0	2
s2r [52]	A,B,C,D	alu	alu	alu	alu	1	1	0	1
	E,F,G,H	alu	alu	alu	mul	1	1	0	2
kim_big [37]	A,C,D	add	sub	cmp	mul	1	1	1	1

In all the distinct cases described for the examples "rotor" and "s2r", a single–port look–up table is also available and it is modeled as an operation with unit delay.

As a last observation, we should mention that the example "kim_big" is obtained from a DFG in [37], but we were obliged to adapt the original graph to our DFG model and also to comply with some restrictions of format imposed by our current implementation. Therefore, the example "kim_big" refers to a DFG slightly *different* from the original DFG.

Appendix

B Auxiliary information

This appendix describes a function used in Chapter 4 when taking the effect of multicycling into account during availability analysis. Algorithm B.1 shows how to evaluate the function max_displacement(o_n, $s_{i,k}$), which is used in Equation 4.5. Let begin($s_{i,k}$) denote the number of cycles required to execute the states of a path from the source to $s_{i,k}$ but excluding $s_{i,k}$. Given a link λ with $o_m \overset{\lambda}{\rightsquigarrow} BB_x$, let end($\lambda$) denote the time when the execution of o_m is completed on some path from the source to BB_x.

Given an available operation o_n and the current state $s_{i,k}$, the described function evaluates the maximal displacement between the time when the current state $s_{i,k}$ starts executing and the time when the output values of the producers o_m are *all* computed. This displacement represents the "distance" between the current state and the state at which o_n is available.

ALGORITHM B.1. Algorithm for the function used in Equation 4.5

```
function max_displacement($o_n$, $s_{i,k}$)
d = 0;
foreach $o_m \in \mathrm{PROD}(o_n)$
  foreach $\lambda$ with $o_m \overset{\lambda}{\rightsquigarrow} BB_x \wedge BB_x \overset{*}{\rightarrow} BB_i$
    if ( begin($s_{i,k}$) < end($\lambda$) )
      d := max(d, end($\lambda$) − begin($s_{i,k}$));
return (d);
```

Biography

Luiz Cláudio Villar dos Santos was born on November 21, 1963 in Arapongas, Brazil. In the period between 1978 and 1981, he worked as a clerk in his hometown, where he concluded high school.

He studied Electrical Engineering at the Federal University of Paraná, in Curitiba, Brazil, from which he graduated with honors on February 17, 1987. He received the degree of Master in Computer Science on May 30, 1990, from the Federal University of Rio Grande do Sul, in Porto Alegre, Brazil.

From June 1990 to April 1991, he worked on IC design at the Catholic University of Louvain–la–Neuve, Belgium. Then, he returned to the Federal University of Rio Grande do Sul, where he worked as an assistant researcher until February 1993. On March 1993, he joined the Computer Science Department at the Federal University of Santa Catarina, in Florianópolis, Brazil. He is on leave from that university since November 1994, when he started working towards a doctorate in the Design Automation Section of the Department of Electrical Engineering of the Eindhoven University of Technology.

He expects to receive the degree of doctor based on the work presented in this thesis on November 23, 1998. Afterwards, he shall return to Brazil and resume his research and teaching activities as an assistant professor at the Federal University of Santa Catarina. His research interests include synthesis of digital circuits and computer architecture.

Stellingen

behorende bij het proefschrift
Exploiting instruction–level parallelism: a constructive approach
van Luiz Cláudio Villar dos Santos

1. The following statement should not be taken to the letter: "The CDFG representation maintains the control structure specified by the designer ... Data dependency information is represented only within the basic block; data dependencies across basic blocks are not explicitly represented. A synthesis system that works from the CDFG representation must maintain the basic block structure ... This is one of the major disadvantages of using a CDFG representation directly for synthesis". [D. Gajski and L. Ramachandran, IEEE Design and Test of Computers, Winter 94, p. 47].

   [this thesis, chapters 2/4]

2. When addressing high–level synthesis scheduling in the presence of conditionals, all "exact" methods are indeed exact, but some are "less exact" than others.

   [this thesis, chapter 3]

3. The use of the notion of mutually exclusive operations is not a main issue in high–level synthesis, because it perpetuates a way of modeling not appropriate for complex control–flows.

4. The developer of a high–level synthesis optimization tool should avoid restrictions, but welcome constraints.

5. CAD consists of a few tools and a lot of HAD (human–aided design).

6. Writing a thesis suggests, after all, a conservative attitude.

7. The systematic use of a diary to make appointments with friends is an exaggeration.

8. Some teachers teach us what we should learn. Some teachers teach us how we should learn. Some teachers teach us how we should not teach.

9. In the early eighties, the Vatican has played distinct political roles in Poland and in Brazil. From the point of view of pursuit of freedom, these roles were contradictory.

10. The following two sentences are equivalent in their ability to motivate further work:

- "Improvements in science are either *evolutionary* or *revolutionary*. The former consists of compounding little steps, each related to perfecting the solutions to some problems. The latter involves radical changes in the way in which problems are modeled and solved. ... Architectural and logic synthesis have been revolutionary, ... An evolutionary growth of synthesis is likely in the near future. Many techniques need to be perfected, and many problems, which were originally considered of marginal interest, need now to be solved." [Giovanni De Micheli, "Synthesis and Optimization of Digital Circuits", p. 560].
- "You are late for dinner and the dishes are piled up."