UNIVERSITY OF CALIFORNIA
Santa Barbara

Symbolic Scheduling Techniques

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

by

Ivan Radivojevic

Committee in charge:

Professor Forrest D. Brewer, Chairman

Professor Malgorzata Marek-Sadowska

Professor Kwang-Ting Cheng

Professor P. Michael Melliar-Smith

Doctor Barry Pangrle

March 1996

The dissertation of Ivan Radivojevic
is approved:

_____

_____

_____

_____

_____
Committee Chairman

March 2, 1996

March, 1996

To my Mother.

Acknowledgments

First, I would like to thank my advisor, Professor Forrest Brewer, for his guidance and support throughout my graduate studies at University of California, Santa Barbara. This work would not have been possible without the most inspiring discussions we have had in the past four years.

Also, I would like to thank the Committee members: Professor Margaret Marek-Sadowska, Professor Kwang-Ting Cheng, Professor Michael Melliar-Smith, and Dr. Barry Pangrle for helpful suggestions and comments helping to improve the presentation of this work.

I would like to gratefully acknowledge contributions from Dr. A. Seawright who took part in early discussions and developed the original C++ BDD package extensively used throughout this project. My special thanks go to A. Crews, C. Monahan, and A. Stornetta for recent efficiency improvements while re-implementing the package.

This work was sponsored in part by fellowship donations from Mentor Graphics Corporation as well as UC-MICRO program. It would not have been possible without their generous support and willingness to help academic research.

I want to use this opportunity to express my thanks, one more time, to all of my teachers and colleagues at: University of California, Santa Barbara, Drexel University, Philadelphia, and University of Belgrade, Yugoslavia, for their contributions to my knowledge and enthusiasm over the fifteen year period.

Finally, my deepest gratitude goes to my mother Miroslava and to April Funcke. Their love, care and patience words cannot express.

VITA

Born — Belgrade, Yugoslavia — August 28, 1962.

EDUCATION

M. S. Electrical Engineering, 1990.
Department of Electrical and Computer Engineering
Drexel University
Philadelphia, PA, U.S.A.

B. S. Electrical Engineering, 1987.
University of Belgrade
Belgrade, Yugoslavia

FIELDS OF STUDY

Major Field: Computer Engineering

Specialization: System Level Computer-Aided Design
Professor Forrest Brewer

PROFESSIONAL EXPERENCE

Graduate Student Researcher, Department of Electrical and Computer Engineering, University of California, Santa Barbara — September 1994.

Teaching Assistant, Department of Electrical and Computer Engineering, University of California, Santa Barbara — September 1991.

Teaching Fellow, Department of Electrical and Computer Engineering, Drexel University, Philadelphia — January 1991.

Teaching Assistant, Department of Electrical and Computer Engineering, Drexel University, Philadelphia — September 1989.

Research Engineer, Faculty of Electrical Engineering, University of Belgrade, Yugoslavia — November 1987.

PUBLICATIONS

Journal papers:

I. Radivojević and F. Brewer, "A New Symbolic Technique for Control-Dependent Scheduling", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no.1, pp. 45-57, January 1996.

I. Radivojević and F. Brewer, "Symbolic Scheduling Techniques", *IEICE Trans. Information and Systems*, vol. e78-d, no. 3, pp. 224-230, March 1995.

I. Radivojević and J. Herath, "Executing DSP Algorithms in a Fine-Grained Dataflow Environment", *IEEE Trans. Software Engineering*, vol. 17, no. 10, pp. 1028-1041, October 1991.

I. Radivojević, J. Herath, and W. S. Gray, "High-Performance DSP Architectures for Intelligence and Control Applications", *IEEE Control Systems Mag.*, vol. 11, no. 4, pp. 49-55, June 1991.

Conference papers:

I. Radivojević and F. Brewer, "Analysis of Conditional Resource Sharing using a Guard-based Control Representation", *Proc. Int. Conf. Computer Design*, Austin, Texas, pp. 434-439, October 1995.

I. Radivojević and F. Brewer, "On Applicability of Symbolic Techniques to Larger Scheduling Problems", *Proc. European Design and Test Conf.*, pp. 48-53, Paris, France, March 1995.

I. Radivojević and F. Brewer, "Incorporating Speculative Execution in Exact Control-Dependent Scheduling", *Proc. 31st ACM/IEEE Design Automation Conf.*, pp. 479-484, San Diego, CA, June 1994.

I. Radivojević and F. Brewer, "Ensemble Representation and Techniques for Exact Control-Dependent Scheduling", *Proc. 7th Int. Symp. High-Level Synthesis*, pp. 60-65, Niagara-on-the-Lake, Ontario, Canada, May 1994.

I. Radivojević and F. Brewer, "Symbolic Techniques for Optimal Scheduling", *Proc. 4th Synthesis and Simulation Meeting and International Interchange (SASIMI)*, pp. 145-154, Nara, Japan, October 1993.

I. Radivojević and J. Herath, "DSP Architectural Features for Intelligence and Control Applications", *Proc. 5th Int. Symp. Intelligent Control*, Philadelphia, PA, September 1990.

Symbolic Scheduling Techniques

by

Ivan Radivojević

ABSTRACT

This thesis describes an exact symbolic formulation of control-dependent, resource-constrained scheduling. The technique provides a closed-form solution set in which all satisfying schedules are encapsulated in a compressed Binary Decision Diagram (BDD) representation. This solution format greatly increases the flexibility of the synthesis task by enabling incremental incorporation of additional constraints and by supporting solution space exploration without the need for rescheduling. The technique provides a systematic treatment of speculative operation execution for arbitrary forward-branching control structures. An iterative construction method is presented along with benchmark results. The experiments demonstrate the ability of the proposed technique to efficiently exploit operation level parallelism not explicitly specified in the input description.

Keywords: Binary Decision Diagrams; Control Dominated Circuits; High-Level Synthesis; Operation Level Parallelism; Scheduling.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Operation Scheduling

Two types of dependencies exist between the operations from a program specification. *Data-flow dependencies* impose precedence (execution order) between the operations. For example, operation O_2 has to be executed after operation O_1, if a result computed by O_1 is used by O_2. *Control-flow dependencies* arise when some portions of the specification are executed conditionally. An example of such conditional behavior is illustrated by a code fragment shown in Figure 1.1. The code indicates that condition *C* is computed and its outcome is used to determine a flow of control of the program. If *C* is "True", operation *A* is executed; otherwise (when *C* is "False") operation *B* is executed. All data-flow and control-flow dependencies have to be satisfied to ensure a correct execution of the specified behavior.

```
if ( C )
        A;
else
        B;
```

Figure 1.1 Control flow dependencies

Additional constraints arise due to finite hardware resources. *Resource constraints* impose bounds on a number of functional units available for the task execution. For example, a microprocessor implementation may incorporate two adder circuits and, consequently, not more than two additions can be executed simultaneously.

Another set of restrictions comes from the *timing constraints*. In many time-critical applications (e.g. aircraft engine control) computer hardware has to react to a recognition of a specific event within a strictly prescribed time interval.

Now we define the *operation scheduling* problem addressed in this thesis:

Definition 1.1 Operation scheduling is the process of determining the assignment of operations to time steps of a synchronous system, subject to data/control flow dependencies and resource/timing constraints.

The goal of operation scheduling is to find an execution order of operations that optimizes specific objective function. In particular, we are interested in applications of scheduling to computer-aided design (CAD) of digital circuits. For example, given bounds on available hardware resources, a goal of finding the fastest possible execution schedule can be set. Alternatively, we can look for a schedule that requires the minimal implementation cost while meeting a pre-specified bound on number of execution steps. Such goal reflects a trade-off between the task's execution time and circuit complexity of VLSI (very large scale integration) integrated circuits.

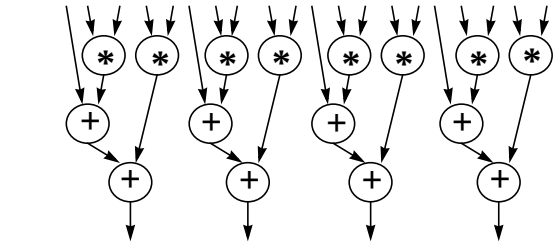
When program includes conditional behavior, some operations may be "mutually exclusive". Operations O_1 and O_2 are mutually exclusive if, during the program execution, either O_1 or O_2 (but not both) is going to be executed. In

Figure 1.1, once the condition *C* is computed, operations *A* and *B* are mutually exclusive and can use the same hardware resource during the execution.

Very frequently, however, it happens that execution order of the condition *C* and operations *A* and *B* is not pre-specified. In such cases, given sufficient hardware resources, operations *A* and *B* can be executed at the same time or even *before* the computation of *C*. This kind of program execution is called *speculative operation execution*. It has been shown that speculative execution can significantly improve execution time by using otherwise idle hardware resources [100][105][119][122]. This, however, increases the complexity of the scheduling task in a dramatic fashion since use of hardware resources has to be determined dynamically during the scheduling process.

Example

Figure 1.2 shows the XMAC example corresponding to a *block-matrix-multiply-and-accumulate* computation. Assume that the XMAC is to be executed on a data-path consisting of a four-cycle pipelined multiplier and a four-cycle pipelined adder. All of the input operands as well as four final output values are to be stored in a multiport general-purpose register file with a single-cycle access



$$\begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} + \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix} \times \begin{bmatrix} c11 & c12 \\ c21 & c22 \end{bmatrix} = \begin{bmatrix} r11 & r12 \\ r21 & r22 \end{bmatrix}$$

Figure 1.2 XMAC example

time. However, to avoid such single-cycle performance penalty, two bypass paths (bypass registers) are available for a direct transfer of operands between the functional units. Figure 1.3 shows the optimal execution schedule for the XMAC in which all of the intermediate results are forwarded using two bypass paths.

Although we intentionally modeled the underlying data-path after a floating-point portion of a recent high-performance commercial microprocessor [34], the XMAC example is, admittedly, very simple and somewhat contrived. In particular, the example does not exhibit conditional behavior. A presence of conditional

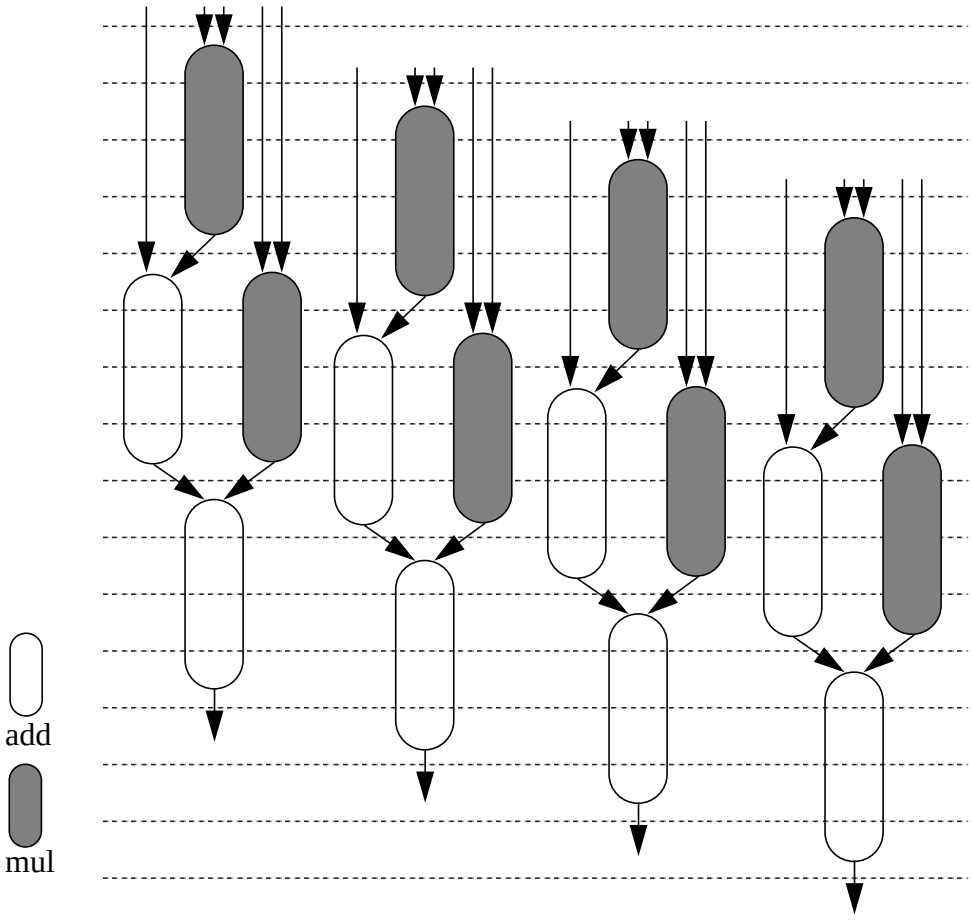


Figure 1.3 XMAC schedule

constructs dramatically increases complexity of the scheduling task. To further elaborate on this point, we will introduce several simple but illustrative examples in Section 1.2, Chapter 2 and Section 6.3.

1.1.1 Complexity of the Scheduling Problem

Throughout this thesis we will assume an input in the form of a *control/data flow graph* (CDFG) specification that describes both data-flow and control-flow dependencies between the operations. Acyclic data-flow graphs (DFGs) are straight-line operation sequences without branching control statements (e.g. *if-then-else*, *case*, *goto*, *exit*) and loops constructs (e.g. *while*, *do-while*). Cyclic DFGs include loop statements but no other forms of branching. Branching statements are present in CDFGs. A CDFG is acyclic if it does not include loop constructs. Otherwise, we say that a CDFG is cyclic.

We discuss a complexity of the scheduling problem using the concept of *time optimality* as defined in [108]:

Definition 1.2 A program *P* is said to be time optimal if for every instruction *I* of *P*, executed at some cycle *c*, there exist at least one operation *op* in *I* and a dependence chain of length *c* ending at *op* [1].

Intuitively, this means that every possible execution path runs in the shortest possible time. Time optimal scheduling of both acyclic and cyclic DFGs [2] is achievable in polynomial time assuming unlimited resources. However, even the problem of acyclic DFG scheduling becomes NP-complete for finite resources [40]. Time-optimal schedule for an acyclic CDFG is always achievable by scheduling all possible execution paths individually and executing them in parallel.

1. For the purpose of a discussion relevant to this thesis, the word “program” can be substituted by “schedule”, and “instruction” corresponds to a set of operations executed at the same cycle (time step).

Unfortunately, this, in general, requires an exponential number of operations as indicated in [105].

Recently, in particular in the area of parallelizing compilers, there has been a considerable interest in software pipelining techniques for cyclic CDFGs [100]. In [108], time optimal scheduling of arbitrary loops is investigated. In that work, the authors consider a theoretical parallel machine with finite but unlimited number of resources. Notice that such an assumption does have practical implications: although a number of resources may grow arbitrarily large in the future, it must be finite in any real-life hardware implementation. For such a machine, it can be demonstrated that, in general, time optimal scheduling of arbitrary loops is impossible.

1.2 Previous Work

High-level synthesis (HLS [31][74][121]) is an automated process that transforms an algorithmic specification of the behavior of a digital system into a hardware structure that implements the behavior. Resource-constrained operation scheduling is one of the crucial tasks in HLS. We say that scheduling is control-dependent if some operations from the control/data flow graph (CDFG) are executed conditionally due to the presence of control-flow constructs such as: *if-then-else*, *goto*, *case*, *exit* etc.

There are two difficult issues in a formal treatment of control-dependent, resource-constrained scheduling:

- concise formulation of the conditional behavior
- treatment of resources.

An efficient formulation should not generate an excessive number of constraints and formulation variables. Moreover, a formal evaluation of resource availability in the face of conditional execution is required. This is particularly difficult when movement of operations across basic code block boundaries is not prohibited. It has been demonstrated that the ability to perform speculative operation execution leads to superior schedules [20][100][105][119].

Current practical methods for solving the scheduling problem involve two basic approaches:

- heuristics
- integer linear programming (ILP).

Priority-based heuristic scheduling (e.g. [18][30][84][87]) can accommodate a variety of control-dependent behaviors, but may fail to find an optimal solution in tightly constrained problems. The reason for this is that heuristic schedulers cannot recuperate from early suboptimal decisions which typically preserve only one representative from a possibly very large pool of qualified candidates.

Conventional ILP methods [49] can solve scheduling exactly but suffer from exponential time complexity and the inability to efficiently formulate control constraints. General applicability of these ILP methods has been improved by re-mapping the constraints [41][42], a mixed ILP/BDD method [127], and heuristic approaches based on ILP [48][59]. However, with the exception of [26] (discussed below), no ILP-based technique provides support for conditional behavior. Similarly, a recent branch-and-bound technique [116] based on execution interval analysis [115] has been applied only to acyclic DFGs.

Finally, we observe that the current scheduling techniques typically produce a *single* representative solution among those which are feasible within the constraints [2]. In subsequent HLS tasks, (e.g. binding and interconnection synthesis) additional constraints that conflict with a particular scheduling solution may arise and the scheduling must be redone to accommodate these new constraints. Using heuristic scheduling, additional constraints can be introduced to help avoid these conflicts (e.g. [25]). However, these additional constraints may adversely affect the heuristic scheduling quality and performance.

1.2.1 Control Dominated Circuits

Many HLS systems *prohibit* code motion in order to avoid problems related to evaluation of resource availability and causality of the solutions. An alternative strategy is to *explicitly* write constraints describing global movement of operations, but such approaches reduce to exhaustive enumeration of potential execution scenarios. In the formulation described in this paper, code motion is allowed *implicitly* -- there is no need to describe freedom already available (although implicit) in a CDFG.

As an example, we consider the formal approach based on algebra of control-flow expressions (CFEs) [26]. In that work, the timing and synchronization requirements for communicating machines are encapsulated in finite-state machine (FSM) description. From this, scheduling constraints are derived and subsequently solved using a BDD-based O/1 ILP solver. The FSM description is constructed from an algebraic CFE specification which implicitly restricts code motion.

2. However, it has been shown that certain HLS benchmark instances have literally billions of optimal solutions ([94], Chapter 6).

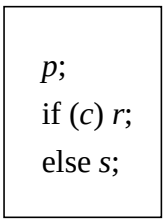


Figure 1.4 Conditional behavior

Consider, for example, the code segment shown in Figure 1.4. A possible CFE specification for this fragment is:

$$p(c{:}r + \bar{c}{:}s) \quad \text{(EQ 1.1)}$$

This requires that *p* be executed before *c*, and *c* before either *r* or *s*. An alternative specification is:

$$c{:}pr + \bar{c}{:}ps \quad \text{(EQ 1.2)}$$

which allows *c* to be executed before *p*. If *c* depends on *p* only the first statement is correct. However, if *c* and *p* are independent, then *both* behaviors (described by Equation (1.1) and Equation (1.2)) are legal. It is possible to create a *specification* which lists all correct execution scenarios, but the number of such scenarios and the size of the specification grow dramatically as the program complexity increases. In contrast, in our approach, only data dependencies are used to impose the execution order of *p* and *c*. In fact, if the data dependencies allow such motion, *r* and/or *s* may be executed before *c* and potentially before *p* as well. These potential execution scenarios are implicitly supported by the formulation.

Since operation level parallelism may not be explicit in the input description, some heuristic schedulers focus on detection of mutual exclusiveness in CDFGs. Tree scheduling (TS) [47] uses a tree-representation of the execution paths to enable movement of operations. In that approach, sub-trees induced by a branch are considered to be mutually exclusive and, consequently, can share resources.

Conditional vector list scheduling (CVLS) [119] uses *condition vectors* [120] to dynamically track mutual exclusiveness of the operations that can be executed in a speculative fashion (i.e. pre-executed). Transformation of a CDFG with conditional branches into one without conditional branches is performed in [55], but there is no support for speculative execution. Furthermore, these heuristics are restricted to nested conditional branches (*conditional tree* control structure). Multiple conditional trees are addressed by Wakabayashi [119], but the trees are either scheduled sequentially (using a priority scheme) or conditional tree duplication is performed.

Some synthesis systems emphasize treatment of behavioral level timing specifications. However, either a predefined order of operation is enforced before the scheduling [18] or the treatment of resource constraints is not fully considered [60]. The PUBSS system [124] forms a product machine of individual behavior FSMs (BFSMs) to statically schedule I/O communication between the components. PUBSS supports a variety of timing constraints. However, parallelism increasing techniques [36] are applied in a static fashion (before BFSM collapsing and scheduling). The issue of resource constraints is either not formally discussed [114][128] or the formulation of exclusivity constraints requires an excessive number of 0/1 ILP variables [113].

1.2.2 Symbolic Techniques

To our knowledge, the first attempt to address the scheduling problem using symbolic computations was made by Kam in [54]. There, several CAD applications of MDDs (multi-valued decision diagrams) were described. Scheduling of acyclic DFGs with function unit constraints was formulated using multi-valued variables, but the approach seemed to be practicable only for tightly

constrained problems. Unfortunately, too few experimental results were left documented to make a critical assessment of that approach.

In the mixed ILP/BDD approach [127], data dependencies were captured in an ROBDD (Reduced Ordered Binary Decision Diagram [12], Appendix A) form to simplify the ILP execution. Inclusion of resource constraints and all other steps towards the final solution were applications of standard ILP techniques. As in the case of [54], the question of control-dependent behavior was not addressed.

An exact symbolic formulation of the *control-dependent*, *resource-constrained* scheduling problem was introduced in [92] and represents a foundation for the work presented in this thesis. In that work, all scheduling constraints are formulated in a Boolean equation form. Unlike other approaches in which a single representative solution is generated, in [92] *all* feasible schedules are encapsulated in a compressed ROBDD representation. This is advantageous since the exact effect of additional constraints derived during subsequent synthesis steps is *incrementally* computable. Also, there is the additional benefit of being able to explore the solution space without the need to reschedule the problem instance. However, the formulation presented in [92] does not support code motion.

An alternative symbolic formulation [125][126] uses finite automata to capture resource/timing/synchronization constraints. A product automaton is built that satisfies the specified behavior. Its ROBDD representation is then traversed to find a minimum-latency schedule. However, similar to [26], the technique lacks support for various forms of a operation-level parallelism to be described in Chapter 2.

In this thesis, we describe a symbolic technique for exact resource-constrained scheduling of arbitrary forward-branching control structures. Scheduling is performed with the assumption that allocation of resources is known. The technique

supports speculative operation execution and global treatment of parallel control structures. To allow a systematic treatment of the problem, a flexible control representation based on *guard variables*, *guard functions*, and *traces* is introduced. A novel *trace validation* algorithm is proposed to enforce causality and completeness of the set of all feasible solutions.

The scheduling technique presented in this thesis supports arbitrary Boolean constraints as well as conventional timing constraints. Scheduling of multi-rate interacting FSMs is not addressed in this work. Similarly, we do not discuss optimizations based on algebraic and retiming transformations [1][64][68][81][90] nor do we discuss scheduling of multi-dimensional applications [85].

1.2.3 Relation to Research in Compilers

Steady advances in VLSI manufacturing technology have made it possible (and economically justifiable) to implement superscalar, superpipelined and VLIW architectures [45][52]. This has had a large impact on research in compilation techniques for instruction-level parallel processing [100]. To find substantial amounts of instruction-level parallelism, it has been demonstrated in numerous experimental studies that optimizing compilers have to be able to schedule code beyond the basic code block boundaries [105]. For the purpose of this discussion, we adopt Fisher's definition of a basic block [37].

Definition 1.3 A basic block is a sequence of instructions having no jumps into the code except at the first instruction and no jumps out of the code except at the end.

Very generally, compilers can be classified based on their ability to perform "linear" or "non-linear" code motions [38]. Typical representatives of the former group are compilers based on "*trace scheduling*" ([35][37][70]) and their

“superblock” derivatives ([19][20][50]). A trace is a loop-free linear fragment of code that might include several basic blocks. A profiling information or the programmer’s directives are used to assign probabilities to outcomes of conditional branches. Based on that information, traces are formed and scheduled sequentially using a priority based on the likelihood of their execution. Since traces span a number of basic blocks, global code motions are possible. However, these code motions are essentially limited to a particular trace being scheduled (thus they are referred to as “linear”). When lower-priority traces are scheduled, this restriction leads to very limited (and unlikely to be very useful) code motions that can potentially “fill the holes” in the machine code already generated for the higher-priority traces. Moreover, to preserve a correct program behavior, trace scheduling may require very complex book-keeping and introduction of additional code blocks in the traces that are yet to be scheduled [3]. Since traces cannot cross back-edges of the loop, loop optimizations are done by means of an aggressive unrolling. To simplify the book-keeping, superblock scheduling introduces a further restriction of a single entry per trace (superblock). Tail-duplication is used to provide the compiler with sufficiently large portions of code.

More global compilation techniques allow “non-linear” code motions. For example, operations from both “then” and “else” branches of an “if-then-else” statement can be simultaneously considered for a speculative execution. These global motions are performed on clusters of code blocks [38] or whole programs [3][80].

Ability to perform global code motions is very useful for software pipelining techniques [3][9][21][32][33][62][80][101][112][118][123]. To maximize throughput, such techniques schedule a number of loop iterations to execute in an

3. For an in-depth analysis of numerous implementation challenges, see Chapter 4 of [35].

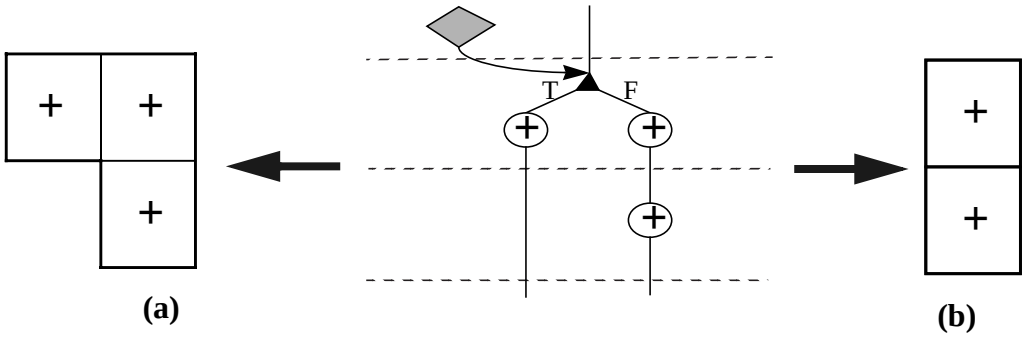


Figure 1.5 Resource management examples

overlapped fashion. Typically, however, these techniques do not perform exact conditional resource sharing analysis.

Shown in Figure 1.5 is a schedule for a CDFG fragment having a simple *if-then-else* structure. *Modulo scheduling* [32][101] converts control dependencies into data dependencies using the *IF-conversion* [5]. Such approach essentially flattens a CDFG and leads to overestimation of resource requirements in a *sum-of-resources* fashion. This is indicated in Figure 1.5(a), where two adders are allocated for execution of mutually exclusive operations at the second step. In *hierarchical reduction* [62], "then" and "else" branches are individually scheduled and encapsulated into a larger node with composite resource usage indicated in Figure 1.5(b). Resource usage is evaluated in a *union-of-resources* fashion. It has been reported that such an approach tends to create nodes with complex and irregular resource usage patterns imposing severe restriction on scheduling of the remaining nodes [123]. Moreover, resource evaluation is still not exact. Observe that in Figure 1.5(b) one adder is allocated at the third time step regardless of the path taken. This leaves a "hole" (NO-OP) that could possible be used to schedule other nodes for parallel execution (e.g. operations belonging to different iterations of a software pipelined loop).

All of the compilation techniques referenced in this section are invariably heuristic and allow very limited (if any) backtracking during scheduling. However, it is unfair to use such an argument as a disqualifying flaw -- in reality, compilers have to deal with programs consisting of thousands of lines of a source code! As a consequence, a premise on impracticality of exact compilation techniques is unlikely to be challenged any time soon. On the other hand, many user applications from a hardware synthesis domain are of a relatively moderate size but have to deal with the underlying hardware intricacies and/or to guarantee that hard real-time throughput constraints are unconditionally met.

Moreover, to our knowledge, all of the competitive compiler implementations do impose significant restrictions on a repertoire of global code motions. Under such circumstances, a strong argument can be made that practical compilation techniques could see benefit from exact techniques capable of handling program fragments under scheduling consideration. In some general-purpose hardware implementations it is undesirable (or even prohibited) to enable speculative transactions potentially resulting in false arithmetic exceptions and memory faults [20][100]. This makes exact techniques even more attractive because of their potential to maximally expose and extract any residual instruction-level parallelism.

1.3 Overview of the Thesis

So far, in Section 1.1 and Section 1.2, the scheduling problem was introduced and related research surveyed. It was our intention not only to provide a necessary background for the reader, but to clearly state the motivation for pursuing a particular research avenue. The rest of the thesis is organized as follows:

- In Chapter 2, we describe several approaches to resource-constrained control-dependent scheduling, as well as a number of features desirable to improve scheduling quality. In particular, we focus on speculative operation execution and treatment of parallel/correlated control structures.

- The formulation is presented in Chapter 3. First, a flexible control representation based on *guard variables*, *guard functions*, and *traces* is described. Next, a speculative execution model is introduced and discussed. A Boolean equation formulation of scheduling constraints follows. Then, a *trace validation* algorithm is proposed to enforce causality and completeness of the set of all feasible solutions. Finally, we discuss extensions to cyclic control and clarify the differences between our formulation and related ILP formulations.

- Aspects related to the ROBDD construction process are considered in Chapter 4. These include: a discussion of the iterative solution construction, ROBDD variable ordering strategies, and techniques employed to improve the run-time efficiency.

- Chapter 5 presents an alternative approach to conditional resource sharing analysis. The approach is not explicitly used in the techniques described in the rest of the thesis. However, it is transparent to a particular scheduling implementation and has relevance to software pipelining techniques. The reader may postpone reading Chapter 5 and the corresponding experimental results (Section 6.5) and treat them as an extra Appendix.

- Experimental results are presented and discussed in Chapter 6.

- Finally, in Chapter 7, conclusions are presented, as well as the questions to be addressed in future.

- Although we assume that the reader has a basic understanding of Binary Decision Diagrams, some necessary background is provided in Appendix A. Most of the results presented in this thesis are derived using reduced ordered binary decision diagrams (ROBDDs [12]). Abbreviations ROBDD, OBDD and BDD will be used interchangeably whenever the correct meaning can be implied from the context of the presentation. When a clear distinction has to be made, more specific abbreviations will be used: for example, 0-sup BDDs or ZBDDs (for Zero-suppressed Binary Decision Diagrams [75]) and MDDs (for Multi-valued Decision Diagrams [54]).

Chapter 2

Control-Dependent Behavior

2.1 High-Performance Scheduling Issues

Our scheduling technique assumes an input in the form of a CDFG specification. The CDFG describes both data-flow and control dependencies between the operations and is similar to the one used by Wakabayashi [119]. Figure 2.1 contains an pseudocode example and its CDFG representation. Operation nodes are atomic actions potentially requiring use of hardware resources (e.g. arithmetic/logical operations, read/write cycles). Conditional behavior is specified by means of fork and join nodes. An operation node generating a control signal for a fork/join pair is called a *conditional*. Directed arcs establish a link between the conditional and a related fork/join pair. In Figure 2.1, the conditional labeled op_2 tests the result of the addition (op_1) and determines the flow of control (i.e. whether “True” (T) or “False” (F) branches should provide operands for op_6).

Figure 2.1 (a), (b), and (c) show three different ways to schedule the example assuming that only one resource of each type is available. The schedule in Figure 2.1(a) uses the knowledge that after a conditional (op_2) is executed, operations

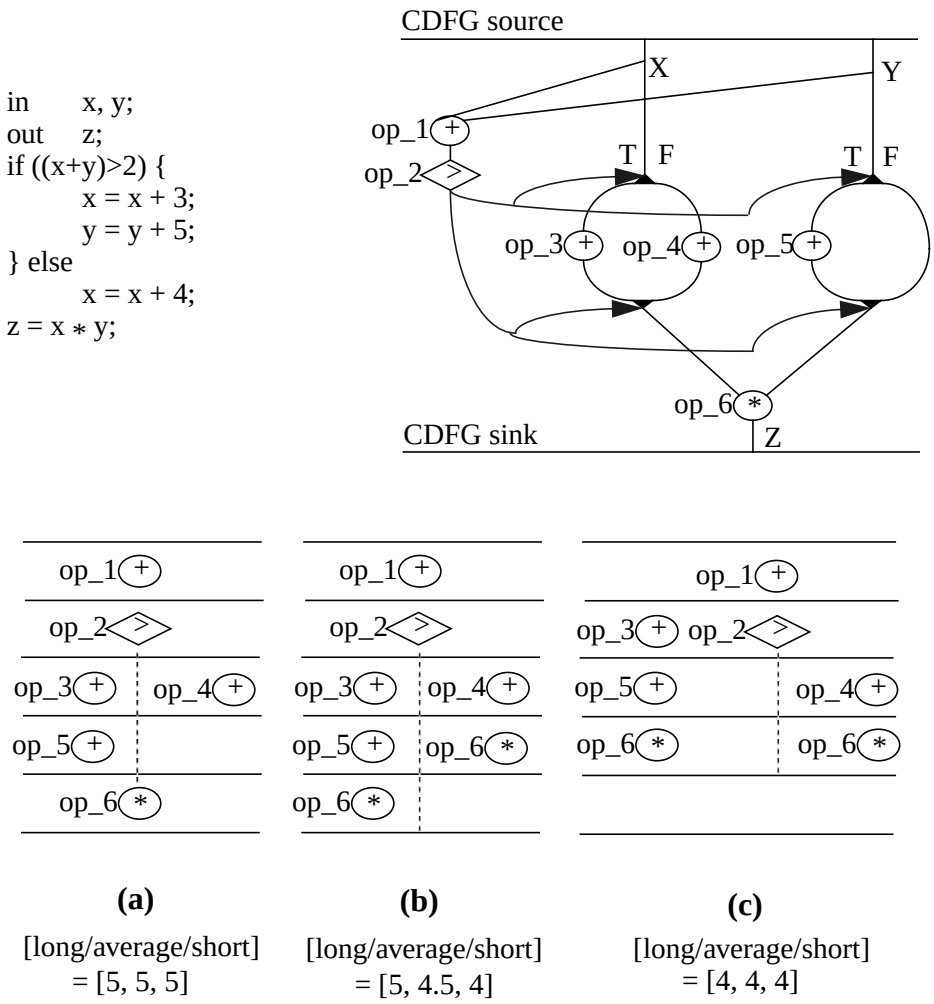


Figure 2.1 Example CDFG and its schedules

belonging to “T” and “F” branch arcs are mutually exclusive. However, the join node is treated as a synchronization point: op_6 cannot be scheduled until both the “T” and “F” branch are executed. This leads to inefficient schedules, since the execution times for alternative branch arcs may differ widely. Consequently, in this example, it takes 5 cycles to execute the schedule no matter what decision is made by the conditional. This approach corresponds to that used by traditional ILP schedulers (e.g. [49]).

The schedule shown in Figure 2.1(b) improves the “average” execution time to 4.5 cycles by scheduling op_6 on the fourth cycle at the “F” branch. Note that the operation execution order is predetermined before scheduling (e.g. op_2 before op_3, although no data dependency exists between these two operations in the CDFG). This approach is supported by a number of heuristic schedulers (e.g. [18]) one recent exact technique [26].

The schedule from Figure 2.1(c) not only further improves the average execution time, but reduces the longest execution path to 4 cycles as well. This is done by scheduling op_3 on the second cycle in a speculative fashion (i.e. before the corresponding conditional, op_2, is resolved). Note that the resource requirements cannot be predicted in a static fashion. For example, if more adders are available, op_4 can be executed in a speculative fashion as well. The mutual exclusion of op_3 and op_4 must be evaluated dynamically by taking into account when the corresponding conditional (op_2) is scheduled. This kind of scheduling is supported by several heuristics ([47][89][104][119]).

Several ways to improve the scheduling quality by exposing and exploiting operation-level parallelism implicit in the CDFG representation are discussed in Sections 2.1.1 through 2.1.4.

2.1.1 Speculative Operation Execution

It is often beneficial to determine the control value simultaneously with branch execution. Operations from branch arcs that are executed before the corresponding conditional value is evaluated are said to be *pre-executed*. Such speculative operation execution allows more flexibility in using given hardware resources. A *conditional* is a scheduled operation that generates a control value. Figure 2.2 shows a CDFG where the control dependencies between the conditionals

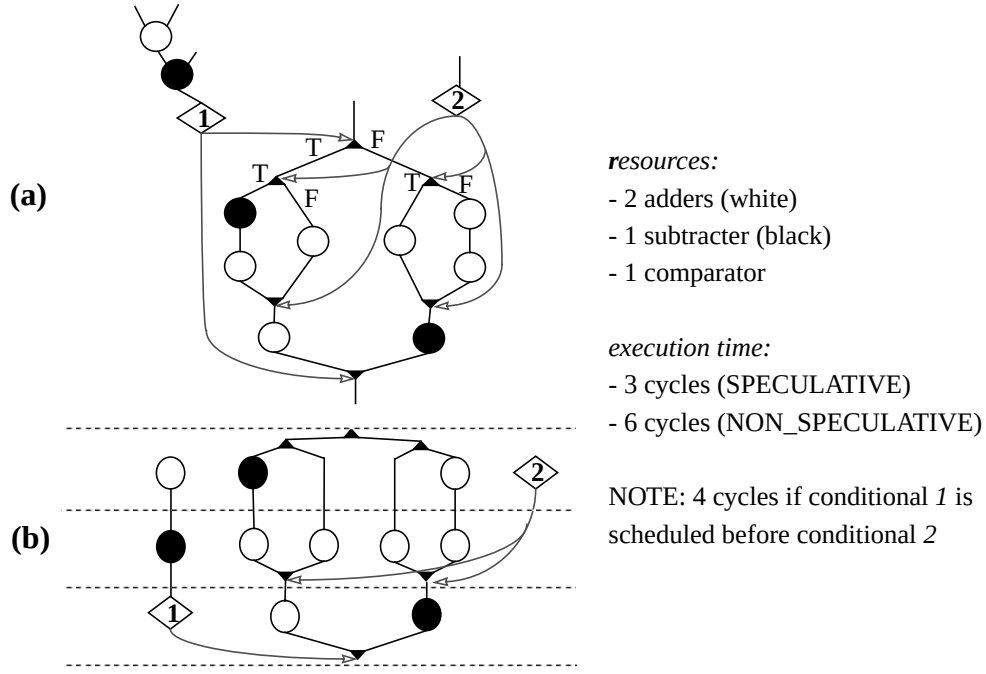


Figure 2.2 Example of speculative operation execution

(comparators *1* and *2*) and the corresponding fork/join pairs are explicitly indicated. Speculative operation execution is not possible if the control precedence between the conditional and the fork node is enforced. In this case, at least six time steps are necessary to execute the CDFG, since the longest dependency chain includes six operations. However, if precedence between the conditional and the fork node is removed, operations from the branch arcs can be pre-executed. Figure 2.2(b) shows a schedule executing in three cycles using the indicated resources. In general, precedence between a conditional and join node need not be enforced either. In this case, the execution time is bounded only by data dependencies (given sufficient resources).

2.1.2 Out-of-Order Execution of Conditionals

It can happen that a faster schedule is obtained if the top-level conditional (in the input specification) is evaluated *after* some other nested conditional. A simple example of this behavior is shown in Figure 2.2(b). The schedule executes in three cycles with the conditional *1* left unresolved until the end of the very last cycle. The knowledge that conditional *2* is resolved during the first cycle is essential to properly interpret resource usage. Since conditional *2* is resolved during the first time step, there are only two distinct execution scenarios for the second time step (corresponding to still unresolved value of conditional *1*). Thus, at the second time step, only two adders are needed.

There is still a considerable discussion on how beneficial conditional re-ordering option is in general-purpose programs, practical compilers and commercial microprocessor architectures ([35][52][71][80][105][111][123]). We do not hope to provide a definitive answer. It should be noted, however, that dynamic re-ordering of conditionals is inherent in a guard representation and comes “for free” in our formulation (Chapter 3). Both *TS* [47] and *CVLS* [119] rely on a conditional-tree representation of the control and cannot accommodate out-of-order execution of the conditionals without dynamically modifying the tree structure.

2.1.3 Irredundant Operation Scheduling

Another way to improve scheduling quality is to identify operations that are not redundant in the input description, but are redundant for certain control paths. The importance of such information has been observed and the algorithms to detect such operations have been discussed in the literature [47][120]. Shown in Figure 2.3 is an example where *op_1* is redundant on “F” path and *op_2* is

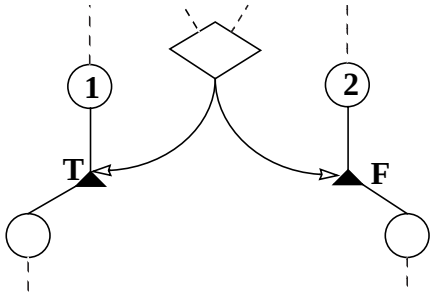


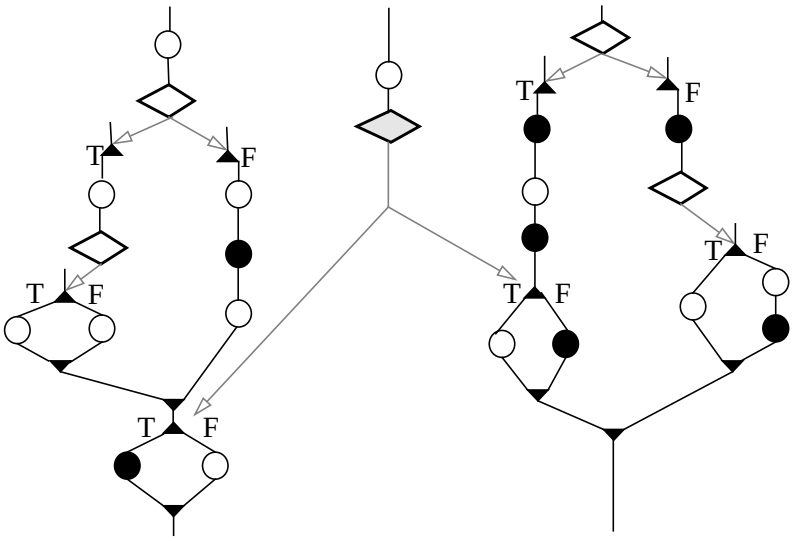
Figure 2.3 Operations redundant on certain control paths

redundant on “T” path -- this knowledge can be used to reduce resource requirements during scheduling.

2.1.4 Parallel and Correlated Control Structures

Control structures that are either fully parallel or have correlated control introduce additional scheduling challenges. As the number of control paths increases, it becomes difficult to keep track of the mutual exclusiveness among the operations. Ideally, the scheduler should evaluate and maintain this information for all control paths. In Figure 2.4, a CDFG is shown in which two parallel trees have a correlated control (shaded comparator). The reader can verify that, given one adder (“white” operation), one subtracter (“black” operation) and one comparator (single-cycle units assumed), a 6-cycle schedule can be found only if the control correlation is properly interpreted (i.e. “false paths” are not scheduled). As indicated in Figure 2.4, speculative execution (and additional or more versatile resources) can further improve the execution time.

Although not typical for conventional structured programs, parallel control structures are likely to result from program transformations performed by parallelizing compilers (e.g. loop unrolling where a conditional behavior is present within the loop body [100]).



no speculative execution:
- 6 cycles (3ALU or 1add/1sub/1comp)

speculative execution:
- 5 cycles (3ALU or 2add/1sub/1comp)
- 4 cycles (5ALU or 3add/2sub/2comp)

Figure 2.4 CDFG with correlated control

2.2 Our Goals

The formulation presented in this thesis supports all of the advanced scheduling features discussed above. In fact, the approach described in the thesis is the only exact technique for resource-constrained scheduling with speculative operation execution.

Definition 2.1 *Minimum latency* of the schedule is the minimum execution delay of the longest path of a scheduled CDFG.

Our goal is to find *all* minimum-latency schedules, given a CDFG specification and resource constraints. By using BDDs we can implicitly (symbolically) capture all feasible solutions to a particular problem instance. This solution format

introduces significant flexibility to a circuit design process by enabling incremental incorporation of additional constraints and by supporting solution space exploration and incremental engineering change without the need for rescheduling. This is potentially very important for practical CAD systems, since some of the relevant issues cannot be always predicted accurately during the early stages of a design process.

Chapter 3

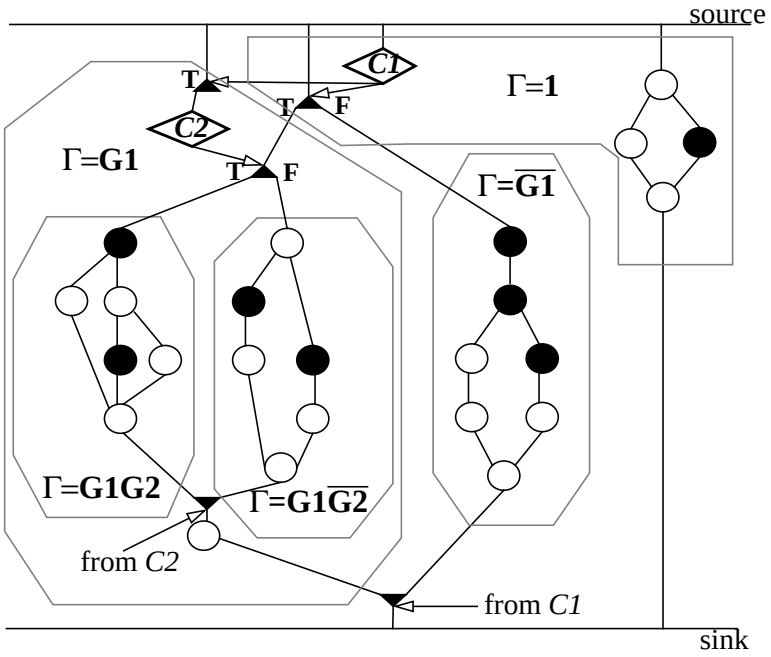
Formulation

In this Chapter, a Boolean formulation of scheduling problem is developed. It consist of four major parts:

- control representation based on *guard variables*, *guard functions*, and *traces*,
- speculative execution model,
- Boolean equation formulation of scheduling constraints, and
- a new *trace validation* algorithm that enforces causality and completeness of the set of all feasible solutions.

3.1 Control Representation

In this formulation, all scheduling constraints are represented as Boolean functions and an OBDD corresponding to the intersection is built. Each variable C_{sj} describes operation j occurring at time step s. C_{sj} is true iff operation j is scheduled at time step s in a particular solution. We assume a unique mapping from operation type to function unit type. To represent control-dependent behavior, a set of *guard variables* is introduced. Each guard G represents a control-flow



Guards: G1 (corresponding to C1 decisions)
G2 (corresponding to C2 decisions)

Figure 3.1 Kim's example

decision by a particular conditional -- the guard is true for one branch and false for the other. Every control path through an arbitrary combination of fork/join pairs is described by a product of the corresponding guard variables. For each operation *j*, a Boolean *guard function* Γ_j (defined on the guard variables) encodes all the control paths on which *j must* be scheduled.

Shown in Figure 3.1 is a CDFG fragment of Kim's example [55] in which two guards (G_1, G_2) encode the conditional behavior. There are three possible execution paths: $(G_1G_2, G_1\overline{G_2}, \overline{G_1})$. Indicated blocks $(1, G_1, G_1G_2, G_1\overline{G_2}, \overline{G_1})$ correspond to operations that share the same guard function Γ. Operations which must be scheduled on all control paths have Γ=1.

```
if ( C1 ) a;
else if ( C2 ) b;
     else goto d;
c; d;
```

Figure 3.2 Pseudo-code fragment

Computation of Γ functions -- Assume that operation *i* has *n* successors (j_1, j_2, ... , j_n) and that none of the successors is a join node. Then a guard function Γ_i can be simply computed as a Boolean *Or* of the successors' guard functions Γ_{j_k} (k=1,2, ... , *n)*. This means that operation *i* has to provide operands to all of its successors. If a successor of *i* is a join node, then its contribution to Γ_i is equal to $\Gamma_{join}G_k$ or $\Gamma_{join}\overline{G}_k$ (depending whether *i* belongs to the "T" or "F" branch). Guard functions corresponding to all of the nodes can be computed by a one-pass traversal of the CDFG that starts from a sink node whose guard function is initialized to '1' (tautology).

We observe that Γ's are not restricted to product terms (thus, they can handle constructs such as: *goto*, *exit*, *case*). In the pseudo-code fragment shown in Figure 3.2, the execution condition for statement *c* is described as: $\Gamma_c = G_{C_1} + \overline{G_{C_1}}G_{C_2} = G_{C_1} + G_{C_2}$. Guard-based representation also applies to parallel or correlated control structures. If two copies of Figure 3.1 are executed in parallel, only two more guard variables are introduced, while the number of control combinations (nine) grows much faster. The number of guards is not proportional to the number of control paths, but is determined by the number of conditionals. For example, in Figure 2.4 (Section 2.1.4), only five guard variables encode 18 possible control path.

In many aspects, the guard-based model is similar to execution conditions from *path analysis* [8]. In that approach, however, Boolean conditions are used in the hardware allocation phase (after AFAP scheduling is performed). Nevertheless, that research demonstrated that BDDs efficiently represent control signals in large scale problems. Similarly, Boolean functions are used to label conditionals in several other recent techniques for control-dependent scheduling [26][126].

In fact, guard representation was used in areas other than HLS -- for example, to perform "IF-conversion" in experimental vectorizing compilers [5] and simplify code generation for VLIW and superscalar machines supporting predicated execution [32][71][88][102].

The technique presented in this thesis generates a solution in the form of a collection of *traces*. A *trace* is a possible execution instance for a particular control path. In BDD form, traces correspond to product terms of the Boolean function. Each trace includes the guard variables (identifying a control path) and operation variables (indicating a schedule for the path). For example, in Figure 3.1, each trace corresponding to the "False" branch of conditional C_1 contains $\overline{G_1}$, as well as 0/1 assignment of C_{sj} variables. Operations with $\Gamma=\overline{G_1}$ or $\Gamma=1$ must be scheduled on that trace. If other operations are scheduled on this trace, they are pre-executed.

The *ensemble schedule* is a set of traces forming a complete deterministic schedule. Conditions for the existence of such a schedule are discussed in Section 3.4. The solution BDD includes only traces belonging to at least one ensemble schedule and implicitly incorporates all feasible ensemble schedules. Note that the number of ensemble schedules can be much larger than the number of traces.

3.2 Speculative Execution Model

In our speculative execution model, only the control precedence between the conditional and join node is enforced. CDFG operations can be scheduled at different time steps on distinct control paths, but cannot be scheduled more than once per trace. Each operation from the CDFG is executed at most once regardless of the actual control decisions made when the schedule is executed. For example, this means that in the current model the following scenario is prohibited: (*i*) operation *j* executes in a speculative fashion using operands *A* and *B* and generates result *R*, (*ii*) a control decision is made and *R* is discarded, (*iii*) operation *j* executes using a different set of input operands (e.g. *C* and *D*) and a correct value of *R* is re-computed.

Figure 2.2 (Section 2.1.1) shows an example where precedences between the conditionals and forks are removed. The critical path length of 6 in the original CDFG is reduced to just 3. All four possible control paths may start executing simultaneously.

Application of the proposed speculative execution model to the *Maha* benchmark [86] is shown in Figure 3.3 (directed arcs represent control dependencies and undirected lines correspond to data dependencies). Notice that a great deal of freedom is added to the schedule: e.g. operations A8 and S8 can be executed during an arbitrary time step subject only to resource constraints. Given sufficient resources, a critical path length of 8 in the original graph can be reduced to just 4 (operations: S6, A6, S7, A7). The current formulation does not allow for operations following the join to be executed in a speculative fashion before the corresponding conditional is resolved (e.g. S7 cannot be scheduled in the second cycle due to a dependency from A2). Notice, however, that there is still a lot of

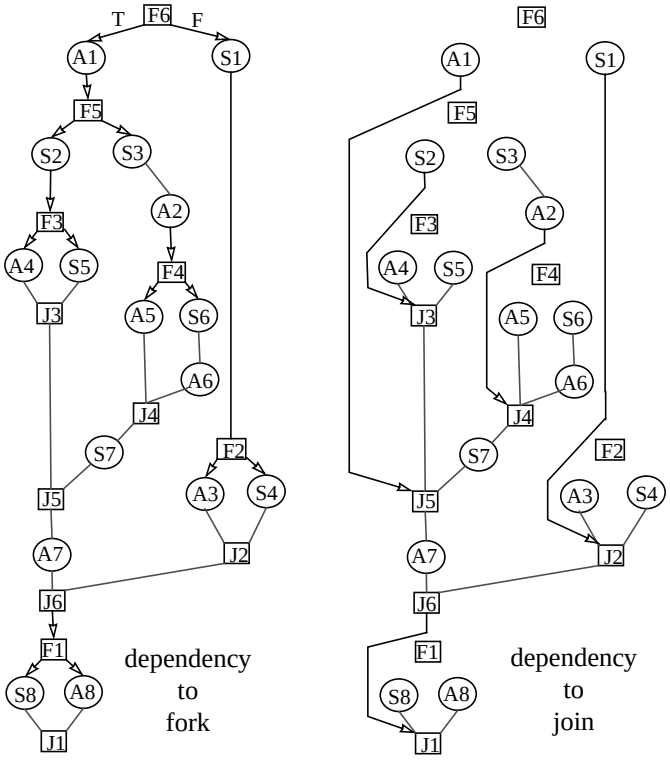


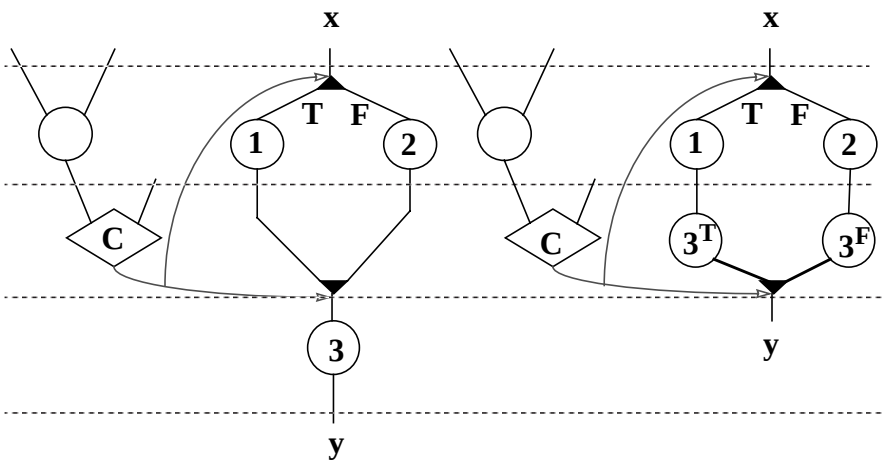
Figure 3.3 CDFG transformation for Maha

freedom to exploit instruction level parallelism: since there are no dependencies left among the conditionals, all 12 control paths can start executing *simultaneously.*

3.2.1 Restrictions of the Proposed Model

The technique we develop in this thesis is exact. Misguidingly, exact methods are frequently referred to as “optimal” as well. However, the exactness implies optimality only up to the extent of generality of the underlying model used in a particular exact method. Thus, it is important to clearly state an answer to the following question:

“Is the speculative execution model described in Section 3.2 capable of generating time optimal schedules?” [1]



(a) Current model (b) Generalized model

Figure 3.4 Speculative execution model

Since the proposed model does not provide a full repertoire of code motions, the answer to this question is obviously: “No”. Figure 3.4(a) illustrates such a case in which our current model cannot achieve time optimality. Although both operations *1* and *2* are scheduled at the first step, execution of operation *3* has to be delayed until the third step (since conditional *C* is resolved at the second step). If the precedence between *C* and operation *3* is not enforced (Figure 3.4(b)), two instances of operation *3* (3^T and 3^F) can be scheduled at the second step, reducing the execution time. In fact, both paths execute in two cycles as dictated by the length of the corresponding chains of data-dependent operations. Thus, the schedule in Figure 3.4(b) is time optimal.

To summarize, we do not allow code motions that can lead to a creation and execution of multiple instances of the same CDFG operation on a control path (in the rest of this discussion we will informally refer to this as “operation renaming”).

1. Time optimality is defined in Section 1.1.1.

Operation renaming may lead to an exponential explosion of operation's instances on a control path and poses a difficult implementation problem even in heuristic schedulers. Since our goal is to develop the exact technique, operation renaming would likely reduce its practicability. Exact amount of instruction level parallelism that is lost due to the restrictions of our speculative execution model is, in general, impossible to be determined. However, the results presented in Section 6.3 indicate that our approach generates schedules that exhibit significant advantage over scheduling without speculative execution. Furthermore, our approach generates comparable or superior results when compared to the best known solutions for a number of HLS benchmarks.

3.3 Derivation of Constraints

For brevity, we assume non-pipelined, unit-time operations. Pipelined and multicycle functional units can be accommodated by incorporating execution delay in the equations presented in Sections 3.3 and 3.4 [92]. Contrary to some other approaches where such operations have to be modeled as chains of single cycle operations (e.g. [3][26][125]), only one variable per multicycle/pipelined operation is used in our formulation. To model operation chaining, a precedence relation can be added between operations that cannot be chained [49].

$(ASAP)_j$ (as-soon-as-possible) and $(ALAP)_j$ (as-late-as-possible) bounds are constructed to limit the time spans over which an operation j can be scheduled. These bounds are not required for correctness, but improve the efficiency of the construction. C_{sj} denotes operation j's instance at time step s. Fork (join) nodes are not explicitly used in the formulation. Precedences to fork (join) nodes are translated in a transitive fashion to the successor nodes of the fork (join). Symbols "Σ" and "+" correspond to Boolean *Or* function, and "Π" stands for Boolean *And*. Product "*ab*" implies "*a And b*".

3.3.1 Uniqueness

Equations (3.1) and (3.2) enforce unique scheduling of operations from the CDFG at time step *s*. If $(ASAP)_j \le s < (ALAP)_j$:

$$\sum_{k \in R_{sj}} \left(C_{kj} \prod_{i \neq k \in R_{sj}} \overline{C_{ij}} \right) + \prod_{i \in R_{sj}} \overline{C_{ij}} = 1 \qquad \text{(EQ 3.1)}$$

where R_{sj} is the range $[(ASAP)_j \ldots s]$.

If time step $s = (ALAP)_j$:

$$\sum_{k \in R_{sj}} \left(C_{kj} \prod_{i \neq k \in R_{sj}} \overline{C_{ij}} \right) + \left(\prod_{i \in R_{sj}} \overline{C_{ij}} \right) \overline{\Gamma_j} = 1 \qquad \text{(EQ 3.2)}$$

Equation (3.1) states that prior to step $(ALAP)_j$, operation *j* is not scheduled more than once. On step $(ALAP)_j$, Equation (3.2) ensures that operation *j* has been executed on all paths covered by Γ_j. On paths not covered by Γ_j, operation *j* can be either uniquely scheduled (pre-executed) or not scheduled at all.

The constraint formulated in Equation (3.1) can be simplified. An iterative form of the Equation (3.1) that enforces uniqueness implicitly (by construction) is formulated in the following equation:

$$\overline{C_{sj}} + \left(\prod_{i \in R_{(s-1)j}} \overline{C_{ij}} \right) = 1 \qquad \text{(EQ 3.3)}$$

where $R_{(s-1)j}$ is the range $[(ASAP)_j \ldots (s\text{-}1)]$.

3.3.2 Precedences

If operation *i* precedes operation *j* (i.e. there is a dependency arc from *i* to *j* in the CDFG) and $\Gamma_i \supseteq \Gamma_j$ (Γ_i covers Γ_j) then for every step *s* in the range $[(ASAP)_j \ldots (ALAP)_i]$ the following must hold:

$$\left(\overline{C_{sj}} + \sum_{ASAP_i \le l < s} C_{li}\right) = 1 \qquad \text{(EQ 3.4)}$$

Equation (3.4) states that either operation *i* has to be scheduled before step *s*, or operation *j* cannot be scheduled at step *s*. The case "Γ_i covers (but is not equal to) Γ_j" ($\Gamma_i \supset \Gamma_j$) occurs when the dependency from *i* to *j* goes through a fork node. When $\Gamma_i \not\supseteq \Gamma_j$ (Γ_j not contained in Γ_i -- e.g. the dependency from *i* to *j* goes through a join node), the precedence relation is enforced only on the paths covered by Γ_i:

$$\left(\overline{C_{sj}} + \sum_{ASAP_i \le l < s} C_{li}\right) + \overline{\Gamma}_i = 1 \qquad \text{(EQ 3.5)}$$

Effectively, Equation (3.4) ensures that the operation can be pre-executed only if all of its predecessors have already been executed. In our model, an operation after the join node cannot be pre-executed before the corresponding conditional is resolved. Thus, according to Equation (3.5), the dependencies to its predecessors are enforced only conditionally.

Figure 3.5illustrates the meaning of Equations (3.4) and (3.5). As indicated by Equations (3.4), operation *2* cannot be scheduled unless operation *1* is scheduled at some earlier time step, since operation *1* provides an operand to operation 2. On the other hand (Equations (3.5)), precedence between operation *3* and operation *4* has to be enforced only on "True" path -- if a decision has been already been made so that "False" path is executed, operation *4* cannot be allowed to wait for an operand coming from operation *3*.

Equation (3.5) requires a special attention. Since operation *i* can be scheduled on control paths not covered by its corresponding guard function Γ_i, it seems that a

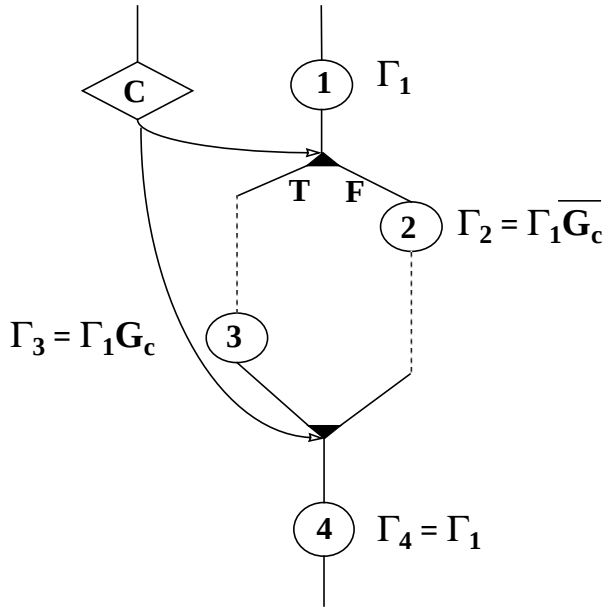


Figure 3.5 Treatment of precedences

solution set may include some traces where a precedence between operations *i* and *j* is not enforced. However, notice that execution of *i* outside of Γ_i corresponds to *i*'s speculative execution. This means that, at a particular time step, decisions by a certain subset of conditionals are yet to be made and that traces for all possible control paths to be distinguished at some later step have to *match* (*be identical*) prior to the moment such decisions are made. Since one of such traces inevitably has to be covered by Γ_i and, consequently, have all of the precedences properly enforced, a trace not covered by Γ_i will preserve all of the precedences as well. The process enforcing trace matching will be described in Section 3.4 (*Trace Validation*).

3.3.3 Termination

A single *sink* variable is used in the OBDD representation to indicate that a particular trace has concluded. It is initialized to '0', and is set to '1' when the terminating condition for the trace is met. Equation (3.6) is used as a terminating

condition for all traces in parallel. The scheduling process can be terminated when *sink* assumes the value '1' on all paths of an ensemble schedule. In these equations, operations ($j_1 ... j_n$) are immediate predecessors of the sink node in the CDFG.

$$\prod_{l=1}^{n} (R_{sj_l} + \overline{\Gamma_{j_l}}) = 1 \text{ , where } R_{sj_l} = \sum_{k=(ASAP)_j}^{s} c_{kj_l} \quad \text{(EQ 3.6)}$$

Function R_{sj_l} is true if operation j_l is scheduled prior to or at step s. The fact that execution of j_l is mandatory only on paths covered by Γ_{j_l} is reflected by Equation (3.6).

3.3.4 Resource Constraints

If k_l resources of a certain type r_l (e.g. multipliers, adders, ALUs, registers, buses) are available, we formulate a "*generalized resource bound*" Equation (3.7):

$$\sum_{1 \le (l_p \ne l_q) \le n_{sl}} \overline{F_{sl_1}} \overline{F_{sl_2}} \ldots \overline{F_{sl_{(n_{sl}-k_l)}}} = 1 \quad \text{(EQ 3.7)}$$

F_{sl} is a Boolean function stating that resource r_l is needed during time step s. Equation (3.7) is applied at each step s for each resource r_l. It ensures that at least (n_{sl}-k_l) resources (among n_{sl} potential candidates at step s) will not be scheduled. For functional units, F_{sl} functions are simply the operation variables. For example, if at step s operation instances C_{sm_1}, C_{sm_2}, C_{sm_3} and C_{sm_4} are candidate multiplications and there are only $k_m = 2$ multipliers available, Equation (3.7) becomes:

$$\overline{C_{sm_1}}\overline{C_{sm_2}} + \overline{C_{sm_1}}\overline{C_{sm_3}} + \overline{C_{sm_1}}\overline{C_{sm_4}} + \overline{C_{sm_2}}\overline{C_{sm_3}} + \overline{C_{sm_2}}\overline{C_{sm_4}} + \overline{C_{sm_3}}\overline{C_{sm_4}} = 1 \quad \text{(EQ 3.8)}$$

Equation (3.7) applies the resource constraint to all traces simultaneously. *Trace validation* (Section 3.4) ensures that there are no resource violations in any ensemble schedule.

Next we discuss how bus and register constraints are generated for acyclic DFGs by a suitable choice of F_{sl}.

A bound on the number of available registers can be implemented using a slightly more complex resource function and then simply plugging this new function into the constructor for Equation (3.7). Equation (3.9) indicates that if an operation *i* precedes the operations (j_1...j_n) at a particular control step *s*, a register is required. This register is required to keep the value of the output of operation i if the successor operations cannot use it immediately.

$$(i \rightarrow (j_1 \ldots j_n)) \Rightarrow F^r_{si} = \left(A_{si} \sum_{j = j_1}^{j_n} \overline{B_{sj}} \right) \quad \text{(EQ 3.9)}$$

where

$$A_{si} = \sum_{l = ASAP_i}^{s} a_{li} \quad \text{(EQ 3.10)}$$

$$B_{sj} = \sum_{l = ASAP_j}^{s} b_{lj} \quad \text{(EQ 3.11)}$$

Notice that this formulation allows for possible chaining of operations since the constraint predicts that no register is required if the operations are all assigned to the same control step. Each Equation (3.9) constraint can be plugged into the typed resource constraint equation Equation (3.7). Note that in this case k_r is the number of registers allowed and n_{sr} is set to the number of candidate Equation (3.9) functions at the s^{th} control step [2]. The construction of the register requirement

2. This approach verifies that the number of variables that are live at a particular time step does not exceed a pre-specified bound. This is compatible with the classical register allocation algorithm based on graph coloring [17].

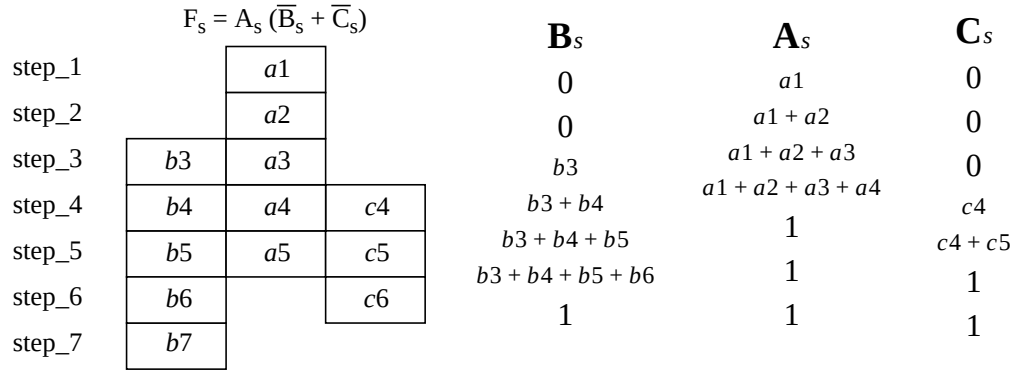


Figure 3.6 Example register constraint

for an example operation with 2 successors is shown in the Figure 3.6. F_s is true if a register is required at time step *s*.

Busses can be treated in a similar fashion. If operation *i* precedes operations ($j_1 ... j_n$), Equation (3.12) indicates that at a particular control step *s* a bus may be needed to read an operand (upper part of Equation (3.12)) or write a result (lower part of Equation (3.12)). Notice that the formulation allows a rather complicated situation (same operand used as an input to a number of operations) to be modeled in a simple fashion.

$$i \rightarrow (j_1 \ldots j_n) \Rightarrow F_{si}^{br} = \sum_{l=1}^{n} C_{sj_l}$$
$$i \rightarrow (j_1 \ldots j_n) \Rightarrow F_{si}^{bw} = C_{si} \qquad \text{(EQ 3.12)}$$

Given the Equation (3.12) constraints, we can again treat them as generic resources and plug them into Equation (3.7) for each time step, since only k_b out of the n_{sb} functions can be active at each particular phase of the time step *s*. The bus constraints apply to for *"read"* and *"write"* phases separately, making no assumption that a number of writes is smaller than the number of read operations at each control step. However, we do assume that read and write transfer phases are

interleaved. Similar constructions can be used to constrain the number of other typed resources. It is important to note that this formulation of these constraints does not require the addition of more implementation variables (as is the case for ILP formulations of Bus constraints [49]). Implicit constraint application allows these resource constraints to be efficiently constructed even for very complex constraints functions (to be discussed in Section 4.4.2).

3.3.5 Removal of Redundantly Scheduled Operations

Assume that a conditional has executed and the "True" branch is selected. Operations from the "False" branch may still be scheduled on the trace corresponding to the "True" branch if there are available resources. Such traces are identified and removed. Assume conditional c_k (whose corresponding guard is G_k) is resolved prior to time step s. Then all the variables that correspond to operation j's instances scheduled for time steps ≥ s have to assume value '0' on traces where G_k is true if:

$$\Gamma_j G_k = 0 \qquad \text{(EQ 3.13)}$$

Similarly, on traces where G_k is false, all the variables that correspond to operation j's instances scheduled for time steps ≥ s have to assume value '0' if:

$$\Gamma_j \overline{G_k} = 0 \qquad \text{(EQ 3.14)}$$

3.3.6 Timing Constraints

Since C_{sj} denotes operation j's instance at time step s, it is possible to describe a variety of timing constraints using Boolean functions. For example, assume that operation i precedes operation j and that both of them execute in a single cycle. Furthermore, assume that operation i can be scheduled at steps 1, 2, and 3 (corresponding variables are: C_{1i}, $C_{2i,}$ and C_{3i}), and that j can be scheduled at steps

2, 3, and 4 (C_{2j}, $C_{3j,}$ and C_{4j}). Then, a constraint "*j* has to be scheduled exactly 1 cycle after *i*" can be written as:

$$C_{1i}C_{2j} + C_{2i}C_{3j} + C_{3i}C_{4j} = 1 \qquad \text{(EQ 3.15)}$$

Minimum/maximum constraints can be represented similarly. For example, a constraint "*j* has to be scheduled at least 2 cycles after *i*" amounts to a Boolean function:

$$C_{1i}C_{3j} + C_{1i}C_{4j} + C_{2i}C_{4j} = 1 \qquad \text{(EQ 3.16)}$$

An iterative formulation of the constraints is possible as well. For example, Equation (3.15) can be applied at step *s* (s=2,3,4) using:

$$C_{(s-1)i} + \overline{C_{sj}} = 1 \qquad \text{(EQ 3.17)}$$

Together with the uniqueness constraint (Equations (3.2) and (3.3)), Equation (3.17) enforces the timing constraint implicitly (by construction). If a timing constraint has to be conditionally enforced, a modification similar to that in Equation (3.5) is necessary. Since we use Boolean functions to represent constraints, more complex timing behavior can be conveniently described using BDD manipulations.

3.3.7 Additional Remarks

The formulation described throughout Section 3.3 is also applicable to scheduling without speculative operation execution. Essentially, a control dependency between the conditional and fork node in the CDFG should be enforced as a hard precedence relation. However, a slightly modified set of the constraints is used to improve efficiency [94]. In addition, timing constraints can be used to enforce precedence between the operations and prohibit speculative execution on individual basis. This is because, in our formulation, precedence constraints are simply a special case of timing constraints.

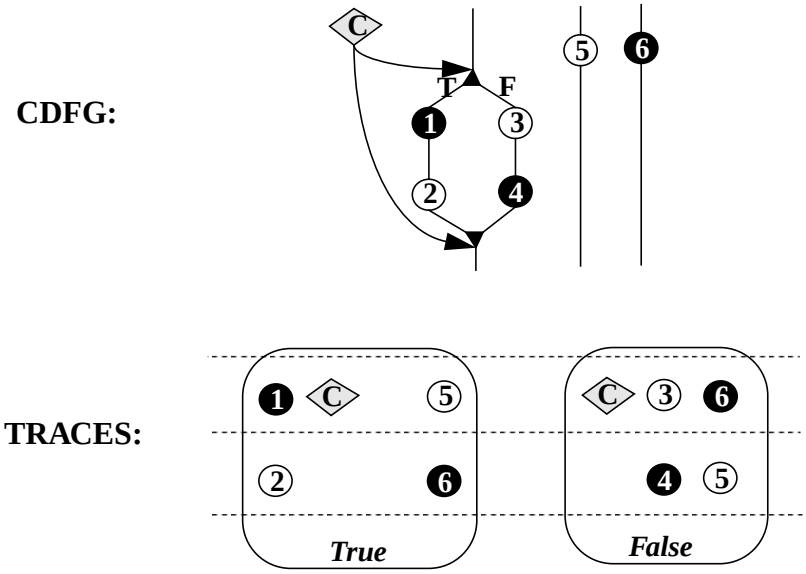


Figure 3.7 Ensemble schedule counterexample

3.4 Trace Validation

A trace satisfying all of the constraints introduced in Section 3.2 may still not be valid in the sense that it cannot be a member of any set of traces forming an ensemble schedule. The example CDFG in Figure 3.7 demonstrates that resource-constrained scheduling of all individual control paths is not sufficient for a proper treatment of control-dependent behavior. Both the “True” and “False” control paths can be scheduled individually in two time steps assuming one single-cycle resource of each type (“white”, “black”, comparator). However, observe that the execution traces shown in the figure cannot be combined into an executable schedule meeting the stated resource constraints. Since a decision as to which path to execute is not known until the end of the first step, the “True” and “False” paths are indistinguishable during that cycle. This means that both *op_1* and *op_5* as well as *op_3* and *op_6* must be executed simultaneously, thus violating the

resource constraint. (A decision to exclusively execute *op_1* and *op_5* or *op_3* and *op_6* depends on knowledge not available until the end of the first cycle!) In fact, no 2-cycle schedule is possible, although both control paths can be individually scheduled in two time steps.

Definition 3.1 A valid *ensemble schedule* is a minimal set of traces which is both *causal* and *complete*.

The *causality* requirement dictates that the schedule cannot use knowledge of the value of a conditional prior to the time when the conditional is executed (resolved). *Completeness* requires that a trace must exist for every possible control combination. An ensemble schedule is a minimal set in the sense that if any trace is removed, the set is no longer complete.

Assume that the conditional c_k is resolved at step j. Causality requires that the traces corresponding to guard values G_k and $\overline{G}_k$ must be identical (match) for all time steps prior to and including j. Completeness ensures that the ensemble schedule includes traces for both G_k and $\overline{G}_k$.

Observations from the previous paragraphs and Figure 3.7 illustrate that an exact formulation of control-dependent, resource-constrained scheduling requires more than the ability to: (*i*) somehow label control paths and operation instances belonging to them, and (*ii*) enforce precedences and resource bounds on individual paths. This might be a reason why no such technique existed until very recently [92], even for scheduling without code motion. In our formulation, the meaning of guards is not only statically linked to conditionals and their decisions, but to the actual moments when the decisions become available. Such a “dynamic” aspect is introduced by means of a *Trace Validation* algorithm.

```
(1)   i = 0;
(2)   do {
(3)         i++;
(4)         S(i) = S(i-1);
(5)         for each time step j {
(6)               S' = ∃_(V − V'(j)) S(i)
(7)               for each conditional c_k {
(8)                     S' = S'R_k(j) + ∀_(G_k)(S'¬R_k(j))
(9)                     if (S'==0) { S(i)=0; exit; }
(10)              }
(11)              S(i) = S(i)S';
(12)        }
(13)  } while (S(i)!=S(i-1));
```

Figure 3.8 Trace Validation algorithm

Trace Validation ensures that each validated trace is part of some ensemble schedule. The validation is efficiently preformed by the iterative algorithm shown in Figure 3.8. The following notation is used:

- f_x ($f_{\bar{x}}$) - *positive* (*negative*) *cofactor* of a Boolean function *f* with respect to a variable *x*
- $\exists_x f = f_x + f_{\bar{x}}$ - *existential abstraction*;
- $\forall_x f = f_x f_{\bar{x}}$ - *universal abstraction*
- ***S*** - set of all traces
- S(i) - set of traces at iteration *i*;

- S(0) - initial set of non-validated traces
- ***V*** - set of all variables not including guard variables
- ***V'(j)*** - subset of ***V*** corresponding to time steps ≤ *j*
- ***S'*** - set of traces from which all variables (***V-V'(j)***) are removed:

$$S' = \exists_{(V - V'(j))} S(i) \quad \text{(EQ 3.18)}$$

- $\boldsymbol{C} = [c_1, c_2 \ldots c_n]$ - set of all conditionals
- $\boldsymbol{G} = [G_1, G_2 \ldots G_n]$ - set of guards corresponding to the conditionals
- $\boldsymbol{R(j)} = [R_1(j), R_2(j) \ldots R_n(j)]$ - *resolution vector*

The *resolution vector* ***R(j)*** is a set of *n* Boolean functions (one for each conditional), where each function $R_k(j)$ indicates whether a conditional c_k was scheduled prior to time step *j*:

$$R_k(j) = \sum C_{lk} \text{ , for } (l < j) \quad \text{(EQ 3.19)}$$

S' is partitioned by ***R(j)*** into a disjoint set of as many as 2^n families, corresponding to the subset of guards that are resolved prior to time step *j* ($\boldsymbol{G_{res}}$):

$$R(j) \rightarrow S'_k(V'(j), G), (0 \le k \le (2^n - 1)) \quad \text{(EQ 3.20)}$$

The guards from ($\boldsymbol{G\text{-}G_{res}}$) (i.e. the unresolved guards) have to be *don't cares* within the family since at time step *j* there is no knowledge about the future values of the unresolved guards:

$$S'_k(V'(j), G) = S'_k(V'(j), G_{res}) \quad \text{(EQ 3.21)}$$

Traces must both *match* and *exist* for all possible combinations from ($\boldsymbol{G\text{-}G_{res}}$), to ensure causality and completeness of the ensemble schedule.

Definition 3.2 We say that the traces belonging to families satisfying Equation (3.21) are *locally valid at time step j*.

Definition 3.3 A *valid trace* is locally valid at every time step.

Trace validation algorithm checks for partial matching up to step *j* for all traces in parallel. However, it is possible that a trace which matched up to time step *j* is invalidated in subsequent steps. Thus its set of matching traces may no longer be complete. The Trace Validation algorithm iterates until a *fixed point* is reached. The number of iterations cannot exceed the number of conditionals (to be discussed in Section 3.4.2). Thus the algorithm generates a polynomial number of constraints regardless of the number of traces.

The intuition behind the Trace Validation algorithm can be provided by means of the schedule from Figure 2.2. Assume that the guards G_1 and G_2 correspond to the conditionals *1* and *2*. There are four possible control paths: $(G_1G_2, G_1\overline{G_2}, \overline{G_1}G_2, \overline{G_1}\overline{G_2})$. At the first step resolution vector components $R_1(1)$ and $R_2(1)$ are both zero since neither conditional is scheduled prior to step 1. To have a causal ensemble schedule, traces for all four control paths must match at the first step. At the next step, $R_1(2)$ is still zero since conditional *1* is not scheduled prior to step 2. However, $R_2(2) = c_{12} = 1$ since conditional *2* is scheduled at step 1. Thus, the matching of traces has to be performed only with respect to conditional *1* (i.e. traces for paths $(G_1G_2, \overline{G_1}G_2)$ must match for the first two steps, as well as the traces for $(G_1\overline{G_2}, \overline{G_1}\overline{G_2})$). The same argument holds for step 3.

Trace Validation implicitly verifies that the ensemble schedules do not violate resource constraints. We indicated in Section 3.3.4 that Equation (3.7) prevents such violations from occurring on individual traces. Since traces match before the

conditional is resolved, resource bounds are met. After the conditional is resolved, the traces are mutually exclusive with respect to that particular conditional and no verification is necessary. Chapter 5 discusses the alternative approach to conditional resource sharing analysis using the guard-based control model (see also [97]).

3.4.1 Proof of Correctness

Theorem 3.1 Results of two consecutive iterations of TV algorithm are same (S(i)=S(i-1)) iff only valid traces are in S(i).

Proof

It is obvious that TV algorithm is not adding any new traces to the set S(i), since at every iteration S(i) is being intersected (restricted) with the set of constraints (see line (11)). The similar statement can be made for set S': while restricting S' at a particular step no new traces are added (line (8)).

$\Rightarrow$ part

In iteration *i* at step *j* the initial value for S' ($S'_{initial}$) is the existential abstraction of S(i) w.r.t. set of variables (V-V'(j)) (line (6)). Notice that S' is a minimal superset of S(i) obtained by factoring out all the variables from (V-V'(j)): if any product term is removed from $S'_{initial}$, S(i) would cease to be a subset of $S'_{initial}$. Since new traces cannot be added to S(i), S(i) has to remain unchanged after applying a restriction S' corresponding to the current step *j* of iteration. That means that S(i) is a subset of the derived restriction S'_{final} at the end of step *j* as well. Thus, S' has to remain unmodified during the processing in step *j*. According to the previously introduced notation and Equation (3.20), S' can be partitioned into a set of disjoint families:

$$S'_{initial} = \sum_{0 \le k \le (2^n - 1)} S'_k(V'(j), G) \quad \text{(EQ 3.22)}$$

Application of the algorithm (lines (6)-(10)) results in:

$$S'_{final} = \sum_{0 \le k \le (2^n - 1)} \forall_{(G - G_{res})}(S'_k(V'(j), G)) \quad \text{(EQ 3.23)}$$

For S'_{inital} and S'_{final} to be same it must hold:

$$(S'_k(V'(j), G)) = (S'_k(V'(j), G_{res})) \quad \text{(EQ 3.24)}$$

This is same as Equation (3.21). Thus, all the traces are locally valid at step *j*. Since the similar conclusion holds for every step within the iteration, only valid traces exist in set S(i).

⇐ part

The valid traces satisfy Equation (3.21). This constraint can be substituted in Equation (3.22) and Equation (3.23), and (⇐ part) of the proof trivially holds.

3.4.2 Convergence Analysis

We have shown that Trace Validation algorithm removes all invalid traces. However, to evaluate its practicality, it is important to demonstrate the existence of the upper bound for the number of iterations necessary for the algorithm to converge to a fixed point. Observe that for the set of all traces obtained from the scheduler it is not possible to establish a temporal order (precedence) for the execution of conditionals. This also cannot be done for a single ensemble schedule, since the same conditional may be resolved at different time steps on different control paths. For a single trace (validity of which we are checking), however, the temporal order for the execution of the conditionals is well-defined. Assume that

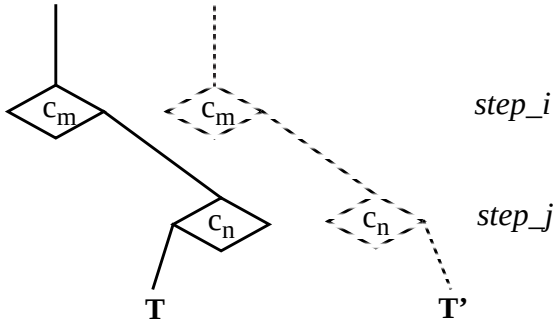


Figure 3.9 Convergence analysis

for trace T (shown in Figure 3.9) the last two conditionals to be resolved are c_m and c_n and that they are resolved at time steps i and j ($i \leq j$), respectively. During the first iteration of TV algorithm the local validity of traces at time step i is enforced before the same thing is done at time step j. Assume that trace T' (shown in Figure 3.9) is a matching trace for T at step j. This means that the following condition must hold:

$$\exists_{(V - V'(j))} T \;=\; \exists_{(V - V'(j))} T' \qquad \text{(EQ 3.25)}$$

It is possible that some traces that were locally valid at time step i may have to be invalidated during the second iteration. However, notice that trace T and T' must match up to step i [3]:

$$\exists_{(V - V'(i))} T \;=\; \exists_{(V - V'(i))} T' \qquad \text{(EQ 3.26)}$$

Thus, during the second iteration, trace matching at step i will either preserve both T and T' or remove both T and T'. In either case, no further matching is necessary at step j at the second iteration. If removal of both traces causes a solution to become incomplete, this will be identified in further iterations.

Consequently, the number of iterations cannot exceed the number conditionals in a temporal chain of conditionals within the trace. In the worst case the number

3. In fact, as indicated in Equation (3.25), they match up to step j.

of iterations is bounded by the number of conditionals (same as the number of guards), since it can happen that all of the conditionals are resolved when speculative execution is allowed. However, our experiments showed that TV algorithm typically converged after only one or two iterations, even on rather complex control structures discussed in Section 6.3. Obviously, a simple way to speed up TV algorithm is to perform the inner loop of the algorithm for only first (k-i+1) time steps at every iteration i, where k is the number of time steps of the minimum latency schedule. If the number of time steps k is smaller than the number of conditionals (possible in case of multiple conditional trees executing in parallel), k is the upper bound on the number of iterations. The efficiency of this algorithm is quite surprising considering that the number of potential control paths can grow very quickly even for relatively small problem instances. Trace validation proceeds in parallel across all potential control paths using a polynomially bounded number ($O((\#steps)(\#conditionals)^2)$) of BDD algorithmic steps.

3.4.3 Extracting One Ensemble Schedule

The technique presented in this thesis generates a solution in the form of a collection of traces. As indicated before, a trace is a possible execution instance for a particular control path. In BDD form, traces correspond to product terms of the Boolean function. Each trace includes the guard variables (identifying a control path) and operation variables (indicating a schedule for the path). Essentially, the solution incorporates all possible execution instances (for all control paths) such that they belong to at least one minimum-latency ensemble schedule.

The question that naturally arises is: “How can we extract *one* ensemble schedule from the solution?”. This is performed using the algorithm shown in

```
P = SetOfAllPaths(S);
while ( P != Zero() ) {
        T = SelectOneTrace(S,P);
        C = Control(T);
        P = P - C;
        S = TraceValidation(T+SP);
}
```

Figure 3.10 Ensemble schedule extraction

Figure 3.10. First a Boolean function *P* is formed that corresponds to a set of all possible paths in the solution set *S*. Then one trace (*T*) belonging to *S* and from a control path *C* ($C \subseteq P$) is selected. *C* is then removed from *P*, indicating the remaining paths from which single traces are yet to be extracted. A new solution *S* is formed as a union of the selected trace *T* and the set of all traces covered by *P*. This set is further trace validated to preserve only the traces that match the selected trace *T*. The process is iterated until set *P* becomes empty.

Individual trace selection is performed heuristically in a greedy fashion (i.e. we do not allow backtracking and do not guarantee a minimal average execution time for all paths). One possibility is to select the shortest possible trace form the set of traces covered by *P* [4]. Since such short traces belong to control paths with a very large number of solutions, the size of *S* is reduced very quickly. This is important since the number of iterations of the algorithm in Figure 3.10 can be as large as the number of distinct control paths in the ensemble schedule. It is possible, however,

4. Whole sets of short traces can be selected as done in some of the experiments in Section 6.3.

to guide the selection heuristic using some other criteria: for example, short traces for the paths with the highest likelihood of execution can receive a favorable treatment.

3.5 Cyclic Control

In a pipelined hardware implementation of a data-path, multiple loop iterations can be executed concurrently. The *latency* is the period of time *l* between initiations of two consecutive iterations. Loop pipelining optimizations have the goal of increasing the throughput by overlapping the execution of loop iterations. In the case of *functional pipelining*, the assumption is that no inter-iteration data dependencies exist. Given sufficient hardware resources, the latency of functionally pipelined data-paths can be made arbitrarily small. In *loop winding* [43], this cannot be done since inter-iteration data dependencies do exist. The *delay* is the number of cycles *d* required to complete one iteration. The number of overlapping iterations is usually referred to as the number of *pipeline stages*.

If a loop body does not contain conditional behavior, our formulation can be extended (similar to the ILP technique described in [49]) to incorporate loop optimization techniques such as loop winding and functional pipelining. The resource constraint procedure has to be modified to capture the fact that operations at time steps s, $s+l$, $s+2l$... share resources [5]. Additional care has to be taken to preserve inter-iteration data dependencies in case of loop winding..

Our technique can also accommodate the approach to cyclic control adopted in path-based scheduling [18]. In that approach, loop cycles are broken, execution is trapped in the last operation of a loop body and, after the scheduling is completed,

5. This approach, also known as “modulo scheduling”, was first described in [101].

transitions are added in the control finite state machine. However, the systematic treatment of speculative execution for parallel branching control with cycles is an open research problem.

In Chapter 5, we presents an alternative approach to conditional resource sharing. The approach is not explicitly used in the techniques described in the rest of the thesis. However, it is transparent to a particular scheduling implementation and has relevance to software pipelining techniques.

3.6 Scheduling Procedure

Pseudocode in Figure 3.11 summarizes our symbolic scheduling procedure.

- First, resource constraints (their type and number) are specified by the user.
- Next, a CDFG is analyzed to determine ASAP and ALAP bounds for individual operations and the length of the critical CDFG path.
- The final pre-processing step determines variable ordering and initializes guard variables.
- Solution construction is performed in iterative fashion (“*for*” loop in Figure 3.11). At every time step (cycle) *s* only scheduling constraints relevant to step *s* are generated. BDDs corresponding to these constraints are placed on a sorted list in decreasing order of their BDD size. Next, a partial solution corresponding to previous (*s*-1) steps is intersected with the constraint BDDs[6]. For efficiency reasons, it is usually beneficial to combine several

6. In reality, we actually build several lists corresponding to different constraint types. Our experiments suggest that, for the variable ordering discussed in Chapter 4, the following order of constraint application typically results in the most efficient construction: 1. functional unit and bus constraints, 2. precedences, 3. uniqueness, 4. removal of redundantly scheduled operations, 5. register constraints.

items from the constraint list into a medium-sized BDD before intersecting it with the partial solution. Trace validation step has to be performed only as a part of the termination test to verify that the set of terminated traces indeed forms an executable schedule

Construction process is discussed in more detail in Chapter 4.

3.7 Relation to ILP

Table 3.1 illustrates some differences between our technique and ILP formulations of resource-constrained control-dependent scheduling. In the symbolic approach, *any Boolean function* can be used as a constraint. Unlike ILP techniques, we can efficiently generate and store *all* feasible solutions to a particular problem instance. More importantly, this requires a very little overhead in terms of formulation variables when compared to the formulation of non-branching scheduling. In the worst case, the number of variables in our formulation is proportional to the product of the number of time steps and the number of operations in the CDFG. In contrast, an identical problem instance formulated using ILP [26] requires, in the worst case, an exponentially larger number of variables. We observe that conventional ILP techniques [41][49] essentially do not provide support for control-dependent scheduling. In such approaches, a CDFG operation has to be scheduled on the *same* cycle on all appropriate control paths.

```
BDDnode*
SymbolicScheduling ( int MAX_STEPS , directed_graph* CDFG ) {
    SpecifyResourceTypesAndBounds ( ) ;
    AnaylizeCDFG ( CDFG ) ;
    InitializeVariables ( ) ;
    BDDnode*     SOLUTION = TRUE ;
    BDDnode*     TEMP ;
    for ( int step=1 ; step<=MAX_STEPS ; step++ ) {
        BDD_List      ConstraintList = BuildConstraints ( step ) ;
        SOLUTION = And ( SOLUTION , ConstraintList ) ;
        SOLUTION = TraceValidation ( SOLUTION , step ) ;          // optional
        if ( SOLUTION == bdd->Zero() )
            break ;
        if ( step >= critical_path_length ) {
            TEMP = Trace Validatation ( TerminationTest ( SOLUTION ) , step ) ;
            if ( ( TEMP != bdd->Zero() ) || ( step == MAX_STEP ) ) {
                SOLUTION = TEMP ;
                break ;      // solution (possibly empty) found at this step
            }
        }
    }
    return SOLUTION ;
}
```

Figure 3.11 Symbolic scheduling procedure

Table 3.1: Symbolic vs. ILP formulation

	constraint type	*#solutions*	*#variables*
Symbolic	*any Boolean function*	*all*	**O**[(*#cycles*)$*$(*#ops*)] + (*#cond*)
ILP	*linear*	*1*	**O**[(*#cycles*)$*$(*#ops*) $*2^{(\#cond)}$]

#cycles - number of time steps, *#ops* - number of operations, *#cond* - number of conditionals.

Chapter 4

Construction

Formulation of scheduling constraints presented in Chapter 3 is just a part of a challenge to develop a practicable scheduling alternative. CPU run-times and memory requirements are critical to applicability of any technique (BDD-based, in particular) to "real-life" problems. In this chapter, aspects related to the BDD construction process are considered. These include a discussion of the iterative construction of a solution, BDD variable ordering strategies, as well as techniques employed to improve the run-time efficiency.

4.1 Iterative Construction Process

First we note that it is not necessary to generate uniqueness (Section 3.3.1) and precedence (Section 3.3.2) constraints on a time-step-by-time-step basis. In fact, in our initial formulation [92][93] all of the constraints were generated during preprocessing before forming their intersection. In [93], a procedure to combine all of the scheduling constraints was described. Essentially, the construction first combined constraints for which "good" orderings were known and then sequentially applied the other constraints. Using this technique, the final BDD typically has relatively small size. However, the size of BDDs at intermediate stages can be rela-

tively large, resulting in slow construction and/or large memory requirements. In particular, these problems were emphasized when the upper bound on execution time used to generate equations was larger than the actual optimal execution time.

To improve the robustness of the algorithm, a new iterative construction is proposed in [94]. The solution is built on a time-step-by-time-step basis: only those constraints relevant to a particular time step *j* are generated and applied to the BDD representing a valid partial solution for the previous (*j-1*) steps. In this way, only partial time sequences of constraints need to be added at each step.

The iterative approach has several advantages:

- It prevents the construction of large set of spurious intermediate solutions.
- Iterative construction generates a larger number of smaller constraints than the non-iterative version and can be slower for small examples. However, for larger cases, it offers far more robust behavior in terms of memory management and allows tighter control over the computation. We observe that the sizes of intermediate BDDs are typically smaller and that generation of "garbage" decreases significantly.
- Using iterative construction one can detect when schedules have completed obviating the need to accurately pre-specify the number of control steps.
- Furthermore, since the valid partial schedules are available after every iterative step, it is possible to devise run-time efficient symbolic heuristic (to be discussed in Section 4.4.3).

Iterative construction process is illustrated in Figure 4.1. During the construction process, some traces (e.g. trace labeled t_2) present after (*k*-1) steps may get

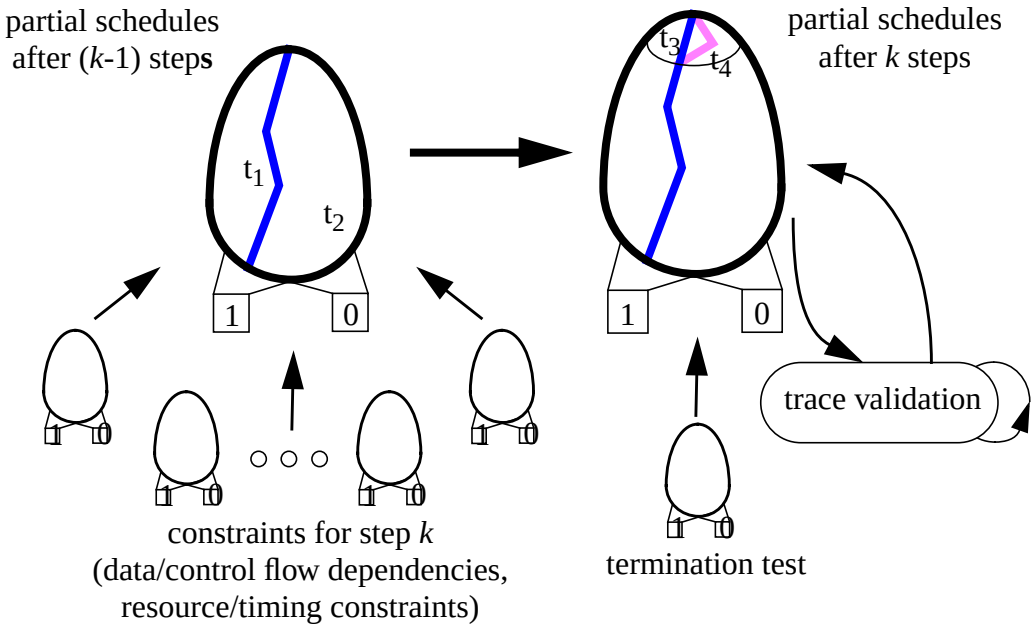


Figure 4.1 Solution construction

eliminated when constraints relevant to step *k* are applied. At the same time, some partial solutions after (*j*-1) steps can generate several new partial solutions (e.g. traces labeled t_3 and t_4 are extensions of trace t_1).

To verify the existence of an ensemble schedule, trace validation *must* be done when a termination test is performed on a set of traces that have concluded execution. Note that this set is typically significantly smaller than the set of all traces in the intermediate solution. To reduce the number of traces (and thus, potentially reduce the intermediate BDD size), it is possible to perform trace validation at the end of every iteration. This can, however, lead to somewhat increased CPU time and more intensive garbage collection. We enforced trace validation only when the intermediate BDD size exceeded a pre-specified threshold. Similarly, the uniqueness constraint for operation *j* can be applied just at step $(ALAP)_j$ (i.e. application of only Equation (3.2), Section 3.3.1, is sufficient). This diminishes the number of

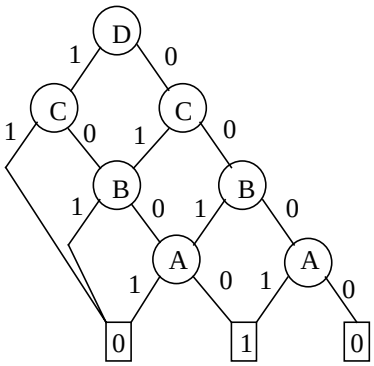


Figure 4.2 Uniqueness constraint (4 time steps span)

constraints that have to be applied and typically increases the speed of construction. However, note that the speed-up techniques described in this paragraph may produce intermediate solutions that temporarily contain invalid traces.

4.2 BDD Form of Constraints

It is of the utmost importance that the individual scheduling constraints have small size and are amenable to an efficient construction. Assume that a particular operation can be scheduled over a time span of four cycles and that the variables corresponding to instances at individual time steps are labeled A, B, C and D. Then a requirement that the operation has to be uniquely scheduled (i.e. at one time step only) can be written as:

$$A\bar{B}\bar{C}\bar{D} + \bar{A}B\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}\bar{C}D = 1 \qquad \text{(EQ 4.1)}$$

A BDD corresponding to Equation (4.1) is shown in Figure 4.2. Since the equation is symmetric, BDD size is independent of a selected variable ordering and the BDD size is guaranteed not to exceed n^2 (where n is the number of variables) [12]. In fact, for the particular type of equation discussed above it can be easily seen that the number of variables is exactly $(2n\text{-}1)$ [1].

However, since the CDFG to be scheduled contains more than one operation, the variable ordering issue plays an important role. Assume that all of the variables corresponding to each individual operation have consecutive BDD indices. In that case, all BDDs corresponding to individual uniqueness constraints would have disjunctive variable support and the resulting overall constraint would be a simple concatenation of individual constraints. On the other hand, if BDD indices corresponding to each individual operation are not consecutive, the overall constraint (intersection of individual constraints) tends to grow rapidly in terms of its BDD size. In the extreme case, when variables corresponding to different operations are completely interleaved, it may even become impossible to build the intersection of the constraints. In one relatively small benchmark problem reported in [93], the size of such constraint varied between 158 and 4,956 nodes.

The generalized resource bound constraint BDD shown in Figure 4.3 (introduced in Equation (3.7), Section 3.3.4) is frequently used as a *construction template* in symbolic scheduling. Some applications include:

- selection of solutions that satisfy a particular resource constraint at a particular time step,
- interior constraints (to be described in Section 4.4.1),
- scheduling heuristics (to be discussed in Section 4.4.3), and
- post-processing (after the scheduling is completed, the bounds can be iteratively tightened/identified).

1. For simplicity, we assume that there is no control-dependent behavior (i.e. the problem can be described as a DFG and all guard functions Γ_i are tautologies).

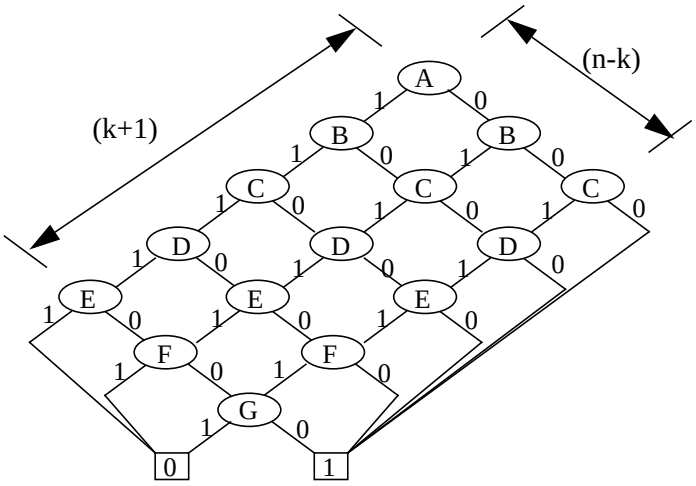


Figure 4.3 *At-most-k-of-n* constraint (k=4, n=7)

Vertices in this *if-then-else* template are not restricted to Boolean variables -- complex Boolean functions (f_1, f_2,... f_n) can be inserted into the template. Note that the number of product terms in a sum-of-products representation of Equation (3.7), Section 3.3.4 is $\binom{n}{k}$. However, its BDD form is compact (O(*nk*) nodes) and can be built efficiently using *ite* [10][12] (if-then-else) calls. This is very important since, due to its regularity, the constraint satisfies both desirable properties state at the beginning of this section (i.e. it is both small and easy to construct)[2].

Assume that all of the CDFG operations execute using single-cycle functional units. In such case, functional unit resource constraints similar to the one shown in Figure 4.3 have to be generated for each time step. This means that the ordering in which variables corresponding to different CDFG operations are interleaved is the

2. This property of the template shown in Figure 4.3 can be utilized in developing efficient "pseudo-polynomial" BDD algorithms for some hard combinatorial problems. For example, in an unrelated set of experiments, we were able to find maximal clique(s) in undirected random graphs with 100 nodes and edge-probability of 0.5. Such instances may pose problems for heuristic algorithms, since for each node the expected number of incident edges is 50, while the expected size of the maximal clique is close to 10.

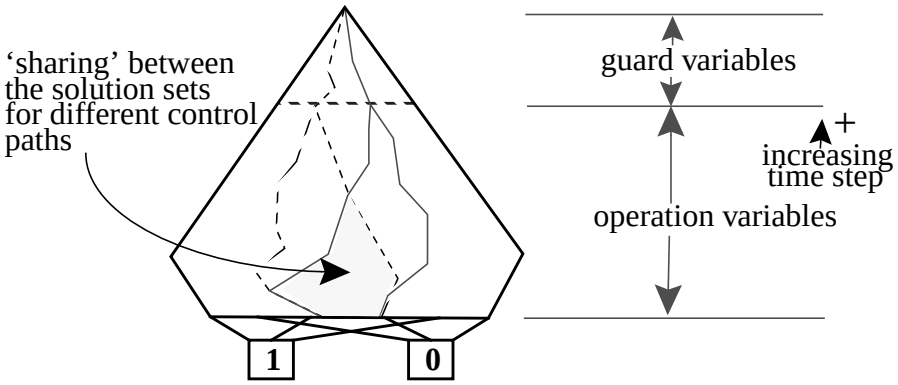


Figure 4.4 BDD representation of the solution

most desirable ordering as far as the construction and intersection of individual resource constraints is concerned. Unfortunately, this is likely to be the worst case ordering for the uniqueness constraint discussed previously. Several examples discussed in [93] provide experimental support for this intuitive observation.

4.3 Variable Ordering

As indicated in Section 4.2., although individual equations have efficient orderings, optimal orderings for different equations frequently contradict. In fact, the optimal BDD variable ordering problem is known to be NP-complete [12][39][65].

Early in this research, numerous strategies were investigated and used to guide non-iterative solution construction [92][93]. In this thesis, however, all of the presented results are generated using the variable ordering shown in Figure 4.4, where non-guard variables are ordered by increasing time step and guard variables are placed on top (i.e. closest to the root of BDD). Essentially, CDFG operations are sorted based on their *as-soon-as-possible* time and operation variables corresponding to the same time step are interleaved [3]. This ordering typically results in small

3. This is similar to heuristic ordering strategies discussed in [72].

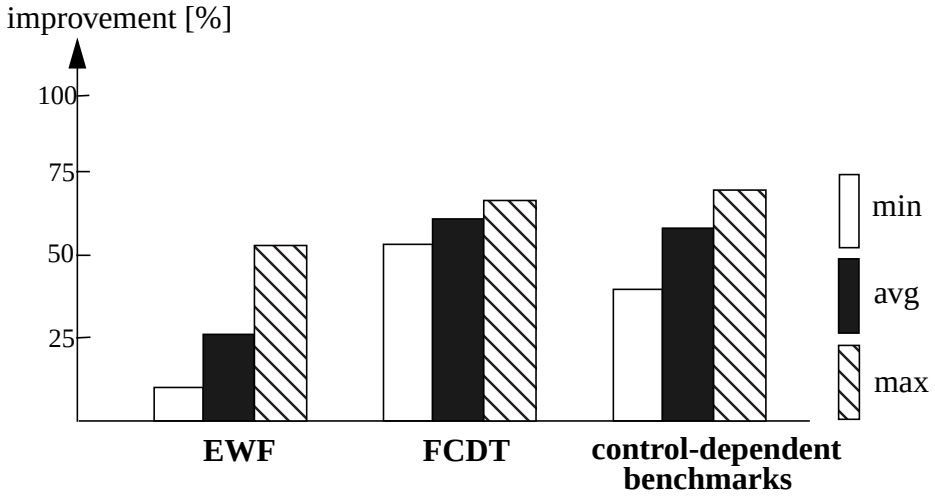


Figure 4.5 Effects of BDD variable ordering

BDDs, enables fast manipulations and accommodates iterative construction. An intuitive explanation can be offered: using the implemented ordering, application of constraints and BDD manipulations are usually localized to a relatively narrow horizontal slices close to the root node of a partial BDD solution.

The effect of the implemented variable ordering using dynamic variable re-ordering option from our BDD software package was investigated in numerous experiments. Re-ordering algorithm is based on a “sifting” procedure described in [106]. Although time-consuming, the algorithm is still not exact. However, the algorithm’s effectiveness has been demonstrated in numerous large problems. One-pass re-ordering was applied to the final solution only. The results are tabulated in Figure 4.5 for three types of benchmarks studied in Chapter 6. The experiments show that for, particular problem instances, BDD size can be improved as much as three times. Thus, devising more complex heuristics (capable of taking into account both the CDFG structure of the problem under consideration as well as the hardware/timing constraints) can be beneficial. However, this leads to some

sort of a “circular” argument, since such ordering heuristics would likely have to be able to resolve many very hard scheduling issues. Additionally, our experimental results from Chapter 6 reveal the solution sizes of the order O(n^2), where n is the number of formulation variables. This is very encouraging since the theoretical results from [65] demonstrate that the vast majority of general Boolean function have exponentially expensive optimal ordering.

4.4 Speed-Up Techniques

In this section, several techniques improving the run-time efficiency of symbolic scheduling techniques are discussed. Accompanying experimental result will be presented in Chapter 6.

- To prevent partial solutions from becoming prohibitively large during the iterative construction process a set of auxiliary scheduling constraints (*interior constraints*) is derived (Section 4.4.1).
- In Section 4.4.2, it is demonstrated how certain complex constraints can be applied *implicitly* (i.e. without explicitly building a constraint BDD).
- Finally, for very large problems consisting of thousands of representation variables, *set-heuristics* are described that preserve whole sets of partial solutions exhibiting desirable properties (Section 4.4.3).

4.4.1 Interior Constraints

In the current implementation, the solution is built iteratively and a termination test is performed after all of the constraints relevant to a particular time step are applied. Although the BDD size of the final solution is typically very moderate, the intermediate solutions can become prohibitively large, resulting in a slower construction and larger memory requirements. Ideally, the intermediate size should

never exceed the size of the final solution. In such a scenario, as long as the final solution fits the memory limits of the run-time environment, the software tool should be able to complete a scheduling task.

To alleviate the problems arising from the uncontrolled growth of the intermediate solution, we identify and discard a set of partial schedules that "hopelessly lag behind" during the construction process and cannot contribute to the set of optimal solutions. This means that at a particular time step such partial schedules cannot terminate for given resources and a pre-specified upper bound on execution time. This consideration leads to a set of *interior constraints* which is dynamically generated during the scheduling in order to prune the BDD.

The main strategy is illustrated by the following example: Assume that at the beginning of step *s* there are *n* addition operations that have ALAP (as-late-as-possible) bounds in the range [*s*... (*s*+*k*-1)] and that there are only *m* single-cycle adders available. At least (*n* - *km*) of these addition operations must be completed prior to step *s* in a feasible solution. Selection of a subset satisfying this property is done efficiently using the constraint template shown in Figure 4.3. Such constraints can be derived for each functional unit type (including multicycle and pipelined units). Interior constraints with *lookahead k* enable an early detection of many (not necessarily all) partial schedules that are destined to be discarded within the next *k* steps. Since the completeness of the solution set is preserved, this does not impact optimality. Experimental results illustrating the benefits from interior constraints application are presented in Section 6.1 and Section 6.4.

A further improvement in run-time efficiency can be expected if execution interval analysis [115] is used for search space reduction. Interior constraints can be viewed as a subset of such analysis.

4.4.2 Implicit Application of Complex Constraints

The generalized resource bound constraint BDD shown in Figure 4.3 is frequently used as a *construction template* in symbolic scheduling. Vertices in this if-then-else template are not restricted to Boolean variables -- complex Boolean functions (f_1, f_2,... f_n) can be inserted into the template.

However, even when (f_1, f_2, ... f_n) are rather simple, the overall constraint may become extremely large. Consequently, it can happen that the partial scheduling solution is of a very moderate size, but the constraint to be applied cannot be built. However, the scheduling constraint need not be explicitly built. The following can be done instead:

- Introduce a new set of auxiliary variables (y_1, y_2, ... y_n) corresponding to the set of functions (f_1, f_2, ... f_n).
- Build the template function *T* (shown in Figure 4.3) using only (y_1, y_2, ... y_n).
- Compute:

$$P^0 = And(P', T) \quad \text{(EQ 4.2)}$$

where *P'* is a partial solution to which the constraint is applied.

- Clearly, a new partial solution *P"* can be obtained using the recursive formula:

$$P^{(i)} = \exists y_i(And(P^{(i-1)}, Xnor(y_i, f_i))) \quad \text{(EQ 4.3)}$$

where $\exists_x f = f_x + f_{\bar{x}}$. This amounts to the standard BDD substitution operation:

$$P^{(i)} = P^{(i-1)}\Big|_{y_i \equiv f_i} \quad \text{(EQ 4.4)}$$

Using this approach, in all of the benchmarks discussed in Section 6.4, we were able to apply register constraints that could not be built explicitly.

4.4.3 Symbolic Heuristics

The main challenge for symbolic techniques can be summarized by Bryant's observation from [13]:

"... In many combinatorial optimization problems, symbolic methods using OBDDs have not performed as well as more traditional methods. In these problems we are typically interested in finding only one solution that satisfies some optimality criterion. Most approaches using OBDDs, on the other hand, derive all possible solutions and then select the best from among these. Unfortunately, many problems have too many solutions to encode symbolically...".

It has been shown that some standard benchmark instances have billions of optimal solutions ([94], Chapter 6). In such cases, the BDD representation can become too large to be practical since both its size and CPU run-time increase significantly.

Heuristic scheduling techniques [18][84][87] are applicable to large problems but may fail to find an optimal solution in tightly constrained problems. This is primarily because the heuristics cannot recuperate from early suboptimal decisions which typically preserve only one representative from a possibly very large pool of candidates.

Since valid partial schedules are available after each time step of our symbolic construction, it is possible to devise heuristic scheduling techniques.The simplest *utility-based* heuristic [94] propagates only the subset of schedules with maximum

```
BDDnode*
SetUtilize(BDDnode* partial, BDDnode* sink, int step, int utility) {
    BDDnode *subset;
    if(step>=minimum_execution_time) { /* check for end */
        subset = And(partial, sink);
        if(subset!=0) return(subset);
    }
    do {
        subset = And(partial,ChooseExactly(utility));
        if(subset==0)  utility--;
        else           return(subset);
    } while(utility>=0);
    return(0);
}
```

Figure 4.6 Utility-based set-heuristic

utilization of resources (see Figure 4.6). Utilization is measured by the number of operations active in each time step. The utility-based heuristic is implemented by iterative application of the generalized resource bound. We enforce maximum utilization of functional units, and then iteratively relax this constraint until satisfying partial solutions are found. Since *all* such schedules are propagated, this simple heuristic has good behavior. An additional (second-level) pruning strategy (*utility+CP*) based on the AFAP (as-fast-as-possible) scheduling of the operations belonging to the critical path(s) is possible as well. Essentially, the scheduler favors the partial solutions where the largest number of operations belonging to the critical path(s) have been scheduled. This strategy is effective when the number of operations that can be scheduled simultaneously is very small or when the sched-

ule is expected to take very large number of steps (some of the problems in Section 6.4 execute in more than 100 cycles).

The algorithm can be made less greedy by applying it over a sliding window of several time steps or over a range of utilizations. Finally, the BDD pruning can be delayed behind the current scheduling step to create “look-ahead”. These manipulations are surprisingly efficient and consist of repeated use of the construction template shown in Figure 4.3.

An accurate estimate of the upper bound on scheduling latency may not be available before scheduling. Unfortunately, the search space increases enormously fast with relaxation of this bound. We show in Section 6.4 that set-heuristic scheduling is very robust: the construction pace shows very weak sensitivity to the upper bound used to initialize the scheduler. It is very important that the heuristics be robust, since they can be used to derive accurate bounds for the exact schedulers whose run-time efficiency is more sensitive to the bound estimates.

4.5 Alternative Representations

This section describes alternative symbolic representations that were considered during this project: Zero-Suppressed BDDs [75] and “log encoding”. We believe that our experience with these approaches may prove valuable for the future implementations and extensions of our work.

4.5.1 Zero-Suppressed BDDs

Zero-Suppressed Binary Decision Diagrams (0-sup-BDDs or ZBDDs) are data structure optimized for an implicit set representation in combinatorial problems [75]. The major difference between 0-sup-BDDs and ROBDDs is the reduction rule shown in Figure 4.7. In ROBDDs, reduction is performed through “don’t

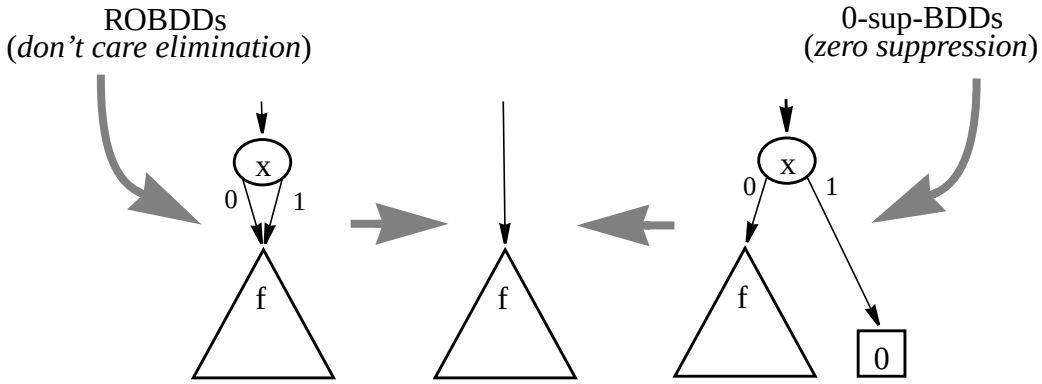


Figure 4.7 BDD reduction rules

care" elimination -- a graph node is eliminated if its both sons are isomorphic (see left-hand side of Figure 4.7). In 0-sup-BDDs, a graph node is eliminated if its "True" son points to a "0" terminal (empty set). Since solutions of many combinatorial problems are rather sparse, this reduction rule allows an implicit representation of the majority of variables that correspond to the elements not present in the final solution. Recently, a 0-sup-BDD were applied to representation/manipulation of polynomials [76][98].

The advantage of using ZBDDs in exact symbolic schedulers is effectively illustrated by the example of the *Elliptic Wave Filter* (EWF) benchmark shown in Figure 4.8. The 28-cycle EWF instance (using one adder and one 2-cycle non-pipelined multiplier) has only 34 operations but 437 variables are used to fully describe the problem. Every solution (a path to "True" node in ROBDD representation) consists of 437 variables out of which only 34 are "1" [4]. A vast majority of variables are equal to "0" (i.e. a particular operation is not scheduled at a particular time step), but are explicitly represented in the ROBDD. Consequently, the solution representation has more than 130,000 ROBDD nodes. Since the 0-variables

4. Actually, 35, including the "sink" node used solely for termination testing.

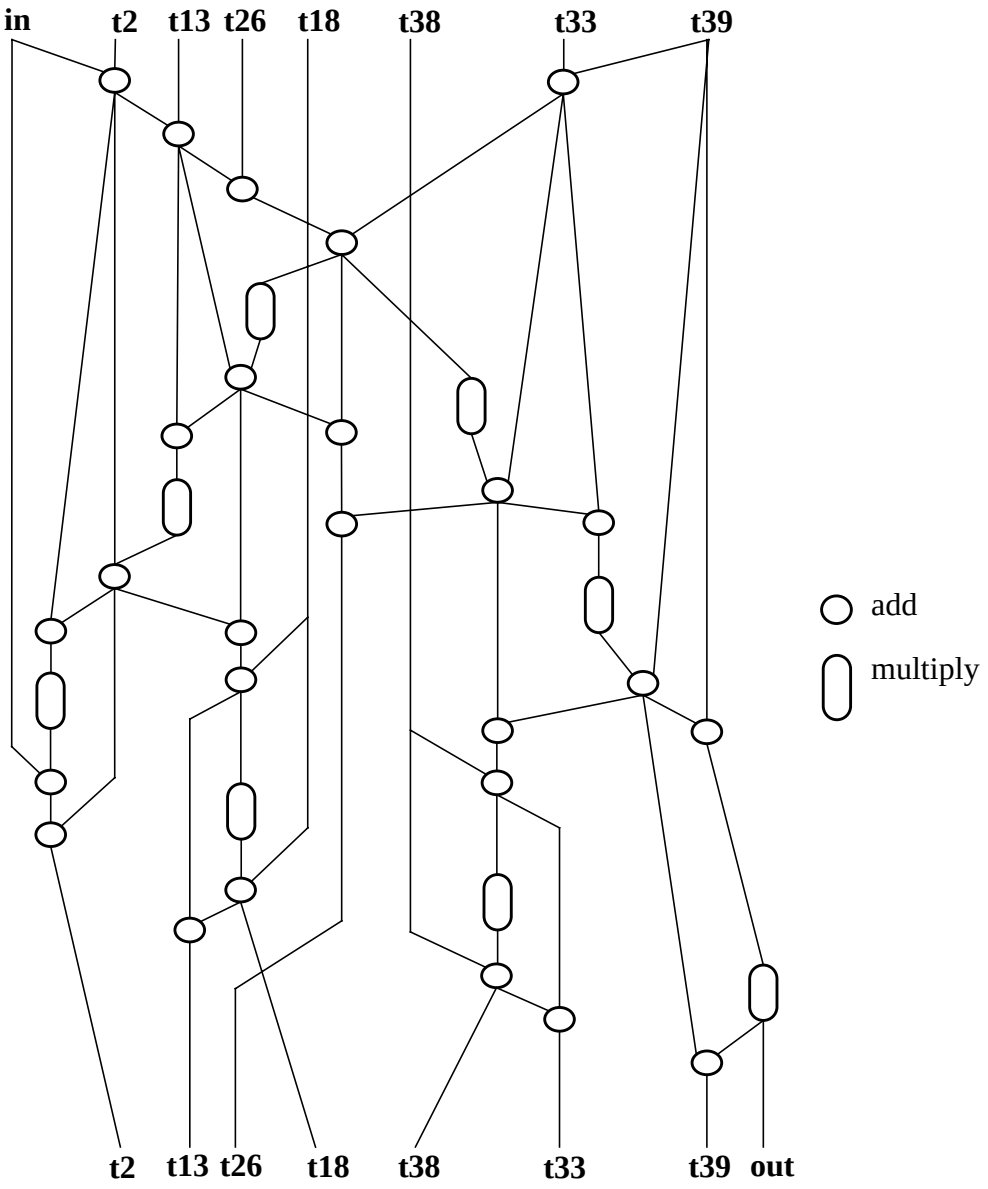


Figure 4.8 Elliptic wave filter (EWF) benchmark

that belong to the solution are implicit (suppressed) in 0-sup-BDDs (ZBDDs), much larger compression of the solution set is possible. In fact, our experiments showed that for a 28-cycle EWF, a nearly ten-fold reduction in size is achieved. A 15-fold reduction is observed when a 2-cycle adder is used (967 variables, 54-

cycle optimal solutions). This marked compression of the representation size offers a potential for analysis of much larger problems than is possible using ROBDDs. In fact, for some examples discussed in Chapter 6, the number of nodes in the optimal solution sets is occasionally smaller than the number of variables describing the problem.

We observed that, when applied to the scheduling problem, 0-sup-BDD manipulations are slower than similar ROBDD manipulations. This seems to be caused by the frequently simpler ROBDD form of constraints. The constraint equation may involve only a few formulation variables and the corresponding ROBDD representation is typically small. However, such constraint may become more complex when converted to a 0-sup-BDD because *don't-care* variables are not implicit in the 0-sup-BDD representation. Another reason for a relatively lower efficiency of 0-sup-BDDs may be due to the implementation specifics of the basic algorithms. In ROBDDs, all of the basic Boolean operations (e.g. *And, Or, Not*) are implemented using a single *ite* (if-then-else) call [10]. For example

$$: And(F, G) = ite(F, G, 0) \ . \qquad \text{(EQ 4.5)}$$

This allows a very efficient hashing and cashing implementation policies that drastically improve computation time [10]. In 0-sup-BDDs, there is no equivalent to *ite*. In fact, separate algorithms are developed for different set operations (e.g. *Intersection, Union, Difference*). Consequently, computational efficiency of a 0-sup-BDD package can be expected to be reduced compared to the ROBDD case. Additionally, in the current scheduler implementation, all of the individual constraints are generated as ROBDDs and then converted to 0-sup-BDDs prior to their intersection. This introduces a very small overhead for large problems, and is beneficial since:

- the ROBDD form of constraints is well-understood and they can be built efficiently,
- RO-to-0-sup BDD conversion is a simple one-pass algorithm, and
- a significant software infrastructure for ROBDD-based symbolic scheduling was already available.

All of the larger DFG instances presented in Chapter 6 were solved using both ROBDDs and 0-sup-BDDs. Not surprisingly, ROBDD implementation required more run-time memory (although the requirements were not excessive). Nevertheless, CPU times for ROBDD version were still typically better. In addition to a discussion from the previous paragraph, it should be acknowledged that a significantly larger amount of programming hours and experience was invested in the ROBDD package implemented by our CAD group.

4.5.2 Log Compression

The formulation presented in Chapter 3 assumes that a variable is assigned to particular instance of a particular operation (i.e. C_{sj}) stands for: instance of operation *j* at time step *s*. This effectively corresponds to a “one-hot” encoding of time and is an inherently redundant representation. In case of schedules where a number of operations in CDFG is large, decreasing bounds on available resources lead to large operation mobilities and, consequently, to large number of BDD variables. For example, 28-cycle EWF benchmark requires 436 variables (on average, 12.8 variables per operation). The question that arises naturally is whether the scheduling formulation can be modified to reduce the number of variables. One possibility is to impose a logarithmic compression of time in the following fashion:

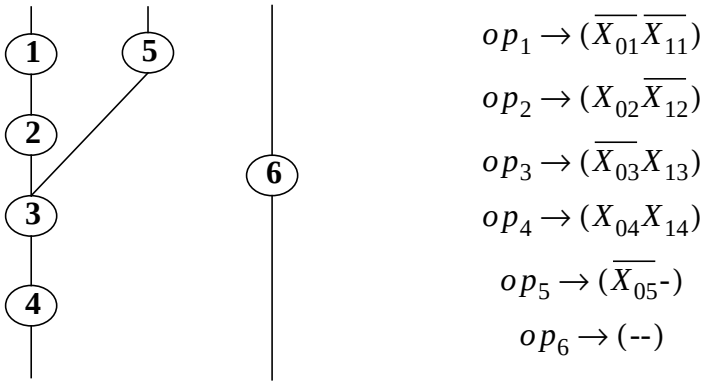


Figure 4.9 Path sharing

- specify the upper bound on execution time for a problem instance to be scheduled (t_{ub}), and
- create $\lceil \log(t_{ub}) \rceil$ variables for each operation from CDFG [5].

For example, all benchmark instances with the upper bound less than 32 time steps can be scheduled using at most 5 variables per operation. Potential asymptotic benefits are obvious: if the number of operations in a CDFG is n_{ops}, than a total number of variables is $O(n_{ops} * t_{ub})$ for the "one-hot" representation and $O(n_{ops} * \lceil \log(t_{ub}) \rceil)$ for the "log" representation. No general claim can be made that "log" compression leads to smaller BDD sizes, but it can be observed that more than one solution to the scheduling problem can be represented in a single BDD path. Figure 4.9 demonstrates this attractive property.

5. This resembles the formulation proposed in [54] and implemented using multi-valued decision diagrams. At the time when we considered "log encoding" representation, we were unaware of that work. Consequently, the approach presented in Section 4.5.2 has unique construction and constraint generation aspects. We thank T. Kam and R. Brayton for referring us to [54] during our visit to Berkeley in February, 1995.

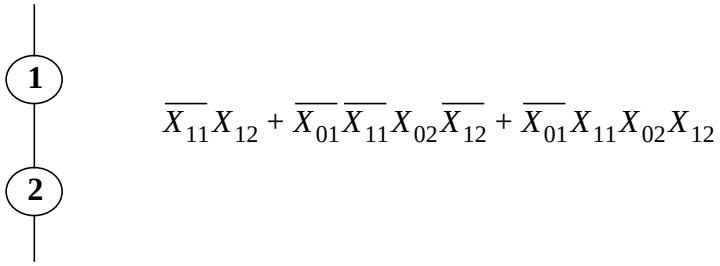


Figure 4.10 More path sharing

Assume that the CDFG has to be scheduled in four time steps using infinite resources. Thus 12 variables are needed in the formulation (two for each operation) [6]. X_{ij} stands for i^{th} variable of operation j ($0 \le i < \lceil \log(t_{ub}) \rceil$). There are eight possible solutions (operation *5* can be scheduled at steps 0 and 1, operation *6* can be scheduled at any step), but only a *single* product term is needed to represent it. Encoding that was used in the previous example was a simple binary one -- it can be observed, however, that Gray's encoding would lead to a similar compression of the result.

Similarly, assume that four time steps are allowed to schedule operations *1* and *2* in Figure 4.10. In this case, there are six solutions represented by only three non-intersecting cubes (paths in BDD). The first term ($\overline{X_{11}}X_{12}$) corresponds to four solutions: operation_1 scheduled at the first two steps (0 or 1) and operation_2 scheduled at the last two steps (2 or 3).

We decided to perform a simple translation from the "one-hot" formulation using the notion of "minterm templates" that are pre-constructed before the actual scheduling is performed. When a particular constraint has to be generated we can use the necessary templates and shift them accordingly. The following example

6. In this simple example without resource constraints, a number of variables using the "one-hot" representation is actually smaller (10).

illustrates the basic idea. Assume that operation *i* precedes operation *j* and that both operations execute in one cycle. Then, at time, step *k* a precedence constraint has to be satisfied:

- (operation *i* was scheduled before step *k*) or (operation *j* is not scheduled at step *k*).

To implement precedence constraint between arbitrary two operations in CDFG, two kinds of templates can be generated for every time step $k \leq t_{ub}$:

- *at[k]* is an encoding (minterm) of a time step value *k*,
- *before[k]* is a Boolean function that is equal to the sum of *at[l]* for $0 \leq l < k$.

Storing all of the templates causes very small memory overhead to the system since there is a very large amount of sharing between the BDD representations of the templates [7].

There are two obvious extreme cases we inspected for implementing variable ordering:

- interleave the corresponding variables belonging to all operations (“interleaved” ordering [54]) -- most significant bits closest to the BDD root
- assign consecutive BDD indices to all the variables corresponding to a particular operation (“natural” ordering [54])

When “one-hot” encoding of time is used, the iterative construction process (Section 4.1) and the corresponding variable ordering (Section 4.3) are efficient

7. We used very simple and unique encodings for each time step (standard binary or Gray’s encoding). Arguably better results might be expected if some redundancy is allowed.

because BDD variables are gradually introduced on time-step-by-time-step basis and manipulations are frequently localized close to the root of the BDD. When "log"' encoding is used, a similar idea can be applied (i.e. variables are introduced on bit-per-bit basis). Effectively, the scheduling problem is solved in $\lceil \log(t_{ub}) \rceil$ steps by dividing the time dimension into the "bins" of equal size and applying gradually more restrictive constraints. We describe the basic principle very informally, using the following example. Assume that a solution is sought for EWF with one 2-cycle non-pipelined multiplier and two single-cycle adders and that a conservative upper bound of 22 time steps is specified (t_{ub}). Five variables per operation are needed ($2^4 < t_{ub} < 2^5$).

STEP_0:

- Divide the problem in two bins (*bin_0* and *bin_1*) of size 16
- Preserve the precedence constraint for every pair of operations (i.e. if *op_i* precedes *op_j*, it is not legal to put *op_j* in *bin_0* and *op_i* in *bin_1*)
- Analyze the CDFG and detect operations that cannot be placed ("chained") in the same bin (e.g. a critical path from *op_i* to *op_j* is larger that the bin size). Generate and apply the corresponding constraints.
- Apply ASAP/ALAP and resource bounds on both bins. The constraints use only MSB variables: there are just two bins and the bits of lower significance should obviously remain "don't cares".

STEP_1:

- Same as STEP_0, just 4 bins of size 8 are used.

...

STEP_4:

- There are 32 bins of size 1. Stop at the earliest time step when termination (schedule completion) is detected.

Notice that only a portion of variables (one bit per operation) is introduced at each step.

We performed experiments using the “log” encoding and both extreme variable ordering (natural and interleaved) introduced earlier in this section. Only functional unit resource constraints were applied. Once the solution was generated, variable re-ordering was used to investigate potential improvements in the BDD size. Somewhat surprisingly, the solutions were typically larger than those obtained using the “one-hot” encoding. We observed that the number of BDD paths decreases (typically, 2-3 times) in the “log encoding” representation due to the ability to encode several solution in a single path. This, however, occurs at the expense of a reduced sharing in a BDD data structure.

Section 4.5.2 illustrates an intriguing idea we considered in the early stages of this project. Although experimental results did not live up to our expectations, some caution should be exercised in labelling this previously undocumented work as definitely impractical. More elaborate encodings and variable ordering schemes may improve applicability of the described approach.

Chapter 5

Conditional Resource Sharing Analysis

Hardware resource optimization of control/data flow graphs (CDFGs) is particularly important when conditional behavior occurs in cyclic loops and maximization of throughput is desired. In this section, an exact and efficient conditional resource sharing analysis using a guard-based control representation (Section 3.1) is presented.

Typical deficiencies observed with previously proposed HLS approaches to conditional resource sharing in acyclic CDFGs include:

- no support for code motion,
- restriction to a fixed order of execution of conditionals,
- restriction to nested if-then-else control structures, and
- no support for parallel and correlated control structures.

Moreover, it has been reported [103] that some representations can occasionally lead to incorrect conclusion on mutual exclusion between the operations [120].

Numerous techniques for cyclic data-flow graph (DFG) optimizations have been proposed, ranging from heuristics [22][63][90][107] to ILP methods [42][49]. However, none of them discusses cases in which conditional behavior occurs within the loop body. The BFSM-based approaches are applicable to cyclic CDFGs, but they either lack a formal treatment of conditional resource sharing [128] or introduce an excessive number of 0/1 variables to model resource and exclusivity constraints [113]. Recently, *rotation scheduling* [22] has been extended to pipelining of CDFGs [110]. This technique is based on a *condition flag* representation restricted to cases where execution conditions can be represented as a Boolean cube. Conditional resource sharing analysis is performed using *usage flags* assigned to individual functional units. Support for *node dividing* [119] is not discussed.

The guard-based analysis presented in this chapter is transparent to a scheduler implementation. The proposed technique systematically handles complex conditional resource sharing for cases when software pipelined loops include conditional behavior within the loop body. Additionally, the analysis is exact, thus avoiding over-estimation of resource requirements (for example, see Figure 1.5, Chapter 1). Throughout this section, *At-most-k-of-n constraint* (a.k.a. “generalized resource bound”) described elsewhere in this thesis (Equation (3.7), Figure 4.3) will be referred to as $B_{k,n}$.

5.1 Acyclic CDFGs

Guard functions may be used to perform conditional resource sharing analysis for an *arbitrary* number of CDFG operations. We illustrate the idea using a CDFG fragment shown in Figure 5.1. Assume that the scheduling has been completed for *step_1* and that operations *1* and *2* have been scheduled in the *step_2*. We want to

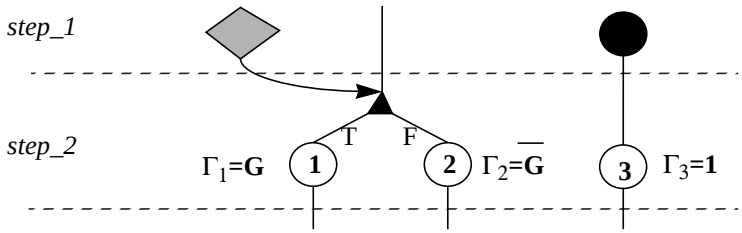


Figure 5.1 Example CDFG fragment

analyze scheduling operation *3* in *step_2* assuming that only one "white" resource is available. Evaluating an $B_{1,3}$ using guard functions Γ_i (i = 1, 2, 3) as arguments we obtain:

$$B_{1,3}(\Gamma_1, \Gamma_2, \Gamma_3) = \overline{\Gamma_1}\overline{\Gamma_2} + \overline{\Gamma_1}\overline{\Gamma_3} + \overline{\Gamma_2}\overline{\Gamma_3} = 0 \quad \text{(EQ 5.1)}$$

Since the constraint evaluates to "0", we conclude that the schedule is infeasible on all paths. If two resources are available, the constraint $B_{2,3}$ evaluates to "1":

$$B_{2,3}(\Gamma_1, \Gamma_2, \Gamma_3) = \overline{\Gamma_1} + \overline{\Gamma_2} + \overline{\Gamma_3} = 1 \quad \text{(EQ 5.2)}$$

indicating that operation *3* can be scheduled on all paths

Let us assume now that operation *1* has been scheduled for execution in a speculative fashion in *step_1*, and that operation *2* is scheduled in *step_2*. Can operation *3* be scheduled in step_2 with only one resource? We evaluate $B_{1,2}$ constraint using Γ_i (i = 2, 3) and obtain:

$$B_{1,2}(\Gamma_2, \Gamma_3) = \overline{\Gamma_2} + \overline{\Gamma_3} = G \quad \text{(EQ 5.3)}$$

This result indicates that the resource bound is met only on path G. In general, the following theorem holds [1]:

1. This reduces to a pair-wise mutual exclusion test ($\Gamma_i\Gamma_j$=0) as a previously observed special case (e.g. [8][53][92]).

Theorem 5.1 *Assume that n operation instances are candidates for scheduling at a particular time step and that there are only k resources available. Then the evaluation of* $B_{k,n}(\Gamma_1, \Gamma_2, \ldots, \Gamma_n)$ *returns all paths where the resource constraint is not violated.*

The proof is straightforward since every individual control path is represented as a product of guard variables. We can evaluate $B_{k,n}(\Gamma_1, \Gamma_2, \ldots, \Gamma_n)$ for every possible combination (minterm) of guard variables and obtain "1" (if the minterm is covered by at most k Γ_i functions) or "0" (if the minterm is covered by more than k functions). Note that although the conceptual complexity of the test is very high, it can be performed efficiently since Γ_i functions are represented by BDDs -- the computation amounts to insertion of guard functions into the template $B_{k,n}$.

We define an operation j's *split-function* S_j as a Boolean intersection:

$$S_j = \Gamma_j B_{k,n}(\Gamma_1, \Gamma_2, \ldots, \Gamma_j, \ldots, \Gamma_n) \quad \text{(EQ 5.4)}$$

Remember that Γ_j indicates all control paths where operation j must be scheduled. Thus S_j indicates all paths where operation j can be scheduled at a particular time step when $B_{k,n}$ is evaluated. If S_j is equal to Γ_j, operation j can be completely scheduled at that time step. If S_j is a proper subset of Γ_j ($\Gamma_j \supset S_j$), node splitting (dividing) may be considered. In the previous example, S_3=G and ($\Gamma_3 \supset S_3$). Thus, operation *3* can be scheduled on path G in *step_2*. On paths:

$$\Gamma_3 \backslash S_3 = \Gamma_3 \overline{S_3} = \overline{G} \quad \text{(EQ 5.5)}$$

operation *3* has yet to be scheduled in the subsequent steps[2].

To support code motion across the basic code blocks, Γ_i functions may have to be modified during the scheduling. For example, if operation *1* (Figure Fig.5.1) is

2. The scheduler, however, has to ensure that node dividing is done in a causal manner (e.g. not to allow dividing of nodes with respect to a conditional whose value is still unknown at a particular time step).

executed speculatively in *step_1*, variable G has to be factored out from Γ_1 (i.e. Γ_1 becomes “1”), since the corresponding conditional (shaded comparator) is unknown at that time. This reflects the fact that during *step_1*, paths G and $\overline{G}$ are indistinguishable.

The proposed approach is memory efficient. We observe that the number of operations in a typical CDFG is significantly larger than the number of potentially distinct guard functions. Only one pointer to a guard function need be stored for each operation instance during the scheduling process. Furthermore, memory overhead for storing guard functions is expected to be very low due to the sharing property of the BDD data structure. Compared to the method proposed here, condition vectors [119] are less efficient and have smaller expressive power since in that approach:

- control paths are “one-hot” encoded,
- no sharing is possible between the vectors, and
- execution order of conditionals is pre-specified.

Guard-based analysis is not restricted to physical hardware resources, but can be applied to modelling more general constraints. As indicated in [69], this property is very important for industrial HLS tools. For example, *mutual exclusion of n signals* is tested by using $B_{1,n}(\Gamma_1, \Gamma_2, \ldots, \Gamma_n)$. A condition for *synchronization of n signals* is evaluated using the complement of $B_{(n-1),n}(\Gamma_1, \Gamma_2, \ldots, \Gamma_n)$ -- this identifies all control paths where all signals occur simultaneously.

5.2 Pipelining of cyclic CDFGs

In a pipelined hardware implementation of a data-path, multiple loop iterations can be executed concurrently. The *latency* is the period of time *l* between

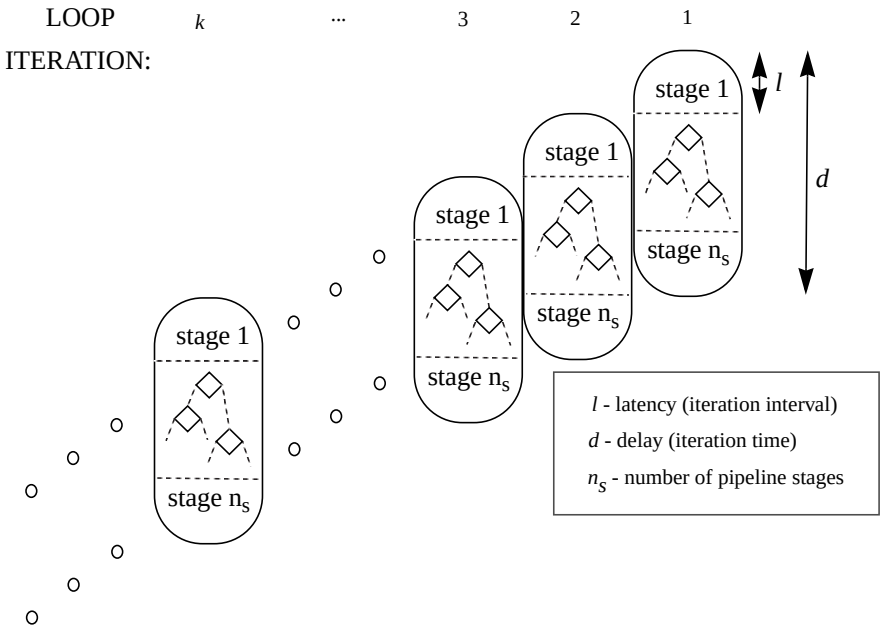


Figure 5.2 Overlapping of loop iterations

initiations of two consecutive iterations. Loop pipelining optimizations have the goal of increasing the throughput by overlapping the execution of loop iterations. In the case of *functional pipelining*, the assumption is that no inter-iteration data dependencies exist. Given sufficient hardware resources, the latency of functionally pipelined data-paths can be made arbitrarily small. In *loop winding*, this cannot be done since inter-iteration data dependencies do exist. The *delay* is the number of cycles *d* required to complete one iteration. The number of overlapping iterations is usually referred to as the number of *pipeline stages*.

Figure 5.2 shows an example of overlapped execution of a loop using n_s pipeline stages. Assume that the loop body exhibits n_p distinct control paths. In Figure 5.2, the number of paths may grow as:

$$(n_p)^{\left(\left\lfloor \frac{(step-1)}{l} \right\rfloor + 1\right)} \qquad \text{(EQ 5.6)}$$

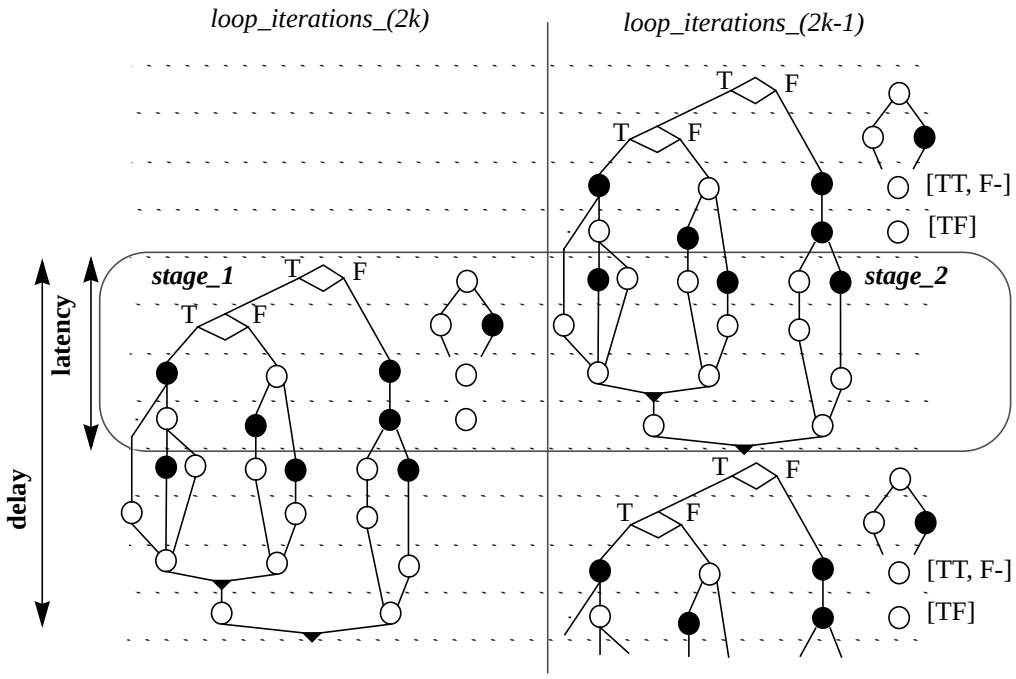


Figure 5.3 Unfolded execution pattern for Kim's example

For example, for time steps $l < step \leq 2l$, the number of paths is potentially $(n_p)^2$, since two iterations co-execute.

Clearly, to have a finite state controller, the number of execution paths must be bounded [3]. This implies limiting state information available to the controller implementing the schedule. At minimum, the state depends on all n_s loop iterations in the pipeline [4].

The unfolded execution of a functionally pipelined version of *Kim* (Figure 3.1, Chapter 3) is shown in Figure 5.3. We assume two adders ("white" operation), one subtracter ("black" operation) and one comparator (single-cycle units assumed).

3. Some compilers use similar constraints to guarantee termination of the scheduling algorithm [3].
4. Increasing the amount of state available for control generation may improve a schedule, but is likely to lead to more complex controllers.

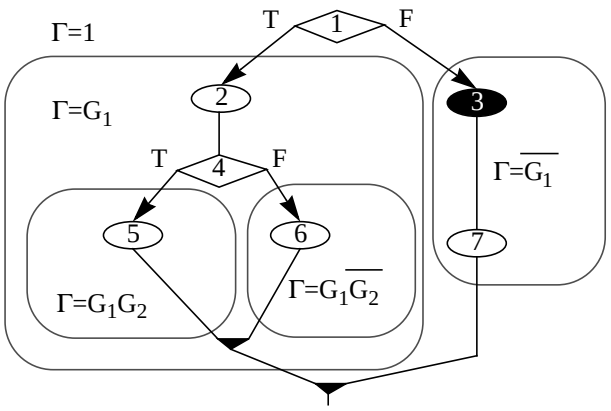


Figure 5.4 Example CDFG to be folded

The example requires 8 cycles on these resources if loop pipelining is not performed [5]. With loop pipelining, a schedule using 2 stages and having latency of 4 (using the same resources) can be found as indicated in the Figure 5.3 (delay remains 8 cycles). One operation is divided as indicated by the values of the guards corresponding to conditionals (*C1*, *C2*). The indicated block in the middle of the figure shows a pipelined loop pattern. Although there are nine control paths, the control is simple since the schedules for the two iterations are independent. In general, this need not be the case: superior schedules may be achieved when a control correlation is introduced among the overlapping iterations.

We now extend conditional resource sharing analysis to the more general case of pipelining of cyclic CDFGs. Consider the CDFG shown in Figure 5.4. Assuming that only one single-cycle resource of each type (comparator, “white”, “black”) is available and that speculative operation execution is not enabled, the CDFG from Figure 5.4 can be scheduled in 4 time steps without loop pipelining. However, latency can be reduced to 2 time steps using three pipeline stages. For

5. Assuming no speculative execution, 8-cycle schedule can be found even using only one single-cycle adder.

simplicity, we assume that the CDFG has to be executed an infinite number of times and that no inter-iteration data dependencies exist. These assumptions do not affect the generality of the approach[6].

Assume that the schedule is to be found using n_s pipeline stages. We specify a bound on the information available to the controller n_i ($n_i \geq n_s$), indicating that the state of the last n_i iterations is preserved and used in decision making. Assume that the CDFG to be scheduled has n_p control paths. Clearly, the bound on the number of distinct control paths grows as $O[(n_p)^{n_i}]$. To accommodate all possible scenarios, guard variables are doubly-indexed. $\mathbf{G_{k,i}}$ stands for "guard corresponding to conditional k in pipeline stage i", where ($1 \leq i \leq n_i$). Index i is called the "**pipe index**". Values of i larger than n_s correspond to loop iterations that left the pipeline. Operations at different pipeline stages correspond to distinct loop iterations. Thus, $G_{k,i}$ corresponds to *any* loop iteration currently present at stage i. Additionally, operation j is guarded by $\Gamma_{j,i}$ (the guard function for operation j at pipeline stage i). The complexity of control representation grows as $n_i n_c$ (n_c is the number of conditionals).

The overlapping iterations are treated as parallel threads of computation, leading to the following resource analysis procedure [7]:

1. For the original CDFG, assign guard variables G_k to the corresponding conditionals and for each operation j compute its guard function Γ_j.

---

6. In the general case, a loop test must be explicit in the CDFG specification and the scheduler has to enforce inter-iteration precedences.

7. Some schedulers first generate a feasible pipelined schedule (in terms of dataflow dependencies) and subsequently resolve resource violations by incremental partial rescheduling [63][89]. Alternatively, the initial non-pipelined schedule can be free of resource violations and the latency is then reduced through incremental operation rotation [22][110].

2. Compute $\Gamma_{j,1}$ by substituting $G_{k,1}$ for each G_k in Γ_j. Resource constraints are evaluated as described for a CDFG without loop folding (Section 5.1).

3.a. If, during scheduling, operation *j* is moved from pipeline stage *i* to pipeline stage (*i*+1), compute $\Gamma_{j,(i+1)}$ by incrementing the pipe indices by 1 for all guard variables in $\Gamma_{j,i}$. Movement of operations that increase the pipe index beyond n_i is not allowed, since this would violate the pre-defined bound n_i.

3.b. If, during scheduling, operation *j* is moved from pipeline stage *i* to stage (*i*-1), compute $\Gamma_{j,(i-1)}$ by decrementing the pipe indices by 1 for all guard variables in $\Gamma_{j,i}$. Operation movement decreasing the pipe index below 1 is illegal, since it would imply non-causal solutions (i.e. control depends on iterations yet to be initiated).

4. Repeat steps 3.a and 3.b for each time step and each pipeline stage. Conditional resource availability is computed as described in Section 5.1.

Steps 3.a and 3.b preserve all inter-iteration and intra-iteration control dependencies. They reflect the fact that overlapping loop iterations flow through the pipeline stages in a synchronous fashion.

We now apply the procedure to the CDFG in Figure 5.4. A feasible schedule using three pipeline stages, achieving latency of 2 is shown in Figure 5.5. Assume $n_i = n_s = 3$ and that stage 1 has been scheduled as shown in Figure 5.5. Since operation 4 is pushed from the first pipeline stage into the second pipeline stage, its new guard function becomes: $\Gamma_{4,2} = G_{1,2}$. If the $B_{1,2}$ constraint at step 1 is evaluated using $\Gamma_{1,1}$ and $\Gamma_{4,2}$ we obtain:

$$B_{1,2}(\Gamma_{1,1}, \Gamma_{4,2}) = \bar{1} + \overline{G_{1,2}} = \overline{G_{1,2}} \qquad \text{(EQ 5.7)}$$

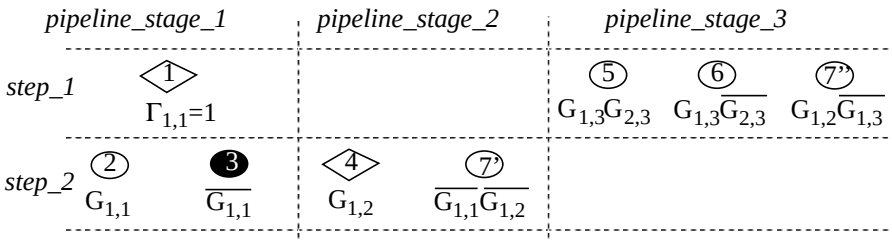


Figure 5.5 Folded CDFG from Figure 5.4

indicating the paths where the resource constraint is not violated. However, the intersection of $B_{1,2}$ and $\Gamma_{4,2}$ is empty (i.e. S_4=0), indicating that operation 4 cannot be scheduled in *step_1*. It is possible to schedule operation 4 in *step_2*, however, since no other comparison is scheduled in that step in pipeline stage 1.

Similarly, operation 7 is guarded by $\Gamma_{7,2} = \overline{G_{1,2}}$ when pushed into stage 2. Although sufficient resources are available, it is clear that operation 7 cannot be scheduled at step 1 if an overall latency of 2 is to be achieved. (Since computation in the first and second pipeline stage are subject to uncorrelated decisions, it can happen that no "white" resources are available for pipeline stage 3 where additional "white" operations have to be scheduled). At step 2:

$$B_{1,2}(\Gamma_{2,1}, \Gamma_{7,2}) = \overline{G_{1,1}} + (\overline{\overline{G_{1,2}}}) = \overline{G_{1,1}} + G_{1,2} \qquad \text{(EQ 5.8)}$$

indicating the paths free of resource violations. Since:

$$S_7 = \Gamma_{7,2} B_{1,2}(\Gamma_{2,1}, \Gamma_{7,2}) = \overline{G_{1,1}}\,\overline{G_{1,2}} \qquad \text{(EQ 5.9)}$$

operation 7 cannot be scheduled at step 2 if the 'T' path is simultaneously taken in the CDFG being executed in the first pipeline stage ($G_{1,1}\overline{G_{1,2}}$) .

However, operation 7 can be split (see Figure 5.5). The guard function of 7' can be set to:

$$\Gamma_{7',2} = S_7 = \overline{G_{1,1}}\,\overline{G_{1,2}} \qquad \text{(EQ 5.10)}$$

Operation 7 has yet to be scheduled on paths:

$$\Gamma_{7'',2} = \Gamma_{7,2} \backslash \Gamma_{7',2} = G_{1,1}\overline{G_{1,2}} \quad \text{(EQ 5.11)}$$

Since this part of operation 7 (7") has to be pushed into pipeline stage 3, its guard function is modified to:

$$\Gamma_{7'',3} = G_{1,2}\overline{G_{1,3}} \quad \text{(EQ 5.12)}$$

During the first step of stage 3, three candidate operations exist: 5, 6 and 7''. If the $B_{k,n}$ constraint is evaluated at step 1 using $\Gamma_{5,3}$, $\Gamma_{6,3}$ and $\Gamma_{7",3}$, we obtain:

$$B_{1,3}(\Gamma_{5,3}, \Gamma_{6,3}, \Gamma_{7'',3}) = 1 \quad \text{(EQ 5.13)}$$

indicating that the resource constraint is satisfied on all paths. Computation of S_j for j=(5, 6, 7") indicates that all operations (5, 6, and 7") can be scheduled at *step_1* (stage 3) and that a feasible schedule has been found.

5.3 Probabilistic interpretation

In a CDFG with n_c conditionals, up to 2^{n_c} control scenarios may occur. Each of these distinct control paths can be represented using a minterm of guard variables. Since the number of minterms covering a Boolean function *f* is typically referred to as *on-set size* of *f*, we define:

$$OnSetSize(1) = 2^{nc} \quad \text{(EQ 5.14)}$$

Assuming that all True/False decisions are equally likely, we offer a probabilistic interpretation of Γ functions:

$$\frac{OnSetSize(\Gamma_j)}{OnSetSize(1)} = P(j) \quad \text{(EQ 5.15)}$$

where *P(j)* indicates the probability that operation *j* will be conditionally executed.

Probability *P(j)* or its variations are frequently used in resource-constrained schedulers to define heuristic priority functions (e.g. [120]). We observe that the

computation of *OnSetSize(f)* amounts to a simple one-pass traversal of an BDD representation of *f*. When the probability of a conditional's outcome is not uniform, behavioral description analysis/simulation can be performed to determine probability values. In such cases, the BDD traversal algorithm for *OnSetSize*(*f*) can be easily modified to take into account individual probabilities $P(G_c)$ [8].

It is also possible to assess the global effects of resource violations using the complement of $B_{k,n}(\Gamma_1,...,\Gamma_n)$:

$$\frac{OnSetSize(\overline{B_{k,n}})}{OnSetSize(1)} \qquad \text{(EQ 5.16)}$$

This ratio indicates the probability of a violation occurrence. Such information is useful for schedulers that resolve resource violations through partial rescheduling.

In Section 6.5 we will present preliminary experiments investigating benefits from applying the proposed conditional resource sharing analysis to software pipelining of cyclic CDGFs.

8. We still assume these probabilities correspond to independent events.

Chapter 6

Experimental Results

The technique described in this thesis was implemented in C++ and executed on a Sun SPARCstation10 with 128 Mbytes of memory. ROBDD package was custom designed. The package implementation follows the approach introduced in [10][1]. Reported CPU times correspond to the complete procedure: CDFG analysis, constraint construction, and all OBDD manipulations leading to the reported results. First, we apply the technique to three typical problem types:

- Section 6.1 presents experimental results of scheduling of acyclic DFGs.
- Section 6.2 demonstrates the ability of our technique to perform loop winding on cyclic DFGs.
- Section 6.3 discuss the scheduling of acyclic CDFGs.

The results are compared to the optimal or best known results. No other work reports competitive results for all three problem types. Subsequently, we perform two additional sets of experiments:

1. “Inverted edges” option was not implemented, however. “Dynamic re-ordering” option was provided, but not used in experiments discussed in Chapter 6.

Table 6.1: EWF experiments

#cycles	17	17	18	18	19	20	20	21	28	28
#adders	3	3	3	2	2	2	2	2	1	1
#multipliers	2(*)	3	1(*)	2	1(*)	2	1(*)	1	1(*)	1
#buses	6	6	6	6	6	4	4	4	4	4
#registers	10	10	10	10	10	10	10	10	10	10
#variables	63	63	97	97	131	165	165	199	437	437
#nodes	82	82	194	209	2,237	2,760	1,905	704	4.9e4	3.2e4
#schedules	18	18	336	18	1.1e4	5.3e4	5,142	2,355	4.3e9	2.6e8
CPU time [s]	0.2	0.2	0.5	0.6	3.4	14.0	12.5	3.5	624.7	391.5
2-cycle multiplier and single-cycle adder except: (*) 2-cycle pipelined multiplier.										

- In Section 6.4, we discuss application of symbolic techniques (both exact and heuristic) to larger DFGs.

- Finally, Section 6.5 presents preliminary experiments investigating benefits from applying conditional resource sharing analysis approach (Chapter 5) to software pipelining of cyclic CDGFs.

6.1 Acyclic DFGs

Table 6.1 summarizes the *elliptic wave filter* (***EWF***) benchmark experiments (See Figure 4.7, Section 4.5.1, for EWF data-flow graph). We found *all* optimal solutions of each instance using BDDs whose size was significantly smaller than $(\#variables)^2$. To reduce the size of partial solutions, an auxiliary set of interior constraints was generated (described in Section 4.4.1). The CPU times are rather moderate for the exact technique generating all optimal solutions. In Section 6.4 we discuss some larger problems (EWF unfolded two and three times, Finite Discrete Cosine Transform, *FDCT* [73]) and demonstrate that efficient symbolic heuristic techniques can treat problems requiring thousands of formulation variables.

Important property of *exact* scheduling techniques is that they are capable of either generating optimal solution(s) or indicating that *no feasible solution* exist for a particular problem instance. On the downside, this requires that an exhaustive search of a complete solution space has to be performed. *Branch-and-bound* techniques are typically employed to speed up the solution space exploration. Unfortunately, problems exist in which it is not possible to derive sufficiently tight bounds. One such example (10-cycle *FDCT* instance with 5 two-cycle non-pipelined multipliers and 3 ALUs) was investigated recently [116]. Due to its very large symmetric search space, this *FDCT* instance is reported to be an extremely hard problem -- both ILP and branch-and-bound techniques take more than 2 CPU hours to reach the infeasibility conclusion. However, we were able to prove its infeasibility in only 100 CPU seconds. This can be explained by the fact that in our technique a very large number of potential solutions are explored in parallel on a time-step-by-time-step basis.

Interior constraints (described in Section 4.4.1) are helpful not only to speed up convergence to a solution set (when one exists), but to improve infeasibility analysis as well. We demonstrate this in Figure 6.1. 54-cycle EWF instance (2-cycle adder, 2-cycle multiplier) introduced in [115] is used as an example. The exact scheduler was initialized with an infeasible execution time of 53 cycles. A logarithm of CPU time elapsed before the infeasibility is detected is indicated as *(log)_CPU_TIME* [s]. A step of iterative construction when the infeasibility conclusion is reached is labeled *INFEASIBLE* [cycle#]. It can be seen that the infeasibility analysis is very fast (less than 0.6 CPU seconds) when interior constraints are applied aggressively (LOOKAHEAD≥40). Almost no penalty is paid for being extremely aggressive (LOOKAHEAD≥50).

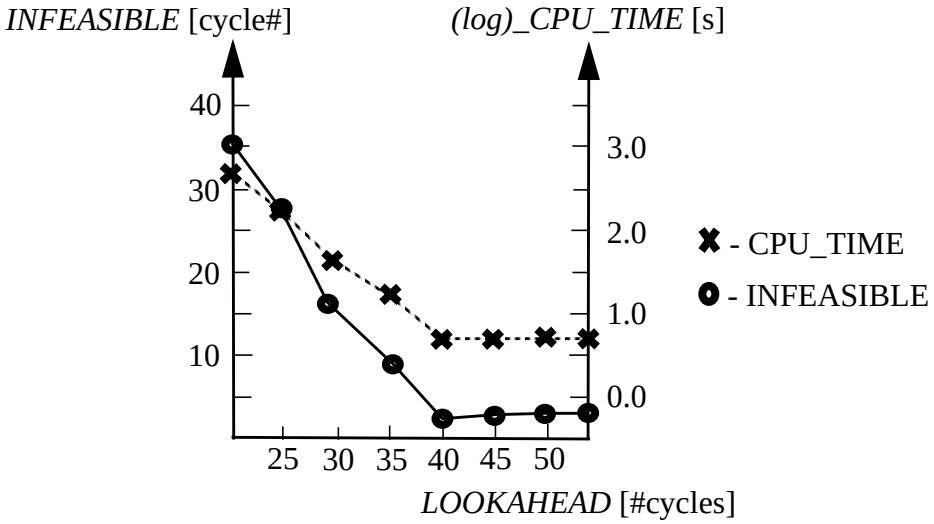


Figure 6.1 Infeasibility detection

6.2 Cyclic DFGs

Loop winding results for *EWF* are indicated in Table 6.2. All optimal schedules, both in terms of latency (iteration interval) and delay (iteration time), are constructed using very moderate computing resources. Several ILP techniques (e.g. [42][49]) report results equivalent to those presented in Table 6.1 and Table 6.2,

Table 6.2: EWF with loop winding

	non-pipelined multiplier				*pipelined multiplier*				
#multipliers	3	2	2	1	3	2	2	1	1
#adders	3	3	2	2	3	3	2	3	2
latency	16	16	17	19	16	16	17	16	17
delay	18	18	19	21	18	18	19	18	19
#variables	97	97	131	199	97	97	131	97	131
#nodes	776	465	689	1,788	799	776	878	258	189
#schedules	2,055	674	108	19,498	2,160	2,055	144	77	19
CPU [s]	4.0	1.2	9.2	7.1	4.4	4.0	9.4	0.5	3.5

with the difference that we provide all optimal schedules. Direct comparison of CPU times is misleading due to machine differences and to the fact that only ILP execution times without preprocessing are typically reported. Similarly, the efficient branch-and-bound technique [116] does not report the time for execution interval analysis.

Table 6.3: Benchmarks with branching

		Maha		*Parker*	*Kim*	*Waka*	*MulT*
#cycles(spec)	longest	5	4	4	6	7	3
	average	3.31	2.25	2.13	5.75	5.0	3.0
#cycles(non_spec)		8	8	8	8	7	4
#adders		1	2	2	2	1	2
#subtracters		1	3	3	1	1	1
#comparators		-	-	-	1	2	1
#variables		65	49	49	71	55	26
#nodes		428	325	220	543	271	116
#traces		15	43	12	124	21	15
CPU time [s]		5.9	3.6	4.7	4.1	2.0	3.3
single-cycle adders, subtracters and comparators assumed							

Table 6.4: Comparison with others: average (longest) path

	Maha		*Parker*	*Kim*	*Waka*	*MulT*
Symbolic	3.31 (5)	2.25 (4)	2.13 (4)	5.75 (6)	5 (7)	3 (3)
TS [47]	3.31 (5)	-	-	-	4.75 (7)	-
CVLS [119]	3.31 (5)	2.38 (4)	2.38 (4)	5.75 (6)	-	2.88 (4)
Kim *et al* [55]	4.62 (8)	-	-	6.25 (7)	4.75 (7)	-

6.3 Acyclic CDFGs

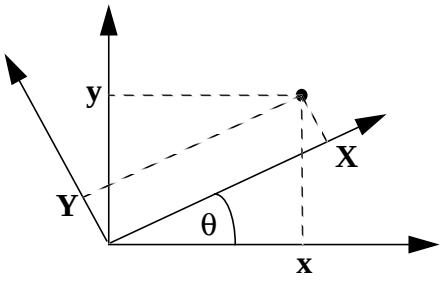
Table 6.3 and Table 6.4 show experimental results for benchmarks exhibiting conditional behavior. The rows *#cycles(spec)* and *#cycles(non_spec)* correspond to scheduling with and without speculative execution using the same set of resources and demonstrate the benefits of performing such code motion. The scheduler terminates when all minimum-latency ensemble schedules are found. The number of cycles for the longest control path is indicated as "*longest*". To compare our results

with schedulers which minimize average path length, a subset of solutions with small average path length is generated in a greedy fashion. Benchmarks *Maha* [86], *Kim* [55] and *Waka* [120] are conditional trees, and *MulT* [119] has two parallel trees. *Parker* is *Maha* with addition *A6* converted into a subtraction (Figure 3.3, Section 3.2).

Our results ("Symbolic") are compared to the best published results. The *Maha* solution with one adder and one subtracter is the same as in [47][119]. Allowing more resources (2 adders, 3 subtracters) an improvement of 0.125 (average path length) is made over the best previous result. In *Parker*, this improvement was 0.25. In most previous work, it is assumed that the comparators incur a small delay within a clock cycle and that the operations following the branch on "True" and "False" paths are mutually exclusive during the *same* cycle. This treatment of the conditionals requires increased cycle time, additional multiplexing, and restricts pipelining of the control. Our results reflect this model in *Maha* and *Parker* only, but this assumption completely eliminates the need for speculative execution in the *Kim* and *Waka* benchmarks. By default, we assume that a single-cycle comparator is used and that its output becomes available for control only in the successive cycle. Even with this assumption, our technique still derives the same result for *Kim* as in [119]. In *Waka* one path is a cycle longer than that reported in [47]. In *MulT* a one cycle *shorter* minimum-latency solution was found by exploiting dynamic scheduling of operations belonging to parallel trees. There is no information on execution times for the results reported in [47][55][119][2].

The *ROTOR* example (Figure 6.2) performs a rotation of coordinate axes by angle θ. This transformation is used in many applications (e.g. graphics applica-

2. More recent implementation of [55] reports CPU times comparable to those listed in Table 6.3 [56].



$$X = \underbrace{x*\cos\theta}_{X'} + \underbrace{y*\sin\theta}_{X''}$$

$$Y = \underbrace{-x*\sin\theta}_{Y'} + \underbrace{y*\cos\theta}_{Y''}$$

```
a = 180-θ;
if (a>=0) {
        b = 90-θ;
        if (b>=0) {
                sinθ = T(θ);
                cosθ = T(b);
        } else {
                sinθ = T(a);
                cosθ = -T(-b);
        }
} else {
        c = 270-θ;
        if (c>=0) {
                sinθ = -T(-a);
                cosθ = -T(c);
        } else {
                sinθ = -T(360-θ);
                cosθ = T(-c);
        }
}
X = x*cosθ + y*sinθ;
Y = -x*sinθ + y*cosθ;
```

Figure 6.2 ROTOR example

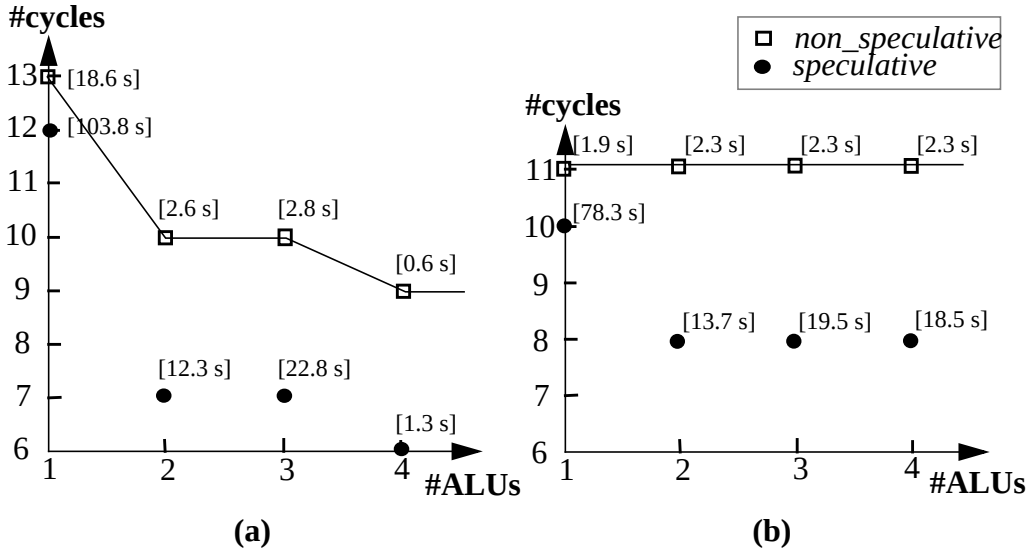
tions and positional control systems). The example requires computation of trigonometric functions (sinθ and cosθ). In high-performance applications, a typical approach is to pre-compute the value of sine and cosine functions and store the sampled values in corresponding tables. However, if high numerical accuracy is required, the size of the storage tends to become rather large. A compromise

approach amounts to storing values for only a quadrant of one trigonometric function (e.g. sine values for arguments $0^\circ \leq \Theta \leq 90^\circ$). It is straightforward to use such a look-up table for obtaining values for both sine and cosine for all possible input arguments ($0^\circ \leq \Theta \leq 360^\circ$).

A pseudocode description of the coordinate rotation using only the first quadrant of the sine function is presented in Figure 6.2. “T(angle)” corresponds to a table read at a location ‘angle’. Similarly, “-T(angle)” corresponds to a table read followed by a negation. We assume that only one single-port look-up table is available and that every ‘read table’ takes one cycle to complete. Although it is possible to simultaneously perform subtraction and comparison of two operands, In the example, we assume *pipelined control* which introduces a two-cycle delay. For example, if operation (a = 180-θ) is executed at step s, result *a* is available at the beginning of step (*s*+1), but control flow is affected by the comparison at the beginning of step (*s*+2) [3].

To simplify interpretation of the results, in Figure 6.3(a) we assume that the available ALUs can perform all arithmetic/logical operations (add, subtract/negate, multiply) in a single cycle. The minimum number of cycles to execute the schedule is presented for cases with and without speculative execution. We observe that, given the same resource constraints, speculative execution enables much faster schedules. In Figure 6.3(b) a more realistic assumption is made. Single-cycle ALUs perform addition and subtraction. Multiplication is performed by two 2-cycle pipelined multipliers. In this case, adding more ALUs cannot improve the performance unless speculative execution is allowed. In Figure 6.3, CPU run-times

3. This corresponds to a “parallel control model” as described in [91]. Data-path operation updates a condition code register at step *s*. The condition code register is inspected at step (*s*+1) and the next state and a new control word are generated. Subsequently, that control word governs data-path activities at step (*s*+2).



- memory constraint: 1 single-port look-up table
- pipelined control delay = 2 cycles
- resource constraints:
  (a) single-cycle ALU (+, -, ∗)
  (b) single-cycle ALU (+, -), 2 two-cycle pipelined multipliers

Figure 6.3 ROTOR experiments

are indicated in brackets. By allowing speculative execution, an average improvement in minimum latency of 25% is achieved using the same resources.

Figure 6.4 shows an 8-cycle ensemble schedule (2 ALUs, Figure 6.3(b)). Operations executed in a speculative fashion are represented using thick lines. If the input angle Θ belongs to the first quadrant, the computation is performed in seven cycles. However, since all ensemble schedules are implicitly encapsulated in an OBDD, the user can search for solutions having other properties. It is relatively straightforward to look for similarities among the traces in order to simplify the control. For example, if the first-quadrant computation takes 8 cycles as well, it is possible to have the same schedule for operations X', X", Y', Y', X and Y for all control paths during the fifth, sixth, seventh and eighth cycles. This sort of design space exploration can be performed without rescheduling the problem instance.

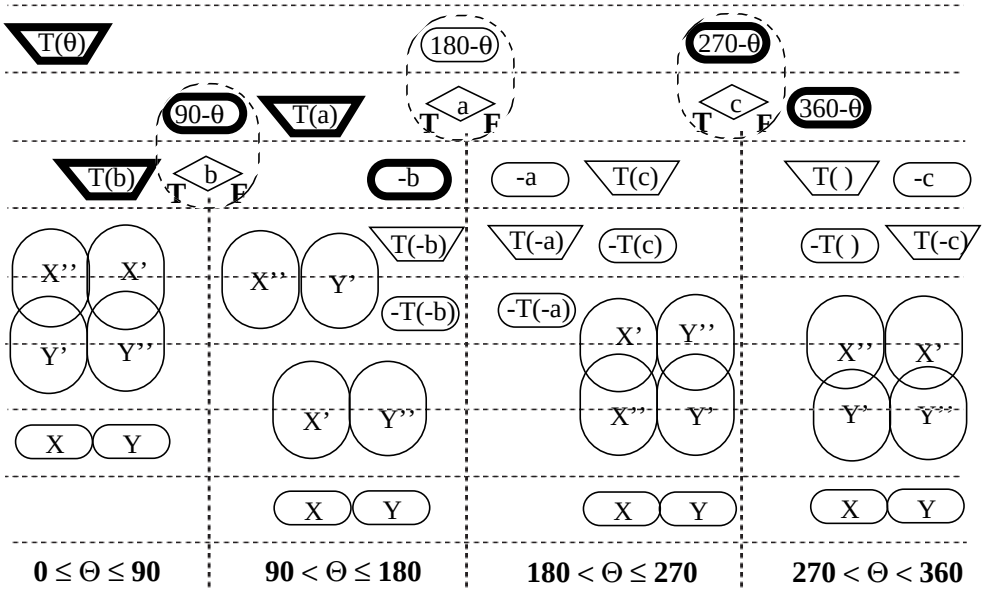


Figure 6.4 8-cycle ROTOR schedule

In Figure 6.5 we introduce the *S2R* example that translates spherical coordinates [R, Θ, Φ] into the Cartesian (rectangular) coordinate values [X, Y, Z]. The problem includes computation of trigonometric functions (as described in the

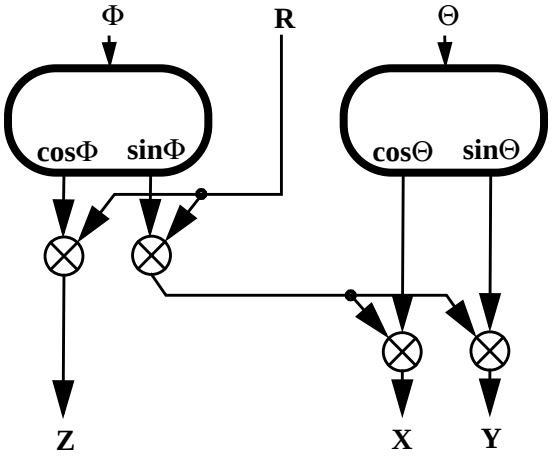


Figure 6.5 S2R example

Table 6.5: S2R experiments

execution_type		*#cycles*	*CPU_time [s]*
speculative	parallel	8	108.8
	serial	10	177.9
non_speculative	parallel	11[a]	56.2
	serial	16[b]	13.5

- memory constraint: 1 single-port look-up table
- pipelined control delay = 2 cycles
- resource constraints: 3 single-cycle ALUs (+,-), 2 two-cycle pipelined multipliers

a. can be achieved with 2 single-cycle ALUs as well (55.2 s)
b. can be achieved with 1 single-cycle ALU as well (12.4 s)

ROTOR example) for two input angles (Θ, Φ). There are 42 operations in the CDFG representation and, if executed in a speculative fashion, as many as 64 execution paths. If a single-port look-up table is used, the scheduling of parallel trees (corresponding to computations for Θ and Φ) has to be done *simultaneously.* This means that the schedule *guarantees* synchronization of the memory accesses without busy/waiting hardware handshaking.

Shown in Table 6.5 (*#cycles*) are *S2R* latencies using one single-port look-up table, 3 ALUs and 2 two-cycle pipelined multipliers. All solutions are exact and correspond to execution with and without speculative execution. In each case, two values are included. An unconstrained version (*"parallel*") allows both trees to be scheduled and executed in parallel. For comparison, we provide the latencies for a *"serial"* version of the problem which imposes an execution order (Φ-tree executed before Θ-tree). The results clearly indicate the benefit from being able to schedule parallel computations in a speculative fashion. Note that none of the results can be further improved by increasing hardware resources.

Table 6.6: Speculative execution model performance

#cycles	*Maha*	*Parker*	*Kim* [a]	*Waka*	*MulT*	*Rotor*	*S2R*
longest chain	4	4	5	7	3	6	8
scheduled	4	4	5	7	3	6	8

a. previously unreported case -- requires 2 adders, 2 subtracters, 1 comparator

6.3.1 Speculative Execution Model Performance

As indicated in Section 3.2.1, the speculative execution model used in the current formulation cannot, in general, guarantee time optimality. In Table 6.6, for all of the benchmarks discussed in Section 6.3, experimental results are shown for both the longest chain of data-flow dependencies and latencies of the scheduled benchmarks. The results are encouraging, since they demonstrate that the restriction in our current speculative execution model did not prohibit, given sufficient resources, achieving theoretically minimum benchmark latencies.

6.4 Larger DFGs

In this section we discuss application of symbolic heuristics (Section 4.4.3) to larger DFGs and perform a performance comparison to the exact approach.

First, in Figure 6.6 and Figure 6.7, we analyze two large instances of EWF benchmark. Both exact and heuristic construction were performed. Two BDD representations were used to generate results: reduced ordered BDDs (ROBDDs [12], see Appendix A) and zero-suppressed BDDs (ZBDDs [75], see Section 4.5.1). Exact results are derived using interior constraints that were applied as aggressively as possible (i.e. all possible lookaheads were allowed at each scheduling step).

Figure 6.6 and Figure 6.7 indicate that utility-based set-heuristics (curves labeled *_heu_robdd* and *_heu_zbdd*, Section 4.4.3) are far superior to exact schedulers both in terms of CPU time and memory requirements, while still finding representative minimum-latency schedules.

When interior constraints are allowed (curves labeled with a prefix *_ic*), the figures indicate controlled growth of the solution in which the intermediate size is never greater than the final size. Although such ideal behavior is not always achievable, our experiments indicate that the use of interior constraints has a dra-

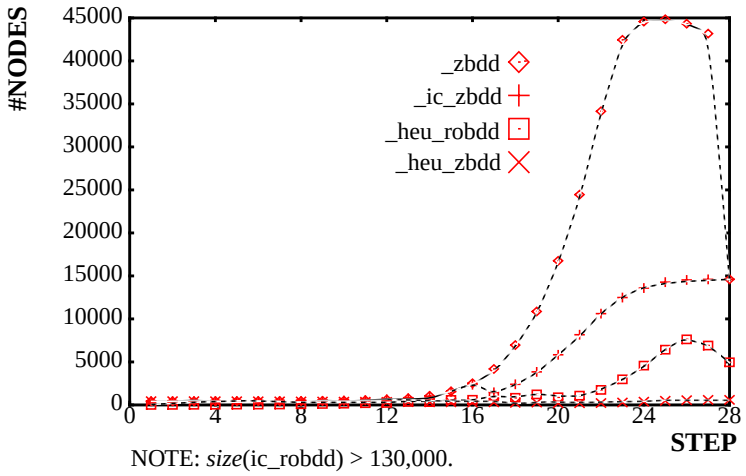


Figure 6.6 28-cycle EWF: exact and heuristic constructions

- *resources*: 1 single-cycle adder, 1 two-cycle multiplier (> 10e+9 solutions)
- *#variables*: 437

_zbdd: exact solution (ZBDD), no interior constraints: ~ 18.5 min

_ic_zbdd: exact solution (ZBDD) built using interior constraints: ~ 9.5 min [1]

_heu_robdd: utility-based set-heuristic solution (OBDD): ~ 17 s [2]

_heu_zbdd: utility-based set-heuristic solution (ZBDD): ~ 102 s

_ic_robdd: exact solution (OBDD) built using interior constraints: ~ 23 min

---

1. For optimal number of registers (10), the size of exact ZBDD solution decreases from ~14.5 to ~3.5 Knodes.
2. ~ 5 s if both utilization and critical path are used as heuristic criteria

matic effect on scheduler efficiency. Without interior constraints, the schedule in Figure 6.7 (labeled *_zbdd*) failed to terminate in several CPU hours.

It is to be expected that heuristics based solely on utilization of the functional unit resources will occasionally produce sub-optimal results in terms of register requirements. For example, if the 28-cycle EWF (Figure 6.6) is scheduled heuristically with no pre-specified register bound, the solution requires at least 13 registers. However, if a register bound of 10 is enforced during the construction, the utility-based heuristic still produces the fastest possible solutions (28 cycles). The

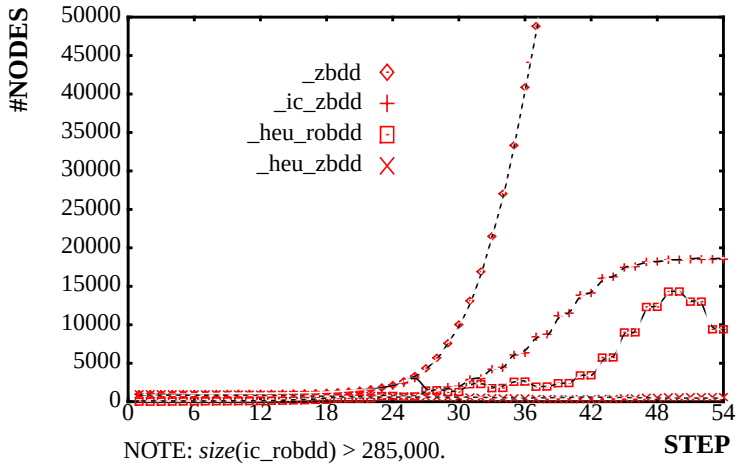


Figure 6.7 54-cycle EWF: exact and heuristic constructions

- *resources*: 1 two-cycle adder, 1 two-cycle multiplier (> 10e+13 solutions)
- *#variables*: 967

_zbdd: exact solution (ZBDD), no interior constraints: could not be constructed

_ic_zbdd: exact solution (ZBDD) built using interior constraints: ~ 1 h [1]

_heu_robdd: utility-based set-heuristic solution (OBDD): ~ 51 s [2]

_heu_zbdd: utility-based set-heuristic solution (ZBDD): ~ 12 min

_ic_robdd: exact solution (OBDD), not constructed (converted from *_ic_zbdd*)

---

1. For optimal number of registers (10), the size of exact ZBDD solution decreases from ~18.5 to ~6.5 Knodes.
2. ~15 s if both utilization and critical path are used as heuristic criteria.

Table 6.7: Robustness analysis of the heuristic scheduler

cycles	upper bound	#vars	*utility-based*		*utility + CP*	
			max # nodes	CPU [s]	max # nodes	CPU [s]
54	54	967	14,328	51	2,210	15
	55	1,001	14,839	52	2,292	16
	56	1,035	15,559	58	2,392	17
	59	1,137	17,151	65	2,616	18
	64	1,307	19,378	81	2,914	20
2-cycle adder, 2-cycle multiplier						

same behavior was observed in the 54-cycle case (Figure 6.7) and further experiments described in Section 6.4.

An accurate estimate of the upper bound on scheduling latency may not be available before scheduling. Unfortunately, the search space increases enormously fast with relaxation of this bound. Set-heuristic scheduling is very robust: the construction pace shows very weak sensitivity to the upper bound used to initialize the scheduler. Although additional constraints are generated (due to an increase in ALAP-ASAP spans for individual operations), the intermediate solution size increases very mildly (Table 6.7). Furthermore, it is very important that the heuristics be robust, since they can be used to derive accurate bounds for the exact schedulers whose run-time efficiency is more sensitive to the bound estimates.

In the rest of this section three larger data-flow graphs are used to investigated:

- ***EWF-2*** (*EWF* unfolded two times, 68 operations, Table 6.8),
- ***EWF-3*** (*EWF* unfolded three times, 102 operations, Table 6.9), and

Table 6.8: EWF-2 experiments

add	mul	bus	**cycles**	optimal	reg [h/e]	#vars	max #nodes	CPU [s]	CPU rel
3	3	6	**33**	yes	11/11	135	178	0.3	0.938
3	$2^{(*)}$	6	**33**	yes	11/11	203	178	0.4	0.930
3	$1^{(*)}$	6	**34**	yes	11/11	203	203	0.8	0.435
3	2	6	**35**	no(34)	11/11	271	291	1.7	0.580
2	$2^{(*)}$	6	**35**	yes	11/11	271	661	2.4	0.171
2	2	6	**35**	yes	11/11	271	639	2.2	0.171
		4	**39**	yes	12/11	543	1,770	10.9	0.004
2	$1^{(*)}$	6	**36**	yes	11/11	339	686	2.2	0.040
		4	**39**	yes	11/11	543	2,064	11.6	0.005
2	1	4	**40**	yes	11/11	611	1,232	9.6	0.005
1	$1^{(*)}$	4	**56**	?	14/ -	1,699	2,603	29.0	-
		2	**68**	?	14/ -	2,515	4,128	69.6	-
1	1	4	**56**	?	14/ -	1,699	2,636	28.8	-
		2	**70**	?	14/ -	2,651	4,403	73.9	-

1-cycle adder, 2-cycle multiplier except: (*) 2-cycle pipelined multiplier

- ***FDCT*** (*Fast Discrete Cosine Transform*, 42 operations, Table 6.10).

For uniformity and the reasons described in Section 4.5.1, all experiments to be presented were run using solely ROBDDs. The results are compared to the *Zone Scheduling* (ZS) [48]. This method subdivides a large problem in zones and solves the subproblems using ILP (integer linear programming) techniques.

EWF-2

To compare our results, the scheduler was run with the same constraints on the number of functional units and buses as in [48]. Register bounds (inputs and outputs included) were identified during the post-processing phase for both heuristic and exact scheduler (column “reg[h/e]”). Maximum size of the partial ROBDD

solution at the end of each iteration step is reported (“max #nodes”), as well as the CPU times (“CPU[s]”) of the heuristic ROBDD scheduler. Column “optimal” indicates whether the result of the heuristic scheduler could be verified by the exact scheduler. A question mark in that column means that we were not able to construct all minimum-latency schedules before exceeding the time limit (one CPU hour). Column “CPU rel” indicates the ratio of the execution time for the heuristic and exact constructions. The CPU times for the exact constructions were generated using interior constraints as aggressively as possible (i.e. all possible lookaheads were allowed at each scheduling step).

Table 6.9: EWF-3 experiments

add	mul	bus	**cycles**	optimal	reg [h/e]	#vars	max #nodes	CPU [s]	CPU rel
3	3	6	**49**	yes	12/12	207	293	0.6	0.800
3	2 (*)	6	**49**	yes	12/12	207	293	0.6	0.923
3	1 (*)	6	**50**	yes	12/12	309	309	1.3	0.398
3	2	6	**52**	no(50)	12/12	513	549	4.4	0.913
2	2 (*)	6	**52**	yes	12/12	513	1,263	5.9	0.041
2	2	6	**52**	yes	12/12	513	1,289	5.7	0.042
		4	**58**	?	13/ -	1,125	3,450	30.1	-
2	1(*)	6	**53**	yes	12/12	615	1,176	4.7	0.010
		4	**58**	?	12/ -	1,125	4,065	31.5	-
2	1	4	**59**	?	12/ -	1,227	2,249	25.1	-
1	1 (*)	4	**84**	?	15/ -	3,777	5,408	86.0	-
		2	**102**	?	15/ -	5,613	7,762	216.2	-
1	1	4	**84**	?	15/ -	3,777	5,408	82.3	-
		2	**105**	?	15/ -	5,919	8,010	233.2	-

1-cycle adder, 2-cycle multiplier except: (*) 2-cycle pipelined multiplier

EWF-3

Benchmark instances with up to 615 variables were solved exactly, and up to 5,919 variables (105-cycle case) heuristically. The heuristic failed in one case to find the optimal execution time (however, as in the case of *EWF-2*, that problem instance was solved exactly). *EWF-3* results were not provided by [48] [4].

Pre-specified register bounds can be used during the construction to minimize the number of registers needed in the heuristically scheduled results. We ran the heuristics with fixed register bounds of 11 (all *EWF-2* instances) and 12 (all *EWF-*

4. To our knowledge, the only reference to this problem is in [41], where a result for the instance with 1 pipelined multiplier and 3 adders is presented. There is no information on the number of registers and buses.

Table 6.10: FDCT experiments

add	sub	mul	bus	**cycles [*our*/ ZS]**	opt.	reg [h/e]	#vars	max #nodes	CPU [s]	CPU rel
2	2	2	10	**10/10**	yes	11/10	251	1,490	32.9	0.332
2	2	2 (*)	10	**11/ -**	yes	9/9	229	2,252	26.2	0.117
1	1	2	8	**13/14**	yes	12/11	377	3,988	26.1	0.018
1	1	2 (*)	8	**14/ -**	?	11/ -	355	4,451	22.7	-
1	1	1	6	**18/20**	yes	11/10	587	12,340	76.0	0.061
1	1	1	4	**18/ -**	yes	11/10	587	5,486	40.3	0.054
1	1	1 (*)	6	**19/ -**	yes	10/9	565	15,346	107.4	0.128
1	1	1 (*)	4	**19/ -**	yes	10/9	565	7,216	60.5	0.114

single-cycle units assumed except: (*) 2-cycle pipelined multiplier

3 instances) and in all cases the solutions that required the same number of cycles as those presented in Table 6.8 and Table 6.9 were found. However, making an accurate estimate on the register bound is a difficult problem, and a further work to directly incorporate a register cost (not just a bound on the number) is needed. This is important for exact scheduling as well, since register constraints can dramatically reduce the solution space. For example, using ROBDDs, all schedules for the 28-cycle EWF with 10 registers can be found in 391.5 CPU seconds (approximately 3.5 times faster than the unconstrained version, Figure 6.6).

FDCT

Although *FDCT* has a relatively moderate number of operations, we include it in this report for two reasons: *(i)* it comes from a practical application, and *(ii)* due to its highly symmetric nature (which should lead to huge solution sets), it is likely to be rather challenging task for exact schedulers. Table 6.10 presents the results for some larger FDCT instances. As before, the scheduler was run with the same constraints on the number of functional units and buses as in [48]. Register bounds

were determined during the post-processing phase for the approach which used both utilization and critical path heuristic. To constrain the solution space, the exact scheduler was run using a pre-specified register bound. The heuristic found the fastest schedules in all cases and performed quite well in terms of the number of registers (typically, off by 1). As can be seen in rows 3 and 5, our heuristic scheduler outperforms *ZS*. This can be explained by the fact that we preserve a complete set of solutions satisfying the heuristic criteria. Even for 2-cycle pipelined multipliers our results are equal (row 4) or better (row 7) that those reported for single-cycle units in *ZS*. Moreover, in rows 6 and 8, we indicate that the problem can be solved with the reduced number of buses (4 instead of 6). The *FDCT* instance with 1 adder, 1 subtracter, 1 pipelined 2-cycle multiplier and 4 buses (row 8) is frequently used to evaluate scheduling results for functional pipelining. However, to our knowledge, the best reported results so far required latency (iteration interval) of 20 cycles [63]. One randomly selected optimal 19-cycle schedule is shown in Figure 6.8.

6.5 Cyclic CDFGs

Two types of experiments are performed. First, we wish to investigate the benefits of exploiting conditional resource sharing. Table 6.11 summarizes results for three examples: the *VerySmall* (Figure 5.4), *Kim* (Figure 3.1, Chapter 3), and *SC* (Figure 6.9, [110]). As assumed in the previous sections, *VerySmall* uses 1 resource of each type (add, subtract, compare) and *Kim* uses 2 adders, 1 subtracter and 1 comparator. *SC* schedule (using 1 multiplier and 2 ALUs) is shown in Figure 6.9. Cycles 3 through 8 form a repetitive pattern that can be pipelined [5].

5. Solution in [110] has latency of 6 as well, but uses 4 pipeline stages.

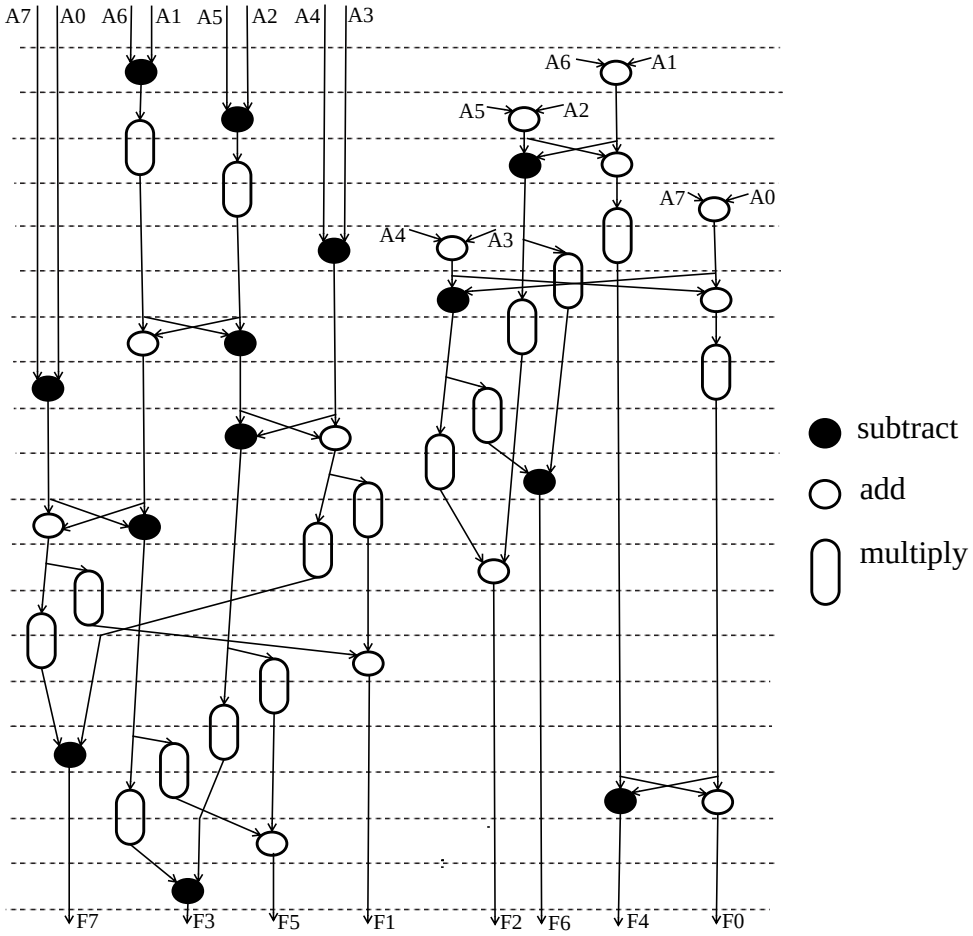
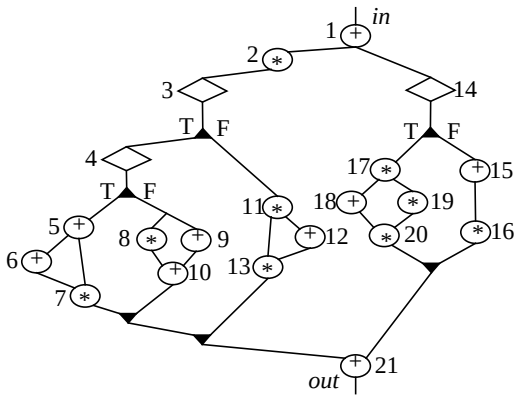


Figure 6.8 19-cycle FDCT with pipelined multiplier

For all examples, we present three results for the same resource bounds. *Original* corresponds to CDFGs without unrolling and pipelining. *Unrolled* corresponds to the unrolled versions of a CDFG (no pipelining), while loop pipelining is allowed in *pipelined*. The *original* and *unrolled* results are obtained using exact symbolic techniques [92][94]. The *pipelined* results were generated in a semi-automated fashion. Essentially, symbolic techniques can be extended to solve a relaxed version of the conditional pipelining by adding some necessary conditions for existence of a repetitive pattern in a schedule of the unrolled loop [6].



cycle:	*pipeline stage:* 0						1
1	1						
2	2, 14						
3	3, 15			3, 17			
4	11	4, 16		11, 18	4, 18, 19		
5	12, 16	8, 9	5	12, 19	8, 9	5	
6	13	10	6	20	10, 20	6, 20	
7	21	21	7	13	21	7	1
8			21	21		21	2, 14
path [3,4,14]:	[F,-,F]	[T,F,F]	[T,T,F]	[F,-,T]	[T,F,T]	[T,T,T]	[-,-,-]

Figure 6.9 SC example and its schedule

All of the presented results were generated using such an approach, but they were manually verified for potential inter-iteration dependency violations. However, although no claim on optimality of the *pipelined* results can be made, Table 6.11 shows that systematic treatment of resource sharing can expose additional operation-level parallelism even in cases when the loop body exhibits conditional behavior.

To investigate the computational and storage overhead of the approach, *“pipelined”* results were verified for potential resource constraint violations. The

6. The basic idea is similar to the GURPR* compiler [9][112].

Table 6.11: Throughput comparisons

example		*#overlapped iterations*	*latency* [cycles]	*delay* [cycles]	*throughput* [1/cycles]
VerySmall	original	1	4	4	0.250
	unrolled	3	7	7	0.429
	pipelined	3	2	5	0.500
Kim	original	1	8	8	0.125
	unrolled	2	11	11	0.182
	pipelined	2	4	8	0.250
SC	original	1	8	8	0.125
	unrolled	2	14	14	0.143
	pipelined	2	6	8	0.167

overhead due to guard variables and functions is very small: 14 OBDD nodes (*Kim*), 18 nodes (*VerySmall*), and 24 nodes (*SC*). In all examples, verification of the resource bounds took less than 0.03 CPU seconds.

Preliminary experimental results form Section 6.5 are encouraging. In the future, a scheduler based on the presented concepts should be implemented. This requires that several additional issues be addressed (e.g. node unification, incremental recalculation of Γ functions when conditionals are rescheduled, timing model for operation chaining, etc.).

Chapter 7

Discussion

7.1 Summary

We described a symbolic formulation that allows speculative operation execution and exact resource-constrained scheduling of arbitrary forward-branching control/data paths. To our knowledge, no other work has been reported on exact techniques supporting speculative execution. An advantage of the formulation is that there is no need to explicitly describe freedom present in the input CDFG description. The execution order of conditionals is not pre-determined and is dynamically resolved allowing gains in scheduling quality. To allow a systematic treatment of the problem, a flexible control representation based on guard variables, guard functions, and traces is introduced. The trace validation algorithm is proposed to enforce causality and completeness of the solution set. An iterative construction method is presented along with benchmark results. The results demonstrate the ability of the technique to efficiently exploit operation-level parallelism implicit in the input description.

The presented techniques provide a closed-form solution set in which all satisfying schedules are encapsulated in a compressed BDD-based representation. This solution format greatly increases the flexibility of the synthesis task by

enabling incremental incorporation of additional constraints and by supporting solution space exploration without the need for rescheduling. Assume that a behavior *B* has to be synthesized to satisfy some desired throughput *T* and that a designer uses a pre-designed and tested data-path *D* as a starting point for implementing a hardware solution. Scheduling B using *D*'s resources (e.g. 2 adders, 1 multiplier, 32-entry register file) produces **S**, a solution set encapsulating all schedules meeting throughput *T.* Once this is done, the designer can incrementally inspect **S** to determine whether *D* can be further simplified (e.g whether the number of adders or registers can be reduced). Similarly, the designer can look for the schedules with some other specific properties. Assume that *B* prescribes a data-flow precedence between two operations *a* and *b.* Designer can then ask the following question: "Is it possible to find a schedule meeting the performance constraint *T* such that *b* is scheduled at least 2 cycles after *a*?". (Such a property can be useful, for example, to simplify design of some interface circuitry, relax clock cycle requirement imposed by layout/interconnect, or allow a use of a slower unit on which *a* is to be performed.) Instead of re-running the complete scheduling procedure, all of the issues raised above can be exactly resolved by incrementally applying additional constraints to the initial solution **S**.

7.2 Future Research Avenues

Despite the advantages summarized in Section 7.1, to make our novel approach more applicable to a wider variety of HLS applications, numerous open research problems deserve further attention. In the following sections, we analyze some of these problems and make an attempt to assess their complexity and possibly suggest some answers.

7.2.1 Complex Operation Mapping

In practice, a designer frequently has an opportunity (and a difficult task) to evaluate trade-offs related to a selection of a functional unit type(s) to use. For example, given a target clock cycle, addition operations can be mapped to a fast single-cycle adder or to a slower (and typically smaller) 2-cycle hardware implementation. In such cases, the formulation presented in this thesis can be extended by triple-indexing operation variables: $C_{s,j,t}$ corresponds to an instance of operation j executing on a functional unit t at time step s. Formulation constraints discussed in Section 3.3 (Chapter 3) have to be modified to exhibit the fact that operation j can be scheduled using different resource types. For example, the uniqueness constraint (Section 3.3.1, Chapter 3) has to include all of the variables corresponding to operation j regardless of a function unit type t.

However, this problem becomes extremely difficult if operation chaining is allowed. For example, assume that a clock period is 40 ns, that a fast adder has a delay of 10 ns and a slow adder has a delay of 25 ns. Remember that operation chaining can be accommodated in our technique by adding precedence constraints between the operations that cannot be chained. (In the example above, two data-dependent additions cannot both use fast adders during the same cycle.) If the mapping from an operation type to a functional unit type is unique, such precedence constraint can be derived by a CDFG traversal in a straightforward fashion. However, if operation mapping is more complex, both the number of constraints as well as the time complexity of the procedure used to derive such constraints increases drastically. Currently, we are not aware of a procedure to perform this task in both a systematic as well as efficient fashion.

7.2.2 Generalized Speculative Execution Model

As indicated in Section 3.2 (Chapter 3), the current speculative execution model imposes a restriction that operations following the join node cannot be scheduled before the corresponding conditional is resolved. This essentially means that at most one instance of a particular operation can exist on any trace. As indicated, in general, this model cannot guarantee time optimal scheduling.

To allow multiple operation instances per trace, two approaches could be considered. A seemingly straightforward extension would introduce a new index for each operations: instead of $C_{s,j}$ we could use $C_{s,j,p}$ corresponding to operation *j*'s instance on path *p* at time step *s*. Unfortunately, aside form re-formulation issues, this approach can result in the exponential increase of the number of formulation variables (O[(*#cycles*)(*#ops*)($2^{(\#cond)}$)], see Section 3.7, Chapter 3) compared to the current model and is unlikely to be efficient even using the BDD-based manipulations.

Another possibility is to first perform a resource-constrained scheduling of all individual control paths with all control dependencies removed and every operation *j* scheduled only on control paths covered by the corresponding Γ_j function [1]. Obviously, selecting the shortest execution instances for all possible control paths produces a time optimal ensemble schedule. Unfortunately, as shown in Figure 3.7 (Chapter 3), resource constraints met on individual control paths are likely to be violated in the ensemble schedule. To properly interpret resource usage, a more powerful version of Trace Validation algorithm has to be developed. This, however, may be a very difficult task, since in the approach discussed in this paragraph, original notion of trace as an execution instance for a particular control

1. As in the current implementation, this can be done simultaneously using the implicit BDD representation.

path is modified. For example, on a trace corresponding to path *p*, pre-executed operations from paths other than *p* may not be seen (again, see Figure 3.7). Moreover, it can happen that the same operation is redundantly scheduled at different time steps on two different control paths, even if these two paths are indistinguishable in the ensemble schedule at the corresponding time steps. Efficient solution for the above mentioned issues is an open research problem.

7.2.3 General Forms of Cyclic Control

In the current formulation, cyclic CDFGs are handled by loop breaking or loop unrolling. For cyclic DFGs, additional optimizations (functional pipelining, loop winding) are available. However, the most general case of multi-rate parallel loops is not dealt with in the current formulation. Such behavior can result from parallelism exposing transformations or the need to schedule a behavior of interacting FSMs.

It is not clear how the techniques presented in this thesis can be expanded to include such cases. The primary difficulty is that a notion of time is “linear” (i.e. not related to FSM states) in our current formulation. On the other hand, we are not aware of any FSM-based model capable of formally incorporating speculative execution.

7.2.4 CDFG Scheduling Heuristics

In Section 6.4, we demonstrated that efficient, near-optimal symbolic heuristics can be developed for larger DFGs. Development of symbolic set-based heuristics for CDFGs is more challenging task, however. In particular, speculative operation execution affects both efficiency and quality of such heuristics. Efficiency is affected by the fact that a very large number of operations might have to be considered for speculative execution at every scheduling step. Thus, in cases

exhibiting complex conditional behavior, partial solutions may grow prohibitively large. Additionally, quality of some global iterative heuristic can be affected if operations belonging to all control paths are treated uniformly at each scheduling step. More promising approach to dealing with larger CDFGs is to apply heuristics (similar to those described in Section 6.4) to individual traces. Similar to well-known compilation techniques ([35][37][70]), the priorities of individual traces can be derived using a profiling information, and the trace validation procedure can be used to maintain the ensemble schedule consistency without the need for a complex book-keeping.

7.2.5 Tightening of Operation Bounds

To further improve the efficiency, additional work is needed to identify tighter operation bounds for the control-dominated case. In the current formulation, very conservative *as-soon-as-possible* and *as-late-as-possible* bounds assuming infinite resources are used. Such bounds are rather loose when speculative operation execution is allowed. Interior constraints (Section 4.4.1, Chapter 4) can be applied to individual control paths, but aside of the critical path(s), they have a rather limited effect.

7.2.6 Lower Level Hardware Implementation Issues

Although the techniques presented in this thesis can cope with certain very difficult issues in control-dependent resource-constrained scheduling, they have a rather global view of a data-path. The current implementation does not deal with detailed layout models, but could possibly be incorporated in feedback-driven closed-loop systems such as [11][58]. In a recent work [78][79], it was demonstrated that symbolic techniques can efficiently and systematically handle numerous hardware issues in existing data-paths (e.g. detailed interconnect

modeling, latches providing a temporary storage within a data-path, timing evaluation etc.).

Furthermore, in this thesis, the cost of controller implementing a particular schedule was not considered (e.g. FSM cycle time, implementation area, serial/ parallel/pipelined controller implementation [11][46][91]). Production-based *Clairvoyant* system [109] generates very fast and area-efficient controllers using non-minimal state encodings. However, other than preserving a prescribed input/ output behavior, Clairvoyant does not perform operation scheduling. It would be very useful to investigate application of techniques described in this thesis to optimization of cycle time and resource usage in the Clairvoyant system.

Bibliography

1. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1988.

2. A. Aiken and A. Nikolau, "Optimal Loop Parallelization", *Proc. ACM SIGPLAN'88 Conf. Programming Language Design and Implementation*, pp. 308-317, June 1988.

3. A. Aiken, A. Nikolau, and S. Novack, "Resource-Constrained Software Pipelining", *IEEE Trans. Distributed and Parallel Systems*, vol. 6, no. 12, pp. 1248-1270, Dec. 1995.

4. S. B. Akers, "Binary Decision Diagrams", *IEEE Trans. Computers*, vol. C-27, no. 6, pp. 509-516, June 1978.

5. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence", *Proc. 10th Ann. ACM Symp. Principles of Programming Languages*, pp. 177-189, Jan. 1983.

6. P. Ashar and M. Cheong, "Efficient Breadth-First Manipulation of Binary Decision Diagrams", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 622-627, 1994.

7. R. I. Bahar *et al.*, "Algebraic Decision Diagrams and their Applications", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 188-191, 1993.

8. R. A. Bergamaschi, R. Camposano, and M. Payer, "Allocation Algorithms Based on Path Analysis", *Integration, the VLSI Journal*, vol.13, no.3, pp. 283-299, Sept. 1992.

9. J. W. Bockhaus, "An Implementation of GURPR*: A Software Pipelining Algorithm", Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

10. K. S. Brace, R. L. Rudell, R. E. Bryant, "Efficient Implementation of a BDD package", *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 40-45, 1990.

11. F. Brewer and D. Gajski, "Chippe: A System for Constraint Driven Behavioral Synthesis", *IEEE Trans. CAD*, vol. 9, no. 7, pp. 681-695, July 1990.

12. R. E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation”, *IEEE Trans. Computers,* vol. C-35, no. 8, pp. 677-691, Aug. 1986.

13. R. E. Bryant, “Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams”, *ACM Computing Surveys*, Vol. 24, No. 3, pp. 293-318, Sep. 1992.

14. R. E. Bryant and Y.-A. Chen, “Verification of Arithmetic Circuits with Binary Moment Diagrams”, *Proc. 32th ACM/IEEE Design Automation Conf.*, pp. 535-541, 1995.

15. R. E. Bryant, “Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification”, *Proc. Int. Conf. Computer-Aided Design*, pp. 236-243, 1995.

16. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, “Symbolic Model Checking for Sequential Circuit Verification”, *IEEE Trans. CAD/ICAS*, vol. 13, no. 4, pp. 401-424, Apr. 1994.

17. G. A. Chaitin, M. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, “Register Allocation via Coloring”, *Computer Languages*, vol. 6, no. 1, pp.47-57, 1981.

18. R. Camposano, “Path-Based Scheduling for Synthesis”, *IEEE Trans. CAD/ICAS*, vol. 10, no. 1, pp. 85-93, Jan. 1991.

19. P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, “The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors”, *IEEE Trans. Computers*, vol. 44, no. 3, pp. 353-370, Mar. 1995.

20. P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, “Three Architectural Models for Compiler-Controlled Speculative Execution”, *IEEE Trans. Computers*, vol. 44, no. 4, pp. 481-494, Apr. 1995.

21. A.E. Charlesworth, “An Approach to Scientific Array Processing: The Architectural Design of the AP-120b/FPS-164 Family”, *IEEE Computer*, vol. 14, no. 9, pp. 18-27, Sep. 1981.

22. L.-F. Chao, A. LaPaugh, and E. H.-M. Sha, “Rotation Scheduling: A Loop Pipelining Algorithm”, *Proc. 30th Design Automation Conf.*, pp. 566-572, 1993.

23. H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi, “Algorithms for Approximate FSM Traversal”, *Proc. 30st ACM/IEEE Design Automation Conf.*, pp. 25-30, 1993.

24. E. M. Clarke, M. Fujita, P. C. McGeer, K. McMillan, J. C.-Y. Yang, and X. Zhao, “Multi-Terminal Binary Decision Diagrams: An Efficient Data-Structure for Matrix Representation”, *Int. Workshop on Logic Synthesis*, pp. 6a1-

6a15, 1993.

25. R.J. Cloutier, and D.E. Thomas, “The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm”, *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 71-76, 1990.

26. C. N. Coelho Jr. and G. De Micheli, “Dynamic Scheduling and Synchronization Synthesis of Concurrent Digital Systems under System-Level Constraints”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 175-181, 1994.

27. O. Coudert, C. Berthet, and J. C. Madre, “Verification of Synchronous Sequential Machines Based on Symbolic Execution”, *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, pp. 365-373, Grenoble, France, 1989.

28. O. Coudert, and J. C. Madre. “A Unified Framework for the Formal Verification of Sequential Circuits,” *Proc. Int. Conf. Computer-Aided Design*, pp. 126-129, 1990.

29. O. Coudert, “Two-level Logic Minimization: An Overview”, *Integration, the VLSI journal*, 17-2, pp. 97-140, Oct. 1994.

30. S. Davidson *et al.*, “Some Experiments in Local Microcode Compaction for Horizontal Machines”, *IEEE Trans. Computers*, vol. c-30, no. 7, pp. 460-477, July 1981.

31. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.

32. J. C. Denhert and R. A. Towle, “Compiling for the Cydra 5”, *J. Supercomputing*, vol. 7, no. 1, pp. 181-227, Jan. 1993.

33. K. Ebcioglu and T. Nakatani, “A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture”, *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nikolau, and D. Padua (*Eds.*), Pitman/The MIT Press, pp. 213-229, 1990.

34. J. H. Edmondson, P. Rubenfeld, R. Preston, and V. Rajagopalan, “Superscalar Instruction Execution in the 21164 Alpha Microprocessor”, *IEEE Micro*, pp. 33-43, Apr. 1995.

35. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, 1986.

36. J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The Program Dependence Graph and Its Use in Optimization”, *ACM Trans. Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, July 1987.

37. J. A. Fisher, “Trace Scheduling: A Technique for Global Microcode Com-

paction", *IEEE Trans. Computers*, vol. C-30, no. 7, pp. 478-490, July 1981.

38. J. A. Fisher, *Global Code Generation for Instruction-Level Parallelism: Trace scheduling-2*, Hewlett Packard Laboratories Technical Report HPL-93-43, June 1993.

39. S. J. Friedman and K. J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams", *IEEE Trans. Computers*, vol. 39, no. 5, pp. 710-713, May 1990.

40. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., 1979.

41. C. H. Gebotys and M. I. Elmasry, "Global Optimization Approach for Architectural Synthesis", *IEEE Trans. CAD/ICAS*, vol. 12, no. 9, pp. 1266-1278, Sep. 1993.

42. C. H. Gebotys, "Throughput Optimized Architectural Synthesis", *IEEE Trans. VLSI Systems*, vol. 1, no. 3, pp. 254-261, Sep. 1993.

43. E. Girczyc, "Loop Winding -- A Data Flow Approach to Functional Pipelining", *Proc. ISCAS*, pp. 382-385, 1987.

44. K. Hamaguchi, A. Morita, and S. Yajima, "Efficient Construction of Binary Moment Diagrams for Verifying Arithmetic Circuits", *Proc. Int. Conf. Computer-Aided Design*, pp. 78-82, 1995.

45. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.

46. S. C.-Y. Huang and W. Wolf, "Performance-Driven Synthesis in Controller-Datapath Systems", *IEEE Trans. VLSI Systems*, vol. 2, no. 1, pp. 68-80, March 1994.

47. S. H. Huang, Y. L. Jeang, C. T. Hwang, Y. C. Hsu, and J. F. Wang. "A Tree-Based Scheduling Algorithm for Control Dominated Circuits", *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 578-582, 1993.

48. C.-T. Hwang and Y.-C. Hsu, "Zone Scheduling", *IEEE Trans. CAD/ICAS*, vol.12, no.7, pp. 926-934, July 1993.

49. C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis", *IEEE Trans. CAD/ICAS*, vol. 10, no. 4, pp. 464-475, Apr. 1991.

50. W.-M. Hwu et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *J. Supercomputing*, vol. 7, no. 1, pp. 229-248, Jan. 1993.

51. S.-W. Jeong and F. Somenzi, "A New Algorithms for the Binate Covering

Problem and its Application to the Minimization of Boolean Relations", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 417-420, 1992.

52. M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.

53. H.-P. Juan, V. Chaiyakul, and D.D. Gajski, "Condition Graphs for High-Quality Behavioral Synthesis", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 170-174, 1994.

54. T. Y. K. Kam and R. K. Brayton, *Multi-valued Decision Diagrams*, Memo. no. UCB/ERL M90/125, UC Berkeley, Dec. 1990.

55. T. Kim, J. W. S. Liu, and C. L. Liu, "A Scheduling Algorithm for Conditional Resource Sharing", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 84-87, 1991.

56. T. Kim, N. Yonezava, J. W. S. Liu, and C. L. Liu, "A Scheduling Algorithm for Conditional Resource Sharing — A Hierarchical Reduction Approach", *IEEE Trans. CAD/ICAS*, vol. 13, no. 4, pp. 425-438, Apr. 1994.

57. S. Kimura, "Residue BDD and Its Application to the Verification of Arithmetic Circuits", *Proc. 32th ACM/IEEE Design Automation Conf.*, pp. 542-545, 1995.

58. D. W. Knapp, "Fasolt: A Program for Feedback-Driven Data-Path Optimization", *IEEE Trans. CAD*, vol. 11, no. 6, pp. 677-695, June 1992.

59. H. Komi, S. Yamada, and K. Fukunaga, "A Scheduling Method by Stepwise Expansion in High-Level Synthesis", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 234-237, 1992.

60. D. Ku, G. De Micheli, "Relative Scheduling under Timing Constraints", *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 59-64, 1990.

61. Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula, "EVBDD-Based Algorithms for Integer Linear Programming, Spectral Transformation, and Function Decomposition", *IEEE Trans. CAD/ICAS*, vol. 13, no. 8, pp. 959-975, Aug. 1994.

62. M. Lam, *A Systolic Array Optimizing Compiler*, Kluwer Academic Publishers, 1989.

63. T-.F. Lee, A. C.-H. Wu, Y.-L. Lin, and D. D. Gajski, "An Effective Methodology for Functional Pipelining", *IEEE Trans. CAD/ICAS*, vol. 13, no. 34, pp. 439-450, Apr. 1994.

64. C.E. Leiserson, F.M. Rose, J.B. Saxe, "Optimizing Synchronous Circuits by Retiming", *Proc. Third Conf. VLSI*, 1983.

65. H.-T. Liaw and C.-S, Lin, "On OBDD-Representation of General Boolean

Functions", *IEEE Trans. Computers*, vol. 41, no. 6, pp. 661-664, June 1992.

66. B. Lin, *Synthesis of VLSI Designs with Symbolic Techniques*, PhD thesis, memo. no. UCB/ERL M91/105, UC Berkeley, Nov. 1991.

67. B. Lin and S. Devadas, "Synthesis of Hazard-Free Multi-level Logic under Multiple-Input Changes from Binary Decision Diagrams", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 542-549, 1994.

68. D. Lobo and B. M. Pangrle, "Redundant Operation Creation: A Scheduling Optimization Technique", *Proc. 28st ACM/IEEE Design Automation Conf.*, pp. 775-778, 1991.

69. T. Ly, D. Knapp, R. Miller, and D. MacMillen, "Scheduling using Behavioral Templates", *Proc. 32th ACM/IEEE Design Automation Conf.*, pp. 101-106, 1995.

70. P. G. Lowney et al., "The Multiflow Trace Scheduling Compiler", *J. Supercomputing*, vol. 7, no. 1, pp. 51-142, Jan. 1993.

71. S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu, "A Comparison of Full and Partial Predicated Execution Support for ILP Processors", *Proc. 22th Int. Symp. Computer Architecture*, pp. 138-150, June 1995.

72. S. Malik, A. R. Wang, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 6-9, 1988.

73. D. J. Mallon and P. B. Denyer, "A New Approach To Pipeline Optimisation", *Proc. European Design Automation Conf.*, pp. 83-87, 1990.

74. M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems", *Proc. IEEE*, vol. 78, no. 2, pp. 301-318, Feb. 1990.

75. S.-I. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems", *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 272-277, 1993.

76. S.-I. Minato, "BDD-Based Manipulation of Polynomials and Its Applications", *Proc. Intl. Workshop on Logic Synthesis*, pp. 5.31-5.43, 1995.

77. S.-I. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publishers, 1995.

78. C. Monahan and F. Brewer, "Symbolic Modeling and Evaluation of Data Paths", *Proc. 32th ACM/IEEE Design Automation Conf.*, pp. 389-394, 1995.

79. C. Monahan and F. Brewer, "STEM: Concurrent Analysis Tool for Data Path

Timing Optimization", *Proc. 33th ACM/IEEE Design Automation Conf.*, to appear.

80. S.-M. Moon and K. Ebcioglu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLWI Processors", *Proc. 25th Ann. Int. Symp. Microarchitecture*, pp. 55-71, 1992.

81. A. Nikolau and R. Potasman, "Incremental Tree Height Reduction For High Level Synthesis", *Proc. 28st ACM/IEEE Design Automation Conf.*, pp. 770-774, 1991.

82. S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 628-631, 1994.

83. S. Panda and F. Somenzi, "Who Are the Variables in Your Neighborhood", *Proc. Int. Conf. Computer-Aided Design*, pp. 74-77, 1995.

84. B. M. Pangrle and D. D. Gajski, "Design Tools for Intelligent Silicon Compilation", *IEEE Trans. CAD*, vol. cad-6, no. 6, pp. 1098-1112, Nov. 1987.

85. N. L. Passos and E. H.-M. Sha, "Push-Up Scheduling: Optimal Polynomial-Time Resource-Constrained Scheduling for Multi-Dimensional Applications", *Proc. Int. Conf. Computer-Aided Design*, pp. 588-591, 1995.

86. A. C. Parker, J. T. Pizarro, and M. Mliner, "MAHA: A Program for Datapath Synthesis", *Proc. 23th ACM/IEEE Design Automation Conf.*, pp. 461-465, 1986.

87. P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Trans. CAD/ICAS*, vol. 8, no. 6, pp. 661-679, June 1989.

88. D. N. Pnevmatikatos and G. S. Sohi, "Guarded Execution and Branch Prediction in Dynamic ILP Processors", *Proc. 21st Ann. Symp. Computer Architecture*, pp. 120-129, 1994.

89. R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation Based Synthesis", *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 444-449, 1990.

90. M. Potkonjak and J. Rabaey, "Optimizing Resource Utilization Using Transformations", *IEEE Trans. CAD/ICAS*, vol.13, no.3, pp. 277-292, March 1994.

91. U. Prabu and B. Pangrle, "Superpipelined Control and Data Path Synthesis", *Proc. 29th ACM/IEEE Design Automation Conf.*, pp. 638-643, 1992.

92. I. Radivojević and F. Brewer, "Symbolic Techniques for Optimal Scheduling", *Proc. 4th SASIMI Workshop*, pp. 145-154, Nara, Japan, 1993.

93. I. Radivojević and F. Brewer, *A New Symbolic Technique for Control-Dependent Scheduling,* ECE Tech. Report #93-16, UC Santa Barbara, Sep. 1993.

94. I. Radivojević and F. Brewer, "Ensemble Representation and Techniques for Exact Control-Dependent Scheduling", *Proc. 7th Int. Symp. High Level Synthesis,* pp. 60-65, 1994.

95. I. Radivojević and F. Brewer, "Incorporating Speculative Execution In Exact Control-Dependent Scheduling", *Proc. 31st ACM/IEEE Design Automation Conf.*, pp. 479-484, 1994.

96. I. Radivojević and F. Brewer, "On Applicability of Symbolic Techniques to Larger Scheduling Problems", *Proc. European Design and Test Conf.*, pp. 48-53, 1995.

97. I. Radivojević and F. Brewer, "Analysis of Conditional Resource Sharing using a Guard-based Control Representation", *Proc. IEEE Int. Conf. Computer Design*, pp. 434-439, 1995.

98. I. Radivojević , "Experiments in BDD Aplications to Algebra of Galois Fields", seminar talk, UC Santa Barbara, June 1995.

99. I. Radivojević and F. Brewer, "A New Symbolic Technique for Control-Dependent Scheduling", *IEEE Trans. CAD/ICAS*, vol. 15, no. 1, pp. 45-57, Jan. 1996.

100. B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview and Perspective", *J. Supercomputing*, vol. 7, no. 1, pp. 9-50, Jan. 1993.

101. B. R. Rau and C. D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing", *Proc. 14th Ann. Workshop on Microprogramming*, pp. 183-198, 1981.

102. B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 Departmental Computer: Design Philosophies, Decisions and Trade-offs", *IEEE Computer*, vol. 22, no. 1, pp. 12-34, Jan. 1989.

103. M. Rim and R. Jain, "Representing Conditional Branches for High-Level Synthesis Applications", *Proc. 29th Design Automation Conf.*, pp. 106-111, 1992.

104. M. Rim, Y. Fan, and and R. Jain, "Global Scheduling with Code Motions for High-Level Synthesis Applications", *IEEE Trans. VLSI*, vol. 3, no. 3, pp. 379-392, Sept. 1995.

105. E. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps", *IEEE Trans. Computers*, pp. 1405-1411, Dec. 1972.

106. R. Rudell, “Dynamic Variable Ordering for Binary Decision Diagrams”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 42-47, 1993.

107. F. Sanchez and J. Cortadella, “Time-Constrained Loop Pipelining”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 592-596, 1995.

108. U. Schwiegelshohn, F. Gasperoni, and K. Ebciouglu, “On Optimal Parallelization of Arbitrary Loops”, *J. Parrallel and Distributed Computing*, 11, pp. 130-134, 1991.

109. A. Seawright and F. Brewer, “Clairvoyant: A Synthesis System for Production-Based Specification”, *IEEE Trans. VLSI Systems*, vol. 2, no, 2, pp. 172-185, June 1994.

110. J. Siddhiwala and L.-F. Chao, “Scheduling Conditional Data-Flow Graphs with Resource Sharing”, *Proc. 5th Great Lakes Symp. VLSI*, pp. 94-97, 1995.

111. M. D. Smith, M. S. Lam, and M. Horowitz, “Boosting Beyond Static Scheduling in a Superscalar Processor”, *Proc. 17th Ann. Symp. Computer Architecture*, pp. 344-354, 1990.

112. B. Su and J. Wang, “GURPR*: A New Global Software Pipelining Algorithms”, *Proc. 24th Ann. Int. Symp. Microarchitecture*, pp. 212-216, 1991.

113. A. Takach and W. Wolf, “Scheduling Constraint Generation for Communicating Processes”, *IEEE Trans. VLSI Systems*, vol.3, no.2, pp. 215-230, June 1995.

114. A. Takach, W. Wolf, and M. Leeser, “An Automaton Model for Scheduling Constraints in Synchronous Machines”, *IEEE Trans. Computers*, vol. 44, no. 1. pp. 1-12, Jan. 1995.

115. A. H. Timmer and J. A. G. Jess, “Execution Interval Analysis under Resource Constraints”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 454-459, 1993.

116. A. H. Timmer and J. A. G. Jess, “Exact Scheduling Strategies based on Bipartite Graph Matching”, *Proc. European Design and Test Conf.*, pp. 42-47, 1995.

117. H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. “Implicit State Enumeration of Finite State Machines using BDD’s,” *Proc. Int. Conf. Computer-Aided Design*, pp. 130-133, 1990.

118. R. F. Touzeau, “A Fortran Compiler for the FPS-164 Scientific Computer”, *Proc. ACM SIGPLAN’84 Symp. Compiler Construction*, SIGPLAN Notices, vol. 19, no. 6, pp. 48-57, 1984.

119. K. Wakabayashi and H. Tanaka, “Global Scheduling Independent of Control

Dependencies Based on Condition Vectors", *Proc. 29th ACM/IEEE Design Automation Conf.*, pp. 112-115, 1992.

120. K. Wakabayashi and T. Yoshimura, "A Resource Sharing and Control Synthesis Method for Conditional Branches", *Proc. 26th ACM/IEEE Design Automation Conf.*, pp. 62-65, 1989.

121. R. A. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, 1991.

122. D. W. Wall, "Limits of Instruction-Level Parallelism", RR-93/6, DEC/WRL, Nov. 1993.

123. N. J. Warter, G. E. Haab, J. Bockhaus, and K. Subramanian, "Enhanced Modulo Scheduling for Loop with Conditional Branches", *Proc. 25th Ann. Int. Symp. Microarchitecture*, pp. 170-179, 1992.

124. W. Wolf, A. Takach, C.-Y. Huang, R. Manno, and E. Wu, "The Princeton University Behavioral Synthesis System", *Proc. 29th ACM/IEEE Design Automation Conf.*, pp. 182-187, 1992.

125. J. Yang, G. De Micheli, and M. Damiani, "Scheduling with Environmental Constraints based on Automata Representations", *Proc. European Design Automation Conf.*, 1994.

126. J. C.-Y. Yang, G. De Micheli, and M. Damiani, "Scheduling and Control Generation with Environmental Constraints based on Automata Representations", *IEEE Trans. CAD/ICAS*, to appear.

127. L. Yang and J. Gu, "A BDD Model for Scheduling", *Proc. CCVLSI*, 1991.

128. T.-Y. Yen and W. Wolf, "Optimal Scheduling of Finite-State Machines", *Proc. IEEE Int. Conf. Computer Design*, pp. 266-369, 1993.

Appendix A

Binary Decision Diagrams

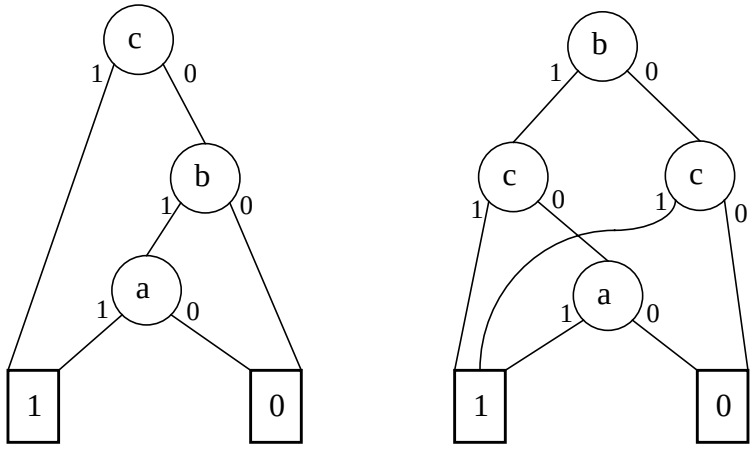
Binary Decision Diagrams (BDDs) are one of the biggest breakthroughs in CAD in the last decade. BDDs are a *canonical* and *efficient* way to represent and manipulate Boolean functions and have been successfully used in numerous CAD applications. Although the basic idea has been around for more than 30 years (e.g. [4]), it was Bryant who described a canonical BDD representation [12] and efficient implementation algorithms [10]. References [13][15][77] are very readable introductions to BDD representations and applications.

Ordered Binary Decision Diagram of a Boolean function *f* can be obtained by iterative application of the Shannon decomposition with respect to a specified variable ordering:

$$f = x f_x + \bar{x} f \qquad \text{(EQ A.1)}$$

A decision tree obtained in such a manner is reduced using two rules: (*i*) eliminate all nodes that have isomorphic sons ("don't care" elimination), and (*ii*) identify and share all isomorphic subgraphs. This process results in a Reduced Ordered BDD which is a canonical representation of a Boolean function for a specific variable ordering.

Using the *ite* (if-the-else) terminology, the Equation (A.1) can be re-written as:



(a) (b)

Figure A.1 ROBDD forms of f=AB+C using different orderings

$$f = ite(x, f_x, f_{\bar{x}}) \quad \text{(EQ A.2)}$$

All basic Boolean function manipulations can be described using *ite* templates. For example:

$$And(g, h) = ite(g, h, 0) \quad \text{(EQ A.3)}$$

and:

$$Not(g) = ite(g, 0, 1) \quad \text{(EQ A.4)}$$

The property that all Boolean manipulations can be treated in a unique manner (using *ite* calls) enables efficient implementations using computer hashing/ cashing techniques [10].

Figure A.1 illustrates ROBDD forms of $f = AB + C$ for two different variable orderings. An edge labeled by "1" ("0") corresponds to a variable's phase x ($\bar{x}$) in the decomposition formula above. The problem of finding the ordering that results in the smallest ROBDD (in terms of the number of nodes in the graph) is NP-complete. An exact variable ordering algorithm was developed in [39], but found a

very limited application due to its computational complexity. Moreover, theoretical analysis of general Boolean functions [65] indicates that, for the majority of functions, “good” orderings do not exist (i.e. the best ordering still leads to exponentially complex graphs). However, ROBDDs have performed extremely well in many practical CAD applications. Typically, the underlying structure of the problem solved using ROBDDs allows development of efficient heuristic ordering strategies (e.g. [72]).

Decision diagrams and their applications are a very active research area. Some interesting, more recent developments include:

- algebraic decision diagrams [7],
- asynchronous circuit synthesis [67],
- binate covering problem (BCP) solver [51],
- BDDs for implicit set representation in combinatorial problems [75] and applications to polynomial algebra [76],
- efficiency improvements through dynamic variable reordering [82][83][106] and breadth-first manipulations [6],
- exact and approximate FSM traversal techniques [23][27][28][117],
- formal verification of arithmetic circuits [14][44][57],
- integer linear programming (ILP) solver based on edge-valued BDDs [61],
- implicit prime generation and two-level minimization [29],
- matrix representation and manipulations using multi-terminal BDDs [24],
- multi-valued decision diagrams [54],

- symbolic model checking [16],
- symbolic synthesis techniques [66].

This list is *by no means* complete!