UNIVERSITY OF CALIFORNIA
Santa Barbara

Automata-Based Symbolic Scheduling

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

Doctor of Philosophy

in

Electrical and Computer Engineering

by

Steve Haynal

Committee in charge:

Professor Forrest D. Brewer, Chairman

Professor Tevfik Bultan

Professor Malgorzata Marek-Sadowska

Professor Louise Moser

December 2000

The dissertation of Steve Haynal
is approved:

_____

_____

_____

_____
Committee Chairman

December 4, 2000

December, 2000

To Heidi,

who motivated me to “schedule” graduate studies in my life.

Acknowledgments

I would like to thank my advisor, Professor Forrest Brewer, for his guidance and support during my graduate studies at UC Santa Barbara. His breadth and depth of intellect is a truly rare gift from which I benefited immensely. This work would not have been possible without his vision and input.

Also, I would like to thank my committee members: Professor Tevfik Bultan, Professor Malgorzata Marek-Sadowska and Professor Louise Moser. They provided helpful suggestions and comments that improved the presentation of this work. Furthermore, I am grateful to colleagues at Intel's Strategic CAD Laboratory, in particular Doctor Mike Kishinevsky, for constructive feedback, industrial examples, and generous equipment donations.

My graduate studies would not have been possible without financial support from a mosaic of sources. I am indebted to the ECE department's teaching assistant program, the National Science Foundation, Randy Fischer at Fischer Computer Systems, Donna Winter at Technical Research Associates, Doctor Ramona Clark, National Semiconductor and Intel for enabling me to pursue academic research while remaining financially solvent.

I am sincerely appreciative of the teachers, fellow students and research group members I associated with at UC Santa Barbara. They created a stimulating and enjoyable environment for intellectual and professional growth.

Finally, my deepest gratitude goes to Heidi, who is my true graduate school love and motivation.

VITA

Born — Loma Linda, California, U.S.A. — December 10, 1968.

EDUCATION

M. S. Electrical Engineering, 1997.
Department of Electrical and Computer Engineering
University of California
Santa Barbara, California, U.S.A.

B. S. Engineering Technology, 1991.
Pacific Union College
Angwin, California, U.S.A.

FIELDS OF STUDY

Major Field: Computer Engineering

Specialization: System-Level Computer-Aided Design
Professor Forrest Brewer

PUBLICATIONS

S. Haynal and F. Brewer, "Automata-Based Symbolic Scheduling for Looping DFGs", *IEEE Trans. on Computers*, to appear, 2001.

S. Haynal and F. Brewer, "Representing and Scheduling Looping Behavior Symbolically", *IEEE Int. Conf. Computer Design*, pp. 552-555, 2000.

S. Haynal and F. Brewer, "A Model for Scheduling Protocol-Constrained Components and Environments", *Proc. of 36th ACM/IEEE Design Automation Conf.*, pp. 292-295, 1999.

S. Haynal and F. Brewer, "Efficient Encoding for Exact Symbolic Automata-Based Scheduling", *IEEE Int. Conf. Computer-Aided Design*, pp. 477-481, 1998.

S. Haynal and B. Parhami, "Arithmetic Structures for Inner-Product and Other Computations Based on a Latency-Free Bit-Serial Multiplier Design", *Proc. of the 30th Asilomar Conf. on Signals, Systems, and Computers*, pp. 197-201, 1996.

PROFESSIONAL EXPERENCE

Graduate Student Researcher, Department of Electrical and Computer Engineering, University of California, Santa Barbara, January 2000 to December 2000.

Research Intern, Strategic CAD Laboratory, Intel, Hillsboro, September 1999 to January 2000.

Freelance Designer/Contract Teacher, Haynal Consulting, Santa Barbara, March 1998 to Present.

Design Intern, National Semiconductor, Santa Clara, June 1997 to September 1997.

Teaching Assistant, Department of Electrical and Computer Engineering, University of California, Santa Barbara, September 1995 to June 1999.

Science and Computer Literacy Teacher, San Fernando Valley Academy, Northridge, August 1993 to August 1995.

Engineer/Technician, Fischer Computer Systems, Angwin, June 1991 to August 1993.

Automata-Based Symbolic Scheduling

by

Steve Haynal

Abstract

This dissertation presents a set of techniques for representing the high-level behavior of a digital subsystem as a collection of nondeterministic finite automata, NFA. Desired behavioral and implementation dynamics: dependencies, repetition, bounded resources, sequential character, and control state, can also be similarly modeled. All possible system execution sequences, obeying imposed constraints, are encapsulated in a composed NFA. Technology similar to that used in symbolic model checking enables implicit exploration and extraction of best-possible execution sequences. This provides a very general, systematic procedure to perform exact high-level synthesis of cyclic, control-dominated behaviors constrained by arbitrary sequential constraints. This dissertation further demonstrates that these techniques are scalable to practical problem sizes and complexities. Exact scheduling solutions are constructed for a variety of academic and industrial problems, including a pipelined RISC processor. The ability to represent and schedule sequential models with hundreds of tasks and one-half million control cases substantially raises the bar as to what is believed possible for exact scheduling models.

Keywords: Scheduling; Binary Decision Diagrams; High-Level Synthesis; Nondeterminism; Automata; Symbolic Model.

Contents

Chapter 1. Introduction 1

1.1 The Scope of Scheduling 2
1.2 Behavioral Representations 5
1.3 The ABSS Methodology....6
1.4 Related Work 8
1.4.1 High-Level Synthesis 8
1.4.2 Scheduling 10
1.4.3 Miscellaneous Related Work....13
1.5 Dissertation Organization 14

Chapter 2. The Scheduling Problem à la ABSS 16

2.1 Information 17
2.2 Tasks 18
2.3 A Composite Task 21
2.3.1 Operand Dependence....22
2.3.2 Control-Dependent Task Interaction 23
2.4 The Scheduling Problem....24
2.5 An Overview of ABSS by Explicit Example....26
2.6 Summary 31

Chapter 3. Specifying Task Behaviors: MA 32

3.1 Acyclic MA....34
3.1.1 Labels, Notation and Definitions....35
3.1.2 Non-Pipelined and Pipelined MA 38
3.1.3 MA with Alternatives 40
3.1.4 Aggregate MA 40
3.1.5 Multiple Output MA....42
3.1.6 Recomputation and Memory 43
3.2 Cyclic MA....44

3.2.1 Duals in Cyclic MA 47
3.3 Protocol MA 49
3.4 Control and Multivalue MA 51
3.4.1 Intuitive Multivalue MA 51
3.4.2 Validation Issues and Local MA Solutions 53
3.4.3 Control-Obviated Task Bypassing 56
3.5 Summary 57

Chapter 4. Composing Modeling Automaton: CMA 59

4.1 Acyclic Data-Flow Composition 60
4.1.1 The Cartesian Product Composition Step 61
4.1.2 Basic Operand Dependencies 63
4.1.3 Alternative Operand Dependencies 65
4.1.4 Undetermined Operand Dependencies 66
4.1.5 Resource Concurrency Bounds 67
4.1.6 A Completely Pruned CMA 68
4.2 Cyclic Data-Flow Composition 69
4.2.1 Basic Operand Dependencies 71
4.2.2 Operand Capacity Constraints 72
4.2.3 Basic Operand Capacities 74
4.2.4 Undetermined Operand Capacities 75
4.2.5 Alternative Operand Dependencies and Capacities 75
4.2.6 Viability Prune 76
4.2.7 Cyclic Concurrency Constraints 79
4.3 Acyclic Control-Dependent Composition 79
4.3.1 Resolved Operand Dependencies 81
4.3.2 Acyclic Task Bypassing 83
4.3.3 Concurrency Constraints 85
4.4 Cyclic Control-Dependent Composition 87
4.4.1 The Impact of Capacity Constraints 88
4.4.2 Relaxing Capacity Constraint Impact 89
4.4.3 Global Capacity Constraints in Iterates 90
4.4.4 The Impact of a Global Capacity Constraint 92

4.4.5 Composing Iterates 93
4.4.6 An Example Iterate 94
4.4.7 Nondeterministic Control 94
4.5 Composition Generalizations 96
4.5.1 Explicit Boolean Constraints Among MA 96
4.5.2 Implicit Boolean Constraints Among MA 98
4.5.3 Task Splitting 102
4.5.4 Operand Buffers 102
4.6 Efficient CMA Representation 103
4.6.1 ROBDD Ordering 104
4.6.2 'Long' CMA Constraints 105
4.6.3 Simplified Operand Resolution Formulation 107
4.7 Summary 108

Chapter 5. Exploring Modeling Automata 110

5.1 Acyclic Data-Flow Exploration 112
5.1.1 Example 112
5.1.2 Basic Exploration Definitions and Algorithms 115
5.1.3 Efficiency of a CMA's Representation 118
5.2 Acyclic Control-Dependent Exploration 120
5.2.1 The Validation Problem 122
5.2.2 Unprioritized Exploration Overview 124
5.2.3 Forward Exploration 126
5.2.4 Backward Pruning with Validation 127
5.2.5 Ensemble Witness Extraction 129
5.2.6 Prioritized Exploration Overview 130
5.2.7 Full Forward Exploration 132
5.2.8 Control-Case Termination and Future Exclusion 132
5.2.9 Backward Pruning for Control-Case Prioritization 133
5.3 Cyclic Data-Flow Exploration 135
5.3.1 Repeating Kernels 135
5.3.2 Overview of Repeating Kernel Algorithms 137
5.3.3 Behavior Cuts and Tags 139

5.3.4 Forward Exploration and Backward Pruning 141
5.3.5 Closure 142
5.3.6 A Witness Schedule 144
5.4 Cyclic Control-Dependent Exploration 145
5.4.1 Behavior Cut 145
5.4.2 Forward Exploration 147
5.4.3 Control-Case Termination and Future Exclusion 147
5.4.4 Backward Pruning with Validation 148
5.4.5 Closure and Prioritized Control Cases 149
5.4.6 Closed Valid Path-Set 149
5.5 Summary 151

Chapter 6. Applications 152

6.1 Data-Flow Applications 152
6.1.1 EWF Case Study 153
6.1.2 EWF Benchmarks 155
6.1.3 FDCT Benchmarks 157
6.1.4 Miscellaneous Academic Benchmark Results 162
6.1.5 Comparison to Other Work 163
6.1.6 Synthetic Benchmarks 164
6.1.7 An Industrial Example 165
6.2 Control-Dependent Applications 167
6.2.1 ROTOR Benchmark 167
6.2.2 S2R Benchmark 172
6.2.3 Industrial Example 175
6.3 A RISC Processor Model 179
6.3.1 An Instruction Task 179
6.3.2 A Processor Composition 181
6.3.3 Modeling an Instruction Task 181
6.3.4 Integer and Load/Store Behavior Subclasses 183
6.3.5 Control Behavior Subclasses 184
6.3.6 Data Hazards 188
6.3.7 Operand Resolution Points 190

6.4 RISC Processor Results 191
6.4.1 Priority Mix Set 1 193
6.4.2 Priority Mix Set 2 196
6.4.3 Representation Growth 197
6.4.4 A Cache Hit/Miss Model 198
6.4.5 2, 3 and 4 Iterates 200
6.4.6 Summary 200

Chapter 7. Discussion 201

7.1 ABSS Novelty 201
7.1.1 Sequential Representation 202
7.1.2 Repeating Behavior with Control 202
7.1.3 Quality 203
7.1.4 Useful Scale 203
7.2 Limitations and Complexity of ABSS 204
7.2.1 Finite State 205
7.2.2 Composition Character 206
7.2.3 Exploration Complexity 207
7.3 Future ABSS Directions 208
7.3.1 Specification 208
7.3.2 Encoding 209
7.3.3 Partitioning 209
7.3.4 Hierarchy of Refinement 210
7.3.5 Heuristic Exploration 211
7.3.6 Synthesis 212
7.4 Conclusions 212

Bibliography 214

List of Figures

Figure 1.1: Three common instructions in a pipelined RISC processor 2
Figure 1.2: Program dependence graph for a RISC processor instruction 5
Figure 1.3: The ABSS Methodology 6
Figure 2.1: A graphical representation of an add task 20
Figure 2.2: A scheduling problem fragment with control blocks 24
Figure 2.3: Example looping behavioral description 27
Figure 2.4: An ABSS specification of figure 2.3's behavior 28
Figure 2.5: Minimum iteration latency schedule 29
Figure 2.6: The example's explicit CMA 29
Figure 3.1: A possible NFA construction for regular expression (a*, b*)* 33
Figure 3.2: An add task and expected sequential behavior 34
Figure 3.3: An MA representing an add task 34
Figure 3.4: An MA representing a 2 time-step add task 38
Figure 3.5: An MA representing a 2 time-step pipelined add task 39
Figure 3.6: An MA representing alternative task implementations 40
Figure 3.7: An MA representing an aggregate MAC 41
Figure 3.8: An aggregate MAC task with chaining and busses 42
Figure 3.9: An MA representing a divide task with two output operands 43
Figure 3.10: An MA representing a 2 time-step pipelined add task with forget 43
Figure 3.11: An MA with explicit physical storage 44
Figure 3.12: Acyclic to cyclic MA transformation 45
Figure 3.13: A labeled cyclic two time-step MA 47
Figure 3.14: An MA for a cyclic 2 time-step task with recomputation 48
Figure 3.15: A protocol constraint MA 49
Figure 3.16: An intuitive acyclic multivalue single-time step MA 52
Figure 3.17: An intuitive cyclic multivalue single-time step MA 53
Figure 3.18: Behavior to highlight validation issues 54
Figure 3.19: A two-phase acyclic multivalue single-time step MA 55
Figure 3.20: A two-phase cyclic multivalue single-time step MA 56
Figure 3.21: An MA representing a 2 time-step pipelined task with bypass 57
Figure 3.22: A labeled cyclic two time-step MA with task bypass 58
Figure 4.1: A simple acyclic data-flow example 60
Figure 4.2: Assigned MA for each task in the example 61

Figure 4.3: Cartesian product composition step for the example 62
Figure 4.4: Dependency-constrained explicit CMA for the example 65
Figure 4.5: Final explicit CMA for the example 69
Figure 4.6: An explicit CMA for a cyclic data-flow composite task 70
Figure 4.7: Simultaneous info forget and accept 74
Figure 4.8: Iteration-sense confusion when alternatives are present 75
Figure 4.9: Cyclic scheduling problem and an acausal partial CMA 76
Figure 4.10: CMA States reachable from Stask start 77
Figure 4.11: A simple acyclic control-dependent example 80
Figure 4.12: Partial CMA for figure 4.11 80
Figure 4.13: A naturally pipelined composition 88
Figure 4.14: A capacity constrained composition 88
Figure 4.15: Two iterates as one composition 89
Figure 4.16: Three global capacity constrained co-executing iterates 92
Figure 4.17: Iteration overlap among iterates with global capacity constraints 93
Figure 4.18: Iterate for cyclic control-dependent example 95
Figure 4.19: “At most 2 of 6” ROBDD 100
Figure 4.20: Task splitting 102
Figure 4.21: Operand buffers 103
Figure 4.22: A ‘chained’ scheduling problem 105
Figure 4.23: A “chained” scheduling problem with “long” constraint 105
Figure 4.24: Operand resolution sequence with immediate task bypass 107
Figure 5.1: Final CMA for acyclic data-flow example from section 4.1 112
Figure 5.2: Forward exploration of figure 5.1 113
Figure 5.3: Backward pruning of figure 5.2 114
Figure 5.4: A final deterministic witness schedule 115
Figure 5.5: Basic forward exploration algorithm 116
Figure 5.6: Basic backward pruning 116
Figure 5.7: Witness extraction algorithm 117
Figure 5.8: Conceptual view of exploration steps 118
Figure 5.9: Nonmerging path set 119
Figure 5.10: Merging path set 120
Figure 5.11: A behavioral example for discussion on validation 122
Figure 5.12: A CMA subset of a valid nonspeculative ensemble schedule 122
Figure 5.13: A CMA subset of an invalid speculative ensemble schedule 123
Figure 5.14: A CMA subset of two valid speculative ensemble schedules 124
Figure 5.15: Conceptual view of control-dependent exploration steps 125
Figure 5.16: Forward exploration algorithm with all-paths check 126

Figure 5.17: Validated preimage computation 127
Figure 5.18: Backward pruning with validation 128
Figure 5.19: Conceptual view of ensemble witness extraction 129
Figure 5.20: Conceptual view of control-dependent prioritized exploration 130
Figure 5.21: Full forward exploration algorithm 132
Figure 5.22: Earliest termination algorithm for control cases cp 133
Figure 5.23: Backward exploration with preservation of terminated states 134
Figure 5.24: Repeating execution unrolled to reveal repeating kernel 135
Figure 5.25: An explicit CMA for a cyclic data-flow scheduling problem 136
Figure 5.26: Cyclic CMA with behavior cut state sets Sbc and Sbc~ 138
Figure 5.27: False closure 139
Figure 5.28: Backward exploration pruning with termination accumulation 141
Figure 5.29: Closure fixed-point 142
Figure 5.30: Closed paths from PS.S0 to RSbc~ 143
Figure 5.31: Backward pruning with validation & termination accumulation 148
Figure 6.1: IP block for reuse 153
Figure 6.2: Target high-level architecture 154
Figure 6.3: FDCT data-flow graph 160
Figure 6.4: Hierarchical resource bounds 165
Figure 6.5: Acyclic MA with communication delay 166
Figure 6.6: ROTOR example 168
Figure 6.7: Results from two contrasting priority lists 170
Figure 6.8: Iteration latency versus ROTOR iterates 171
Figure 6.9: S2R example 173
Figure 6.10: High-level behavior of industrial control-dependent example 176
Figure 6.11: Time-zone partitioning 177
Figure 6.12: Abstract view of interconnect in industrial example 178
Figure 6.13: Abstracted instruction task 179
Figure 6.14: A processor composition of three instruction iterates 180
Figure 6.15: Possibilities for next-pc calculation and instruction prefetch 185
Figure 6.16: Speculative instruction prefetch in the default case 186
Figure 6.17: Speculative pc preincrement in the default case 186
Figure 6.18: Single bypass modeling 188
Figure 6.19: RISC processor instruction iterate 192
Figure 6.20: ROBDD node usage 198
Figure 7.1: Hierarchy of Refinement 210

List of Tables

Table 3.1: Label definitions 36
Table 4.1: Summary of Composition Steps 109
Table 6.1: Constrained IP-block results 154
Table 6.2: Results with Constrained Registers and Busses 155
Table 6.3: Elliptic Wave Filter Results 156
Table 6.4: Fast Discrete Cosine Transform Results 158
Table 6.5: Witness Schedule for FDCT1_c 161
Table 6.6: Miscellaneous Academic Benchmark Results 163
Table 6.7: Academic Benchmark Comparisons 163
Table 6.8: Synthetic benchmark results 164
Table 6.9: Acyclic ROTOR Results 169
Table 6.10: Results for Cyclic ROTOR with Two Iterates 169
Table 6.11: Acyclic S2R Results 174
Table 6.12: Cyclic S2R Results 175
Table 6.13: Control-Dependent Industrial Example Results 176
Table 6.14: Low-Level Tasks 181
Table 6.15: Generic Operand Names used in an Instruction Task 182
Table 6.16: Integer Arithmetic and Load/Store Instruction Class 183
Table 6.17: Control Class 187
Table 6.18: Expected CPI Given Mix Set 1 195
Table 6.19: Expected CPI Given Mix Set 2 197
Table 6.20: Expected CPI For Model with Cache Hit/Miss 199
Table 7.1: Select Largest ABSS Examples 205
Table 7.2: Summary of Exploration Algorithms 207

Chapter 1

Introduction

Webster's dictionary defines the verb *schedule* as "to appoint, assign, or designate for a fixed time." Scheduling is an integral step during the design of a digital subsystem. Throughout the manual design process, an engineer decides *when* operations, transactions and other events must take place. Such scheduling decisions are constrained by hardware resource availability, operand dependencies and control decisions. Also, interface protocols, either devised to simplify scheduling or resulting from implementation constraints, must be observed during scheduling. An engineer often uses *ad hoc* methods, such as timing diagrams and simulation models, to reason through the scheduling process. The goal is to create a correct implementation that meets design objectives such as minimum execution latency, reduced silicon area and low power.

This dissertation presents automated scheduling techniques useful in digital subsystem design. With these techniques, a designer need only specify desired behavior. Automated scheduling then assigns behavioral events (computations, memory access, operand communication, etc.) to specific time-steps. This relieves a designer from manual scheduling and shortens the design cycle. Because of the reduced work requirement, more emphasis may be placed on design exploration.

Ideally, automated scheduling leverages a designer's insight in tandem with a machine's computational ability to synthesize superior schedules. The end result is substantial automation of the design process.

1.1 The Scope of Scheduling

Characteristics from a RISC processor design are used to delineate the scope of scheduling for this dissertation. Figure 1.1 shows a traditional view of a single-issue four-stage pipelined RISC processor [55] and some low-level tasks required to implement an instruction. The first stage fetches the instruction and increments the program counter. The second stage decodes the instruction and accesses register file data. Next, the third stage performs a particular ALU computation. Finally, the fourth stage writes a result back to the register file. When engineering a RISC processor, the designer must schedule[1] when such low-level tasks (instruction fetch, write back, etc.) occur so that any and every processor instruction may correctly execute.

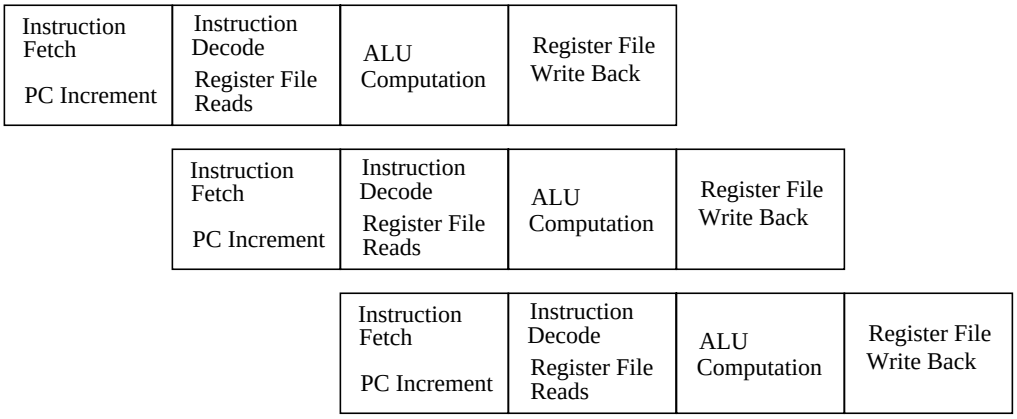


Figure 1.1 Three common instructions in a pipelined RISC processor

1. This is *not* scheduling to find good orders of assembly instructions, as is done by a compiler, but rather scheduling of the many low-level tasks needed to implement any and every processor instruction.

The implementation in figure 1.1 observes **operand dependencies**. A particular task, such as the write back, may execute only if the correct data exists. It would be incorrect to place the write back in a pipe stage before the ALU computation as the write back requires the result of the ALU computation. Scheduling must observe operand dependencies.

There is considerable **control-dependent behavior** in this RISC example. For instance, depending on the decoded instruction, an integer or a floating-point computation may be required. In other cases, such as a jump instruction, no computation and write back are necessary. In fact, for a RISC example scheduled in chapter 6, there are over 500,000 distinguishable control-dependent execution sequences! Scheduling must correctly consider every control case.

It is naive to assume that an instruction fetch always completes in a single clock cycle. Real implementations must communicate to a memory hierarchy and contend with a memory-fetch protocol. For instance, an instruction fetch may complete in a single clock cycle if a cache hit occurs but require several clock cycles if a cache miss occurs. All digital subsystems exhibit some sort of sequential behavior. In fact, for large digital subsystems, this sequential behavior may be complex and is often simplified and understood through use of protocols. Scheduling must handle all expected **sequential behavior constraints** in and for a digital subsystem.

Every RISC processor implementation has **hardware resource constraints**. To illustrate, a register file may have only two ports and allow 2 reads or 1 read/1 write concurrently. Other bounded hardware resources may include local storage, bus interconnect, function units, IO ports, etc. Scheduling must produce solutions which observe such hardware resource constraints. In fact, contention for hardware

resources makes scheduling intractable. Were it not for resource contention, all events could execute as soon as operand dependencies were satisfied.

RISC processor execution repeats endlessly. Execution sequences for various instructions may be linked end-to-end to create an infinite number of infinite length execution sequences. Scheduling must correctly generate such **infinite repeating solutions** yet do so in a finite and bounded manner even for control dependent and nondeterministic behaviors.

A RISC processor must be **complete**. It must observe *all* imposed design constraints yet correctly implement *all* behaviors, whether common or special cases. For example, a processor often executes without interruption from data hazards, but must sometimes stall until data hazards are resolved. Typically far more design effort and complexity is required to address special cases than streamlined common cases. To be practical, scheduling must solve all cases, including special cases, while prioritizing the common cases.

A RISC processor must achieve a certain level of **quality**. A throughput of 1 instruction per cycle is expected for common instructions in a single-issue pipelined implementation. In fact, to achieve this performance in figure 1.1, the register file fetches must be performed **speculatively** or before certain they are needed. To illustrate, the decoded instruction may be a jump and hence discards the unneeded register file values. On the other hand, the decoded instruction may be an add and the register file values are available for immediate computation. To be used widely in industry, automated scheduling must produce results with similar quality to and preferably better than manual implementations.

1.2 Behavioral Representations

Automated scheduling requires a behavioral specification in which events are *not* preassigned to time-steps. This may be expressed in a program-like textual description and then converted into a program dependence graph [36] called a data flow graph, DFG, or control/data flow graph, CDFG. A program dependence graph clearly identifies operand dependencies and reveals inherent parallelism yet does not assign events to time-steps. Figure 1.2 shows a program dependence graph for one class of RISC processor instructions. Nodes represent operations or tasks while edges represent operands. Scheduling takes this graph and assigns tasks and operands to specific time-steps and in so doing creates an implementation.

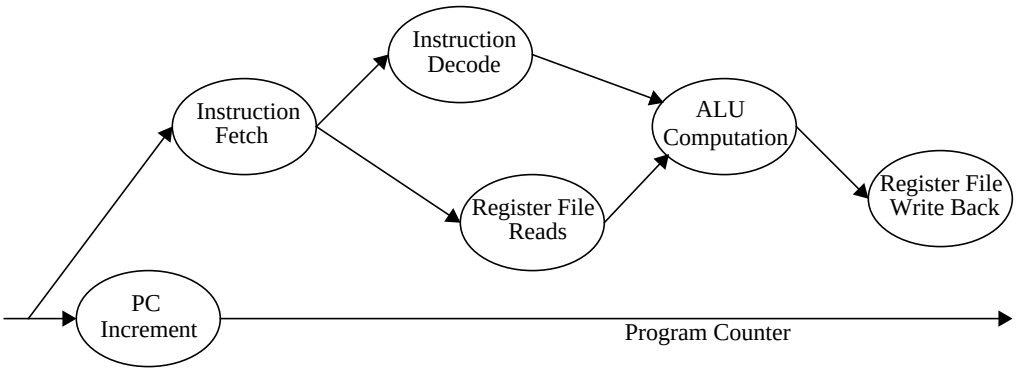


Figure 1.2 Program dependence graph for a RISC processor instruction

There is no formal language for program dependence graphs. Related work in scheduling often use representations based on program dependence ideas but with little specification compatibility among different work. As detailed in chapter 2, this work, Automata-Based Symbolic Scheduling, ABSS, also uses an input specification based on a program dependence graph. The ABSS specification goes beyond the basic program dependence graph and allows for sophisticated control-dependent behavior, cyclic behavior, sequential constraints as well as composition, abstraction and hierarchy.

1.3 The ABSS Methodology

The ABSS methodology is illustrated in figure 1.3. To begin, three input are required. The behavior input is based on a program dependence graph. It describes

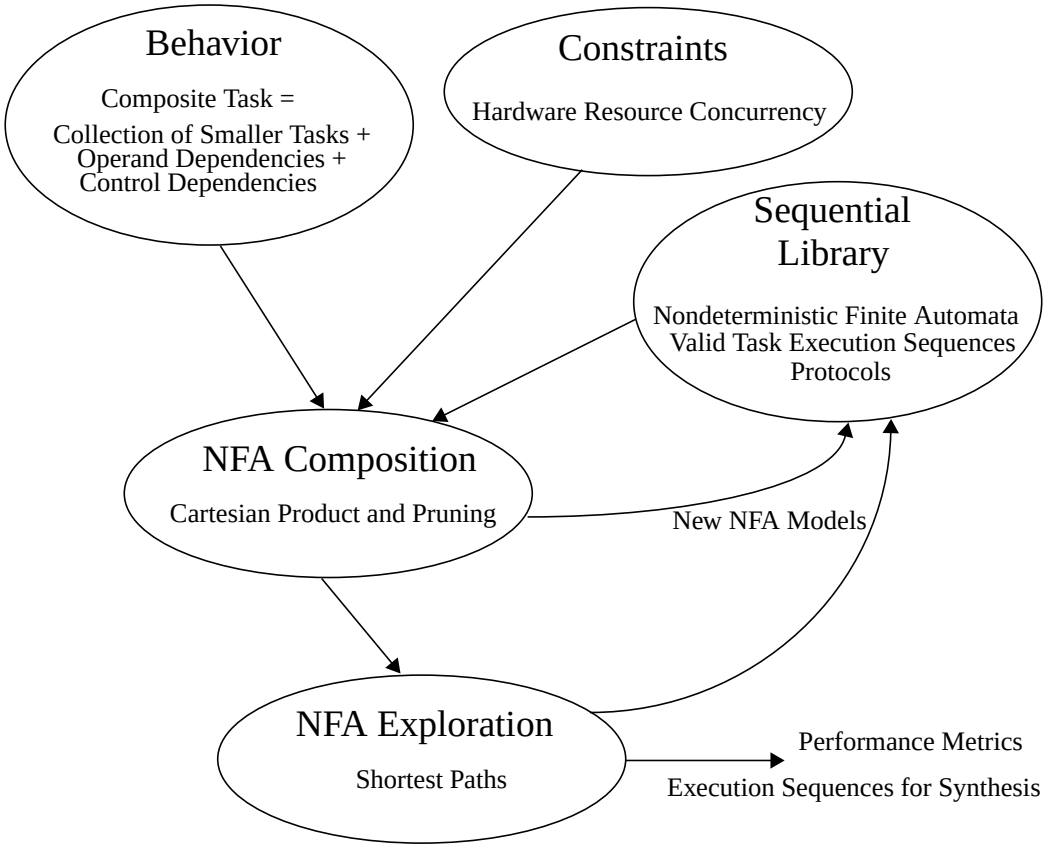


Figure 1.3 The ABSS Methodology

behavior as a collection of tasks with operand and control dependencies. This collection of tasks is itself a task, called a **composite task**, and hence a natural hierarchy is formed. For the second input, each of the tasks within a composite task is assigned expected sequential behaviors from the sequential library. These are nondeterministic finite automata, NFA, represented implicitly with Reduced Ordered Binary Decision Diagrams, ROBDDs [16][48][90][93]. These NFA, called **modeling automaton**, **MA**, encapsulate anticipated sequential behaviors for

targeted low-level hardware units. Additional sequential constraints, such as protocols, drawn from the sequential library, may also be included as input. The final and third input is resource concurrency constraints. These bound concurrent use of hardware resources such as function units, local storage, ports and interconnect.

These three input are presented to the core of ABSS: NFA Composition. This step creates a new NFA, called a **composite modeling automaton**, **CMA**, that represents *all* valid sequences and hence implementations of the behavior input. This **CMA** and its sequences encompass the scope of scheduling described in section 1.1: operand dependencies, control-dependent execution and speculation, sequential constraints, hardware resources constraints as well as infinite repeating solutions with bounded state. Furthermore, since *all* valid sequences are found, it is possible to be complete and guarantee optimality. Finally, a **CMA** is also an **MA** and is therefore of the same format used in the sequential library. Hence, scheduling hierarchy and abstraction are possible.

The final step in the ABSS methodology is NFA exploration. Although NFA composition produces a **CMA** that encapsulates all valid sequences, some of these sequences are more desirable than others. For example, minimum latency schedules are generally preferred over other schedules. NFA exploration employs an implicit symbolic implementation of Dijkstra's shortest path algorithm to determine all minimum latency schedules. This provides implementation performance metrics for the behavior input. Furthermore, NFA exploration may extract deterministic minimum latency schedules suitable for finite state machine controller, FSM, synthesis. Finally, as with NFA composition, results from NFA exploration may be used to create new sequential library members.

1.4 Related Work

1.4.1 High-Level Synthesis

ABSS is most related to work in high-level synthesis [35][39][40][66][88][137]. High-level synthesis is an automated process that transforms an algorithmic specification of a digital system's behavior into a hardware structure that implements the behavior. High-level synthesis offers simple and fast design specification, short and highly automated design cycles, and hopefully competitive implementations. Although high-level synthesis has received considerable research attention, it has yet to achieve wide-spread use in industry. This may be attributed to two primary failings. First, high-level synthesis is often unable to address the scale or complexities of modern designs. Design teams have tens of millions of transistors at their disposal. Current high-level synthesis tools are incapable of dealing with such large designs in an unpartitioned manner and have only primitive means for problem partitioning and abstraction. Furthermore, modern designs include complex control-dependent implementation and require interface through sophisticated protocols. Such complexity issues are either unaddressed or poorly addressed by existing high-level synthesis tools. Second, high-level synthesis rarely produces competitive implementations. When implementation value is measured in terms of performance and silicon area, current high-level synthesis implementations compare unfavorably to manual implementations.

ABSS directly addresses these two primary failings of current high-level synthesis. First, ABSS utilizes an automata-based representation to address design sequential and protocol complexities. Furthermore, although ABSS successfully solves unpartitioned problems of meaningful scale, ABSS provides a route to problem abstraction based on a hierarchy of sequential behaviors. Second, ABSS determines *all* valid sequences. Hence, ABSS is guaranteed to find the best

possible sequences. This, combined with the ability to handle considerable control complexity, certain types of speculation, and endlessly repeating behavior, enables ABSS to compare favorably with manual design.

The greatest high-level synthesis success stories are for digital filter synthesis [24][25][42][60][104][110][120][134]. Such designs contain very simple control structure, if any, and may be behaviorally specified with a relatively small set of mathematical equations. The majority of papers in high-level synthesis report results for digital filter benchmarks. Although ABSS produces exceptional results for digital filters, it distinguishes itself from the bulk of high-level synthesis by addressing the full scope of scheduling. Control-dependent behavior, sequential and protocol constraints, hardware resource constraints and cyclic behavior are addressed in concert yet exact and complete schedules are generated. This enables ABSS to be successfully applied to a much wider range of high-level synthesis applications.

High-level synthesis is traditionally broken into four general steps: Input specification, Allocation, Scheduling and Binding. Although not set in stone, variations of these four steps appear in all high-level synthesis flows. Input specification is typically a textual description of the desired behavior. It is often converted into a program dependence graph as described in section 1.2. Allocation is “determination of the type and quantity of resources to implement a design for given performance and area constraints” [40]. Scheduling is “the partitioning of design behavior into control steps such that all operations in a control step execute in one clock cycle” [40]. Binding is “assignment of operations, memory accesses, and interconnections from the behavioral design description to hardware units for optimal area and performance” [40].

ABSS is fundamentally a scheduling technique. It assumes that behavioral input specification and allocation have been performed and produces all valid control-step sequences such that every event in the behavior input is assigned to a control step. ABSS performs resource-constrained scheduling and hence adheres to a restricted set of subsequent bindings. In fact, ABSS may incorporate a large number of prespecified constraints, both sequential and resource utility, so as to simplify or eliminate the final binding step. A fundamental premise of this dissertation is that a scheduling technique that generates all valid execution sequences may be initially or subsequently constrained to also represent all optimal bindings. Hence, attention is focused on comprehensive scheduling that adheres to flexible and varied constraints.

1.4.2 Scheduling

Scheduling is a well-studied problem, with a rich literature of previous work. For this reason, discussion of related scheduling work is organized around three general approaches: Heuristic, Integer Linear Programming (ILP), and Symbolic. Furthermore, success of these approaches is measured by how completely and thoroughly they address the scope of scheduling introduced in section 1.1. In summary, the scheduling scope includes: operand dependencies, control-dependent behavior, sequential behavior and constraints, hardware resource constraints, repeating behavior, completeness and quality.

Heuristic scheduling techniques are by far the most common [22][25][44][45][58][68][71][104][109][110][121][132][133][134]. All heuristics, and in fact all scheduling techniques, address operand dependencies. If a scheduling technique only considers operand dependencies, then it only schedules data-flow graphs, DFGs. Within the DFG paradigm, there is considerable work regarding repeating or cyclic behavior. Pioneering work in DFG loop scheduling

and pipelining was done by Girczyc[44], Paulin[109] and Goosens[45]. Chao's rotation scheduling[25] uses a series of transformations to perform DFG loop pipelining with function-unit utility constraints. Lee[71][72], Sanchez[120], and Wang[134] all require an initial prespecified loop iteration latency and then adjust for function-unit utility constraints. Lee employs as-soon-as-possible scheduling and then resolves resource constraint violations. Sanchez computes the minimum initiation interval of the loop and then iteratively retimes, schedules and adjusts resources. Wang, who has perhaps the most complete heuristics for digital filters, uses a novel cycle-finding and covering scheme. In general, heuristics that ignore control-dependent behavior can perform well for repeating behavior. Solution quality is equivalent or close to optimum. On the other hand, solution completeness may suffer as only single solutions are found and protocols are not accommodated.

Heuristics also exist for control-dependent scheduling. The most influential early work is attributed to Wakabayshi[132][133]. Two recent state-of-the-art heuristics are by Lakshminarayana[68] and Dos Santos[121]. Lakshminarayana's technique, which handles some repeating behaviors, uses an explicit breadth-first elaboration of the available operations on each time-step that is similar to the implicit NFA exploration used in ABSS. Dos Santos' heuristic is guided by code-motion pruning and includes some forms of speculation but does not handle repeating behavior. In general, heuristics for CDFG scheduling perform poorly in terms of quality and completeness. Quality suffers since early decisions related to control often eliminate good solutions. Furthermore, if a substantial portion of the solution space is explored to reveal good solutions, only small problems may be solved. Finally, as before, solution completeness suffers as only single solutions are found and protocols are not accommodated.

Some of the best known exact scheduling techniques are based on integer linear programming, ILP [42][43][61]. Although shown to be relatively efficient, ILP techniques do not readily generalize to control-dependent scheduling and have formulation difficulties with sequential constraints other than pure or bounded delays. Of the few ILP techniques that handle control, Coelho's is perhaps the most mature [30]. Even so, only a small number of control points are allowed and code motion is substantially impacted by control formulation. In general, ILP techniques do not handle the amount of control nor sequential protocols necessary for practical design.

Symbolic scheduling techniques were first suggested by Kam[64]. A few preliminary experiments for tightly constrained acyclic DFG scheduling problems were formulated using multi-value decision diagrams. ROBDD-based exact symbolic scheduling was pioneered by Radivojević [113][114]. His work addressed acyclic control-dependent resource-constrained scheduling in a fairly complete manner. As with ABSS, *all* execution sequences, including those with some types of speculation, are found. As such, his work is a significant foundation and motivation for ABSS. The differences between Radivojević's technique and ABSS lie in fundamentally different problem formulations. Radivojević's technique represents a complete schedule as a ROBDD-based implicit Boolean logic function. On the other hand, ABSS represents a complete schedule as an execution sequence of an implicit nondeterministic finite automaton. This allows ABSS to naturally handle repeating behavior. Radivojević's technique only handled repeating behavior for DFG scheduling with prespecified latencies and not at all for CDFG scheduling. Furthermore, since Radivojević's technique requires that all schedule control steps are represented in a single Boolean logic function, schedules with long latencies require lengthy logic functions. ABSS does not record all past control steps in a single NFA state vector and hence has no such

difficulty with long latency schedules. Finally, Radivojević's technique is based on a simple non-sequential operator model and can not be generalized to an arbitrary sequential model as can be done in ABSS. Thus, ABSS can represent abstractable sequentially-constrained control-dependent repeating schedules while Radivojević's technique does not.

Automata-based symbolic scheduling did not originate with ABSS but was introduced by Yang[138]. His formulation did not address repeating behavior or sequential constraints in a general way. Although control was incorporated, necessary correctness issues related to validation and causal ensemble schedules were ignored. Yang's formulation also did not use nondeterminism and hence suffered significantly from ROBDD representation growth. Monahan[94][95] also proposed an automata-based scheduler. His was for predefined datapaths subject to limited memory constraints. His work did not address control-dependent or repeating behavior. Finally, Yen[140][141] introduced the notion of Behavioral FSM scheduling. Although this technique was symbolic and automata-based, it was explicit rather than implicit. Hence, only a single solution is found. Furthermore, substantial difficulty is encountered when constraining resources and formulating control for general BFSM models.

1.4.3 Miscellaneous Related Work

A novel ability of ABSS is protocol and sequential constraint accommodation. Related work regarding this has focused primarily on interface synthesis separated from scheduling. Influential interface synthesis work is attributed to Borriello[12].

Some scheduling techniques expose more problem parallelism through graph transformations [1][81][99][111]. Algebraic and retiming transformations restructure the DFG or CDFG. Hence, a new number of problem graph vertices or edges may result. Although ABSS may implicitly perform some algebraic and

retiming transformations through use of nondeterministic alternatives, this is not the focus of this dissertation. Hence, unless otherwise stated, a scheduling problem and solution correspond to a single static graph.

Scheduling is an essential step in code compilation[1]. Here, a heuristic is usually employed to order processor instructions from thousands of lines of code. Unlike the scheduling presented here, there is predefined processor hardware. This typically constrains the problem and guides the heuristic scheduler. The focus is on general improvement in thousands of lines of code rather than on determining a "best" processor architecture.

As ABSS is an implicit symbolic technique, it is related to symbolic model checking [89]. With ABSS, a model is constructed to represent only and all correct execution sequences. This model is then used to evaluate or generate a correct implementation. With symbolic model checking, a supposedly correct implementation model is provided. This model is then evaluated to determine if the implementation is indeed correct.

ABSS represents the execution sequences of atomic tasks in a scheduling problem as NFA. These NFA closely resemble certain types of Petri Nets [98][108]. This relation provides alternative conceptual models and leverage of a wider range of existing research.

1.5 Dissertation Organization

This dissertation is organized around the ABSS methodology introduced in section 1.3. Chapter 2 presents the ABSS problem formulation and behavioral input specification. Both are information-centric and describe how tasks interact. Care is taken to support hierarchy and abstraction. Chapter 3 describes how sequential library members, **MA**, are specified. **MA** are specified to model low-

level hardware components with the simplest of sequential behaviors. Only a handful of states and transitions are required for these specifications. Chapter 4 describes how a collection of **MA** represent tasks in the behavior input. A new **MA** is formed through composition. After a series of constraint applications, this **CMA** represents all valid sequences and hence implementations of the behavior input. Chapter 5 presents **CMA** exploration. Exploration seeks to find shortest paths and hence minimum latency schedules. Such schedules provide performance metrics as well as deterministic synthesis candidates. Throughout chapters 2 through 5, the RISC processor example from section 1.1 is used as a touchstone. It directs the potentially tedious yet necessary details in these chapters to one common goal: automated design of a RISC processor. Chapter 6 presents applications of ABSS. Scheduling problems, drawn both from academia and industry, are formulated and solved completely and precisely. Furthermore, these problems are of sufficient size and scale to demonstrate the viability of ABSS for industrial design. A complexity discussion is included here. As a capstone, RISC processor behavior for all MIPS integer instructions, similar to and beyond the example in section 1.1, is scheduled. These scheduling solutions are comparable to, and in some cases, better than what is expected from high-quality manual design of a single-issue pipelined processor. Finally, chapter 7 draws conclusions. ABSS novelties, complexities and limitations are summarized and future ABSS research directions are outlined.

Chapter 2

The Scheduling Problem à la ABSS

Automata-Based Symbolic Scheduling, ABSS, was conceived to answer the practical high-level synthesis questions raised in chapter 1. At the heart of these questions is a scheduling problem: *when* should operations occur to provide good performance given hardware constraints? Unfortunately, traditional scheduling problem definitions fail to capture enough real constraints to be representative of practical designs. As such, when formally defining the scheduling problem ABSS solves, a different and expanded problem viewpoint must be taken. ABSS approaches the scheduling problem from an operand or information viewpoint. To be precise, ABSS models all feasible, causally and sequentially correct sequences of information consumption and production in a digital system where such information is finite. This may be contrasted with traditional operation-centric scheduling techniques. Rather than assigning operations to time-steps, ABSS determines available information at each time-step and sequentially models how new information may be produced from this available information. Once all desired information has been produced, the behavior is complete. If this is done in a minimum number of time-steps, a minimum latency schedule or execution sequence results. Finally, by taking an information-centric approach, ABSS may separate a behavioral task from an actual sequential implementation. An ABSS

scheduling problem specifies the desired behavior in terms of information creation and consumption as well as anticipated sequential constraints of protocols and hardware function units used to implement this behavior. Consequently, the same behavior may be scheduled for various target hardware with differing sequential constraints.

This chapter is organized as follows. First, *information* is defined and described as used with ABSS. Next, the concepts of *task* and *composite task* are introduced and defined. These definitions provide a foundation to formulate a new scheduling problem which is amenable to abstraction and hierarchy. Both abstraction and hierarchy are essential for scheduling large digital systems as the resource-constrained scheduling problem is intractable. Finally, an explicit example provides an overview of the entire ABSS technique.

2.1 Information

Definition 2.1 **Information** is any bit, signal or piece of data (besides a clock) communicated in to or out of a digital subsystem. An atomic portion of information is referred to as an **operand**.

This is a very general definition as it makes no distinction concerning *types* (control, data, bit width, etc.) or *values* of information. When symbolically modeling system behavior for scheduling, the emphasis is not on the *correctness* of a behavior's computations but rather on the behavior's *feasibility, performance*, and *cost.* Hence, the information's type or value may often be ignored. For this reason, determining only whether or not a particular operand *exists* is critical. This existence or nonexistence of information is represented as a Boolean variable. For example, suppose *info* refers to some operand necessary in the behavior. In ABSS, *info* is *true* if this operand exists and is available in the system, and *false* if not. Although *info* corresponds to a particular operand, *info* **does not** contain the actual

value of this particular operand. If it is necessary to distinguish between various values of one operand, it is always possible to treat each necessarily distinguishable value as a separate operand and hence reduce the problem to the original simplicity. For instance, the operands $info^{val=5}$ and $info^{val=7}$ are two operand names, not values, and refer to the Boolean *existence* of two separate operands. Although there are additional system modeling considerations, this minimalist operand existence/nonexistence view is a key element of ABSS.

As described, information is encoded in “one-hot” fashion. A single Boolean variable identifies existence or nonexistence of a single operand in the digital subsystem. Although pure logarithmic encodings were experimented with [51], this sparse “one-hot” encoding generally provides more efficient ROBDD representation and manipulation. Still, when necessary to represent several mutually exclusive values of an operand, a logarithmic encoding is beneficially employed.

The example RISC processor requires and produces numerous operands during execution. If considered at the level of an executing instruction, a register file read produces an operand that is required by the ALU. Likewise, the ALU produces an operand that is required by the register file write back. For scheduling purposes, the operand’s value is often irrelevant, yet a correct dependency-ordered sequence of operand production and consumption is required.

2.2 Tasks

When viewed as a black box, a digital subsystem *accepts* (input) and/or *produces* (output) various operands in various time sequences. A digital subsystem may be as simple as an ALU, which requires two input operands and produces one output operand a short time later, to something as complex as a RISC processor core, which requires input instructions and memory data and produces memory

addresses and memory data while observing a complex dynamic sequential protocol. Furthermore, large digital systems are typically composed of smaller and simpler digital subsystems. At any level of this natural hierarchy, this view of sequential information consumption and production holds.

All digital systems or subsystems implement a desired *task* or portion of behavior. For instance, an ALU may implement an add or a subtract as well as other tasks. In traditional high-level synthesis, these tasks are called operations and are scheduled and bound to available hardware. Since ABSS takes a more information-centric sequential view, operations are not necessarily trivial but can range from simple combinatorial ALU functions to the transaction tasks performed on large digital systems with sophisticated protocols. Hence, the term *task* is used to describe a particular portion of behavior.[1]

Definition 2.2 A **task** is a four-tuple *task*(*A*, *P*, *R*, *Q*). Each operand $a \in A$ may[2] be accepted to complete the task. Each operand $p \in P$ may be produced during implementation of the task. *A* and *P* are thought of as sets of Boolean variables where truth value represents existence or nonexistence of the particular operand. Each $res \in R$ is a named type of hardware resource (function units, buses, local registers, etc.) which may be required to complete the task. *Q* encapsulates all allowed sequential executions of the task either imposed by hardware constraints or desired by the designer.

As an example, a simple add task may be specified as ADD((*a*, *b*), (*c*), (ALU), (Single time-step)). Here, *a* and *b* are the input operands, *c* is the result operand, an ALU hardware resource is required, and the

1. The term task is used commonly in hardware/software codesign. In that context, it typically describes fairly complex behavior. In the context of ABSS, tasks describe behaviors ranging from very simple to complex.
2. Operand events are potential rather than necessary as control decisions may make some operand events unnecessary.

result operand is computed in one time-step. It is rather imprecise to specify ‘Single time-step’ as *Q*. Chapter 3 describes in detail how potential sequential behaviors for a task are described with a nondeterministic finite automaton.

The sets *A* and *P* include *all* operands the task potentially accepts and produces. This should not be confused with a hardware data port. It is possible that some ALU must communicate all operands through one port, yet there are still typically three operands: 2 input and 1 result. Any single information communication event that provides unique information to or produces unique information from a task is considered a distinct operand and must be included in either *A* or *P*. Hence, although some data may be packetized, each discrete distinguishable piece of a packet is considered a unique operand. On the other hand, tasks often repetitively execute within loops. In this case, the sets *A* and *P* may grow infinitely large given infinite repeated execution of a task. When a repeating task is encountered, exactly two instances of a particular operand, which distinguish between past and present values, are included in *A* and *P* and hence the representation is bounded. Section 3.2 addresses issues related to modeling cyclic tasks.

When graphically representing a task, elements *R* and *Q* may be suppressed as shown in figure 2.1. Only elements *A* and *P* are drawn explicitly. With this

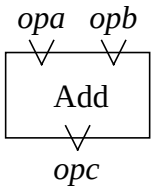


Figure 2.1 A graphical representation of an add task

representation, a task may be thought of as a vertex (although with possibly numerous input and output operands) in a traditional data-flow graph. This is primarily a behavioral representation. Resource requirements, *R*, and sequential

constraints, *Q*, are separate artifacts of implementation and hence may vary for the same behavioral task.

In the definition of a task, definition 2.2, a substantive set hierarchy was introduced. For example, $p3 \in P \in task$ identifies an operand *p3* which is one of the produced operands of *task*. This type of set hierarchy is common to ABSS. To ease notation of this concept, dot notation from object oriented programming is borrowed. Hence, *task.P.p3* is shorthand for $p3 \in P \in task$. This notation, as formally defined in definition 2.3, is used throughout this dissertation.

Definition 2.3 Dot notation represents $SET1 \in SET2 \in \ldots \in SETn$ as $SETn.\ldots.SET2.SET1$ or represents $element_i \in SET1 \in SET2 \in \ldots \in SETn$ as $SETn.\ldots.SET2.SET1.element_i$.

2.3 A Composite Task

A collection of tasks may interact to represent a larger behavior. For example, in a RISC processor, small tasks, such as register file reads and writes, ALU computations, etc., represent behaviors required to execute a complete instruction. When these tasks are collected and organized appropriately, their collective behavior may be viewed as a larger task representing behavior of a single RISC instruction.

A collection of interacting tasks, called a **composite task**, is also a task albeit at a higher level of abstraction. It falls within definition 2.2 for a task as it accepts and produces operands, requires resources, and exhibits sequential behavior. Furthermore, a composition specifies a particular organization of tasks and task interaction. This particular organization defines a behavior in much the same was as a control/data-flow graph does for traditional scheduling. Loosely, a composite task is a set of tasks and a set of task interactions. Before formally defining a

composite task, it is helpful to define and describe how tasks may interact with each other.

2.3.1 Operand Dependence

Fundamentally, tasks accept and produce information. Task interaction is when one task accepts information produced by another task. This is an operand dependence as the accepter *depends* on the producer's information.

Definition 2.4 An **operand dependence** is defined as $e = (f, a)$. The Boolean expression f is written in terms of produced operands, $p \in P$, from tasks in the composite task. The variable a is a single accepted operand, $a \in A$, for some task in the composite task. When f is *true*, the required operand a is available. When f is *false*, the required operand a is not available.

A novel aspect of this operand dependence definition is the Boolean expression f. If f consists of just a single operand p, then this operand dependence reduces to a simple data-flow graph data dependency edge. Since f may be an arbitrary Boolean expression of operand existence variables, considerable flexibility is added. For instance, the operand selection that occurs typically at a CDFG join may be expressed. A task might depend on operand $info^{spec}$ in some cases but operand $info^{nospec}$ in other cases. The Boolean expression for this might look like $info^{spec} \cdot rinfo^{val\,=\,1} + info^{nospec} \cdot rinfo^{val\,=\,2}$ where $rinfo^{val=1}$ and $rinfo^{val=2}$ are two separate operands whose existence indicates different control cases. Operands $rinfo^{val=1}$ and $rinfo^{val=2}$ are said to *guard* operands $info^{spec}$ and $info^{nospec}$ respectively. As another example, there may be two legitimate sources of some operand and the accepting task should use whichever one is available and convenient. This nondeterministic alternative might be expressed as $info^{source1} + info^{source2}$ where $info^{source1}$ and $info^{source2}$ are two equivalent yet alternative instances of the required operand.

2.3.2 Control-Dependent Task Interaction

Most high-level behavioral descriptions require control-dependent execution and/or interaction of tasks within a composition. A decision as to what tasks to execute or what information to use is made based on the value of some operand. For instance, depending on the decoded instruction in a RISC processor, the final write back task may or may not be required. Hence, this write back task execution is control dependent. Control-dependent execution of a task requires a *task tuple* while control-dependent selection of an operand requires a guarded operand dependence expression.

Definition 2.5 A **task tuple** is defined as (*t*, *cb*). The variable *t* is a task as in definition 2.2. Boolean expression *cb* is written in terms of produced operands, $p \in P$, from tasks in the composite task. When *cb* is *true*, the task *t* is required for successful completion of the composite task. When *cb* is *false*, the task *t* is not required for successful completion of the composite task.

Consider the composite fragment shown in figure 2.2. The tasks on the left belong to the control block $d^{val=0}$ while those on the right belong to the control block $d^{val=1}$. The tasks on the bottom and top belong to the control block $d^{val=0}+d^{val=1}$ as they accept or produce operands in either control case.

An operand within a control block expression is still viewed as information and hence is included in definition 2.1. However, it is sometimes necessary to distinguish between several values of a control operand and not just its existence or nonexistence. As described in section 2.1, all necessarily distinct values may be represented as unique operands. Consequently, each value of a control operand may thus be represented by a unique operand.

With these preliminary definitions completed, it is possible to formally define a composite task.

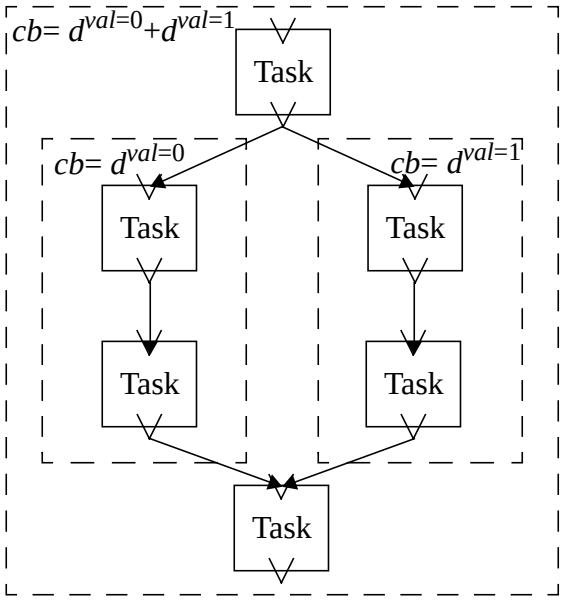


Figure 2.2 A scheduling problem fragment with control blocks

Definition 2.6 A **composite task** is a **task** as defined in definition 2.2. A **composite task** is also a **composition** defined by $C = (T, E)$. Each $(t, cb) \in T$ is a task tuple as in definition 2.5. Each $e \in E$ is an operand dependence as in definition 2.4. An operand dependence exists for every accepted input operand, $a \in A$, of every task $t \in T$ in a composition.

Definition 2.7 A c**ontrol-dependent composite task** is a composite task in which the dynamic *value* of some operand alters behavior.

2.4 The Scheduling Problem

A scheduling problem is not a composite task, but rather the question of what is a correct and desirable sequential behavior, *Q*, for a composite task. A scheduling problem asks the important question, “What finite state machine controllers exist which will execute this collection of tasks in acceptable time yet require reasonable hardware and datapath?” Although answering this question is potentially intractable, ABSS provides techniques to solve problems of reasonable

size exactly while also providing a route to higher abstraction for very large problems.

A scheduling problem asks what are correct and desirable sequential behaviors for a composite task given internal tasks that interact through operand dependencies, exhibit their own sequential behavior and contend for resources. Furthermore, other arbitrary sequential constraints may be imposed to constrain how internal tasks interact or how a composite task interfaces externally.

Definition 2.8 A **scheduling problem** is defined by the three-tuple $SP = (C, R, Q)$. The set C is a composition as in definition 2.6. Each $(bound, T_r) \in R$ is an ordered pair where natural number *bound* is the maximum permitted concurrent uses of class r resources and $T_r \subseteq C.T$ is the set of all tasks requiring a class r resource at some time. Q represents a set of additional sequential constraints which may represent external protocols or constrain how several tasks within the composition must interact. A solution to the scheduling problem is a correct and desirable set of sequential behaviors suitable for $Q \in$ *composite task* that observe all constraints of a scheduling problem.

As stated, solutions to the scheduling problem are sequential behaviors suitable for $Q \in$ *composite task*. In ABSS, solutions, as well as all other sequential behaviors Q, are represented as nondeterministic finite automaton. For modeling purposes, this entire nondeterministic finite automaton may be used. For synthesis purposes, a deterministic finite automaton contained within the nondeterministic solution must be extracted. Solutions observe all internal task sequential behavior, operand dependencies, hardware concurrency limits and additional sequential constraints. A desirable solution strives to satisfy other objectives such as minimum execution latency for the composite task. For the RISC example, the

scheduling process determines correct minimum-latency sequential behaviors for several concurrently executing RISC instructions.

A scheduling problem has a potentially infinite number of solutions. Suppose that a scheduling solution takes *n* time-steps. It is often possible to add a delay at some point in the schedule and thus require *n*+1 time-steps. In this way, an infinite number of solutions may exist which are encapsulated via nondeterminism of the ABSS formulation. Alternatively, suppose that a scheduling solution for repeating behavior produces an intermediate operand *o* at some point in the schedule. If dependency constraints permit, this schedule may produce the next iteration instance of *o,* called o^2, while o^1 is still in use. Indeed, it may be possible that schedules exist where *n* (possibly infinite) iteration instances of operand *o* are in use. In another case, a valid yet impractical scheduling solution might recompute the *same* iteration instance of intermediate operand *o ad infinitum*. ABSS models schedules as instances of finite state automata and hence must bound these potential infinite state situations.

Once scheduling solutions meeting certain objectives are found, a composite task is completely defined --its *Q* is specified. This composite task itself may be composed with other tasks and scheduled at a now higher level of abstraction. In this fashion, very large tasks, represented as a hierarchy of refinement, may be described. A task's sequential behavior, *Q*, at any level of this hierarchy, is the vehicle of refinement. The scheduling process communicates refinements across hierarchy levels.

2.5 An Overview of ABSS by Explicit Example

To provide an example of how ABSS works, an ABSS automaton model is presented explicitly. This example is not representative of all ABSS capabilities. Its purpose is to highlight what is at the core of ABSS models while ignoring, for

now, other important considerations such as protocols and control-dependent behavior. In practice, all ABSS models are implicitly and efficiently represented with Reduced Ordered Binary Decision Diagrams, ROBDDs [16][48][90][93].

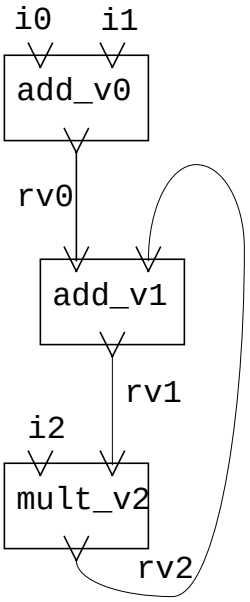
Figure 2.3is a looping pseudocode behavioral description which may be cast as

```
rv2 = 0;
while (TRUE) {
   i0 = read();
   i1 = read();
   i2 = read();
   rv0 = i0 + i1;   // Task v0
   rv1 = rv0 + rv2; // Task v1
   rv2 = rv1 × i2;  // Task v2
   write(rv2);
}
```

Figure 2.3 Example looping behavioral description

a scheduling problem. For each iteration of the loop, the subsystem implementing this behavior reads three input values and writes one result. Furthermore, the result of the multiplication, `rv2`, is required by an earlier addition and hence a operand dependence between different iterations of the loop, an *inter-iteration* dependency, exists. Consequently, `rv2` must be initialized upon entering the loop.

Figure 2.4 shows both graphically and textually how figure 2.3's behavior is specified as an ABSS problem. Read and write events and associated data dependencies are not represented as separate tasks but rather as resource requirements in this particular specification. Input and output operands are assumed to be unbuffered. Hence, external read and write events occur when input and output operands are required or produced by tasks. Finally, for brevity, the control blocks for each task tuple are suppressed in SP as they are all don't care for this control-less example. Also, the sets of tasks requiring a resource, T_r, are suppressed as they may be inferred from the individual task specifications.




```
add_v0 = (   (i0, i1), (rv0),
             (ALU, IO Port, IO Port),
             (Single time-step) )

add_v1 = (   (rv0, rv2), (rv1),
             (ALU),
             (Single time-step) )

mult_v2 = (  (i2, rv1), (rv2),
             (Multiplier, IO Port),
             (Single time-step) )

SP = (  ( (add_v0, add_v1, mult_v2),
          ((add_v0.rv0, add_v1.rv0),
           (add_v1.rv1, mult_v2.rv1),
           (mult_v2.rv2~, add_v1.rv2))),
        (  (1, ALU),
           (1, Multiplier),
           (3, IO Port) ),
        (  (No additional Q) ) )
```

Figure 2.4 An ABSS specification of figure 2.3's behavior

Correctly scheduling this DFG requires assigning each operand, and hence task in this case, to a time-step while observing several criteria. First, all operand dependencies must be observed. Second, resource bounds, such as one available adder or maximum four simultaneous external data transfers, must be adhered to. Finally, a scheduling objective, such as latency minimization typically guides schedule selection.

If one single time-step adder, one single time-step multiplier and three simultaneous external IO transfers are allowed, then the example's only minimum iteration latency schedule is shown in figure 2.5. Although the *delay*, or required

time-steps for a single loop iteration, is three, the *iteration latency*, or time-steps between successive loop iterations, is only two time-steps[3]. This *loop winding* is possible because operands and tasks from successive iterations are allowed to overlap as seen with tasks `v2` and `v0` and hence with operands `rv2` and `rv0`.

```
v0   v1   v2
          v0   v1   v2
                    v0   v1   v2
 1    2    3    4    5    6    7    ...
```

Figure 2.5 Minimum iteration latency schedule

ABSS constructs a composite modeling automaton, a **CMA**, that encapsulates all solutions for a given scheduling problem, *SP*, subject to problem and model-imposed constraints. For this example, the desired **CMA** is *explicitly* shown in figure 2.6. Each *transition* in this nondeterministic state graph represents a time-step[4]. Task activities are assigned to time-steps and are identified through transition labeling. The distinction between successive loop iterations or *iteration sense* is made with the symbol '~'. If a task activity is labeled with no '~', such as *add_v0*, then *add_v0~* represents the same task activity in the successive iteration and vice versa. For instance, the transition from state 100 to state 001 is labeled *add_v0* and *mult_v2~* as these two tasks from successive loop iterations occur during this transition time-step.

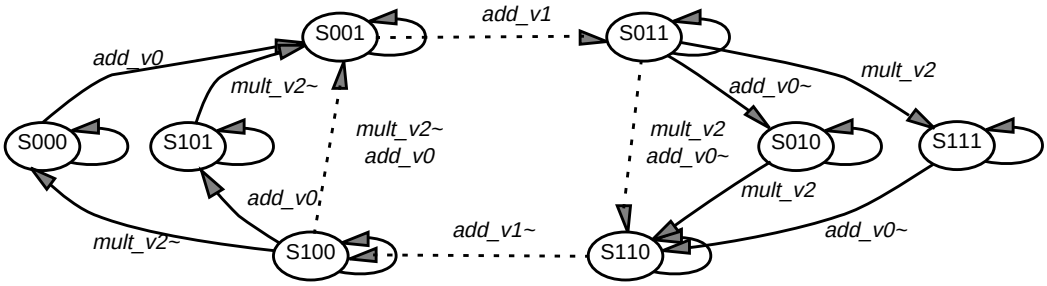


Figure 2.6 The example's explicit CMA

3. Minimizing iteration latency in this case is equivalent to maximizing throughput.
4. A time-step typically corresponds to a synchronous clock period but may be interpreted as any integral time unit.

While transitions denote task activities, states encode in which sense operands currently exist in the system. The existence/nonexistence truth value of any single operand is evident from a state's encoding. In the example, state vector bits are ordered *rv2, rv1, rv0*. To illustrate, consider the transition from state 100. The one indicates that operand *rv2* is *known* (present) in the regular iteration sense but *unknown* (not present) in the '~' iteration sense. Likewise, the two zeros indicate that operands *rv1, rv0* are *known* (present) in the '~' iteration sense but *unknown* (not present) in the regular iteration sense.

Any path in this **CMA** represents a valid execution sequence of the behavior. Every transition belongs to some path and is hence a step in some valid execution sequence. Consider the transition from state 100 to state 001. Since operand *rv1* is *known* in the '~' iteration sense, and an IO port is free to read *i2~*, the task *mult_v2* may execute and does so during this transition time-step. This task produces the next iteration result of operand *rv2* which is indicated by the '0' in the *rv2* position of the state transited to. Likewise, since the limit of three IO ports has not been reached, operands *i0* and *i1* may be read and the task *add_v0* is allowed to execute and produce a new *rv0* result operand.

All valid execution sequences of a scheduling problem, including sequences which arbitrarily stall, are modeled by a **CMA**. These sequences observe all constraints, such as resource utility limits, included as part of the scheduling problem. Given such a structure, it is possible to find subsets of paths with specific properties. As an example, minimum iteration latency repeated executions of the loop are represented by shortest cycles. In figure 2.6, the minimum iteration latency schedule is highlighted with dashed edges. Two iterations (for both iteration senses) are scheduled in one complete traversal of this cycle. A loop initialization sequence is a shortest path from state 000, where no information is *known*, to entry of the solution cycle at state 001. Symbolic exploration, described

in chapter 5, finds such paths meeting a scheduling objective, if they exist. These paths may then be synthesized into a finite state machine controller.

A **CMA** is built through symbolic composition as described in chapter 4. The composition process insures that all scheduling problem tasks interact in a causal and concurrency-bounded manner. The pieces of this composition are also nondeterministic finite automata and are called *modeling NFA*, **MA**. They typically model the sequential behavior of one task in the scheduling problem. Chapter 3 describes how **MA** are specified.

2.6 Summary

This chapter first defined terms important to the ABSS problem definition. Information was defined as typeless and often valueless operands. Information is represented by Boolean variables indicating the information's existence or nonexistence at some time-step. Tasks were defined as digital subsystems that implement some desired piece of behavior. Tasks accept, process and produce information. Furthermore, tasks require specific hardware resources, which exhibit constrained sequential behavior, for implementation. Next, a composite task was defined as a collection of interacting tasks. With a composite task, a natural hierarchy of refinement and route to abstraction is provided. After the concepts of information and tasks were defined, the ABSS problem was formulated as determining correct and desirable sequential behavior for a composite task. Hardware utility limits as well as additional sequential constraints were included in the problem definition. Control-dependent scheduling was shown to fall within the definitions given so far provided that actual values of 'control' operands are distinguished. Finally, ABSS solutions were introduced by way of example. An implicit nondeterministic automaton model called a **CMA** efficiently encapsulates *all* valid execution sequences (schedule solutions) of the ABSS problem.

Chapter 3

Specifying Task Behaviors: MA

The fundamental building blocks in ABSS are nondeterministic finite automata called *modeling NFA* or **MA**. An **MA**'s sequential behavior captures the local sequential behaviors possible or allowed when implementing one task[1], $t \in C.T$, from a scheduling problem, *SP*. In this way, a task's *local* timing constraints, *Q*, are specified via its **MA**. Furthermore, a task's operands, sets *A* and *P*, as well as resource requirements, set *R*, are mapped to states and transitions of an **MA** through a labeling system. Every task in the scheduling problem is modeled by its own instance of an **MA**. In other words, to represent the entire behavior of the scheduling problem, a *new* **MA** is instantiated for each task. This is a fundamental departure from other symbolic modeling techniques as primarily *behavior* (use of abstract resources) rather than *implementation* (actual hardware resources) is modeled. Finally, as related to the RISC example, specifying **MA** may be thought of as specifying the sequential behaviors of atomic hardware resource *uses* such as register file reads and writes, ALU computations, memory accesses, etc.

Once every task from a scheduling problem is modeled by an **MA**, all task **MA** are composed into a larger composite **MA** called a **CMA**. This **CMA** is refined to

1. When tasks become too complicated, several **MA** may be composed to model one task.

encapsulate precisely *all* valid execution sequences of the scheduling problem. Hence, a **CMA** represents all correct sequential behaviors of a composite task. Finally, a **CMA** may be implicitly explored or further pruned to determine system execution sequences or schedules with desired qualities.

This chapter focuses on specifying various **MA**. As all **MA** are fundamentally nondeterministic finite automata, they inherit the infinite language and correspondence to regular expression's of all NFA. For example, figure 3.1 shows one possible NFA construction which encapsulates the regular expression (a*b*)*. Although this represents an infinite number of infinite sequences, it is still compactly represented with two states and four transitions. It is impractical to define entirely the descriptive power and general applications of a NFA representation. Instead, the intent of this chapter is to demonstrate how a NFA representation may be used to model sequential behaviors of system tasks that are important to a digital system designer. Typically, designers strive to simplify subsystem tasks, interfaces and protocols in practical design. Because of this, only a handful of explicit states and transitions are usually needed to specify any **MA**. Once an **MA** is specified, it is kept in a library so that it can be reused easily. Although small **MA** are often specified manually, a key power of ABSS is the ability to compose many **MA** and correctly represent all valid sequential behaviors of truly complicated and large composite tasks.

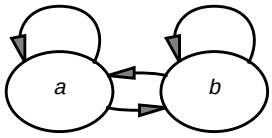


Figure 3.1 A possible NFA construction for regular expression (a***,*** b*)*

This chapter is organized as follows. First, a basic *acyclic* **MA** is introduced and a formal definition of an **MA** is provided. Examples and techniques for several

more complicated and descriptive **MA** are presented within an acyclic framework. Second, these acyclic **MA** concepts are generalized and extended to a *cyclic* **MA** representation. Finally, specific considerations for modeling control are covered.

3.1 Acyclic MA

A basic **MA** might specify the sequential behavior of one *use* of a combinatorial ALU. Implementing this task requires two input operands, *opa*, *opb* ∈ *A*, at the beginning of a clock cycle and produces a result operand, *opc* ∈ *P*, by the end of the clock cycle. Furthermore, one ALU hardware resource is required, *alu* ∈ *R*. Figure 3.2 shows the graphical representation of this task and its expected sequential behavior. Figure 3.3 shows a sparsely labeled **MA** representing this sequential behavior. Execution begins in state(s) labeled *opc unknown*. If *opa* is present, *opb* is present, and an ALU resource is available, this machine may nondeterministically choose to transit to state(s) labeled *opc known*. In general, nondeterminism allows an **MA** to delay execution in favor of another **MA** when in a composition. This enables full context-independent exploration of the solution space and hence exact results.

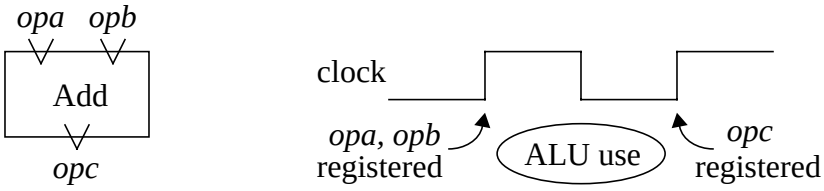


Figure 3.2 An add task and expected sequential behavior

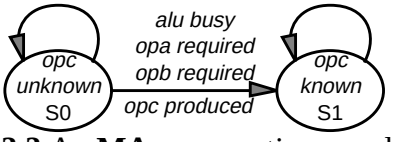


Figure 3.3 An **MA** representing an add task

Several properties of figure 3.3's **MA** are important to note. First, a synchronous system is assumed with clock-period activity and duration

corresponding to all **MA** transitions (not states). The transition labeled *alu busy* requires and occupies an ALU resource for the *entire* clock period. Also, the transition labels *opa required*, *opb required*, and *opc produced* indicate that these communications occur sometime during the clock period. For this example, *opa required* and *opb required* occur at the beginning of the clock period while *opc produced* occurs towards the end of the clock period. On the other hand, states correspond to system knowledge present at a clock edge. The state labeled *opc known* indicates that operand *opc* is registered and available for use. Second, this particular model represents an acyclic computation of *opc*. In this acyclic model, operand *opc* may be computed only once and in fact persists forever in the system.

Real systems contain tasks with more complex sequential behavior than this example. Fortunately, **MA** are generalized easily to represent complex tasks. Since designers strive to simplify subsystem interfaces and protocols in practical design, only a handful of explicit states and transitions are typically needed to specify subsystem **MA**. When behavior becomes too complex to specify with a handful of states and transitions, the behavior can often be decomposed into interacting simpler tasks represented by several **MA** and then composed into a larger more complicated **CMA** as described in chapter 4.

3.1.1 Labels, Notation and Definitions

A task's operands, sets *A* and *P*, and resources, set *R*, as well as other externally important information are mapped to an **MA** with a labeling system. In section 3.1, the labels *alu busy*, *opc unknown*, *opc known*, *opa required* and *opb required* were introduced. Table 3.1 lists and describes a subset of the most commonly used labels. Labels consist of two grammatical parts: a SUBJECT followed by a PREDICATE. A valid SUBJECT is an operand (*A* and *P*), a resource (*R*) or the word

task, which is a generic reference to the behavioral task modeled by an **MA**. In the table, operands are represented by the generic name *info* but are eventually replaced with unique operand names as specified in task sets *A* and *P*. Likewise, resources are represented by the generic name *res* but are eventually replaced with appropriate resource names as specified in task set *R*. Examples of resources are ALU, multiplier, bus1, etc. Currently supported PREDICATES are defined in table 3.1. A PREDICATE'S purpose is to describe SUBJECT properties. With this background, it should be clear that the label *rv0 known* identifies operand *rv0* has been produced in the system and label *alu busy* indicates that an ALU is currently occupied.

Table 3.1: Label definitions

Label	Description	Applied To
info known	The operand *info* has been produced in the system.	State(s)
info unknown	The operand *info* has not been produced in the system.	State(s)
info accepted	The operand *info* has been accepted.	State(s)
info stored	The operand *info* is occupying storage.	State(s)
info resolve	The operand *info* is resolving to multiple values.	State(s)
task start	The task has yet to begin execution.	State(s)
task final	The task has completed execution.	State(s)
info required	The operand *info* is required.	Transition(s)
info produced	The operand *info* if produced	Transition(s)
info forget	The operand *info* is forgotten.	Transition(s)
res busy	A resource *res* is busy.	Transition(s)
task bypass	The task is bypassed to its final state(s).	Transition(s)

Although table 3.1 describes commonly used labels, a designer is free to create additional labels. Consequently, **MA** states and transitions may be interpreted in any way a designer chooses. In general, **MA** states correspond to system knowledge present at a clock edge[2] and should receive labels which reflect this. State labels such as *info unknown*, *info known* and *info accepted* are **historical labels** as they provide a record of past events. State labels such as *info stored* and *info resolve* are **current labels** as they identify current information availability. On

2. This may be a time-step edge if finer synchronous granularity is used.

the other hand, **MA** transitions correspond to system activity during a clock period[3] and should receive labels which reflect this. As examples, the label *info required* implies that *info* was communicated to hardware implementing the task, and the label *res busy* indicates a particular hardware resource *res* is required and occupied during this clock period. Finally, **MA** state encoding and labels are distinct. In fact, one state or transition may be referenced by multiple labels and one label may reference multiple states or transitions.

Specific notation, based on this labeling system, may be used to describe transitions and states of an **MA**. The notation S is reserved to represent a set of states. The notation S_{label} represents a set of all states referenced by *label*. A single state is denoted as s. A single state referenced by *label* is denoted s_{label}. Likewise, the notation Δ is reserved to represent a set of transitions. The notation Δ_{label} represents a set of all transitions referenced by *label.* A single transition may be denoted as δ. A single transition referenced by *label* is denoted δ_{label}. Alternatively, a transition may be denoted as (s, s') where s is the **present state** or predecessor and s' is the **next state** or successor. When referring to all transitions to or from a set of states S_{label}, the notations $(—, S_{label})$ and $(S_{label}, —)$ are used respectively.

Definition 3.1 A **modeling NFA** is defined by the seven-tuple, **MA** = $(S, \Delta, I, R, L, LS, LT)$. S is a set of states. $\Delta: S \rightarrow S$ is the next state function or transition relation. In some cases, Δ may be partitioned into two phases, Δ^{data} and Δ^{control}. I is the set of operands, $info \in I$, produced, accepted or constrained external or internal to an **MA**. When a precise value of an operand *info* must be distinguished, the notation $info^{val=i}$ is used, where the meaning of i will be made clear from the problem context. R is the set of all resource types (function units, buses, local registers,

3. This may be a time-step period if finer synchronous granularity is used.

etc.), *res* ∈ *R*, which may be required in production of an **MA**'s operands. *L* is the set of all labels, *label* ∈ *L*, used in an **MA**. *LS* is a family of labeled sets of states. $S_{label} \in LS$ where $S_{label} \subseteq S$ and is the set of states referenced by *label*. *LT* is a family of labeled sets of transitions. $\Delta_{label} \in LT$ where $\Delta_{label} \subseteq \Delta$ and is the set of transitions referenced by *label*.

Definition 3.2 A **path** in an **MA** is a potentially infinite sequence of states, $(s_0, s_1, s_2, \ldots)$, such that for each successive pair of states $(s_i, s_{i+1}) \in \Delta$.

3.1.2 Non-Pipelined and Pipelined MA

Figure 3.4 shows a labeled **MA** representing the sequential behavior of a 2 operand in/1 operand out 2 time-step non-pipelined task. Concretely, this might

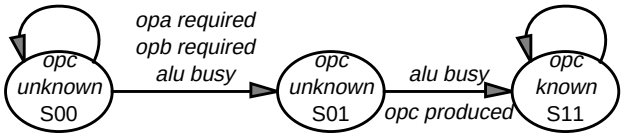


Figure 3.4 An **MA** representing a 2 time-step add task

model an addition task implemented on an ALU hardware resource requiring two clock periods. Once the required input operands are present, this machine may nondeterministically choose to begin a sequence that eventually leads to the output operand being known after two transitions. Also, this sequence to $S_{opc\ known}$ requires that an ALU resources is occupied for two consecutive transitions. Although not shown in figure 3.4, states S01 and S11 are also labeled *opa accepted* and *opb accepted.* In general, $S_{info\ accepted}$ includes all states sequentially following $\Delta_{info\ required}$, $S_{info\ known}$ includes all state sequentially following $\Delta_{info\ produced}$ and $S_{info\ unknown}$ includes all state sequentially proceeding $\Delta_{info\ produced}$.

Figure 3.4 may be modified easily to represent the sequential behavior of a 2 operand in/1 operand out 2 time-step *pipelined* task. Instead of two *alu busy*

labels, a single *alu entry busy* label, as shown in figure 3.5, throttles the number of alu tasks initiated during any time-step. All *alu entry busy* labels are implicitly limited to a maximum concurrent bound during composition. In this way, non-stallable pipelined behavior is simulated. Through the use of additional **MA** state and appropriate **MA** transition labeling, non-pipelined and pipelined behavior of various depths may be represented.

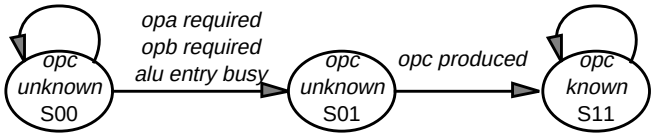


Figure 3.5 An **MA** representing a 2 time-step pipelined add task

Given these more descriptive **MA** examples, several general properties of an **MA** may be highlighted. First, although not explicitly labeled in the figures, there are well defined start state(s) and final state(s) in an acyclic **MA**. A start state has no knowledge of any information produced by the modeled task. A task begins when transiting out of a start state. For the **MA** in figure 3.5, the start state is S00. In general, the set of start states are referred to as $S_{task\ start}$. A task completes at a final state. For a task to complete, all desired information must have been successfully produced. For the **MA** in figure 3.5, the final state is S11. In general, the set of final states are referred to as $S_{task\ final}$. Furthermore, there are nondeterministic paths which allow states in $S_{task\ start}$ or $S_{task\ final}$ to idle indefinitely. The nondeterminism of $S_{task\ start}$ allows an **MA** to delay its task in favor of another **MA**'s more critical task when in a composition. The nondeterminism of $S_{task\ final}$ sequentially records that this task has completed. Finally, paths from $S_{task\ start}$ to $S_{task\ final}$ in an **MA** represent a task's allowed or imposed local sequential behaviors.

3.1.3 MA with Alternatives

Because an **MA** is nondeterministic, it easily encapsulates alternatives. For example, some scheduling problem task may be implemented by a three time-step multiplier or by two executions of a single time-step ALU. This freedom may be directly specified in an **MA** as shown in figure 3.6. Nondeterminism is exploited to provide two (or more) paths from $S_{task\ start}$ to $S_{task\ final}$. Each alternative path may have a different sequential behavior and typically requires a different set of hardware resources. The better choice, if one is better, is discovered during exploration of the composition.[4]

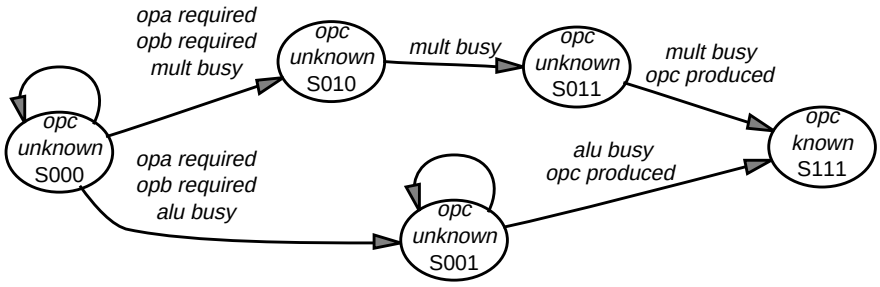


Figure 3.6 An **MA** representing alternative task implementations

3.1.4 Aggregate MA

It is possible to specify an **MA** which may require several fundamental hardware units and implement a more constrained sequential behavior. For instance, a multiply and accumulate (MAC), $r = a \times b + c \times d$, is a commonly implemented task which requires two multiplier uses, one adder use and involves multiple operands. A possible **MA** representing this task is shown in figure 3.7. This **MA** requires operand pairs a,b and c,d to be presented at successive clock edges yet allows either ordering. A three time-step pipelined multiplier and a two

4. Alternative paths should not be confused with control cases (section 3.4). Alternative paths offer multiple possible implementations where *only one* needs to appear in a final solution. On the other hand, control cases offer multiple possible choices where *all* choices must appear in a final solution.

time-step pipelined adder are assumed. After six time-steps, the result operand, *opr*, is produced. This **MA** trades scheduling freedom for less state as well as datapath guidance. By imposing successive ordering on the input operands and forcing the add to begin immediately when its input operands are available, several constrained MAC datapath elements may be inferred.

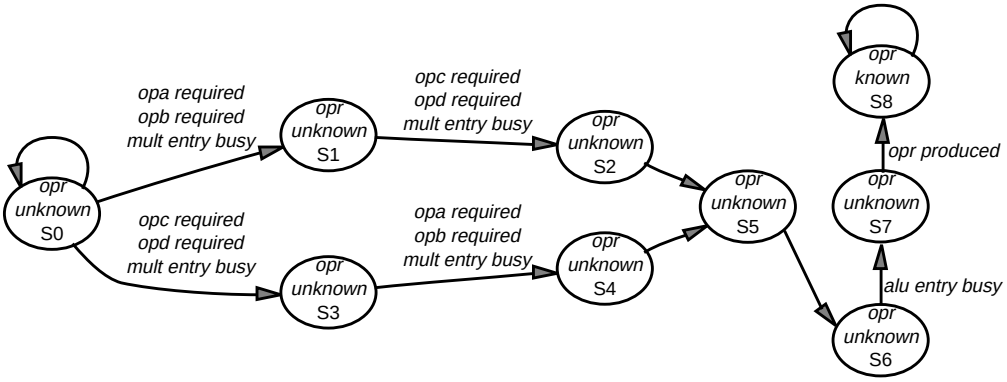


Figure 3.7 An **MA** representing an aggregate MAC

Another MAC datapath element might consist of a non-pipelined two time-step multiplier and a single time-step adder. Furthermore, the timing of the last multiply stage may be such that it may be *chained*[5] with the add. Figure 3.8 shows what an **MA** representing this sequential scenario might look like. Notice that the transition (S5,S6) is labeled with both *alu busy* and *mult busy.* This simply indicates that for this transition to occur, both an alu *and* a multiplier are required at some time within that time-step. This may be interpreted as chaining or as two independent but necessarily concurrent resource uses. Hence a designer is allowed to decide the exact physical meaning of *res busy* labels and the correct interpretation.

Figure 3.8 highlights some additional uses of *res busy* labels. The transition (S5,S6) is also labeled *bus busy.* This assumes that the add result must be communicated over one of possibly several busses to a register. Hence, one use of

5. Both the multiply and add occur in *sequence* within the same time-step.

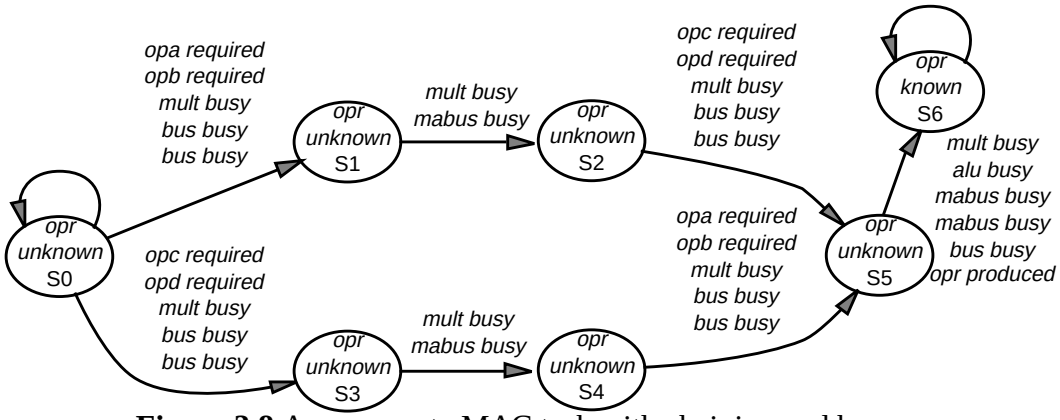


Figure 3.8 An aggregate MAC task with chaining and busses

this bus resource must be counted. Also, the transition (S0,S1) is labeled with two *bus busy* labels. This assumes that a multiplication requires *two* input operands and hence must occupy *two* busses to begin.[6] All *res busy* labels have weight one and multiple *res busy* labels are attached to the same transition to create larger weights. The flexibility and potential of *res busy* labels enables specification of sophisticated concurrency constraints. For example, complex resource hierarchies and partitions may be specified. These are meaningful in the context of a composition and therefore are addressed in chapter 4.

3.1.5 Multiple Output MA

So far, all **MA** have modeled the production of just one output operand, $|P| = 1$. An **MA** with multiple output operands may be specified. As an example, figure 3.9 shows an **MA** for a three time-step divide task which produces quotient and remainder output operands. Although it is possible to specify multiple output operands, it often becomes state costly to represent all desired permutations of when and how long various output operands are *known*. For this reason, it is typically easiest to partition tasks with $|P| > 1$ into several **MA**, each modeling the production of just one output operand. When composed, these **MA** can still be

6. MAC internal transfers are via the multiplier-to-adder bus network, *mabus*.

sequentially related through use of a sequential constraint or protocol as described in section 3.3.

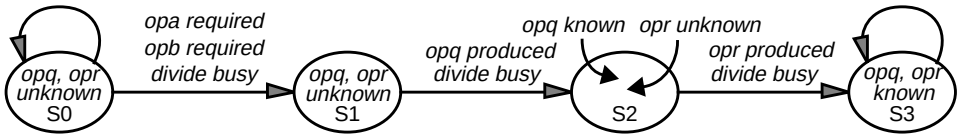


Figure 3.9 An **MA** representing a divide task with two output operands

3.1.6 Recomputation and Memory

The **MA** described thus far have had persistent memory. Once a task computes an output operand, it is never forgotten. It may be that register storage is limited and it is preferred not to store a result but rather recompute it when needed. Figure 3.10 shows an **MA** with the option to *forget* a result operand. This transition forces the **MA** to $S_{task\ start}$ and the task must be reimplemented to recreate the result operand. If all **MA** in a composition have $\Delta_{info\ forget}$ transitions, then considerable freedom is added and hence solution space expansion occurs. This is especially true if an **MA** is allowed to *forget* under any circumstance --even when the result operand was never used! Hence, to use $\Delta_{info\ forget}$ transitions effectively, a designer should only specify them when and where potentially helpful. Furthermore, when constructing the composition, $\Delta_{info\ forget}$ transitions should only be enabled after at least one child task has accepted the result operand.

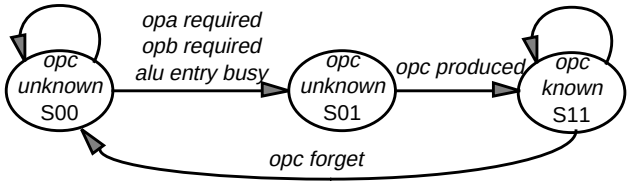


Figure 3.10 An **MA** representing a 2 time-step pipelined add task with forget

Register usage is another important resource constraint. This may be modeled either in an **MA** or through interpretation of a composition. Figure 3.11 shows an **MA (**with label SUBJECTS suppressed**)** which explicitly differentiates between

scheduling history, *info known*, and physical storage, *info stored*.[7] Modeling physical storage in an **MA** requires additional state and hence leads to compositions with additional state. This expense can be avoided by enforcing physical storage constraints at the composition level. In a composition, an *info known* label may be inferred to imply physical storage if *info* has yet to be accepted by children tasks. Operand acceptance is indicated with *info accepted* labels. The details of this preferred technique are presented in section 4.1.5.

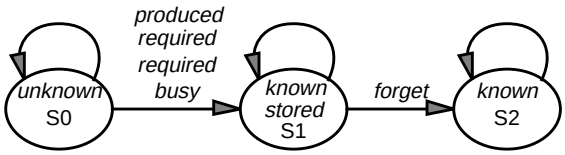


Figure 3.11 An **MA** with explicit physical storage

3.2 Cyclic MA

With looping behavior, a task is scheduled multiple times and iteratively produces result operands. For instance, a RISC processor executing an instruction stream must repeatedly perform a memory read task to access every instruction. Unfortunately, keeping track of multiple coexisting result operands for one particular task can quickly become complex and costly. In fact, it may be that unbounded state is required for full loop exploration. Consider a loop which is unrolled to obtain as much parallelism as possible. Some tasks may produce an infinite number of result operands concurrently if this parallelism is taken to its limit. Specifically, for the RISC example, it is possible that *all* instructions are prefetched from memory and stored locally before any one instruction is executed. This requires tremendous local state and offers little practical benefit.

7. Only one execution of the task is allowed here. With $\Delta_{info\ forget}$ transitions, $S_{info\ known}$ states may be directly interpreted as implying physical storage and another means of determining scheduling history must be utilized.

ABSS bounds this complexity and cost in the formulation of a cyclic **MA**. At a minimum, two successively produced result operands (an *odd* operand labeled with '~' or an *even* operand requiring no additional label) must be distinguished. This can be done with two separate acyclic **MA** but will require additional state and will not have a simple natural repeating behavior. Instead, two acyclic **MA** are overlaid on one set of state encodings as shown in figure 3.12. By doing this, a single state bit can distinguish between *known/unknown* operands in the odd or even iteration sense for a one time-step task behavior. Furthermore, a naturally repeating automaton structure results. Complexity and cost are bounded because an operand may only exist in one iteration sense at any given time-step. Since $S_{info\ known}$ and $S_{info\sim\ known}$ states are mutually exclusive by construction, it is impossible to have simultaneously knowledge of an operand in both iteration senses. The "pigeon hole" analogy provides another way to think of this. A cyclic **MA** reserves one pigeon hole per operand and each pigeon hole has room for one operand. Finally, consider a RISC processor composition where *n* register file read tasks are modeled by *n* cyclic **MA**. Hence, only at most *n* register file read results are ever concurrently available. The sense, odd or even, of a particular register file read result distinguishes whether it belongs to the current or previous instruction.

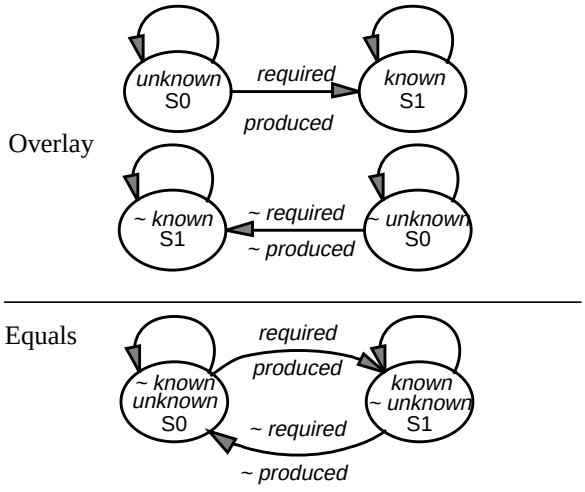


Figure 3.12 Acyclic to cyclic MA transformation

Two senses of information is the minimum needed to distinguish between successive iterative results. It is possible to generalize to a larger number of information senses but is not practical from the local **MA** perspective. Suppose three iteration senses were used. An **MA** could be constructed which sequences through three result operand instances, *r known*, *r~ known* and *r# known*. As all operand instances in an **MA** are mutually exclusive by construction, this is no better than the two iteration sense model which represents the same sequence as *r known*, *r~ known*, *r known*. More importantly, the two iteration sense model requires less state.

What might be advantageous for some scheduling problems is if a cyclic **MA** allowed multiple *concurrent* (not mutually exclusive) result operand iteration instances. This could be achieved through direct specification of an **MA**. Unfortunately, as pointed out in section 3.1.5, multiple output operand **MA** specification quickly becomes complex. Instead, several (rather than one) single result-operand cyclic **MA** are used to model each task when in a composition. Consequently, additional modeling state is naturally included to represent result operands from multiple concurrent instances of a task.[8] This is analogous to "unrolling the loop" in conventional methods. To give a specific example, a RISC processor composition may represent execution of just one instruction. A pipelined RISC processor executes several instructions concurrently although at different stages. To model multiple concurrently executing instructions, a RISC processor composition is duplicated so as to contain additional modeling state. The duplication process is described in chapter 4. Finally, duplication is demonstrated in chapter 6 to potentially improve solution quality at the possible expense of greater scheduling complexity.

8. When additional modeling state is included in a composition, solutions belonging to minimum state models are still implicitly covered.

3.2.1 Duals in Cyclic MA

As with acyclic **MA**, cyclic **MA** are capable of representing complex sequential behavior and requirements of tasks. The main difference is that a cyclic **MA** contains two symmetric sequences, even and odd, of a task's sequential behavior. Figure 3.13 illustrates this for a two time-step non-pipelined function unit. The top path (S00, S01, S11) represents the sequential behavior for production of an odd result operand. Input operands are ~ *required* in the odd sense and then two time-steps later the result operand is ~ *known*. Likewise, the bottom path (S11, S10, S00) represents the sequential behavior for production of an even result operand. These two paths, as well as the states and transitions along these two paths, are symmetric by construction and are referred to as **duals**. For example, states S00 and S11 are duals, states S10 and S01 are duals, transitions (S00, S01) and (S11, S10) are duals, and finally paths (S00, S01, S11) and (S11, S10, S00) are duals. Although not shown in figure 3.13, states S01 and S11 are labeled *info~ accepted* and states S10 and S00 are labeled *info accepted*. In general, $S_{info\ accepted}$ includes all states sequentially following $\Delta_{info\ required}$ and $S_{info\sim\ accepted}$ includes all states sequentially following $\Delta_{info\sim\ required}$. Finally, both $S_{task\sim\ start}$ and $S_{task\ final}$ include S00. Likewise, both $S_{task\ start}$ and $S_{task\sim\ final}$ include S11.

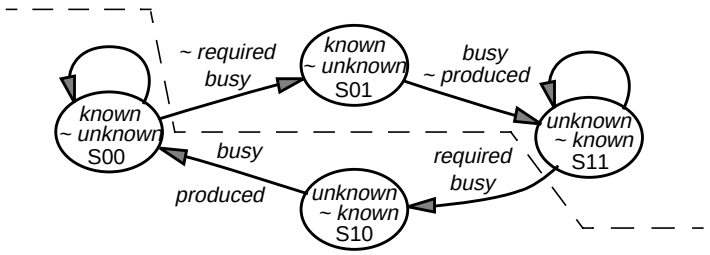


Figure 3.13 A labeled cyclic two time-step MA

An acyclic **MA** may be transformed into a cyclic **MA** by mirroring states and transitions to form dual tasks: *task* and *task~*. Furthermore, $S_{task\sim\ start}$ and $S_{task\ final}$ are overlaid on one set of states. Likewise for $S_{task\ start}$ and $S_{task\sim\ final}$. These states

demark where one iteration of the tasks ends and another begins. Also, when one task iteration ends, the next is ready to begin. Paths in a cyclic **MA** still represent allowed or imposed task sequential behaviors but now may repeat *ad infinitum*. More specifically, a cyclic **MA** may be described as a connected nondeterministic Büchi automaton (accepting infinite sequences) with two symmetric mutually exclusive halves (duals). Finally, to clearly identify duals and simplify upcoming composition exploration algorithms, encodings for dual states are assigned such that a dual encoding may be found by simple bitwise complementation. For example, the dual of a state encoded ‘1011’ would be ‘0100’.

Section 3.1.6 described how an acyclic **MA** is specified to model physical storage and task result operand recomputation. This flexibility is not lost when specifying cyclic **MA**. Figure 3.14 shows a cyclic **MA** which models a two time-step task that allows result operand recomputation. A dashed line divides this **MA** into symmetric halves. Task start/final states are S0 and S7. A cyclic **MA** that models physical information storage may be specified in a similar fashion.

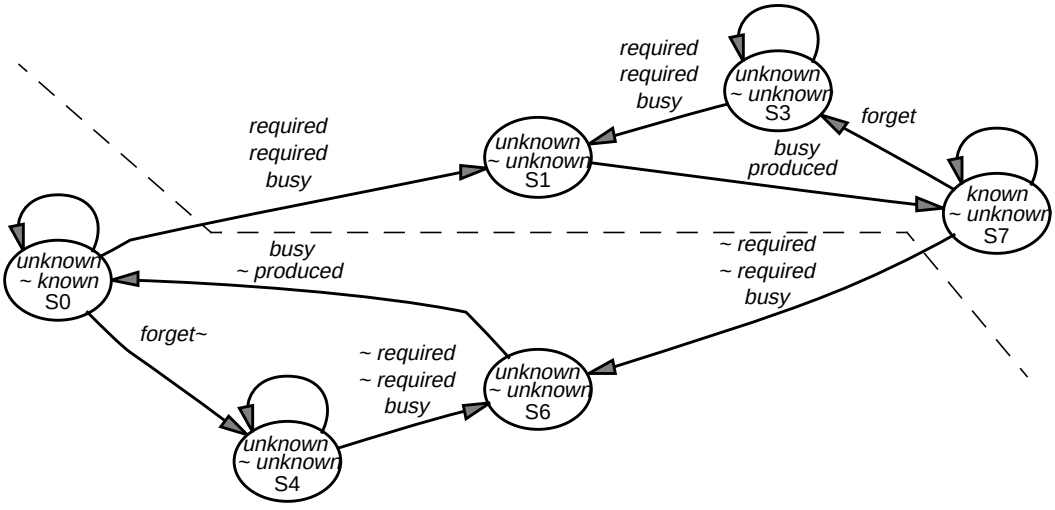


Figure 3.14 An **MA** for a cyclic 2 time-step task with recomputation

3.3 Protocol MA

All **MA** discussed up to this point are specifications of task sequential behavior, $Q \in t \in C.T \in SP$. They model anticipated sequential behaviors of hardware units implementing tasks. An **MA** may also specify sequential or protocol constraints desired by the designer or required for external interface. Such **MA** are elements of scheduling problem sequential behavior, $Q \in SP$. These **MA** constrain how several other **MA** sequentially interact. This can range from governing how a few task **MA** interact to how an entire composition must interface to the external world.

A scheduling problem may represent a portion of a larger design that must interface to the remainder of the design via specific IO protocols. In this case, the IO protocol constraints may be represented as several **MA** and are elements of $Q \in SP$. For example, suppose a designer knows that a subsystem must communicate through one IO port and alternate between input and output transactions. Furthermore, an arbitrary delay is permitted between input and output transactions. Figure 3.15 represents this sequential constraint as an **MA**. When IO protocol **MA** are composed and co-executed with other traditional **MA**, composition constraints ensure that appropriate local **MA** transitions are synchronized across the composition. In this example, all task **MA** transitions requiring an input operand via the IO port must be synchronized with *input* or *input~ required* transitions in this protocol **MA**. Likewise for IO port output.

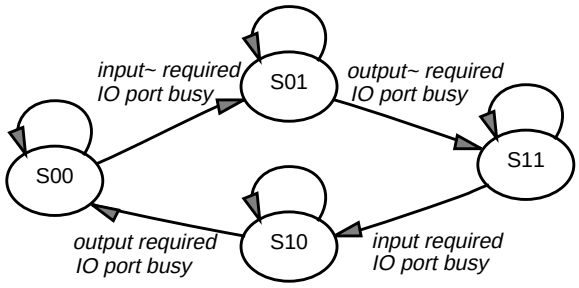


Figure 3.15 A protocol constraint **MA**

In other situations, the scheduling problem may contain a complicated internal task which is not easily described with a single **MA**. Instead, several simple **MA**, each describing only the existence or non-existence of one of the task's result operands and called **operand MA**[9], are co-executed with a sequential **MA** in the composition. In this way, the sequential **MA** imposes the task's correct sequential behavior while the several operand **MA** keep track of information existence. Although decomposing complex tasks into multiple **MA** may require more state than a single **MA** representation, it is not necessarily more expensive to represent as a ROBDD. Furthermore, complex tasks may now be easily specified in understandable small pieces yet be represented completely and in full complexity when composed.

Sequential and protocol constraint **MA** belong to $Q \in SP$ and constrain how several **MA** may sequentially interact. The cardinality of $Q \in SP$ may be greater than 1. These **MA** do not represent tasks which consume and produce information but instead regulate how information interacts sequentially in a composition. Sequential and protocol constraint **MA** may be applied at varying scopes. The simplest sequential **MA** might constrain how two internal task or operand **MA** are sequentially ordered. A complex sequential **MA** might dictate how an entire scheduling problem composition must sequentially interface to the outside world. As an example, the RISC processor may require that a sequential reconfiguration penalty is paid to switch from successive memory reads to memory writes and vice versa. Finally, sequential and protocol constraint **MA** have labeled transitions and states, drawn from table 3.1, that are synchronized with transitions and states in other **MA** during the composition process described in chapter 4.

9. Operand **MA** are single time-step acyclic or cyclic **MA** as shown in figure 3.12.

3.4 Control and Multivalue MA

The **MA** discussed thus far have assumed that actual operand *values* are independent from system behavior. Hence, only the existence or non-existence of a result operand, not its type or value, are modeled. Furthermore, no predefined mutual exclusiveness of operands has been made. Each **MA** independently models sequential production of typically one or a small set of result operands. When several **MA** are composed to represent system behavior, this operand independence is still retained. Consequently, in a composition, potential for all possible concurrent combinations of operand existence are modeled. Although this does encapsulate all potential system behavior executions, it is an overestimation.

With control-dependent behavior, the assumption that actual operand values are independent from behavior is not valid. The very name ‘control-dependent behavior’ implies that control operand values do alter behavior. Hence, some new **MA**, which does distinguish between some possible operand values, is needed. In general, maintaining complete possible concurrency for all operand *values* is potentially costly and not necessarily required. For example, the value of a decoded RISC processor is necessary to select appropriate executions. On the other hand, once an instruction is decoded, it refers to exactly one instruction and need not represent multiple concurrent instructions. In this section, multivalue **MA**, meeting these needs, are introduced.

3.4.1 Intuitive Multivalue MA.

A generic control structure may require a control operand with potential values in the range (0,1,...,*n*) which identify 0 to *n* possible cases of behavior. Figure 3.16 shows an intuitive acyclic multivalue **MA** that models single time-step production of a result operand required for this case statement. Rather than just *info unknown* and *info known* labels, labels identifying the *known* operand value are used. Since

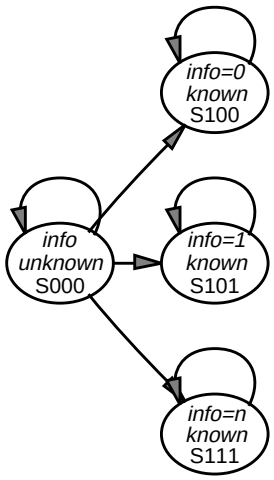


Figure 3.16 An intuitive acyclic multivalue single-time step **MA**

the desired final system execution sequences must be deterministic[10], the model need not support multiple *concurrent* values of this result operand. Mutually exclusive states, each uniquely and typically logarithmically encoded, differentiate between *values* of this result operand. Figure 3.17, is figure 3.16 transformed through standard steps of mirroring and overlaying task start/final state(s) to create a cyclic multivalue **MA**. Finally, as with regular **MA**, multivalue **MA** only allow one known operand instance at a time and are subject to the same 'pigeon hole' state bounds as discussed in section 3.2.

As with a regular **MA**, a multivalue **MA** may encapsulate complex sequential behaviors and nondeterministic alternatives. This tends to be tedious and complex as both sequential behaviors and all possible values of result operands must be considered in tandem. To avoid this, a multivalue **MA** is expressed as a composition of two **MA**. One **MA** specifies sequential behaviors and result

10. The modeling technique uses nondeterminism. The modeled system is deterministic. In a real system, *after* an operand is computed, its precise value is known. Hence, there is no need for multiple operand values to exist concurrently *after* computation. This does not exclude speculation which presumes *all* or *some* operand values *before* actual computation.

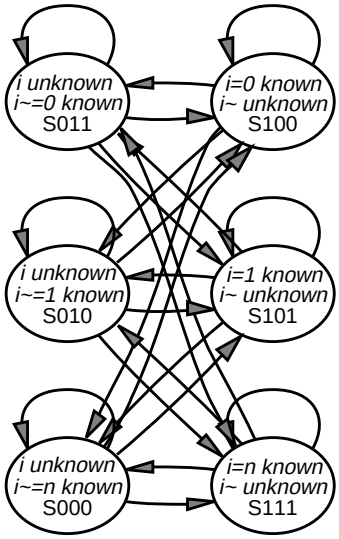


Figure 3.17 An intuitive cyclic multivalue single-time step **MA**

operand existence/non-existence but ignores result operand values. This **MA** is in fact a regular cyclic or acyclic **MA** as previously discussed in detail. The second **MA** specifies result operand existence/nonexistence as well as result operand values but no complex sequential behavior. This **MA** is a single-time step multivalue **MA** similar to what is shown in figures 3.16 or 3.17. During composition, result operand existence/nonexistence for these two **MA** are synchronized. This more understandable partitioned approach achieves the desired sequentially complex yet multivalue **MA**. Consequently, only single time-step acyclic and cyclic multivalue **MA** need be discussed.

3.4.2 Validation Issues and Local MA Solutions

When a multivalue **MA** is used, execution for *every* result operand *value* must be guaranteed. Consider if a particular control case is known to never occur, then it need not be modeled as a distinct control operand value. Hence, by problem construction, execution completion for every result operand value of a multivalue **MA** is necessary. Also, if a particular result operand value does not differentiate behavior, then it need not be distinguished as a unique case in a multivalue **MA**.

Consequently, all multivalue operand cases are distinct, impact behavior in some way and must be covered by some correct execution sequence of the system. Finally, a collection of execution sequences covering all cases, called an **ensemble schedule**, must be causally compatible or valid. Validation is the process of determining causally compatible ensemble schedules. It becomes important during the exploration of a composition, **CMA**, and is discussed in detail in chapter 5.

To simplify validation, a local change, discussed here, is made to every multivalue **MA**. Consider the behavior described in figure 3.18. There are three

```
if (c > 100)
   res = a + b;
else
   res = a - b;
```

Figure 3.18 Behavior to highlight validation issues

tasks to execute: *compare*, *add* and *subtract*. If only two single time-step ALUs are available to implement these three tasks, a possible minimum latency execution sequence is *add* and *compare* in one time-step. This covers the *compare true* case and speculates[11] on the *add*. The *compare false* case must be covered also and may be done with *subtract* and *compare* in one time-step. This execution sequence speculates on the *subtract*. Although these two execution sequences form an ensemble schedule (they cover all *compare* result operand cases), the ensemble schedule is not valid. The proposed ensemble schedule requires speculation of both the *add* and the *subtract* as well as performing the *compare* all in one time-step. This requires three ALUs and is hence infeasible. In chapter 5, validation avoids invalid ensemble schedules by insisting that the following proposition is always true: “At any *single* state in any ensemble schedule occurring *before* a case operand is *known*, *every* possible case operand still eventually completes

11. To speculate on a task is to begin execution of a task before it is absolutely certain that the result operand(s) will be required.

execution." Although the details of validation are left to chapter 5, validation is facilitated by adding *resolve*-labeled states to all multivalue **MA**.

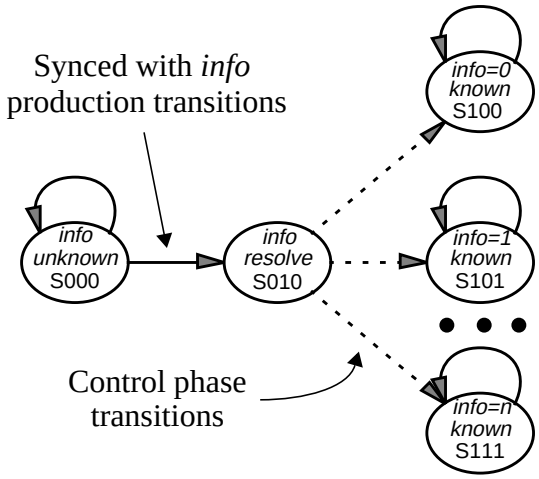


Figure 3.19 A two-phase acyclic multivalue single-time step **MA**

Figure 3.19 shows figure 3.16 modified to include a *resolve*-labeled state, $s_{info\ resolve}$. This new state is inserted between $S_{info\ unknown}$ and $S_{info\ known}$ states. It is a deterministic state and provides a checkpoint for validation to ensure that execution sequences for *every* possible operand value exist. Validation checks that all $s_{info\ resolve}$-exiting transitions still exist in a valid ensemble. When in a composition, a constraint insures that the multivalue **MA**'s $s_{info\ resolve}$ occurs in sync with *info* production in the mated regular **MA**. Figure 3.20 shows a cyclic multivalue **MA** with states $s_{info\ resolve}$ and $s_{info\sim\ resolve}$.

The **MA** in figures 3.19 and 3.20 are also the first examples of **multiple phase MA**. So far, a transition in an **MA** represents activity during a complete time-step, typically a clock period. In a multiple phase **MA**, a transition still represents activity during a complete time-step, but there may be *different sets* of transitions for *different* time-steps. Imagine a clock period divided into two phases, data and control. During the data phase, all activity associated with data production and communication is handled. During the control phase, all activity associated with

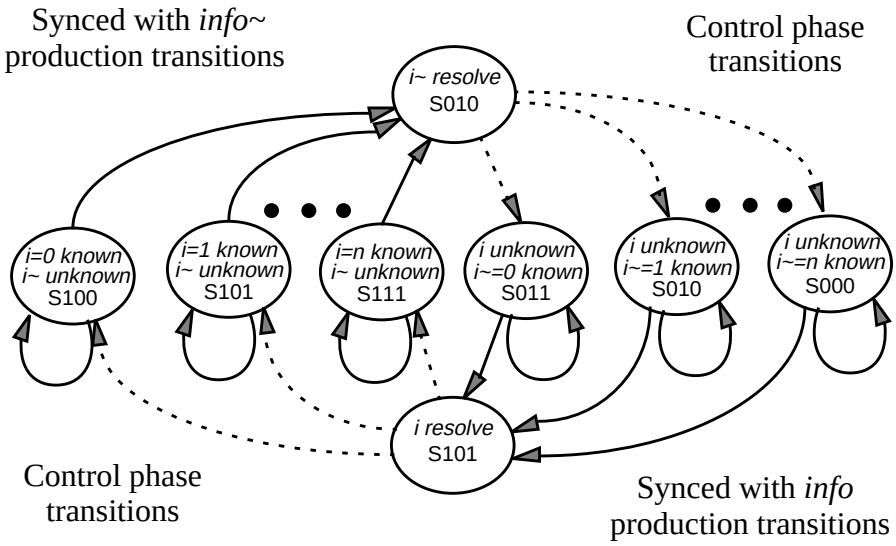


Figure 3.20 A two-phase cyclic multivalue single-time step **MA**

actual operand values and control decisions is handled. Transitions belonging to the data phase are shown solid while those belonging to the control phase are dotted. A two-phase approach permits addition of $s_{info\ resolve}$ states while still maintaining a consistent synchronous view in all models. Finally, a two-phase approach simplifies other control related issues such as task bypassing.

3.4.3 Control-Obviated Task Bypassing

With control-dependent behavior, some portions of the behavior are never required. For example, if the *compare* in figure 3.18's resolves true, then it is not necessary to execute the *subtract*. For acyclic **MA** compositions, a control-obviated task may remain unexecuted. On the other hand, for cyclic **MA** compositions, control-obviated tasks must be force bypassed to the next iteration's task start state(s), $S_{task\ start}$ or $S_{task\sim\ start}$. If this were not done, confusion would occur in a composition as to what particular iteration various control blocks were executing. Furthermore, by bypassing control-obviated tasks, they are correctly primed and may immediately begin speculative or nonspeculative execution of the next iteration. Although task bypassing is a necessity for constructing a correct

composition, additional bypass transitions must be added at the **MA** specification level to facilitate this. Finally, although acyclic **MA** compositions do not require task bypassing, it does simplify composition behavior termination detection. Hence, task bypassing is included for both acyclic and cyclic **MA** which model control-dependent behavior.

Figure 3.21 shows a two time-step pipelined task with task bypassing. A

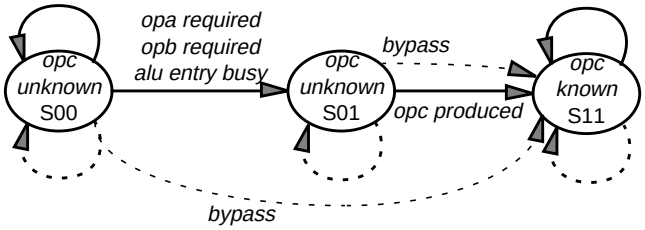


Figure 3.21 An **MA** representing a 2 time-step pipelined task with bypass

$\delta_{task\ bypass}$ transition is added from any state which is *not* in $S_{task\ final}$ to $S_{task\ final}$. More formally, $\Delta_{task\ bypass} = (\overline{S_{task\ final}}, S_{task\ final})$. All $\Delta_{task\ bypass}$ occur during the control-phase and are hence shown as dotted arcs. Furthermore, figure 3.21 also shows what other transitions exist during the control-phase for regular acyclic **MA**. These transitions simply hold the current state during the control-phase so that no change (other than an enabled bypass) may occur. Figure 3.22 shows a cyclic **MA** with task bypassing. Symmetric transition sets, $\Delta_{task\ bypass}$ and $\Delta_{task\sim\ bypass}$, are added. These take any state on a *task* or *task~* execution sequence to $S_{task\ final}$ or $S_{task\sim\ final}$ respectively.

3.5 Summary

This chapter described modeling NFA, **MA**, specification. Modeling NFA are the base building blocks used to describe sequential production of a task's result operands. They are the atomic pieces necessary for a first composition. For instance, sequential behavior for atomic tasks in a RISC processor such as register

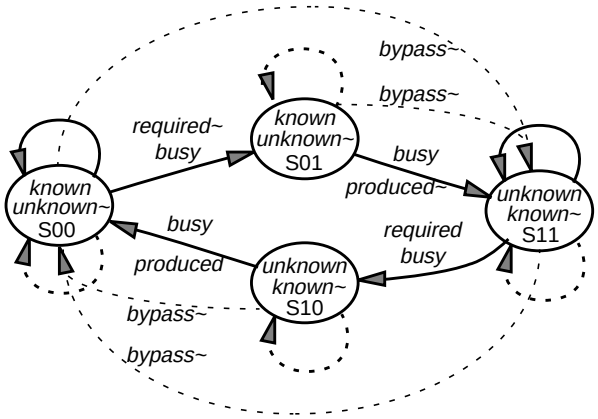


Figure 3.22 A labeled cyclic two time-step MA with task bypass

file reads and writes, ALU computations, memory access, etc., are specified via **MA**. As described, a designer manually and explicitly specifies an **MA** as a labeled nondeterministic state graph. This is kept relatively simple as only a handful of states and transitions are used in any single **MA** specification. When complexity becomes too great, a designer relies on and leverages the power of composing several **MA**. State and transition labels assign meaning to local sequential behavior as wells as provide “hooks” for creating correct compositions. Various useful types of acyclic, cyclic and multivalue (control) **MA** were illustrated and described.

Chapter 4

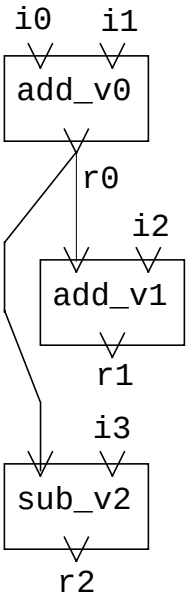
Composing Modeling Automaton: CMA

Chapter 3 describes how the sequential behavior of one task in a composite task, $Q \in$ *composite task*, is constrained and represented by a modeling automaton, **MA**. In this chapter, sequential behavior is assumed to be assigned for each task in a composition via its **MA**. All such **MA** are composed through a Cartesian product step to create a new, larger composite modeling automaton, **CMA**, which represents the as yet unknown sequential behaviors of the composite task. The Cartesian product step is not sufficient to create a correct **CMA**. This chapter describes how a **CMA** is pruned through a series of dependency, capacity and viability refinements. Furthermore, concurrency constraints are applied to a **CMA** to limit available system hardware resources such as function units, busses and local memory. After a **CMA** is correctly pruned, it represents *all* valid execution sequences of a composite task. It is possible to systematically explore this composite model and determine *best*-possible execution sequences. FSM controllers and datapath portions may be synthesized from a **CMA** path or ensemble subset of **CMA** paths. Finally, a pruned **CMA** is itself an **MA** and may be used in further compositions in a hierarchical fashion.

With regard to the RISC example, the end goal of composition is to create a NFA that represents all valid executions of all instructions. This requires modeling control-dependent execution as well as cyclic behavior. To reach this level of sophistication, it is helpful to first discuss the composition process for simpler scheduling problems. This chapter first presents the composition process for acyclic data-flow scheduling problems. After this introduction, composition differences for cyclic data-flow, acyclic control-dependent and cyclic control-dependent scheduling problems are presented and discussed in that order. Common constraint formulations for all types of compositions are generalized. Finally, particularly efficient **CMA** construction and implementation techniques derived both from experience and problem insight are described.

4.1 Acyclic Data-Flow Composition

Consider the simple acyclic data-flow example shown in figure 4.1. This




```
add_v0 = (   (i0, i1), (r0),
             (ALU, In Port, In Port),
             (Single time-step) )

add_v1 = (   (r0, i2), (r1),
             (ALU, Out Port, In Port),
             (Single time-step) )

sub_v2 = (   (r0, i3), (r2),
             (ALU, Out Port, In Port),
             (Single time-step) )
```

Figure 4.1 A simple acyclic data-flow example

composite task represents a behavior that requires four input operands and produces two output operands. This problem is small enough that the minimum

latency schedule, `add_v0` in time-step 1 followed by `add_v1` and `sub_v2` in time-step 2, is obvious. Note that `add_v1` and `sub_v2` must follow `add_v0` as they depend on the result, `r0`, produced by `add_v0`. Each of the tasks in figure 4.1 is represented by its own **MA** in the composition as shown in figure 4.2. These **MA** have label SUBJECTs that reflect operands and hardware resources in the composite task. For example, there is a specific **MA** for task `add_v0` which requires input operands `i0` and `i1`, occupies an ALU resource[1], and produces result operand `rv0` all in a single time-step. A **CMA** is created by first composing these **MA** and then applying pruning steps. The remainder of section 4.1 discusses the composition and pruning steps with respect to this example.

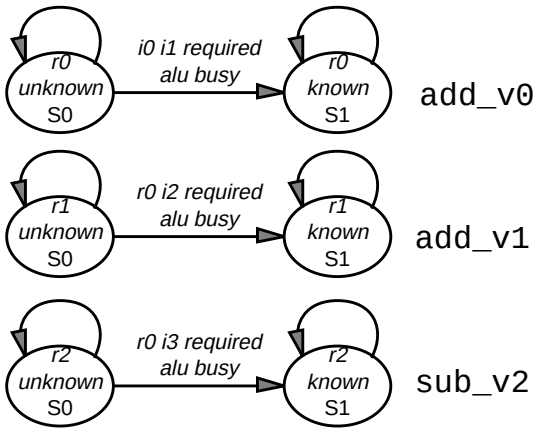


Figure 4.2 Assigned **MA** for each task in the example

4.1.1 The Cartesian Product Composition Step

A Cartesian product of figure 4.2's three **MA** is shown in figure 4.3. There are two important traits to note about a **CMA** that are evident from this figure. First, to form a true Cartesian product, state variables for each task **MA** are encoded with a *unique* set of ROBDD variables in a **CMA**. This is seen in figure 4.3's composition state vectors. The composition state vectors require three Boolean state variables,

1. *Bus busy* labels are left out to simplify the figures.

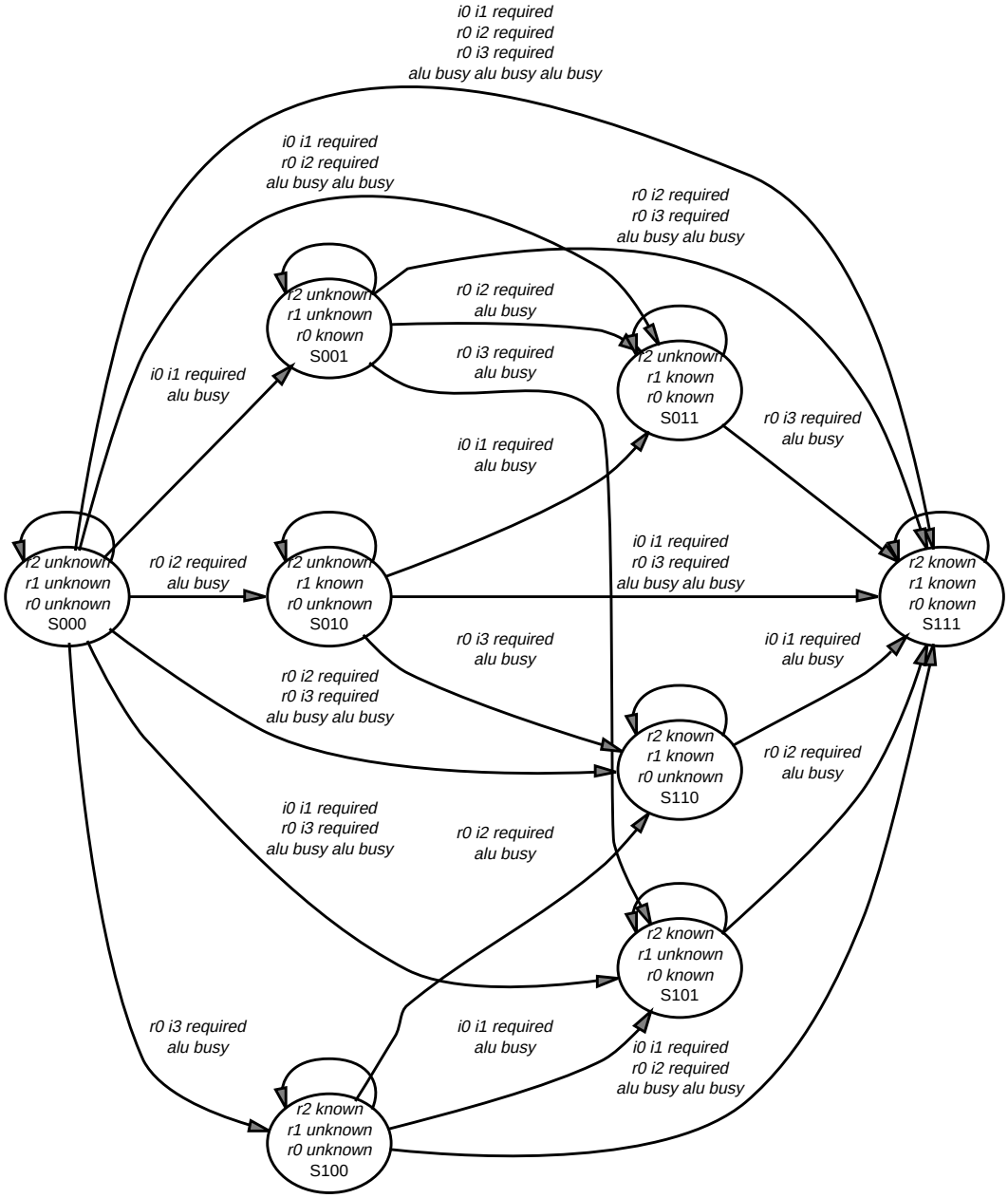


Figure 4.3 Cartesian product composition step for the example

as each task **MA**'s state encoding contributes exactly one Boolean state variable. In fact, the composition state vectors are ordered {`sub_v2`, `add_v1`, `add_v0`}. Second, because a true Cartesian product machine is created, no task **MA** labels or associated properties are lost. Figure 4.3 shows transition and states with multiple labels that identify specific task **MA** properties. These labels provide the necessary

references to subsets of states and transitions with specific properties required during the composition pruning steps. Finally, although figure 4.3 is shown explicitly, all **CMA** of meaningful scale are implicitly represented as ROBDDs.

Definition 4.1 **CMA** denotes the Cartesian product automaton of several **MA** and is itself an **MA**. Let M denote the set of all **MA**, $m \in M$, composing a **CMA**. Then a **CMA** $= (S, \Delta, I, R, L, LS, LT)$ is composed as $S = \prod_{m \in M} m.S$ and[2] $\Delta = \prod_{m \in M} m.\Delta$ where $\prod$ represents a Cartesian product. As with all **MA**, Δ may be partitioned into at least two phases, Δ^{data} and Δ^{control}, and is not necessarily single valued. $I = \cup_{m \in M} m.I$ where $\cup$ represents set union. Likewise, $R = \cup_{m \in M} m.R$ and $L = \cup_{m \in M} m.L$. A labeled set $S_{label} \in LS$ for a **CMA** is constructed as $S_{label} = \cup_{m \in M} m.S_{label}$ when $m.S_{label}$ exists. A labeled state set S_{label} is created for each $label \in L$ for a **CMA**. The set union of all S_{label} forms the composition family of sets LS. Likewise, a labeled set $\Delta_{label} \in LT$ for a **CMA** is constructed as $\Delta_{\text{label}} = \cup_{m \in M} m.\Delta_{label}$ when $m.\Delta_{label}$ exists. A labeled transition set Δ_{label} is created for each $label \in L$ for a **CMA**. The set union of all Δ_{label} sets forms the family of sets LT.

4.1.2 Basic Operand Dependencies

For a single operand *info*, each possible use of *info* must follow its creation. In the RISC example, a result operand must be produced by an ALU computation before it may be written back to the register file. The Cartesian product shown in figure 4.3 has transitions labeled *i0 required* leaving states labeled *i0 unknown*. This violates the core of scheduling: operand dependence, which requires that *i0* may be used only if it exists in the system. These dependency violating transitions and any subsequently isolated states are removed through dependency constraints.

2. Dot notation was introduced in chapter 2, definition 2.3.

As presented in definition 2.8 of the scheduling problem, an operand dependence, $e \in C.E$, pairs produced operand(s) with a consumption point. An operand dependence is specified for every required input operand, $a \in A$, of every task, $t \in C.T$, in the scheduling problem. Operand dependency constraints apply this information, operand dependence, to a **CMA** model of the scheduling problem.

A basic operand dependence is of the form (*task1.P.info*, *task2.A.info*). In the basic case, the operand dependency's Boolean expression *f* consists of just a single operand. *Task1's* locally produced operand *info* is solely required as *task2's* local input operand *info.* This is equivalent to a data dependency edge in a traditional DFG.

A single **basic operand dependence**[3], (*taskp.P.info*, *taska.A.info*), is modeled by the implication,

$$ma.\Delta_{info\ required} \Rightarrow (mp.S_{info\ known}, —).^{4} \tag{4.1}$$

In this implication, *ma* is the accepting task's **MA** while *mp* is the producing task's **MA**. This construct insures that any transition labeled *info required* may only occur if the required information is present in the system. The labeled state set $S_{info\ known}$ in the present state indicates that the required information exists in the system. Another way to view this is that a present state labeled $s_{info\ known}$ *enables* (yet does not demand) an information accepting $\delta_{info\ required}$-labeled transition. Let ζ_e represent expression 4.1 for one operand dependence, *e.* The refinement of Δ^{data} for a **CMA** is,

$$\Delta^{n+1} = \Delta^{n} \bullet \prod_{e \in C.E_{basic}} \zeta_e . \tag{4.2}$$

3. Other dependency constructions (Alternative, Resolved and Undetermined) will be introduced.

4. In a ROBDD, implications may be built as $p \Rightarrow q = \overline{p\bar{q}}$.

Equation 4.2 only shows refinement of $\Delta^{data} \in$ **CMA**. All refinements are defined on Δ^{data} and/or $\Delta^{control}$ while refinements of other sets of any **CMA** are implicit. In practice, $S \in$ **CMA** is never explicitly stored but always determined from Δ.

Figure 4.4 shows figure 4.3 after all dependency violating transitions and states

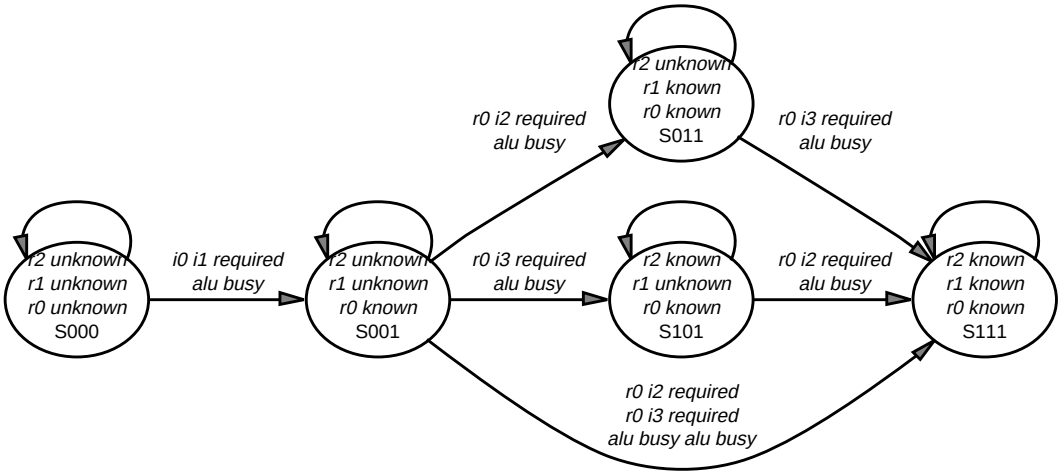


Figure 4.4 Dependency-constrained explicit **CMA** for the example

are pruned. Now if a transition is labeled *info required*, it will only exit a state labeled *info known*. Hence, only execution sequences which satisfy dependency conditions remain. In fact, *all* such valid execution sequences, even of infinite latency, remain.

4.1.3 Alternative Operand Dependencies

It is possible that two sources for one operand exist in a scheduling problem. For instance, some operand *info* may be computed through multiplication or repeated addition. In section 3.1.3 this same alternative was represented directly in an **MA**. It is also possible to represent this alternative at the composite task level by using several **MA**. Some **MA** represent the repeated addition behavior while another **MA** represents the multiplication behavior. Two equivalent *info* instances are produced, $info^{by\ adds}$ and $info^{by\ multiplication}$. During the exploration process, a

winning alternative, if one exists, becomes evident. This type of nondeterministic scheduling problem behavior has no equivalent in traditional DFG/CDFG representations, but is somewhat similar to optimizing compiler strength-reduction. In general, alternative operand dependencies are useful to represent a single operand produced in several different ways. The precise difference between alternatives is left to the designer. Some useful differences are behavioral (i.e. algebraic transformation, etc.) and spatial (i.e. equivalent operand instances are computed in physical design partition a or partition b [129]).

A single **alternative operand dependence**, (*taskp1.P.info+taskp2.P.info+…+taskpn.P.info, taska.A.info*) is modeled by the implication,

$$ma.\Delta_{info\ required} \Rightarrow ((mp^1.S_{info\ known}, —) + (mp^2.S_{info\ known}, —) + \ldots + (mp^n.S_{info\ known}, —)) \quad . \tag{4.3}$$

In this implication, *ma* is the accepting task's **MA** while mp^1 through mp^n are individual **MA** each producing an equivalent yet alternate instance of *info*. This consequent directly corresponds to an alternative operand dependency's expression *f*. As long as at least one source has produced *info*, then the *info required* condition is satisfied. Implications for all scheduling problem alternative operand dependencies are built and prune a **CMA**'s Δ^{data} in similar fashion to equation 4.2.

4.1.4 Undetermined Operand Dependencies

An **MA** in a composition may produce operands yet require no input operands. A random number generator is an instance of this. Or, an **MA** may require input operands but these input operands are always available or will be supplied later from an external source with no timing constraints. Unconnected or unnecessary input operand consumption points such as these require an undetermined operand

dependence of the form (—, *a.info*). Since the Boolean operand production expression is don't care, —, no pruning of Δ^{data} for a scheduling problem's **CMA** is necessary.

4.1.5 Resource Concurrency Bounds

If only a single ALU is available for the example in figure 4.1, then the two time-step minimum latency schedule is impossible as it requires two concurrent uses of an ALU during the second time-step. Prohibiting these resource violating executions corresponds to removing transitions from the **CMA** where the number of *res busy* labels exceeds the number of concurrent *res* uses allowed in the scheduling problem, $(bound, T_r) \in R \in SP$. This constraint may be built by enumerating all combinations of 0 up to *bound* busy-labeled transitions for a particular resource class. At first glance, this constraint appears to be exponential, but its ROBDD representation requires only time and nodes of order $O(bound \times |\Delta_{res\ busy}|)$ (see section 4.5.2). Let $\mathbf{A}_{res\ busy}$ be the set of all combinations of at most *bound* transitions $\delta \in \Delta_{res\ busy}$. Then,

$$\sum_{A \in \mathbf{A}_{res\ busy}} \left(\prod_{\delta \in A} \delta \right) \tag{4.4}$$

represents all possible transitions which observe resource bounds. Expression 4.4 is built for all resource types and intersected with Δ^{data} to prune resource violating transitions. Let ξ_{res} represent expression 4.4 applied to a resource type $res \in R$. The refinement of Δ^{data} is,

$$\Delta^{n+1} = \Delta^{n} \bullet \prod_{res \in R} \xi_{res}. \tag{4.5}$$

This construct may be generalized to bound any type of transition-based *concurrency* in a **CMA**. As transitions in ABSS represent activity during a time period, any such concurrent activities may be bounded.

It is also desirable to bound the number of operands which are concurrently kept in local storage. This local storage concurrency constraint must be formulated on states as they represent what is stored or available in the system at time-step boundaries. One way to do this is to explicitly add additional **MA** state that differentiates between *info known* and *info stored* as described in section 3.1.6. An alternative preferred method is to identify memory care conditions at the composition level without adding state. A **memory care condition** for some operand *info* is expressed as,

$$mcc_{info} = \overline{mp.S_{info\ known}} \bullet \left(\overline{ma^1.S_{info\ accepted}} + \overline{ma^2.S_{info\ accepted}} + \ldots + \overline{ma^n.S_{info\ accepted}} \right) \quad (4.6)$$

In this equation, *mp* is the **MA** producing *info* while ma^1 through ma^n are all composition **MA** which require *info*. This equation says that if *info* is *known* yet some task requiring *info* has not accepted *info,* then *info* must occupy storage. Let $\mathbf{A}_{memory\ care}$ be the set of all combinations of at most *bound* memory care conditions, *mcc,* for all operands $info \in I$ in a composition. Then,

$$\sum_{A \in \mathbf{A}_{memory\ care}} \left(\prod_{mcc \in A} mcc \right) \quad (4.7)$$

represents all possible **CMA** states which observe memory concurrency bounds. Expression 4.7 is intersected with the present state portion of Δ^{data} to prune memory concurrency violating states.

4.1.6 A Completely Pruned CMA

If a single ALU resource constraint is imposed on the example in figure 4.1, then the completely pruned explicit **CMA** is shown in figure 4.5. This, like any other **MA,** represents all possible constrained sequential behaviors for a task. Execution begins at the task start state where all produced information is *unknown* and proceeds until all produced information is *known*. There are infinite number of

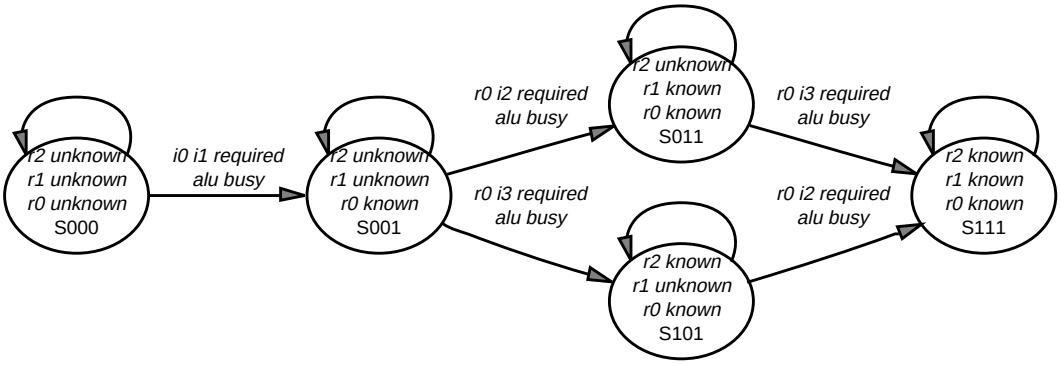


Figure 4.5 Final explicit **CMA** for the example

execution sequences represented as execution may stall indefinitely at various points. A minimum latency execution sequence is the shortest path from task start to final state. For this example, there are two such minimum latency schedules requiring three time-steps.

4.2 Cyclic Data-Flow Composition

Cyclic data-flow composition requires two extensions of the acyclic data-flow composition in section 4.1. First, there are two senses, odd and even, of every operand in a cyclic model. Dependency constraints must be built for both cases. Second, with the notion of two operand senses comes the possibility of iteration-sense confusion.[5] A cyclic data-flow composition must guarantee that only operands from correct iteration instances are produced and accepted. This requires the formulation and application of a capacity constraint as well as a viability pruning step.

To provide an intuitive introduction to cyclic data-flow composition, the explicit **CMA** introduced in section 2.5[6] is duplicated in figure 4.6. An abbreviated

5. Conventionally, iteration sense is maintained by iteration counters for each operand. Iteration counters add substantial overhead to the scheduling process without significantly adding to the representation power.
6. The ABSS scheduling problem for this example is presented in figure 2.4.

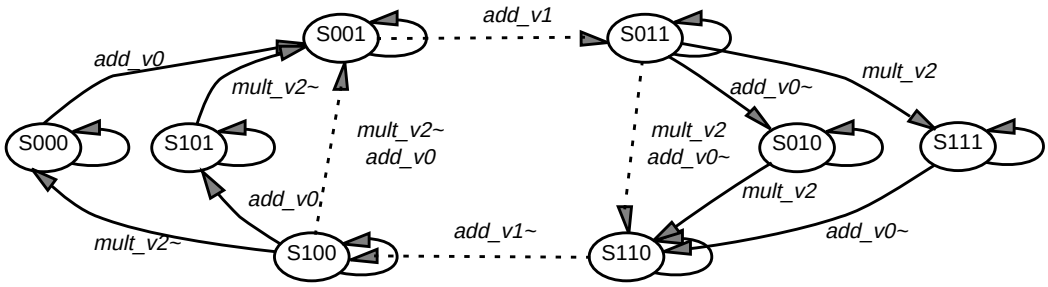


Figure 4.6 An explicit **CMA** for a cyclic data-flow composite task

label system is used here which identifies actual task execution on a transition. State label information must be derived from a state vector ordered, {`mult_v2`, `add_v1`, `add_v0`}, where '1' represents the single output operand *known* in the even sense and '0' represents the operand *known* in the odd or '~' sense. This **CMA** has been completely pruned and allows only one ALU and one multiplier resource. Notice that dependency is observed in both senses. The task `add_v1` depends on the result of `add_v0` as input. Consequently, transitions that schedule `add_v1` only leave states where `add_v0` is *known* and likewise `add_v1~` is scheduled on transitions that must leave states where `add_v0~` is *known*. Furthermore, to prevent iteration-sense confusion, an `add_v0` result is not forgotten until *after* it is accepted by dependent `add_v1`. Finally, only a single ALU and multiplier use occur on any transition, regardless of the task sense.

This, like any other cyclic **MA,** represents constrained sequential behavior for a cyclic composite task. Execution may begin at the task start state where all produced information is *unknown* in one sense and proceed with infinite executions of the loop. There are infinite execution sequences represented as execution may stall indefinitely at various points. Although infinite execution sequences are represented, state is bounded. State represents the set of concurrently *known* operands. As only a single **MA** is assigned per task in this composition, only three operands may be concurrently *known* in the system,

{mult_v2, add_v1, add_v0}. These operands may be known in one of two senses, odd or even, depending of which iteration produced them. Although an operand is always *known* in one sense or the other, this does not imply that it is occupying physical storage but only that it has been produced in that iteration sense. Finally, unlike an acyclic **MA,** a minimum latency execution sequence is not the shortest path from task start to final state. Rather, a *repeating kernel*[7] which overlaps iteration instances achieves minimum iteration latency. For this example, the minimum iteration latency is shown dotted and requires two time-steps as two complete iterations are represented in this four time-step cycle.

4.2.1 Basic Operand Dependencies

In a **CMA** modeling cyclic behavior, there are two senses, odd and even, of every operand. Two dependency implications, for both operand senses, must be built for each basic operand dependence. For *intra*-iteration dependencies, or dependencies within the same execution iteration, the two implication expression 4.8 is built.

$$\begin{array}{l}(ma.\Delta_{info\ required} \Rightarrow (mp.S_{info\ known}, —)) \bullet \\ (ma.\Delta_{info\sim\ required} \Rightarrow (mp.S_{info\sim\ known}, —))\end{array} \tag{4.8}$$

For *inter*-iteration dependencies, or dependencies between two successive loop iterations, the two implication expression 4.9 is built.

$$\begin{array}{l}(ma.\Delta_{info\ required} \Rightarrow (mp.S_{info\sim\ known}, —)) \bullet \\ (ma.\Delta_{info\sim\ required} \Rightarrow (mp.S_{info\ known}, —))\end{array} \tag{4.9}$$

These implications insure that for either operand sense, the required operand must be *known* to the system to allow any *required*-labeled transitions. Let ζ_e represent either expression 4.8 or expression 4.9 depending on *inter*/*intra*-iteration depen-

7. Repeating kernels are discussed in section 5.3.1.

dency type. The refinement of Δ^{data} for a cyclic **CMA** is again as shown in equation 4.2.

An operand dependence tuple contains no explicit information regarding *inter*- or *intra*-iteration type. Rather, the scheduling problem is defined to be acyclic or cyclic. If a scheduling problem is cyclic, then an operand dependence is written with consumption point, *a*, always in the even sense while the produced operands in *f* may be in the even or odd '~' sense depending on *inter*- or *intra*-iteration type. For example, the operand dependence (*info, info*) represents an *intra*-iteration dependence while (*info~, info*) represents an *inter*-iteration dependence. All operand dependencies only specify the first implication of either expressions 4.8 or 4.9 while the second implication is inferred.

Since a cyclic **MA** represents only two instances of an operand, it is only possible to formulate *intra*-iteration dependencies (within the same iteration), or *inter*-iteration dependencies (between successive iterations) with a single **MA**. If dependencies must be written across several iterations, then additional **MA** are added to model the additional operand instances. This may be done in two ways. First, *iterates*, as described in section 4.4.2, build several instances of the entire composite task. Hence, several iteration instances of all operands are available. This approach is required to model several executing instructions in a pipelined RISC example. Second, operand buffers, as described in section 4.5.4, provide a local way to model storage for several instances of one particular operand within a **CMA**.

4.2.2 Operand Capacity Constraints

Operand capacity constraints maintain operand iteration-sense consistency in cyclic scheduling problems. They are not needed for acyclic scheduling problems. Consider a basic operand dependence (*taskp.P.info, taska.A.info*). It is possible,

due to **MA** nondeterminism, that the source, *taskp*, may continuously produce operand *info*, (*info, info~ info, info~*, …), yet the sink, *taska*, idles and never accepts or accepts some *info* operand a few iterations later. This is not only wasteful but also leads to operand iteration-sense confusion. Which iteration of *info* did *taska* actually accept? What iteration do any of *taska*'s result operands really belong to? Information capacity constraints avoid this confusion by insisting that a particular produced operand remains *known* (is not forgotten) in the current iteration sense until all accepting tasks have accepted this operand in the current iteration sense. Hence, operand iteration-sense consistency is maintained throughout a composition.

Definition 4.2 Iteration-Sense Confusion occurs when an operand dependency implication is satisfied strictly by operand sense yet the satisfying operand is *not* from the iteration expected by the accepting task.

For the RISC example, operand capacity constraints insist that intermediate operands relevant to one instruction are accepted before intermediate operands for the next instruction are produced. For example, suppose instructions *i4* and *i5* both require an operand from the register file. The correct register file read operand for instruction *i4* may be in the even sense, *rfread known,* while the correct operand for instruction *i5* may be in the odd sense, *rfread~ known*. Capacity constraints insist that the *i4* register file read operand, *rfread known,* is not overwritten by the *i5* register file read operand, *rfread~ known,* until all *i4* tasks requiring *rfread known* have accepted. Since only one **MA** represents this particular *rfread* operand, only one instance may exist. Capacity constraints insure that this single operand instance is kept in the iteration sense expected by all accepting tasks.

4.2.3 Basic Operand Capacities

The capacity constraint for a single basic operand dependence, (*taskp.P.info*, *taska.A.info*), is modeled by the implication,

$$mp.\Delta_{info\ forget} \Rightarrow ((ma.S_{info\ accepted}, —) + (—, ma.S_{info\ accepted})) . \quad (4.10)$$

This is opposite from the dependency implication in expression 4.1 as the enabled transition in expression 4.10 belongs to the *producer* rather than the *accepter.* Expression 4.10 insures that transitions labeled *info forget* may only occur if the required information is accepted in the present state or will be accepted by the next state. The implication's consequent is written in terms of the next state so that the producing task's **MA** may forget an operand during the same time-step that an accepting task's **MA** accepts. Figure 4.7 illustrates how this relates to a real sys-

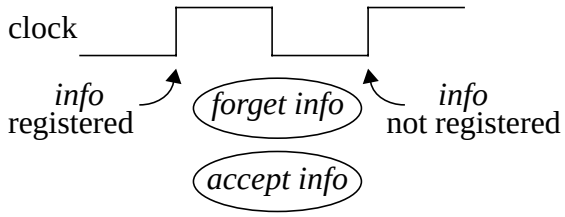


Figure 4.7 Simultaneous *info* forget and accept

tem. ABSS transitions correspond to activity during time-step periods. The operand *info* is forgotten and accepted during the same time-step period. ABSS state corresponds to information available at a clock edge or time-step boundary. At the start of this time-step period, *info* was registered and hence may be accepted. During this period, *info* is forgotten and hence not registered at the next clock edge or time-step boundary. More concretely, this correctly represents pipelined behavior where pipe stage a is forgetting *resulta1* and computing *resulta2* while at the same time pipe stage b is accepting *resulta1* and computing *resultb1*.

Let ξ_e represent expression 4.10 for one basic operand dependence, *e*, yet built for both operand iteration senses. The refinement of Δ^{data} for a **CMA** is,

$$\Delta^{n+1} = \Delta^n \bullet \prod_{e \in E_{basic}} \xi_e . \tag{4.11}$$

4.2.4 Undetermined Operand Capacities

As with undetermined operand dependencies, no refinement of a **CMA**'s Δ^{data} is required for undetermined operand capacities. Capacity constraints are important for produced operands of tasks with undetermined operand input dependencies. Since such tasks have no constraints on when *info required*-labeled transition may occur, they would be free to behave in any way if it were not for capacity constraints on their produced operands. For example, a cache hit/miss multivalue **MA** cannot produce its next cache hit/miss value until the current operand is no longer required.

4.2.5 Alternative Operand Dependencies and Capacities

An alternative operand dependence leads to iteration-sense confusion in a cyclic composition. Consider the sequence shown in figure 4.8 where task `acp` may use the operand from `src_a` or `src_b`. In frame 1, `src_a` is *known* in the

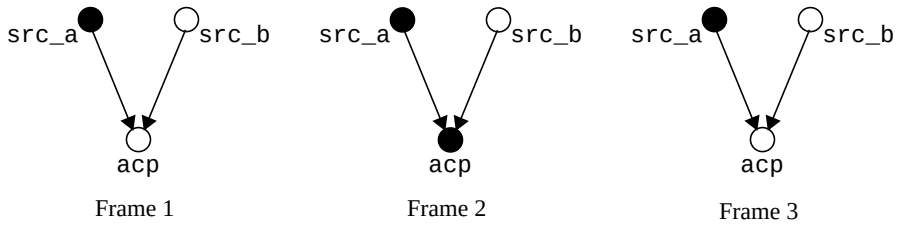


Figure 4.8 Iteration-sense confusion when alternatives are present

even sense as indicated by the solid circle. Consequently, in frame 2, `acp` accepts this alternative result. In frame 3, it appears as if `src_b` is ready in the odd sense so `acp` accepts this alternative result for the next iteration. Unfortunately, `src_b` is really from a previous odd iteration and iteration-sense confusion results.

Furthermore, `src_b` may not even produce a current even result as a capacity constraint from `acp` to `src_b` prevents `src_b` from forgetting its odd result. Although `acp` has in fact moved forward, `src_b`'s odd result is still kept since `acp` has not accepted in the odd sense. This situation is referred to as *capacity deadlock*.

Definition 4.3 Capacity Deadlock occurs when an operand is kept in its current iteration sense even though it is not needed. It typically occurs when an accepting task some how accepts *ahead* of a potential producer.

Alternative dependencies are not allowed in cyclic compositions as they lead to iteration-sense confusion and capacity deadlock. Instead, alternatives are represented through use of nondeterministic control as discussed in section section 4.4.7.

4.2.6 Viability Prune

Although all **CMA** states and transitions now observe dependency and capacity constraints, there are still execution sequences present which exhibit iteration-sense confusion. Figure 4.9 shows a three task loop and some dependency

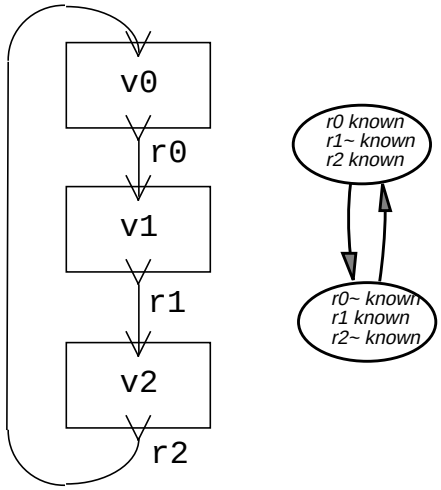


Figure 4.9 Cyclic scheduling problem and an acausal partial **CMA**

and capacity refined **CMA** states and transitions. From these **CMA** states and transitions**,** it appears as if a new iteration instance result, `r2`, is produced *every* time-step. However, the inter-dependency edge from `v2` back to `v0` requires that one iteration complete before the next iteration may begin. Hence, a minimum iteration latency schedule must require at least three time-steps.

If execution were to begin at a **CMA** task start state set[8], the state(s) where all composition member **MA** are in their task start state(s), $S_{task\ start}$, then it is impossible, due to dependency and capacity constraints, to reach iteration-sense violating states and transitions. Consider the illustration in figure 4.10. All states

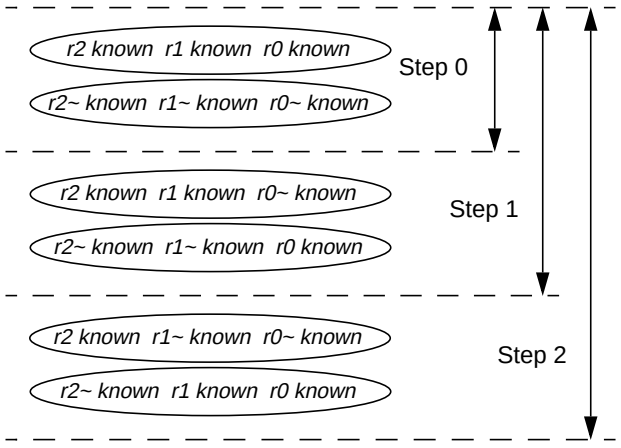


Figure 4.10 CMA States reachable from $S_{task\ start}$

reachable from $S_{task\ start}$ are found after two time-steps and do not include the erroneous states shown in figure 4.9. These erroneous states represent multiple iteration instances of a single loop which is impossible given a natural initial entry of a loop.

Symbolic image, preimage and reachability computations[31][32][90] are important for viability pruning as well as future **CMA** exploration. Given the

8. There is a single $S_{task\ start}$ for acyclic models while there are dual task start state sets, $S_{task\ start}$ and $S_{task\sim\ start}$, for cyclic models. Likewise for **CMA** task final state sets, $S_{task\ final}$ and $S_{task\sim\ final}$.

relation $\Delta \in$ **CMA**, all next states of some set $S_{label} \subseteq S$ may be determined by the expression,

$$\exists_{(X, —)}(\Delta \bullet (S_{label}, —)) . \tag{4.12}$$

In expression 4.12, the inner product determines all transitions of Δ with predecessors belonging to S_{label}. The set X represents all ROBDD variables used to encode **CMA** state. All predecessors states are removed through existential quantification leaving only successor states. This is commonly referred to as a next state or image computation. Likewise, a previous state or preimage computation can be expressed as,

$$\exists_{(—, X)}(\Delta \bullet (—, S_{label})) . \tag{4.13}$$

All states reachable from $S^0 \subseteq S$ may be found with a least fixed-point of expression 4.12. This fixed-point is found by computing

$$S^{n+1} = S^n \cup \exists_{(X, —)}(\Delta \bullet (S^n, —)) \tag{4.14}$$

until $S^{n+1} = S^n$ for some natural number n. When the fixed-point is reached, S^n is the set of all states reachable from S^0. A similar fixed-point may be formulated to determine all states which eventually reach some state set $rs^n \in S$.

Given a cyclic **CMA**'s $S_{task\ start}$, the set of reachable states RS is determined with equation 4.14. The viability pruning of Δ^{data} for this **CMA** is,

$$\Delta^{n+1} = \Delta^n \bullet (RS, RS) . \tag{4.15}$$

Theorem 4.1 The viability prune leaves only iteration-sense consistent states in a **CMA.**

Proof Suppose s is any state reached from $S_{task\ start}$. Dependency implications prevent tasks from accepting operands unless the required input operands are *known*. Capacity implications prevent loss of an operand before all children tasks have accepted. Since s is reached from $S_{task\ start}$, where no operands exist in

the even sense, then paths to *s* produce any and every operand as dictated by dependency and capacity implications. Hence, *s* is not subject to iteration confusion because it identifies no **MA** in a composition that has produced an operand without proper required input operands. As *s* is arbitrary, all states reached from $S_{task\ start}$ are iteration-sense consistent.

4.2.7 Cyclic Concurrency Constraints

Cyclic transition-based concurrency constraints are formulated as in section 4.1.5. There is no dual sense for resource usage, such as *alu busy* and *alu~ busy.* All uses of a particular resource *res* are derived from a composite member **MA** and are implicitly counted and prune a **CMA** as in expressions 4.4 and 4.5. On the other hand, since there are two senses of operands, two sense-distinct memory care conditions must be formulated for each operand with expression 4.6. These are implicitly counted as in expression 4.7 and intersected with the present state portion of Δ^{data} to prune memory concurrency violating states.

4.3 Acyclic Control-Dependent Composition

Acyclic control-dependent composition introduces two new concepts beyond those introduced for acyclic data-flow composition in section 4.2. First, a control operand may select one operand from several choices to satisfy an input operand dependence. This requires a new resolved operand dependency implication. Second, some tasks may be deemed unnecessary under certain control conditions. These tasks are bypassed to their task final state. A control-dependent composition contains a two-phase transition relation, Δ^{data} and $\Delta^{control}$, reflected in the **MA** forming the composition, to facilitate task bypassing and control resolution.

Figure 4.11 presents an acyclic control-dependent scheduling problem to aid discussion. Each task identifies which control block, `cb`, it is valid in. Task `mt`

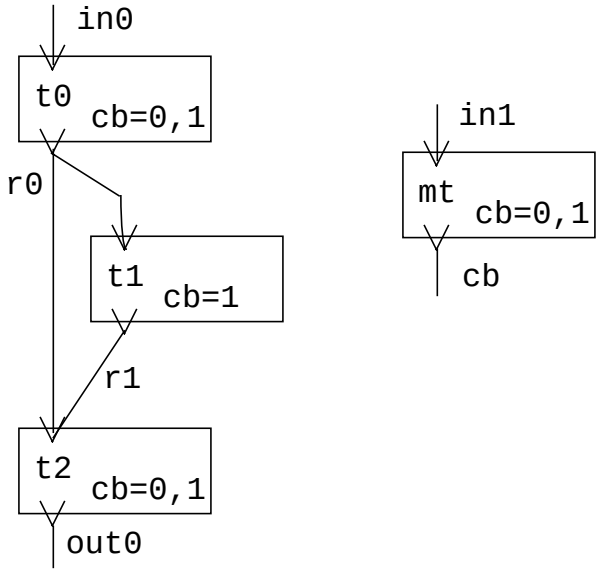


Figure 4.11 A simple acyclic control-dependent example

produces the two-value control operand `cb`. Task `t2` has two potential sources for its single input operand. If `cb=0` at this operand resolution point, then `r0` is used, else `r1` is used.

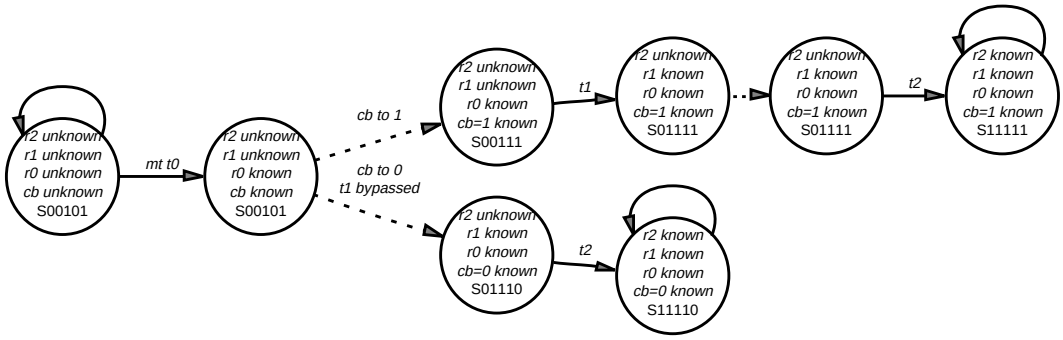


Figure 4.12 Partial **CMA** for figure 4.11

Figure 4.12 shows a partial **CMA** with minimum latency execution sequences for the example in figure 4.11. In the top path, where `cb=1`, task `t2` waits until `r1` is available before executing. In the bottom path, where `cb=0`, task `t1` is bypassed during a control phase (dotted transition) as it is deemed unnecessary. Also, when `cb=0`, `t2` executes with `r0` as an input operand.

4.3.1 Resolved Operand Dependencies

A guarded operand dependence specifies several possible source operands, one of which is selected by another operand to satisfy an input operand dependence. For example, $a^{spec} \cdot d^{val=1} + a^{nospec} \cdot d^{val=0}$ may be interpreted as "use speculative operand a^{spec} if multivalue operand d equals 1 or use nonspeculative operand a^{nospec} if multivalue operand d equals 0." Although the true operand dependence is on either a^{spec} or a^{nospec}, the multivalue operand d is needed to select which operand dependency is correct. This selection process is called **operand resolution** and corresponds to a join in a traditional CDFG representation. A guarded operand dependence forms an **operand resolution point** where potential source operands are called **preresolution operands** and the final accepting task is called the **postresolution task**. The multivalue control operand, which selects a particular postresolution operand, is called the **resolver operand**. Finally, as an operand resolution point example, behavior in a RISC processor selects an operand from several possible sources to write back to the register file. Some instructions require that a memory fetch be written to the register file while other instructions require that an ALU computation result be written to the register file.

It is possible that both preresolution operands a^{spec} and a^{nospec} are *known* but the resolver operand, *d*, is *unknown*. Although all input data dependencies for the postresolution task are satisfied, appropriate *a required*-labeled transitions still may not be enabled. This is because the postresolution task fundamentally expects a single *a* input operand. Without resolver operand *d*, it is unclear which *a* should be used. Furthermore, the postresolution task **MA** is only allocated enough state to model result operands for one task instance and hence may not process both a^{spec} and a^{nospec}. Consequently, the postresolution task **MA** must stall until the resolver operand *d* is *known* and operand resolution occurs.

This artificial dependency on resolver operand *d* bounds the state in a **CMA**. Consider how this dependency might be removed. Instead of a single postresolution accepting task **MA**, two accepting task **MA** may be specified. One accepts and processes a^{spec} while the other accepts and processes a^{nospec}. Each task could then begin execution immediately when its particular *a* input operand is present. No dependency on resolver operand *d* exists. Also, enough state exists to represent production of result operands from processing both a^{spec} and a^{nospec}. This *task splitting* could be carried to the limit such that no resolver operands are ever required. At the limit, a **CMA** may require excessive state for acyclic and infinite state for cyclic control-dependent behavior. Furthermore, controllers synthesized from such a **CMA** are potentially infinite while synthesized datapaths must correctly handle increased numbers of concurrent operands. On the other hand, operand resolution points are convenient and appropriate places for a designer to bound **CMA** state growth as well as any implied FSM controller and datapath complexity. Specification and adjustment of operand resolution points is a mechanism by which a designer architects the design.

A single operand resolution point dependence is modeled by the implication,

$$\begin{aligned} ma.\Delta_{info\ required} \Rightarrow\ & (mp^{1}.S_{info1\ known}, —) \cdot (mr.S_{rinfo^{val=1}\ known}, —) + \\ & (mp^{2}.S_{info2\ known}, —) \cdot (mr.S_{rinfo^{val=2}\ known}, —) + \ldots + \\ & (mp^{n}.S_{infon\ known}, —) \cdot (mr.S_{rinfo^{val=n}\ known}, —) \end{aligned} \tag{4.16}$$

In this implication, multivalue **MA** *mr* produces resolver operand *rinfo* which selects the actual *info* operand from *n* possible choices. For some case *i,* if $rinfo^{val=i}$ is *known* AND *infoi* is *known,* then the *info required*-labeled transition is allowed. Let Ψ_e represent expression 4.16 for one guarded operand dependence, *e.* The refinement of Δ^{data} for a scheduling problem **CMA** is,

$$\Delta^{n+1} = \Delta^{n} \bullet \prod_{e \in E_{guarded}} \Psi_{e}. \tag{4.17}$$

Basic, Alternative and Undetermined dependencies may be formulated for cyclic control-dependent composition as described for acyclic data-flow composition in section 4.1.

4.3.2 Acyclic Task Bypassing

The **CMA** constraints discussed thus far, dependencies, capacities and concurrency, have all been applied to the data-phase transition relation, Δ^{data}. They enforce causal ordering of operands and task execution, limit resource use, as well as maintain iteration-sense consistency in cyclic behavior. For scheduling problems without control dependencies, these constraints[9] and Δ^{data} are sufficient. For control-dependent scheduling problems, constraints applied to a control-phase transition relation, Δ^{control}, are required.

A **CMA**'s control-phase transition relation serves two purposes. First, it provides a resolve step for multivalue **MA**. As introduced in section 3.4, a convenient deterministic *resolve*-labeled state is added to simplify validation during a **CMA**'s exploration. Validation is an exploration step and is described in detail in section 5.2. Second, Δ^{control} allows tasks to be completely bypassed if deemed unnecessary by control resolution. This section focuses primarily on when and how such *control-obviated* task are bypassed.

To establish the need for task bypassing, consider a memory write task in a RISC processor. For some instructions, this task is required while for others it is not. In the event that it is not required, it may be bypassed to its task final state. This simplifies termination detection for acyclic models and primes all tasks for the next iteration in cyclic models.

9. Operand resolution dependencies are not required for data-flow only behavior as well.

As defined, the scheduling problem associates a control block, *cb*, with each task. This control block is a Boolean expression indicating when the paired task *t* is necessary. Suppose *task1* and its **MA** are in the control block,

$$cb = rinfo^{val = 0} + (rinfo^{val = 1} \cdot binfo^{val = 1}).$$

Task1 is necessary when *rinfo* equals 0 or *rinfo* and *binfo* both equal 1. Assuming multivalue operands *rinfo* and *binfo* are limited to the range 0 to 1, *task1* is unnecessary when *rinfo* equals 1 and *binfo* equals 0. When known to be unnecessary, *task1* may be bypassed to simplify determining a task final state in an acyclic **CMA**. With task bypassing, $S_{task\ final}$ for a **CMA** simplifies to the set of states where *all* composition **MA** are in their final task state(s) and all control cases are included. This can be contrasted to a possible $S_{task\ final}$ without bypassing where some tasks remain unexecuted. This varies from control case to control case and is difficult to determine exactly without *a priori* knowledge of which tasks have speculatively executed. Finally, task bypassing is necessary to maintain iteration-sense consistency in a cyclic control-dependent **CMA**.

A single scheduling problem task's **MA** is **control-obviated** when all operands in the task's control block expression have resolved yet the control block expression is *false.* Consider the control block expression,

$$cb = rinfo^{val = 0} + (rinfo^{val = 1} \cdot binfo^{val = 1}).$$

In terms of composition **MA** states, the control-obviated expression for this control block is,

$$\frac{(S_{rinfo\ known} \cdot S_{binfo\ known}) \bullet}{(S_{rinfo\ known^{val = 1}} + (S_{rinfo\ known^{val = 1}} \cdot S_{binfo\ known^{val = 1}}))} \quad . \qquad (4.18)$$

The top term in this expression insists that both multivalue operands *rinfo* and *binfo* are known regardless of value. The bottom term insists that the values of

rinfo and *binfo* are something other than those required for this control block. Hence, all control block expression operands have resolved and the control block expression is false.

The precise form of a control-obviated expression, *coe*, depends on the task's Boolean *cb* expression. In general, the following two rules are used to create a correct control-obviated expression. First, each operand *info* that appears in a control block contributes a $S_{info\ known}$ to a control-resolved product term. The top portion of expression 4.18 is an example of this. Second, the entire *cb* term is expressed as **MA** states and complemented. The bottom portion of expression 4.18 highlights this. The product of these two terms is a correct control-obviated expression for a task.

An **MA**'s *task bypass*-labeled transition is forced if the **MA**'s control-obviated expression, *coe*, is *true* and the **MA** is not yet bypassed.

$$\Delta_{task\ bypass} \Leftrightarrow (—, coe \cdot \overline{S_{task\ final}}) \tag{4.19}$$

The right side of expression 4.19 is written in the next state so that the **MA** appears bypassed in lock step with control block resolution. Before the control block is resolved, the **MA** is considered necessary and may be executed speculatively. If the **MA** has executed speculatively, it may often be in its task final state and hence need not be bypassed. Let Ω_m represent expression 4.19 for an **MA** *m* in the composition. The refinement of Δ^{control} for a **CMA** is,

$$\Delta^{n+1} = \Delta^n \bullet \prod_{m \in M} \Omega_m . \tag{4.20}$$

4.3.3 Concurrency Constraints

Both transition-based and state-based (memory) concurrency constraints are built for acyclic control-dependent compositions as described in section 4.1.5. Transition-based concurrency constraints are applied to Δ^{data} and hence, task

bypassing, which occurs in $\Delta^{control}$, does not interfere. Furthermore, before control resolves in a **CMA** path or execution sequence, only a single path exists. Consequently, all task resource uses --both speculative and nonspeculative-- are implicitly counted from the same pool of resources. After control resolves in a **CMA**, a path branches into various mutually exclusive control cases and subsequent tasks are allocated resources in a correct mutually exclusive fashion.

State-based concurrency constraints for acyclic control-dependent compositions highlight two issues. First, a memory care condition should not include control-obviated dependents. Fortunately, once a task is control-obviated, it is immediately bypassed and subsequently appears as if all input operands had been accepted. It would not be counted when using the existing memory care condition (expression 4.6). Second, memory care conditions through an operand resolution point must be counted correctly. If a resolver operand has yet to resolve, then all *known* preresolution operands must be counted as memory care. Once a resolver operand has resolved, then only the selected preresolution operand must be counted as memory care. To formulate this, an expression very similar to the control-obviated expression (4.18) is built. The difference is that the preresolution operand's selection condition rather than a task's control block is used as a starting point. For example, suppose a preresolution operand is only selected when the resolver is *known* and equal to 3. The term $info3 \cdot rinfo^{val\,=\,3}$ will be one term in the sum of a resolved operand dependence tuple's Boolean expression *f*. Using this term in place of a control block expression, a resolved memory care condition, *rmcc*, is constructed in the same manner as expression 4.18. Then, assuming ma^1 depends on *mp* through an operand resolution point, a updated memory care condition is,

$$mcc_{info} = mp.S_{info\ known} \bullet \frac{}{(\overline{rmcc_{info}} \cdot \overline{mc^1.S_{info\ accepted} + mc^2.S_{info\ accepted} + \ldots + mc^n.s_{info\ accepted}})} \quad . \qquad (4.21)$$

Note that if a child depends on an operand through a resolution point, it is conditioned by a resolved memory care condition.

4.4 Cyclic Control-Dependent Composition

Cyclic control-dependent compositions are the most complicated to formulate as they require concepts from all previously discussed composition types. The Cartesian product step, dependency constraints, iteration-sense issues, task bypassing requirements and operand resolution all apply to cyclic control-dependent composition. What differs is how iteration-sense confusion concerns are handled. In data-flow composition, it is relatively straight-forward to formulate a capacity constraint that maintains iteration-sense consistency locally at the operand dependence level. For control-dependent composition, capacity constraints are not straight-forward. First, task bypassing toggles the sense of tasks within a control block and consequently creates additional sources of iteration-sense confusion. Second, operand resolution points allow postresolution tasks to accept before all preresolution operands are known. This also adds to potential iteration-sense confusion. Capacity constraints that address these new sources of iteration-sense confusion are difficult to formulate. A new method to maintain iteration-sense consistency is developed and presented here.

This section begins by with a general discussion on how capacity constraints impact ABSS. Next, the impact of capacity constraints is relaxed through duplication of a scheduling problem composition. With this background, a global capacity constraint is developed for cyclic control-dependent models. An example

illustrates how control-dependent pipelining is modeled with this technique. Finally, nondeterministic control is presented.

4.4.1 The Impact of Capacity Constraints

An **MA** in a composition typically models one instance of its produced operands. For cyclic behavior with capacity constraints, this implies one 'live' iteration instance of an operand at a time. The operand must remain in its current sense until all dependent tasks have accepted. This bounds, at a cost, the potentially infinite state behavior of highly parallel cyclic behavior. To illustrate how capacity constraints impact cyclic problems, two examples are considered.

Figure 4.13 shows how a simple chain of three cyclic tasks are pipelined in

Figure 4.13 A naturally pipelined composition

ABSS. In time-step 1 the composition is in a composite task starting state as all internal tasks are *known* in the odd iteration sense. By time-step 2 task *t1* executes (produces its operand) in the even sense. In time-step 3, task *t2* executes in the even sense while task *t1* may execute in the odd sense as no capacity constraints are violated. In time-step 4, tasks *t1* and *t3* execute in the even sense while task *t2* executes in the odd sense. Now the pipeline is full and all tasks execute in one iteration sense at each time-step by toggling between states shown in time-steps 3 and 4. Capacity constraints do not impact solution quality for this composition.

Suppose that figure 4.13's example has an additional operand dependence from task *t1* to task *t3* as shown in figure 4.14. Now capacity constraints require task *t1*

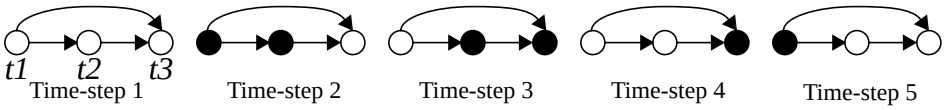


Figure 4.14 A capacity constrained composition

to stall and hold its current result so that task *t3* may accept the correct operand. Full pipeline behavior sequences from time-step 5 back to time-step 2 and achieves minimum iteration latency of two time-steps as two iterations require four time-steps to complete. Thus, pipelined behavior is impacted by the constraint of representing at most only one result operand for task *t1*.

The impact of capacity constraints on control-dependent cyclic composition can be more severe than what is seen in this three task data-flow example. For example, a postresolution task has a dependency on a resolver to determine which preresolution operand to accept. This dependency requires a capacity constraint from postresolution task to the resolver to prevent iteration-sense confusion. This capacity will hold a resolver in its current iteration sense until all postresolution tasks have accepted. This is potentially costly as a resolver typically resolves numerous operand resolution points. Pipelining control operands is difficult if state for only one 'live' control operand is allocated. This, coupled with the complexity of formulating correct capacity constraints in the presence of task bypassing and through operand resolution points, makes the locally applied capacity constraint route a rocky choice for control-dependent cyclic composition.

4.4.2 Relaxing Capacity Constraint Impact

Sometimes it is desirable to relax the impact of capacity constraints in a controlled manner. This may be done by composing several copies of a composition, called **iterates**, as one larger **CMA**. Figure 4.15 shows how two

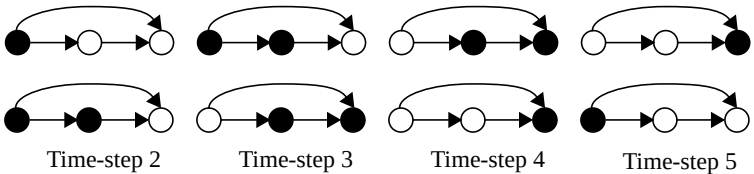


Figure 4.15 Two iterates as one composition

iterates of figure 4.14's example may be co-executed yet still under a single

resource constraint set. Even though a single iterate has iteration latency of two time-steps, final iteration latency for the two iterate composition is one time-step as both iterates execute. This improved performance is obtained because there are now two **MA** for task *t1* and hence two *t1* result operands may exist concurrently. Expanding the searched solution space by duplicating the entire scheduling problem is analogous to “unrolling a loop” in traditional scheduling and compiler theory. Although there are no inter-iteration dependencies in this example, it is often required that dependencies from one iterate of the scheduling problem to the next are added. In particular, including artificial dependencies from one iterate to the next orders the iterates and hence avoids representation of symmetric solutions.

Duplicating an entire composition exceeds what is minimally necessary to avoid impact of capacity constraints. In the above example, a single additional **MA** that represents another instance of *t1*’s result operand would be sufficient to avoid capacity constraint impact. This additional **MA**, inserted to break the dependency and capacity from *t1* to *t3*, would ‘buffer’ *t1*’s result operand. Consequently, *t1* is free to produce the next iteration’s result while this buffer holds the current value required by *t3*. The difficulty with this approach is determining *a priori* where capacity constraint impact occurs. Furthermore, with control-dependent scheduling and buffered resolver operands, task bypassing becomes illdefined. Which resolver should control-obviate a task and when? For these reasons, duplication of an entire control-dependent cyclic composition is preferred.

4.4.3 Global Capacity Constraints in Iterates

Although duplicating a composition relieves capacity constraints, it does not eliminate them. For control-dependent scheduling, capacity constraints must be correct even when task bypassing and operand resolution points are present. These

are difficult to formulate at the local operand dependence level and hence a global capacity constraint is introduced.

Suppose that an iterate did not mix the iteration sense of internal **MA**. Beginning with iterate start states, $S_{task\ start}$, execution proceeds until iterate final states, $S_{task\ final}$, are reached and vice versa for an odd execution. At no time during an even execution is a task allowed to execute or produce result operands in the odd sense and vice versa. This behavior is precisely what exists in a cyclic **MA** where execution in one sense is always mutually exclusive from execution in the other sense. An iterate with such a constraint could be thought of as a single, although complex and control-dependent, **MA**. This has the advantage that no internal operand capacity constraints need be imposed. Since an iterate final state for either sense must be reached *before* the next iteration may begin, it is guaranteed that all internal tasks of an iterate have completed, either through execution or bypassing, in the current iteration sense. This is, by definition, a task final state, $S_{final\ state}$. This, coupled with knowledge that an iterate is globally executing either in one sense or the other, eliminates the need for capacity constraints among internal operands. Another way to view an iterate with a global capacity constraint is as a cyclic **MA** created from dual acyclic control-dependent **MA** similar to section 3.2. The acyclic control-dependent **MA** are constructed as discussed in section 4.3 and as such do not require capacity constraints.

An iterate may be forced to not overlap iterations by introducing an artificial **sense operand MA**, *mso*. This is a single time-step cyclic **MA** as shown in figure 3.12 that identifies the current sense of the entire iterate. It may be thought of as a sequential constraint, $mso \in Q$, of the scheduling problem. Constraints are applied to an iterate **CMA** which restrict all internal tasks to even iteration executions if the sense operand is *known* in the even sense and vice versa for odd iteration executions. In other words, an internal task of an iterate may only leave its

even iteration-sense task start state if the sense operand is in the even iteration sense and vice versa for the odd sense. This may be enforced with the implication,

$$(m.S_{task\ start}, \overline{m.S_{task\ start}}) \Rightarrow (mso.S_{sense\ known}, —)\,. \tag{4.22}$$

This is built for all $m \in M$ in an iterate and for both iteration senses. Furthermore, a capacity constraint prevents *mso* from forgetting the current sense until all internal tasks of an iterate are *known* or will be *known* in the current iteration sense. This may be enforced with the implication,

$$mso.\Delta_{sense\ forget} \Rightarrow \langle (m.S_{task\ final}, —) + (—, m.S_{task\ final}) \rangle\,. \tag{4.23}$$

This too is built for all $m \in M$ in an iterate and for both iteration senses. The product of all such implications prunes an iterate's $\Delta^{\text{data}} \in$ **CMA** as in expression 4.2.

4.4.4 The Impact of a Global Capacity Constraint

Although a global capacity constraint simplifies the construction of an iterate by eliminating local capacity constraints, an additional impact on the solution space, due to the granularity of the global capacity constraint, occurs. Figure 4.16

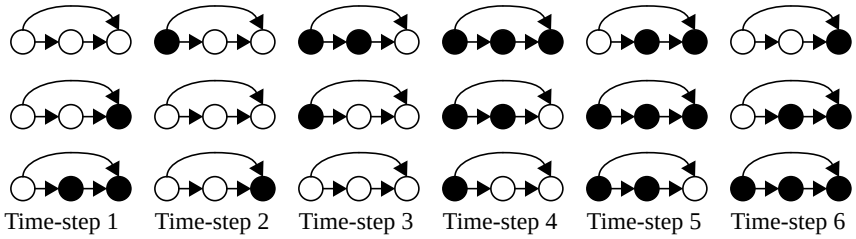


Figure 4.16 Three global capacity constrained co-executing iterates

shows the same three task pipelined example but now all iterates have a global capacity constraint applied. Now, a single iterate has an iteration latency of three time-steps and three iterates must be co-executed to achieve single time-step iteration latency. In fact, the maximum number of 'in processing' instances of a scheduling problem is exactly equal to the number of co-executing iterates.

Although iteration overlap is now prohibited within an iterate, iteration overlap is still possible among iterates as shown in figure 4.17.

Figure 4.17 Iteration overlap among iterates with global capacity constraints

Figure 4.17 further motivates the need for iterates. Imagine a **CMA** that models execution of a single RISC instruction. Because of the global capacity constraint required for cyclic control-dependent models, only one instruction may be ‘in flight’ at a time. By creating several iterates of this **CMA**, several instructions may be simultaneously ‘in flight’. This models what happens in a pipelined or even superscalar RISC processor implementation.

4.4.5 Composing Iterates

Each iterate is an entire instance of a scheduling problem that is made to appear as a single cyclic **MA**. All control considerations are completely encapsulated within an iterate. Hence, when composing several iterates into a new, larger **CMA**, all formulation steps for cyclic data-flow composition, section 4.2, apply.

Within an iterate there is control. Since an iterate with control will have a global capacity constraint applied, it may be constructed as described for acyclic control-dependent composition, section 4.3. The only difference is that an iterate and all **MA** in an iterate are cyclic and hence all section 4.3 constraints must be built for both iteration senses.

When several independent iterates are used within a composition, all sequence permutations of one iterate executing before another are represented. This freedom

may lead to inefficient representation. Furthermore, from a final system implementation perspective, one ordering permutation may be indistinguishably symmetric to all other ordering permutations. Artificial dependencies, from one iterate's *mso* to a next iterate's *mso*, may be used to arbitrarily choose one ordering permutation.

4.4.6 An Example Iterate

Figure 4.18 shows an iterate for the acyclic control-dependent composition example of figure 4.11. This iterate is a cyclic **MA**. Starting at the iterate's task start state, $S_{task\ start}$={S000000, S000001}, only even sense execution occurs until $S_{task\ final}$={S111110, S111111} is reached. The same is true for odd sense execution from $S_{task\sim\ start}$ to $S_{task\sim\ final}$. Notice that iterate task start and final state sets contain states representative of every possible control case. Furthermore, it is still possible to execute a *cb*=0 control case in two time-steps while a *cb*=1 case requires three time-steps. Several of these iterates may be composed into a new **CMA** as described in section 4.2 to model high-level task pipelined behavior.

4.4.7 Nondeterministic Control

An **MA** in a composition may produce operands yet require no input operands as described in section 4.1.4. This is particular valuable for multivalue **MA** serving as resolvers. To model an alternative, an operand resolution point with a nondeterministic resolver is specified. Then, during exploration, whatever resolver choice appears best is kept. This allows for alternatives in the composition yet does not require that every alternative eventually produce as some may be bypassed.

As a specific example of nondeterministic control, consider a memory access task in a RISC processor. This memory access task exhibits control-dependent behavior. If a cache hit occurs, the memory access task produces the requested operand relatively quickly. If a cache miss occurs, a sequential penalty is paid. In

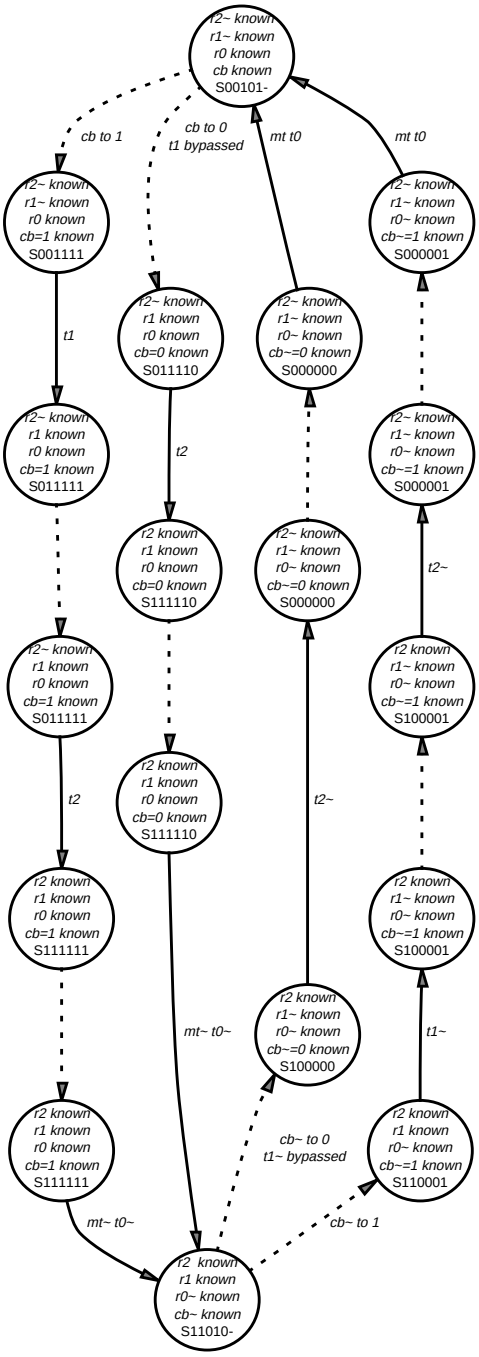


Figure 4.18 Iterate for cyclic control-dependent example

this example, a multivalue **MA** produces a control operand which decides between cache hit or miss. This multivalue **MA** may be modeled with no required input operands (undetermined operand dependencies). Then, a cache hit/miss

probability determines how schedules should be prioritized during exploration. For example, if schedules favoring cache hits are desired, then exploration will give higher priority to schedules where the cache hit/miss multivalue **MA** resolves to hit. This is not an internal timing constraint but rather a probabilistic selection of various control-dependent behaviors.

4.5 Composition Generalizations

This section highlights and generalizes some of the composition techniques used in sections 4.1 through 4.4. In particular, Boolean constraints among different **MA**, which dependency and capacity constraints are instances of, are discussed. Concurrency constraints, which are also Boolean constraints albeit combinatorially large when written explicitly, are discussed. In a more general setting, these Boolean constraints enable additional sequential constraints and hierarchy of concurrency constraints. Finally, task splitting and operand buffers are presented as ways to increase freedom in a **CMA**.

4.5.1 Explicit Boolean Constraints Among MA

Any Boolean constraint that may be expressed explicitly in terms of composite-member **MA** states and/or transitions may be applied to a **CMA**. This is a substantial amount of expressive freedom which may not always lead to correct system execution models. Iteration-sense consistency, causal operand ordering and valid control behavior are a few important system execution considerations that must be maintained by Boolean constraints. Several types of explicit Boolean constraints have proven useful and valid in ABSS and are generalized here.

Both dependency and capacity constraints fit into an implication constraint form. In a ROBDD, implications may be built as $p \Rightarrow q = \overline{p\bar{q}}$. The two most useful formulations are,

$$mi.\Delta \Rightarrow (mj.S, —) \tag{4.24}$$

and,

$$mi.\Delta \Rightarrow ((mj.S, —) + (—, mj.S)) . \tag{4.25}$$

Implication 4.24 may be interpreted as activity *mi.*Δ is enabled yet not forced if *mj.S* is *true* in the present state for two **MA** *mi* and *mj* in a composition. This implication is typically written with consequent state sets labeled with **historical labels** such as *info known* or *info accepted.* Historical labels identify that certain knowledge exists or has existed earlier during the execution of a task.[10] This allows an implication to enforce an ordering of events. Implication 4.24 may be interpreted as, "Activity *mi.*Δ occurs some time after the activity recorded by *mj.S.*" Implication 4.25 differs only in that the consequent is written in the present state *or* the next state. This may be interpreted as, "Activity *mi.*Δ occurs at the same time or some time after the activity recorded by *mj.S.*"

The consequent of these implications is extended to handle alternatives or resolved operand dependencies. Alternatives are simply multiple terms related by Boolean OR in the consequent. For example,

$$mi.\Delta \Rightarrow ((mj.S, —) + (mk.S, —)) \tag{4.26}$$

may be interpreted as, "Activity *mi.*Δ occurs some time after the activity recorded by *mj.S* OR the activity recorded by *mk.S.*" Although it is possible to form a similar implication with multiple terms related by Boolean AND in the consequent, it is not necessary as the intersection of multiple single-term consequent implications is equivalent. With resolved operand dependencies, alternative choices are conditioned or guarded by an additional term. For example,

$$mi.\Delta \Rightarrow \langle (ml.S \cdot mj.S, —) + (mn.S \cdot mk.S, —) \rangle \tag{4.27}$$

10. Historical state labels are contrasted with current state labels such as *info stored* which identify currently available information and do not remember that this information once was available after it is lost.

may be interpreted as, “Activity *mi*.Δ occurs some time after the activities recorded by *mj*.*S* AND *ml*.*S* OR the activities recorded by *mk*.*S* AND *mn*.*S*.” In typical application of implication 4.27, guard terms are chosen which are known by construction to be mutually exclusive. Hence, at most only one alternative is ever valid. Both implications 4.26 and 4.27 may include consequent states in the next state as in implication 4.25.

A double implication is used to synchronize activity or information in an ABSS composition. The double implication,

$$mi.\Delta \Leftrightarrow mj.\Delta \tag{4.28}$$

may be interpreted as “Activity *mi*.Δ must occur during the same time-step as activity *mj*.Δ.” Although a transition set is specified, Δ, transitions in terms of states, (*s*,*s*^) or even just states may be used. A double implication is used to require a task bypass transition if a bypass is deemed necessary and no bypass has occurred yet. A double implication is also used when sequential constraint or protocol **MA** are included in a composition. These **MA**, as described in section 3.3, do not necessarily represent sequential behavior of operand production but rather describe how several **MA** in a composition must sequentially interact. Consequently, activities labeled *info required* and *info produced* of the various operand **MA** are synchronized to appropriate transitions and states of the sequential constraint or protocol **MA** with double implications.

These implications are the most applicable constructs for ABSS composition. In fact, all presented ABSS compositions only require these implications and the implicit constraint described next for all constraint pruning.

4.5.2 Implicit Boolean Constraints Among MA

Some Boolean constraints are prohibitively large when expressed explicitly. Consider all ways of choosing only *r true* Boolean variables from a set of *n*

Boolean variables. In sum of products form, this Boolean expression requires $\binom{n}{r}$ terms. Rather than expressing all such choices explicitly, an implicit ROBDD construction technique builds all such choices efficiently. Statements such as, “at most *r* registers are available for these *n* result operands” or “at most *r* ALUs are available to implement these *n* tasks” require formulation of *r*-combination expressions. Formulation of ROBDD *r-combination* expressions may be traced to several sources [16][64][80][114].

Figure 4.19 shows an “at most 2 of 6” ROBDD. Only 12 ROBDD nodes are required to efficiently encapsulate all 127 combinations. Although figure 4.19 chooses at most 2 of 6 single Boolean variables, *r-combinations* of Boolean *functions* are possible. Memory concurrency constraints provide a sophisticated example or *r-combinations* involving Boolean functions. A memory care condition is a fairly complicated expression as it contains Boolean variables from a producing **MA** and all accepting **MA**. Still, regardless of ROBDD ordering and overlap, an *r-combination* constraint may be built implicitly from a set of memory care conditions. This requires time and nodes proportional to $n \times r$. A modern implementation of ROBDD *r-combination* constraints may be found in PYCUDD[53].

All applications of *r-combination* constraints so far have considered just one set of size *n* and at most *r* combinations from this set. It is possible to use several *r-combination* constraints which choose combinations from several not necessarily disjoint sets to generate stronger concurrency constraints. Suppose a scheduling problem has a single register file available. This register file has two ports and supports at most 2 reads or 1 read/1 write during a single time-step. Numerous tasks in the scheduling problem may access this register file. Each task’s **MA** may have $\delta_{rf\ read\ busy}$ and/or $\delta_{rf\ write\ busy}$ transition sets. Two *r-combination* constraints are required. One considers all $\delta_{rf\ read\ busy}$ and $\delta_{rf\ write\ busy}$ transitions and creates

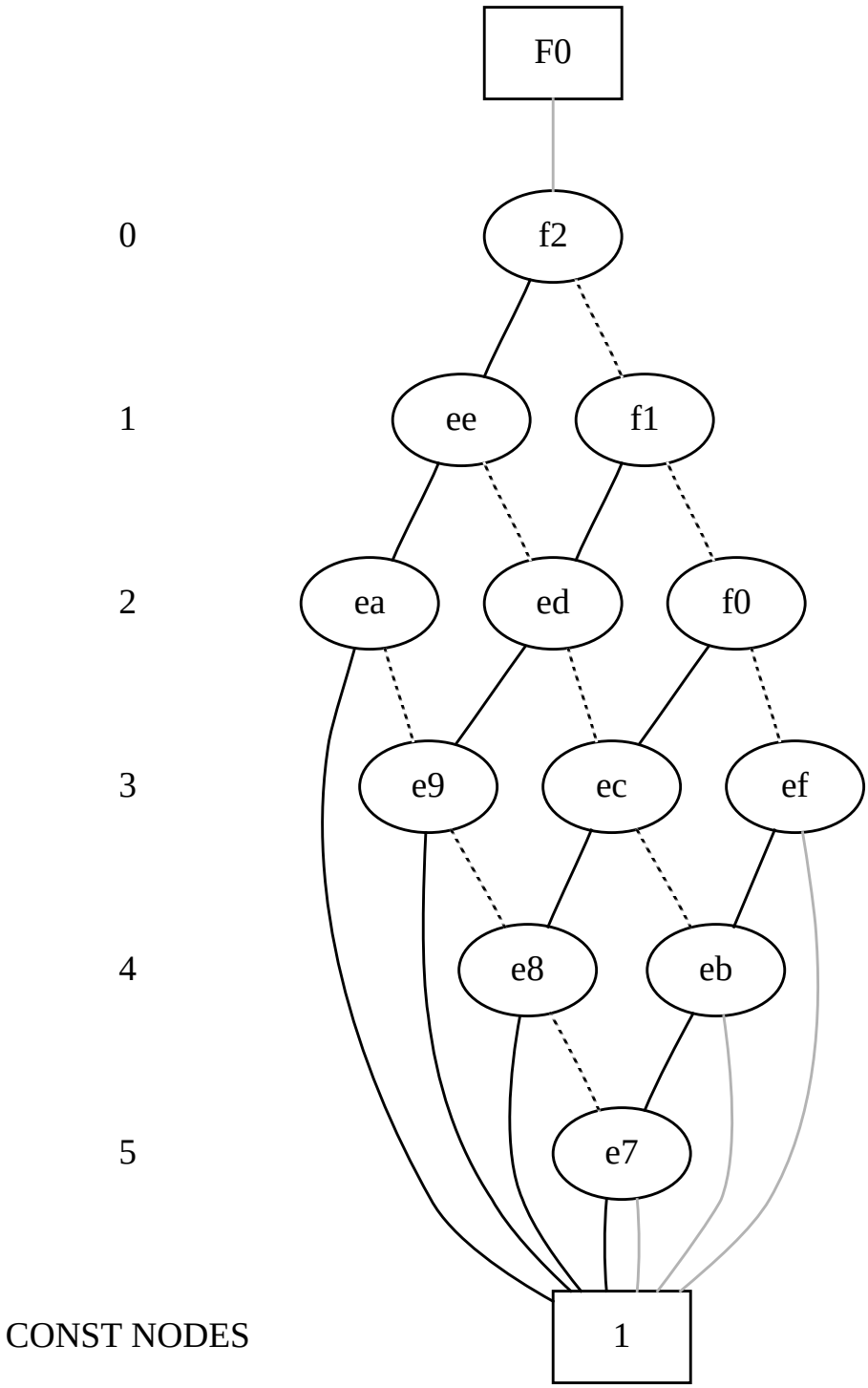


Figure 4.19 “At most 2 of 6” ROBDD

all combinations of at most 2. The second considers all $\delta_{rf\ write\ busy}$ transitions and creates all combinations of at most 1. Together, these two constraints enforce at most 2 reads or 1 read/1write during any time-step.

Some *res busy*-labeled transitions may require more than a single resource *res*. As discussed in section 3.1.4, *i* unit-weight *res busy* labels may be attached to a single transition to represent *i* resource uses. The set with cardinality *n* from which all combinations are chosen must contain *i* instances of a function with weight *i*. When a ROBDD *r-combination* constraint is constructed from such a set, choosing one instance automatically chooses all other instances and hence the appropriate weight results. Suppose tasks within a scheduling problem produce both single word and double word operands. More precise local storage constraints may be formulated by assigning weight 2 to double word memory care conditions and weight 1 to single word memory care conditions. By using concurrency weights as well as hierarchical concurrency constraints, sophisticated bounded resource use may be modeled.

Two final points remain for concurrency-type constraints. First, conditions may be applied to a concurrency constraints. Just as an operand value selects among alternatives in an operand resolution, an operand value (or some other condition) may select among several static or dynamic constraint scenarios. Second, although a transition-based concurrency constraint applies for an entire time-step, a time-step may still represent several concurrency-limited activities. For example, an add task may require two busses to transfer two input operands from local storage, an arithmetic function unit, as well as an additional bus to transfer a result to local storage. A single time-step implementing this task may be broken down into three time periods: input transfer, computation and output transfer. This may still be modeled by occupying all required resources for the duration of the time-step while allowing the actual order of use to remain implicit.

4.5.3 Task Splitting

The notion of ‘task splitting’ was introduced in section 4.3.1. This permits postresolution accepting tasks to begin execution before a resolver resolves by duplicating the postresolution task and executing a copy for all preresolution values. Figure 4.20 shows how a postresolution task is split into three to increase

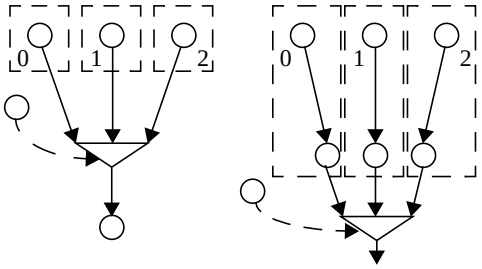


Figure 4.20 Task splitting

speculation. Although more state is added, it may be done in a controlled manner to enlarge the solution space and determine potentially better execution sequences.

4.5.4 Operand Buffers

As pointed out in section 4.4.1, the number of concurrent instances of a particular operand is bound by the number of **MA** modeling the production of this operand. This bound may impact cyclic composition execution sequences and also prohibits dependency implications between executions iterations that are not successive. Section 4.4.2 described how this bound could be relaxed with iterates of an entire composition. In some scheduling problems, several instances of *all* operands in a composition are not necessary and costly to represent. It is possible to relax this bound locally through use of operand buffers.

Figure 4.21 illustrates how a particular operand *info* may be buffered. Each operand buffer is a single time-step cyclic **MA** as introduced in section 3.2. If dependency implications are written as in implication 4.24, with no ‘look ahead’

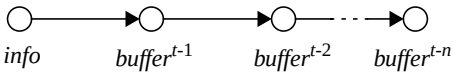


Figure 4.21 Operand buffers

as is commonly done, then this chain of buffers is similar to a synchronous shift register where the last *n* result operands are preserved. This type of operand buffering is useful when operand dependencies exist across several execution iterations. With this style of operand buffering, operand dependencies for *info* use the desired iteration result, $buffer^{t-i}$, as producer. The original producer is considered iteration 0.

If dependency implications for figure 4.21 are written as in implication 4.25 with 'look ahead', then all buffers in this chain may simultaneously transit. Hence, $buffer^{t-n}$ may represent any result from the current iteration, *t*, to the past iteration *t-n*. This type of operand buffering is useful to relax capacity constraints. If required by capacity constraints, a past result may be stored by a buffer while a new result is produced. But if not impacted by capacity constraints, all buffers may immediately represent the current result with no delay. With this style of operand buffering, all operand dependencies use $buffer^{t-n}$ as the sole producer and typically only one buffer is useful. In general, operand buffers provide a localized way to increase model freedom yet maintain an information-centric view.

4.6 Efficient CMA Representation

Composition may result in large ROBDD structures. In particular, a **CMA**'s transition relations may be exponentially large, even when expressed as ROBDDs. As with all ROBDD-based techniques, a good ordering can significantly reduce the size of ROBDD structures. Also, targeted partitioning of a **CMA**'s transition relations reduces ROBDD representation size. Finally, the operand resolution

formulation may be simplified for some cases to reduce ROBDD size. This section discusses these three strategies for efficient **CMA** representation.

A **CMA**'s final encoding is relatively sparse. It more closely resembles a 'one-hot' encoding of operand existence rather than a logarithmic one. Although additional research may determine a best encoding, the current sparse encoding has one key advantage. Because all composite-member **MA** are encoded over their own unique set of ROBDD variables, a Cartesian product is easily and efficiently represented. In fact, a Cartesian product requires only total nodes equal to the summation (not product) of all nodes for all **MA** in a composition. This may be contrasted to a logarithmic encoding that may identify guaranteed mutually exclusive operands in the scheduling problem and reuse the same Boolean variables to represent both. This would result in less total Boolean variables but not necessarily less ROBDD nodes. With the 'one-hot' approach, constraints are written between a small set of typically local Boolean variables. This does not greatly disturb the original efficient Cartesian product. With a logarithmic approach, constraints would have to be written between a larger, more dispersed set of Boolean variables. From experience, constraints which involve a large number of dispersed Boolean variables typically cause the most severe representation growth. Hence, ABSS strives for sparse encodings that lead to efficient ROBDD representation rather than a minimal number of state variables.

4.6.1 ROBDD Ordering

A good ABSS ROBDD ordering places two task **MA** related by an operand dependence as close together as possible. In other words, a good ABSS ROBDD ordering follows the flow of the original behavioral description. Consider a scheduling problem consisting of n single time-step tasks where task i+1 depends on the result of task i. Figure 4.22 illustrates how this 'chained' scheduling

problem looks for 10 tasks. When ordered as shown in figure 4.22, only 72 nodes

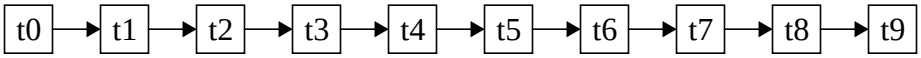


Figure 4.22 A ‘chained’ scheduling problem

are required for $\Delta \in$ **CMA**. If ordered {0,2,4,6,8,9,7,5,3,1}, 3638 nodes are required to represent the same problem! In this bad ordering, the dependency from task 0 to task 1 extends across all other tasks. An ordering which insists that all constraints cross over as small a number of **CMA** ROBDD variables as possible is typically good. With real scheduling problems, it is not always possible are productive to find this best ordering. Rather, a heuristic task ordering, which tends to minimize constraint length, is used. Traditional ROBDD sift reordering then finds a better ordering from this good starting point. Finally, when employing traditional ROBDD ordering techniques such as sift, all **MA** in a composition are first sifted as whole blocks and then sifting occurs within individual **MA**. This grouping of **MA** variables during sifting speeds up reordering and results in smaller ROBDDs.

4.6.2 ‘Long’ CMA Constraints

Suppose that a ‘long’ inter-iteration dependency is added to the example as shown in figure 4.23 to create a loop. Even though all other dependencies are of

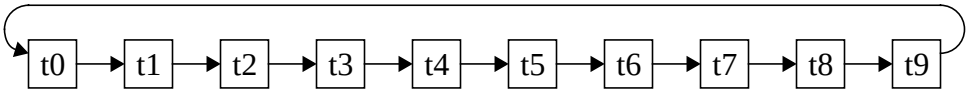


Figure 4.23 A “chained” scheduling problem with “long” constraint

optimal length, this one long constraint causes the **CMA** transition relation to grow from 72 to 237 nodes. Instead of attempting to find a better average order, a **CMA**’s transition relation may be partitioned. Consider a transition relation partitioned into two partitions, 0 and 1. Both partitions are originally the Cartesian product of all composition **MA**. Most constraints are applied to partition 0 while a

few long constraints are applied to partition 1. Care must be taken when computing image and preimage of such a partitioned transition relation because both state and transition information are important to ABSS. To compute an image, a present state set is intersected first with partition 0. Present state variables of all **MA** that do not involve any of the long constraints in partition 1 are then existentially quantified out. Next-state variables are not mapped or shifted to present-state variables yet. Transitions of all **MA** involved with long constraints, whether accepting or producing, are preserved by delaying existential quantification of their present state-variables. Finally, this intermediate set is intersected with partition 1 and all present-state variables are existential quantified out and next-state variables are shifted to present-state variables. This may be generalized to several additional partitions for long constraints. There are trade-offs and it is not necessarily best to have numerous partitions. With more partitions, more **MA** have constraints appearing in later partitions. These **MA** may not have present-state variables existentially quantified out during early partition computations. This can lead to larger intermediate set sizes.

The scheduling problem in figure 4.23 was built in two partitions, 0 and 1. These two partitions require a total of 72+7=79 nodes. Tasks 1 and 9 involve long constraints and hence have present state variables existentially quantified out during image computation only with partition 1. A partitioned transition relation for long constraints is typically only used during the viability prune described in section 4.2.6. The viability prune removes states and transitions with iteration-sense confusion. These erroneous states and transitions are the primary contributors to non-partitioned transition relation growth. Once they are removed, all transition relation partitions are merged into one transition relation through intersection. For the example in figure 4.23, the post-viability ‘flattened’ transition

relation requires only 73 nodes. Finally, concurrency constraints are typically applied only after a transition relation is flattened.

4.6.3 Simplified Operand Resolution Formulation

Operand resolution may be simplified in compositions without capacity constraints. This includes cyclic control-dependent iterates and regular acyclic control-dependent composition. In these cases, tasks are bypassed immediately when their control-obviated expression is *true*. Figure 4.24 illustrates operand resolution for these cases. In frame 1, the resolver has yet to resolve while the task in control block 2 has speculatively produced. In frame 2, the resolver resolves and

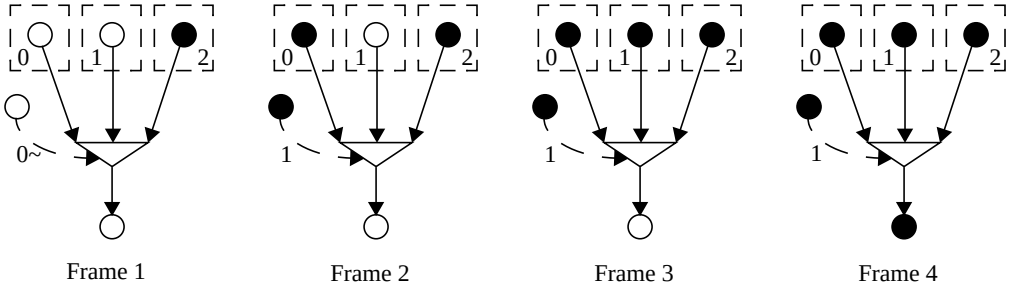


Figure 4.24 Operand resolution sequence with immediate task bypass

the task in control block 0 is immediately bypassed. This is possible as there is no chance of iteration-sense confusion for the composition cases considered. In frame 3, the task in control block 1 executes. In frame 4, the postresolution task may accept. The postresolution task's dependency on preresolution operands need not be conditioned to refer to just a single preresolution operand but may be written in terms of *all* preresolution operands. Provided that all preresolution operands are in control blocks which are only active when they supply the postresolution task, then *all* preresolution operands will be known in the correct sense through bypassing or speculative execution by the time the resolver resolves. Hence, an operand resolution point may be written as,

$$(ma.\Delta_{info\ required} \Rightarrow (mp^1.S_{info1\ known}, —) \cdot (mr.S_{rinfo\ known}, —)) \bullet \quad (4.29)$$
$$(ma.\Delta_{info\ required} \Rightarrow (mp^2.S_{info2\ known}, —) \cdot (mr.S_{rinfo\ known}, —)) \bullet \ldots \bullet$$
$$(ma.\Delta_{info\ required} \Rightarrow (mp^n.S_{infon\ known}, —) \cdot (mr.S_{rinfo\ known}, —))$$

For postresolution $\Delta_{info\ required}$ to be enabled, the resolver must be *known*, $s_{rinfo\ known}$, as well as *all* preresolution operands, $s_{info1\ known}$ through $s_{infon\ known}$. From experience, this **flat operand resolution** formulation is often more efficient to represent.

4.7 Summary

This chapter described how a composite modeling automaton, **CMA**, is constructed. The Cartesian product of all **MA**, each representing a task or sequential constraint from the scheduling problem, creates an initial **CMA**. Dependency implication constraints, applied through intersection, prune acausal states and transitions from a **CMA**. Capacity implication constraints and a viability pruning step prevent iteration-sense confusion in cyclic models. Task bypassing double implications, operand resolution point implications and iterates help model correct execution of acyclic and cyclic control-dependent scheduling problems. The required **CMA** pruning steps for various types of scheduling problems are summarized in table 4.1.

This chapter stressed that concurrent operand instances are bounded in a cyclic **CMA**. As an **MA** represents existence or nonexistence of a finite number of operand instances, only operand instances proportional to the number of **MA** in a composition may be simultaneously represented. Thus, a cyclic **CMA** bounds the solution space. Several techniques were presented to relax this bound in a controlled fashion. These include using iterates, adding operand buffers and performing task splitting.

Table 4.1: Summary of Composition Steps

	Acyclic Data-Flow	Cyclic Data-Flow	Acyclic Control-Dependent	Cyclic Control-Dependent
Cartesian Product	Yes	Yes	Yes	Yes
Transition Relations	Δ^{data}	Δ^{data}	Δ^{data}, $\Delta^{control}$	Δ^{data}, $\Delta^{control}$
Task Bypassing	None	None	Yes	Yes
Dependency Constraints	Basic Alternative	Basic	Basic Resolved	Basic Resolved
Capacity Constraints	None	Basic	None	Global Iterates
Viability Pruning	No	Yes	No	Yes
Concurrency Constraints	Transition State	Transition State	Transition State	Transition State

This chapter also discussed composition techniques in general terms. Implications and double implications are necessary for all dependency and capacity constraints as well as synchronization with sequential constraints. Concurrency constraint hierarchies enforce sophisticated resource bounds of both system activities and local storage.

The final discussion in this chapter focused on efficient ROBDD representation of a **CMA**. A good ordering is found by following the expected flow of a scheduling problem's behavior. Some dependency and capacity constraints cause excessive ROBDD growth. This is avoided by partitioning a **CMA**'s transition relation. Finally, operand resolution points may be formulated in a more efficient manner for certain types of scheduling problems.

Chapter 5

Exploring Modeling Automata

MA and **CMA** as described in chapters 3 and 4 encapsulate all legal execution sequences of a system. Still, they may not be used directly as a finite state machine controller. Fundamentally, they represent multiple legal execution sequences via nondeterministic choices, yet a real implementation must make deterministic choices. If nondeterministic choices are pruned in a **CMA** to leave only one deterministic choice, or if multiple choices are made deterministic by conditions, then a finite state machine controller may be directly synthesized. But this raises questions as to which nondeterministic choices should be kept and made deterministic and which choices should be pruned. This chapter introduces exploration techniques that answer these types of questions.

Optimization requires an objective. The objective directs how and what type of deterministic sequences are extracted from an **MA**. A common objective is minimum latency. This may be stated as, “What execution sequences implement the entire scheduling problem in the smallest number of time-steps?” Variations of this exist for cyclic behavior, where minimum iteration latency is often desired, and control-dependent behavior, where some control cases are more favored than others. The exploration techniques presented here are directed by a minimum

latency objective. They first determine if, given all constraint imposed on a **CMA**, any valid execution sequence of any length exists. Then, they determine all such sequences of minimum latency.

At the core of ABSS exploration techniques is an implicit implementation of Dijkstra's shortest paths algorithm. For acyclic models, shortest paths from $S_{task\ start}$ to $S_{task\ final}$ represent minimum latency execution sequences. For cyclic models, shortest repeating sustainable paths in a **CMA** correspond to minimum iteration latency or maximum throughput execution sequences. Control-dependent models require that a set of shortest paths, covering all possible control cases in a deterministic and causal way, is found. Finally, with control-dependent models, some control cases may be favored over others and hence shortest paths for these favored paths are found at the expense of other control cases.

With regard to the RISC example, exploration determines execution sequences that exhibit high instruction throughput for the most common instruction cases. This requires finding shortest repeating paths in a **CMA** that execute all iterates. This is a difficult problem as paths for all control cases must be found and furthermore must each exhibit sustainable repeating behavior.

This chapter is organized as follows. First, exploration for acyclic data-flow models is presented. Basic exploration ideas and a detailed example are introduced here. Next, validation and ensemble path sets, which are both required for control-dependent exploration, are discussed in regard to acyclic models. Finally, exploration techniques relevant to cyclic models, such as behavior cuts, repeating kernels and closure, are presented first for data-flow and then for control-dependent **CMA**.

5.1 Acyclic Data-Flow Exploration

This section introduces exploration techniques for acyclic data-flow models. First, a detailed example illustrates basic exploration concepts required here and in later sections. Next, these exploration concepts are more formally defined and described. Finally, this section provides reasons why a **CMA**'s representation and exploration is efficient.

5.1.1 Example

Section 4.1 introduced a simple acyclic data-flow example with a final pruned **CMA** presented in figure 4.5 and again here in figure 5.1. The two minimum latency paths in this example are evident. Either the top or the bottom path may reach $S_{task\ final}$ from $S_{task\ start}$ in three time-steps. Although this is evident here, it is not evident in most real scheduling problems. Consequently, an implicit implementation of Dijkstra's algorithm is used to find all such shortest paths.

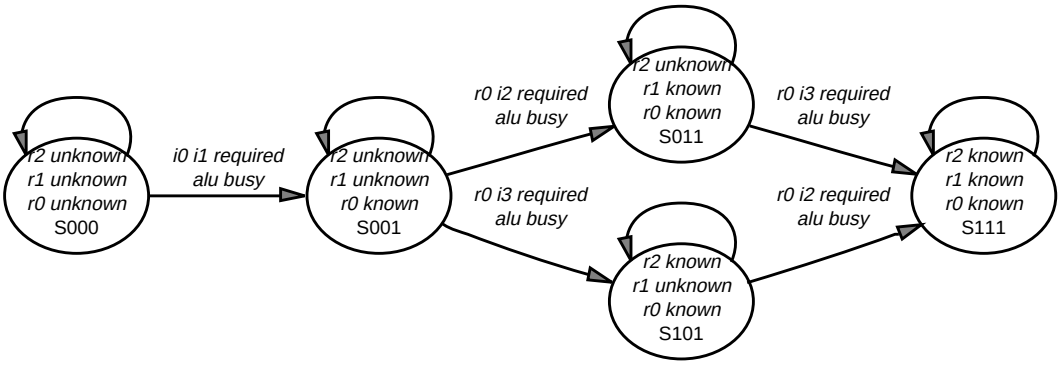


Figure 5.1 Final **CMA** for acyclic data-flow example from section 4.1

Consider states, as shown in figure 5.2, that are reached during a forward exploration starting with state S000. The state set at time-step 0 only contains S000. After a single image computation[1], the state set at time-step 1 contains states S000 and S001. Only at time-step 3 are $S_{task\ final}$ states present in a time-step set.

1. A image computation is described in section 4.2.6.

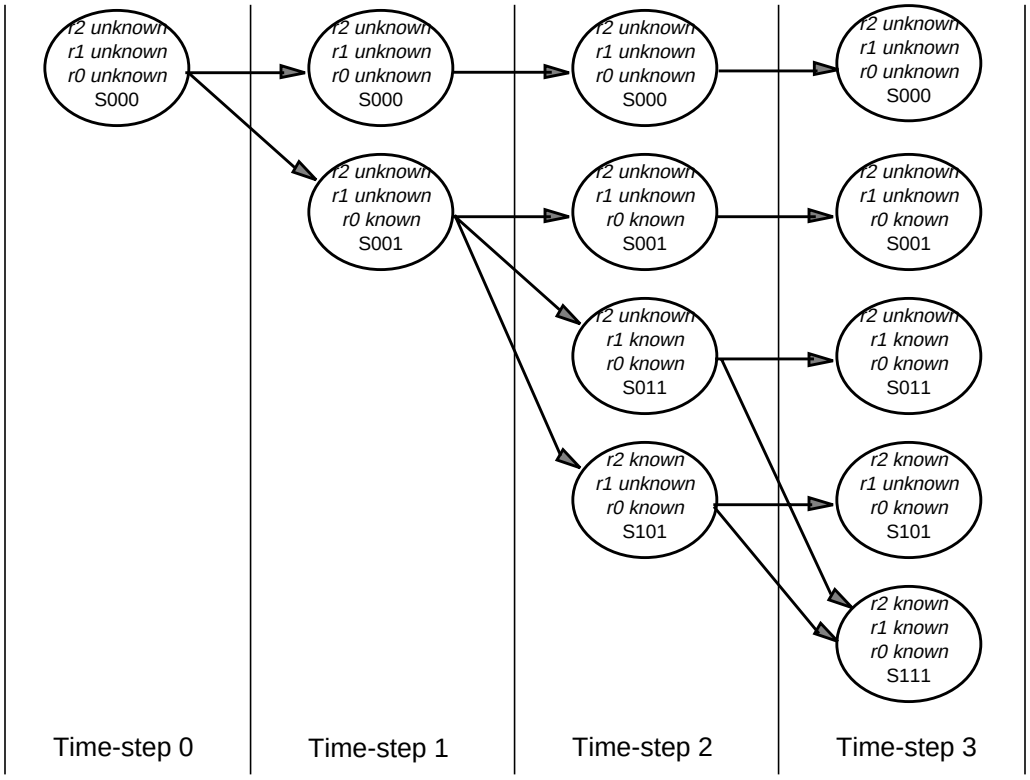


Figure 5.2 Forward exploration of figure 5.1

Reaching states in $S_{task\ final}$ is the forward exploration termination condition. This **forward exploration** process is similar to labeling graph vertices with their distances from $S_{task\ start}$ if all transitions are considered unit weight. For instance, S011 appears in the state sets at time-steps 2 and 3 because it is at least two image computations away from any state in $S_{task\ start}$. There is no guarantee that states in $S_{task\ final}$ are ever reached during forward exploration as constraints applied to a **CMA** may prohibit this. When this occurs, no solution is possible since ABSS implicitly searches the entire solution space.

Once forward exploration terminates, a **backward pruning** step occurs. This pruning starts with any states reached in $S_{task\ final}$, computes a preimage, and then uses this preimage set to prune the previous time-step set. Figure 5.3 illustrates how backward pruning is applied to figure 5.2. The preimage of S111, the only

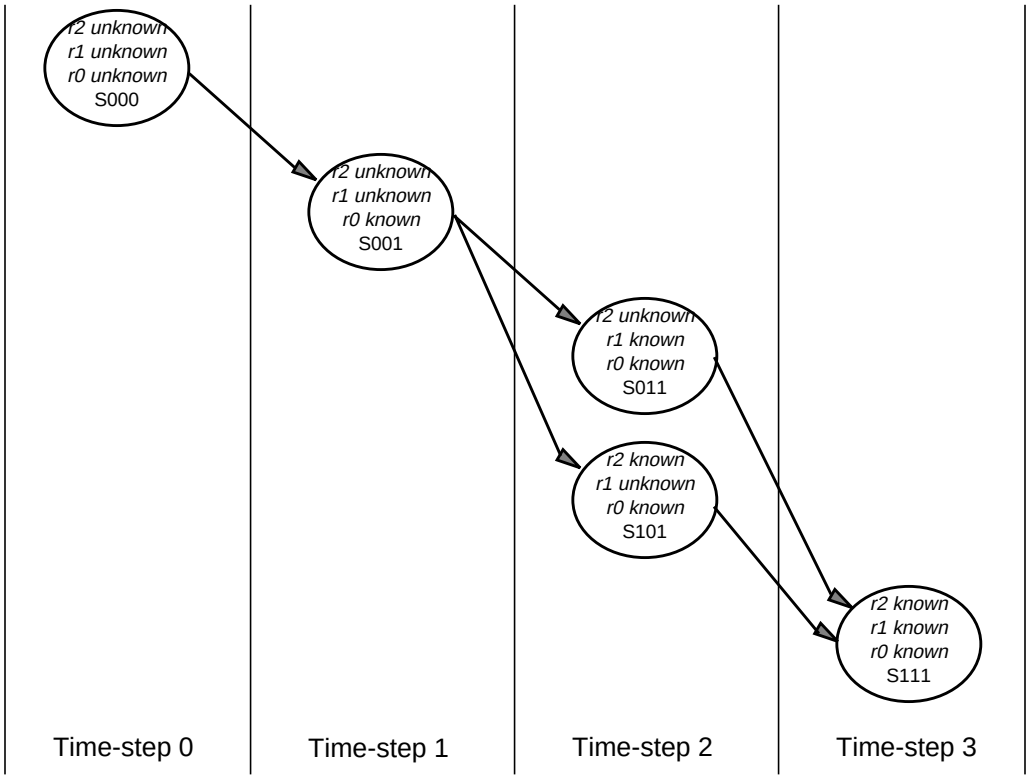


Figure 5.3 Backward pruning of figure 5.2

state reached in $S_{task\ final}$, returns states S011 and S101. This is intersected with the time-step 2 state set to prune states not on a path that reaches $S_{task\ final}$. Next, the preimage of this pruned time-step 2 state set is used to prune the time-step 1 state set. This continues until states at time-step 0 are reached. All shortest paths from $S_{task\ start}$ to $S_{task\ final}$ remain.

Nondeterministic choices are still present in figure 5.3 as all shortest paths are represented. It is possible to arbitrarily pick and preserve a single choice whenever faced with a nondeterministic choice to create a single deterministic execution sequence or **witness schedule** as shown in figure 5.4. This assumes that all choices are of equal cost as they all require the same minimum latency to reach $S_{task\ final}$. There may be additional cost considerations that prefer one choice over another.

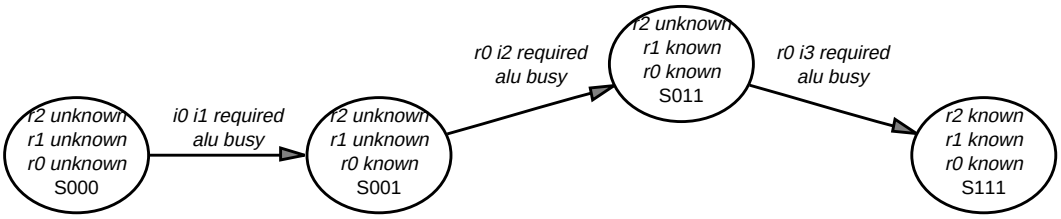


Figure 5.4 A final deterministic witness schedule

5.1.2 Basic Exploration Definitions and Algorithms

To facilitate a more formal discussion of exploration, some new definitions and notations are needed. In the example, states reached at a certain time-step were preserved in a time-step state set. This family of time-step state sets is called a path set.

Definition 5.1 A **path set** of a **CMA** is an indexed family of state sets $\{S^0, S^1, \ldots, S^n\}$ where S^{i+1} or S^{i-1} is contained in the image or preimage of S^i respectively. This encapsulates sets of **CMA** paths as defined in definition 3.2.

Symbolic preimage and image computations, as defined in section 4.2.6, are used extensively during exploration. To denote an image or next-state computation, the function,

$$Img(S_{label}, \Delta) = \exists_{(X, —)}(\Delta \bullet (S_{label}, —)) \tag{5.1}$$

is defined. This function returns all next states of states in S_{label} mapped to present state ROBDD variables. If the argument Δ is suppressed, than the default transition relation for the **CMA** under consideration, Δ^{data}, is implied. Likewise, a preimage or previous-state computation is denoted by the function,

$$Img^{-1}(S_{label}, \Delta) = \exists_{(—, X)}(\Delta \bullet (—, S_{label}))\,. \tag{5.2}$$

Minimum latency exploration of an acyclic data-flow **CMA** employs three algorithmic steps: forward exploration, backward pruning and, if desired, witness schedule extraction. Forward exploration begins with a **CMA**'s task start state,

$S_{task\ start}$, and constructs a path set, *PS*, of cardinality i+1 such that $PS.S^i \cap S_{task\ final} \neq \varnothing$. Figure 5.5 details the basic forward exploration algorithm. As long as an image computation does not contain states in $S_{task\ final}$, a time-step set is added to *PS*. Forward exploration terminates when the last time-step state set in *PS* contains some states in $S_{task\ final}$.

$i = 0$
$PS.S^i = S_{task\ start}$
`while(` $(PS.S^i \cap S_{task\ final}) == \varnothing$ `){`
    $PS.S^{i+1} = Img(PS.S^i)$
    $i = i+1$
`}`

Figure 5.5 Basic forward exploration algorithm

Once a candidate path set, *PS*, is created via forward exploration, backward pruning only preserves *all* paths from $S_{task\ start}$ to $S_{task\ final}$. As shown in figure 5.6, the last state set in *PS* is restricted to states that also appear in $S_{task\ final}$. A series of preimage computations prunes all states sets in *PS* such that only and all paths from $S_{task\ start}$ to $S_{task\ final}$ remain. A path set that contains all paths (not just a single path) is useful if further refinement is desired. For example, since all minimum latency paths are found, it is possible to further prune this path set by additional design objectives.

$i = |\ PS\ | - 1$
$PS.S^i = (PS.S^i \cap S_{task\ final})$
`while(` $i > 0$ `){`
    $PS.S^{i-1} = (PS.S^{i-1} \cap Img^{-1}(PS.S^i))$
    $i = i-1$
`}`

Figure 5.6 Basic backward pruning

The pruned path set, *PS*, contains all minimum latency execution sequences. This set may be restricted to represent a single witness schedule if desired. This begins by choosing a single state at time-step 0, computing an image, restricting time-step 1 by this image, and continuing through all time-step state sets in like manner. The witness extraction algorithm[2] is shown in figure 5.7.

$i = 0$
$s \in PS.S^i$
$PS.S^i = s$
`while(` $i < |PS| - 1$ `) {`
    $i = i+1$
    $PS.S^i = (PS.S^i \cap Img(PS.S^{i-1}))$
    $s \in PS.S^i$
    $PS.S^i = s$
`}`

Figure 5.7 Witness extraction algorithm

A witness schedule, as well as any other execution sequence in a **CMA** or a path set of a **CMA**, must be interpreted through transition labels. The path set *PS* contains state sets representing time-step boundaries. To determine what *activity* occurred during the *i*th time-step, the transition(s) $(PS.S^{i-1}, PS.S^i)$ must be considered. Any label which references $(PS.S^{i-1}, PS.S^i)$ identifies an activity that occurred during the *i*th time-step. In general, for each $\Delta_{label} \in LT$, if $(PS.S^{i-1}, PS.S^i) \subseteq \Delta_{label}$, then *label* occurs for at least one transition in $(PS.S^{i-1}, PS.S^i)$. If $(PS.S^{i-1}, PS.S^i)$ represents a single transition, then *label* definitely occurs for that transition.

2. If only a single witness schedule and no path set representing all minimum latency schedules is desired, then backward pruning and witness extraction may be merged into one step for acyclic data-flow **CMA**.

Figure 5.8 shows a conceptual view of the basic exploration steps. At the top, forward exploration has created a path set that includes minimum latency paths from $S_{task\ start}$ to $S_{task\ final}$. Both time-step state sets and transitions are shown. Backward pruning has been applied to the path set shown in the middle. Every minimum latency path is present. Finally, a single witness schedule remains in the path set shown at the bottom. This represents one minimum latency deterministic execution sequence for the **CMA**.

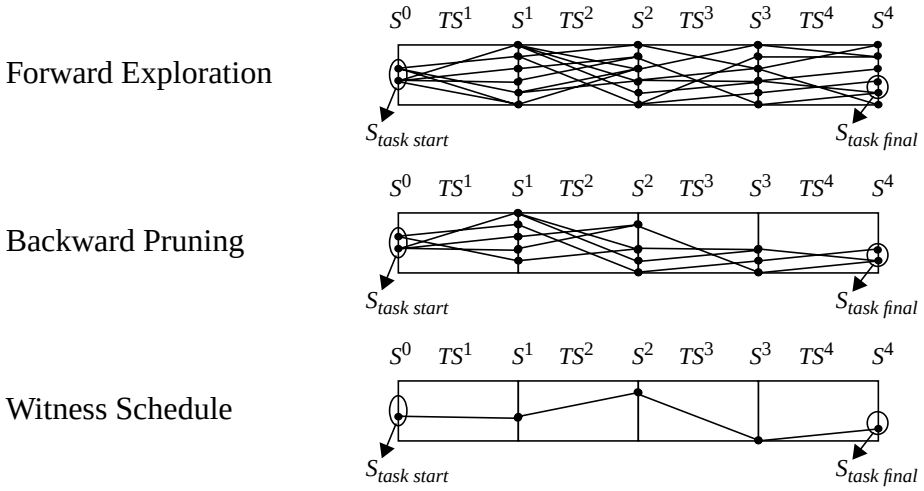


Figure 5.8 Conceptual view of exploration steps

5.1.3 Efficiency of a CMA's Representation

Representation and exploration of a **CMA** is potentially very costly as up to approximately 2^{200} states and an even larger number of paths may be dealt with. Still, exploration and representation of a **CMA** is often efficient for several reasons. First, use of implicit ROBDD representation often, though not necessarily always, provides dramatically efficient representation and manipulation of large sets. Second, particular properties of the ABSS formulation also enhance efficiency. Note that during forward exploration, the entire reached state set, not just the frontier, is propagated forward. Consider a task starting state S000000. Suppose the task represented by the single time-step **MA** in the most significant

state bit position executes during time-step 1. Hence, S000000 and S100000 are contained in $PS.S^1$. Only the cube S-00000 need be represented and no ROBDD variable is needed for the most significant state bit. On the other hand, if just the frontier is preserved and propagated, $PS.S^1$ will only contain S100000. This requires an ROBDD node to represent the most significant state bit. Propagating the entire reached state set rather than the frontier results in a ~3× improvement for ABSS.

Another reason for efficiency is merging of paths in a **CMA** and its path sets. Imagine a scheduling problem with three tasks: *a*, *b* and *c*. These tasks are completely independent --there are no operand dependencies among them. These tasks all require a single time-step ALU and only one such ALU is available. Given these assumptions, there are six possible orderings of *a*, *b* and *c*. A path set that doesn't support path merging is shown in figure 5.9. All six distinct paths reach six distinct task final states. A path set that does support path merging is

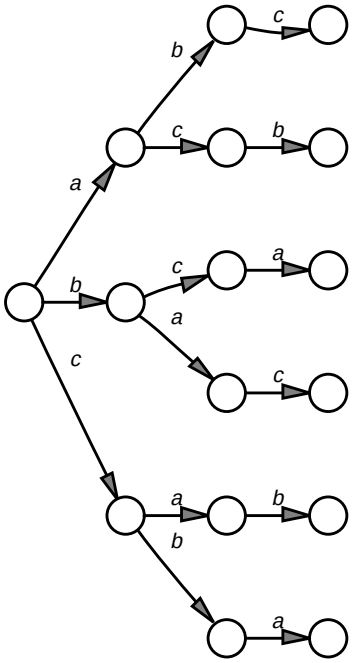


Figure 5.9 Nonmerging path set

shown in figure 5.10. Only half as many states are required when paths are allowed to merge. Path merging is similar to the sharing of isomorphic subgraphs that makes ROBDDs compact. A **CMA**'s nondeterminism and 'one-hot' Cartesian encoding naturally enables this efficient merging of paths. Although path merging is possible in an explicit representation, the number of paths for exact search of the scheduling solution space is typically so large that an implicit representation, which still supports path merging, is efficient and preferred in practice.

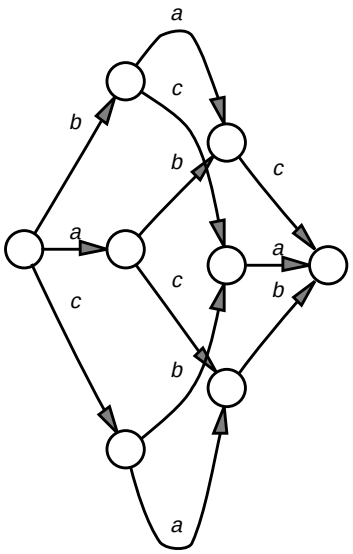


Figure 5.10 Merging path set

Even with these reasons for efficiency, representation and especially exploration of a **CMA** may be complex and costly. Still, ABSS is practiceable for problems of useful scale. Specific discussion regarding ABSS cost and complexity is presented in section 6.1.3 and section 7.2.3.

5.2 Acyclic Control-Dependent Exploration

To determine minimum latency schedules from an acyclic control-dependent scheduling problem's **CMA** also requires finding shortest paths. Similar to section 5.1, symbolic reachability finds shortest paths from a **CMA**'s task start states, $S_{task\ start}$, to task final states, $S_{task\ final}$. Unlike section 5.1, a witness

schedule is not a single path in a path set but rather an ensemble of paths from $S_{task\ start}$ to $S_{task\ final}$. Such an ensemble of paths is called an **ensemble schedule** and must include a path for each distinct control-dependent execution sequence. For instance, a RISC processor must be able to execute *all* instructions and therefore an ensemble schedule for a RISC processor contains sequences for every instruction. As a more specific example, consider some control-dependent behavior that branches into two sets of behaviors depending on a *true/false* control resolution. An ensemble schedule for this example's **CMA** must contain a path from $S_{task\ start}$ to $S_{task\ final}$ that represents execution of *true* control resolution behavior and another path that covers *false* control resolution behavior. Furthermore, all paths of an ensemble schedule must be mutually compatible. As will be shown, it is possible to find paths from $S_{task\ start}$ to $S_{task\ final}$ that cover all control cases yet do not represent a casual ensemble schedule. A fixed-point pruning, called **validation**, is performed during backward exploration pruning to insure that only valid ensemble schedules remain.

Control dependent models contain two transitions relations: Δ^{data} and $\Delta^{control}$. A time-step still corresponds to a single image computation of either Δ^{data} or $\Delta^{control}$. When modeling clock synchronous systems, two time-steps, one for Δ^{data} and another for $\Delta^{control}$, correspond to a single clock period. As one Δ is now not enough to create an entire path set, a transition set is associated with a path set to identify which Δ should be used to relate one time-step state set to the next.

Definition 5.2 If multiple transition relations are required in the computation of a path set, a **transition set** is associated with a path set. A transition set, $\{\Delta^0, \Delta^1, \ldots, \Delta^{n-1}\}$, is an indexed set containing all transition relations used in the computation of a path set. A transition relation Δ^i relates sets S^i, S^{i+1}.

5.2.1 The Validation Problem

To illustrate the validation problem, three possible ensemble schedules for the behavior in figure 5.11 are discussed. Notice that this behavior contains four tasks

```
if (d > 100) {        // Task c1
   y = a × b; }       // Task m1
else {
   y = c × b; }       // Task m2
z = x + y;            // Task a1
```

Figure 5.11 A behavioral example for discussion on validation

(*c*1, *m*1, *m*2, and *a*1), two control blocks (d>100, $d\leq$100), and one operand resolution point ($y^{d>100}$ or $y^{d\leq 100}$ resolves to y for task *a*1). For discussion simplicity, assume that each task is implemented in a single time-step. A **CMA** representing this behavior is constructed and subsets of paths in this **CMA** are explicitly presented for discussion.

Figure 5.12 shows a **CMA** portion representing the behavior in figure 5.11

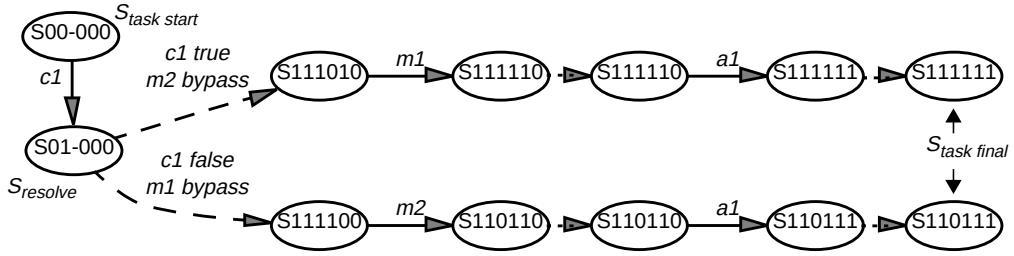


Figure 5.12 A **CMA** subset of a valid nonspeculative ensemble schedule

where no speculation occurs. Bits in the state vector are ordered {*c*1, *c*1, *c*1, *m*1, *m*2, *a*1}. In the first time-step, task *c*1 executes as seen by the change in state from 00- to 01- in *c*1's state portion. As *c*1 is modeled by a multivalue **MA**, resolution occurs during the control-phase. A path covering every possible control resolution leaves the *resolve*-labeled state, S01-000. Furthermore, during control-phase resolution, either tasks *m*1 or *m*2 are bypassed. The *true* path,

shown at the top, executes *m*1 and finally *a*1. Likewise, the *false* path, shown at the bottom, executes *m*2 and finally *a*1. This results in a valid ensemble schedule requiring 6 time-steps or 3 clock ticks for either control case. It is a valid ensemble schedule as all control cases are covered and all possible value resolutions leaving a *resolve*-labeled state continue to $S_{task\ final}$.

Figure 5.13 shows an explicit **CMA** portion where speculation for the behavior

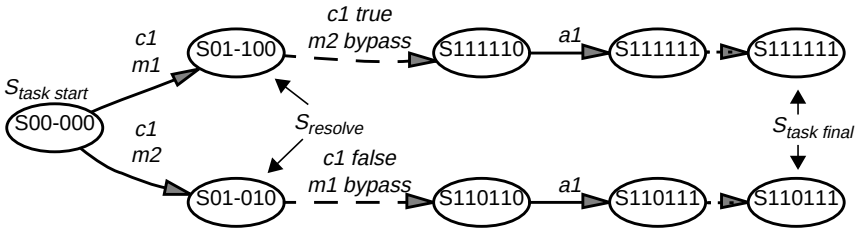


Figure 5.13 A **CMA** subset of an invalid speculative ensemble schedule

in figure 5.11 does occur. In the first time-step, task *c*1 executes and either task *m*1 or *m*2 executes speculatively. Tasks *m*1 and *m*2 may not both execute speculatively during the same time-step as only one multiplier resource is available. Now there are two *resolve*-labeled states, S01-100 and S01-010. Only paths favoring speculation are shown leaving these *resolve*-labeled states. The paths covering cases for incorrect speculation are *not* included. It appears as if either control case may complete and reach $S_{task\ final}$ in just four time-steps. In fact, there are legal speculative paths for either control case which do complete in just four time-steps. Thus, all control cases are covered, yet this is not a valid ensemble schedule. Since the control operand is not known at time-step 0, a deterministic machine can not choose the appropriate speculation *a priori*. It must choose to speculate either on *m*1 or *m*2. If it chooses to speculate incorrectly, then it must have a recovery path. Unfortunately, the *resolve*-labeled states only have exiting paths which presume a correct speculation was done. Hence, a deterministic implementation can not be synthesized which requires just four time-steps for both control cases.

Figure 5.14 shows a **CMA** path subset that contains recovery paths. Although speculation may occur for either *m*1 or *m*2, paths exiting *resolve*-labeled states S01-100 and S01-010 now cover *all* possible control resolution cases. Hence, when a control value resolves, a deterministic machine always has an appropriate path for the particular value resolution. In fact, figure 5.14 contains two valid

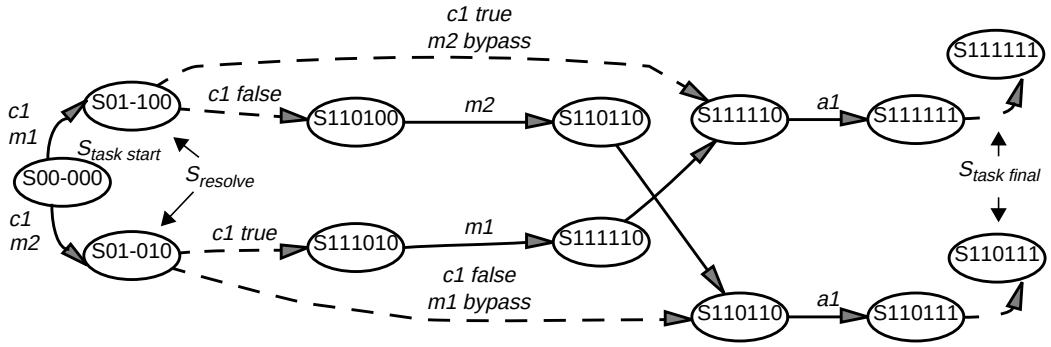


Figure 5.14 A **CMA** subset of two valid speculative ensemble schedules

ensemble schedules. One speculatively executes *m*1 and may complete a *true* resolution in four time-steps yet requires six time-steps for a *false* resolution. The other speculatively executes *m*2 and may complete a *false* resolution in four time-steps yet requires six time-steps for a *true* resolution. This observation leads to a deeper scheduling objective question, “Which subset of control cases should be favored for minimum latency and at what expense to other control cases?”

A validation step is added to backward pruning to guarantee every path is a member of some valid ensemble schedule. The validation step insures that all *resolve*-labeled states in a path set have exiting paths covering all control resolutions. Finally, control cases may be prioritized during backward exploration to favor minimum latency of some control cases at the expense of others.

5.2.2 Unprioritized Exploration Overview

Two types of exploration are presented for acyclic control-dependent models: unprioritized and prioritized. Unprioritized exploration does not favor some

control cases at the expense of others but rather finds a minimum latency execution sequence that includes all control cases. This is the simpler exploration strategy and is presented first.

Figure 5.15 shows a conceptual overview of control-dependent exploration

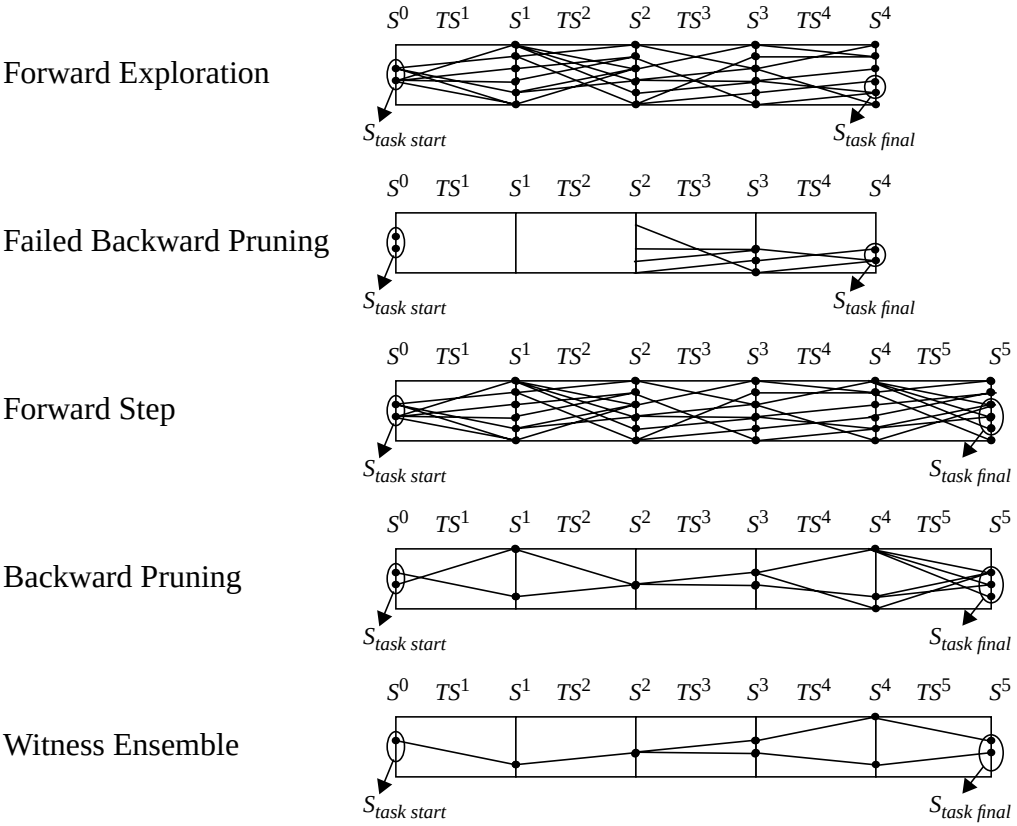


Figure 5.15 Conceptual view of control-dependent exploration steps

steps. These steps are described in detail in subsequent sections. First, forward exploration creates a path set with shortest paths from $S_{task\ start}$ to $S_{task\ final}$. A further criteria is that states in $S_{task\ final}$ covering all possible control resolutions are reached. Next, backward pruning is attempted. During backward pruning, a validation step insures that when a multivalue **MA** resolves, all control cases are still present and continue to $S_{task\ final}$. As shown in the failed backward pruning, states in S^2 are removed as they fail validation. No states remain and hence no

valid ensemble schedules exist. Therefore, another forward step is added to the *original* path set and backward pruning is attempted again. In the second backward pruning attempt, one state in S^2 survives validation as it has exiting paths for both control values. Still, several ensemble schedules exist. The final step reduces the path set to a single witness ensemble schedule.

5.2.3 Forward Exploration

Forward exploration for acyclic control-dependent models proceeds in similar fashion as previously described for acyclic data-flow models. With basic data-flow

$i = 0$

$PS.S^i = S_{task\ start}$

`while(` $apc(PS.S^i \cap S_{task\ final}) \neq apc(S_{task\ final})$ `){`

$PS.S^{i+1} = Img(PS.S^i)$

$i=i+1$ `}`

Figure 5.16 Forward exploration algorithm with all-paths check

forward exploration, termination occurs once any states in $S_{task\ final}$ are reached. With control-dependent forward exploration, termination occurs once states in $S_{task\ final}$ are reached which also cover all control cases. An **all-paths check** is formulated to determine if a state set covers all control cases. Let $X_{no\ values}$ be the set of all ROBDD variables in a **CMA** less those variables that encode a control value.[3] The all-paths check for some state set S is expressed as,

$$apc(S) = \exists_{(X_{no\ values}, —)} S. \tag{5.3}$$

All variables, except those indicating control value, are existentially quantified[4] out. This leaves only control case values. If *apc* exactly equals all expected control

3. When encoding a multivalue **MA**, some state variables serve only to distinguish operand values. These variables are denoted X_{values}. $X_{no\ values}=X-X_{values}$.

4. Existential abstraction is used rather than universal abstraction as states reached in $S_{task\ final}$ may cover all control cases yet still differ when considering other state variables.

case values for a **CMA**, then the all paths check is successful. All expected control cases for a **CMA** may be determined by applying equation 5.3 to a **CMA**'s $S_{task\ final}$. Finally, figure 5.16 shows a forward exploration algorithm modified to include an all-paths check.

5.2.4 Backward Pruning with Validation

Backward pruning for acyclic control-dependent models requires a validation step at each state set in a path set that may contain $S_{resolve}$ states. Validation simply insures that for any state labeled *resolve*, a transition to every possible resolved value exists. Due to the way a multivalue **MA** is specified, $S_{resolve}$ states are only reached during backward pruning in a preimage of Δ^{control}. Hence, validation is only required when computing a preimage of Δ^{control}.

$V^0 = \varnothing$

$V^1 = \exists_{(—,\, X - X_{values})}((—, S) \bullet \Delta^{control})$

$j = 1$

`while (` $V^j \neq V^{j-1}$ `){`

$V^{j+1} = \forall_{m \in M_{multivalue}}((\forall_{(—,\, m.X_{value})} V^j_{m.resolve}) + V^j_{\overline{m.resolve}})$

$j = j + 1$

`}`

`return` $\exists_{(—,\, X_{values})} V^j$

Figure 5.17 Validated preimage computation

Figure 5.17 describes a validated preimage computation. An initial set, V^1, computes a *partial* preimage of state set *S.* This is a partial preimage as not all next state variables are existentially quantified out. Next-state variables which distinguish value for any multivalue **MA** in a composition remain. The validation fixed point, within the while loop, acts on this set *V.* For every multivalue **MA** *m*

within the composition, two partitions are formed. $V_{\overline{m.resolve}}$ contains all states in V where m is not in the *resolve*-labeled state while $V_{m.resolve}$ contains all states in V where m is in the *resolve*-labeled state. Next-state variables for a m which distinguish value are universally quantified. Only *resolve*-labeled states with transitions to next states covering all possible resolved values remain. As validation for some multivalue m may remove required transitions for other multivalue m, validation is repeated for all m until no additional pruning occurs. Finally, remaining next-state variables are existentially quantified out. The returned set is the validated preimage of S.

This validated preimage computation is incorporated directly into backward exploration pruning as seen in figure 5.18. Within the while loop, two preimages

$i = |PS| - 1$
$PS.S^i = (PS.S^i \cap S_{task\ final})$
`while(` $i > 0$`){`
    $PS.S^{i-1} = (PS.S^{i-1} \cap ValImg^{-1}(PS.S^i))$
    `if(` $PS.S^{i-1} = \varnothing$ `) break`
    $i = i - 1$
    $PS.S^{i-1} = (PS.S^{i-1} \cap Img^{-1}(PS.S^i))$
    $i = i - 1$
`}`

Figure 5.18 Backward pruning with validation

for both $\Delta^{control}$ and Δ^{data} are computed. A validated preimage computation is only used with $\Delta^{control}$. This assumes that the initial path set always contains an odd number of time-step state sets and forward exploration is altered to guarantee this. If the validated preimage returns an empty set, then backward pruning fails. Backward exploration pruning is attempted again with another path set that is

extended by two time-step sets with forward exploration. For this reason, a copy of PS is always used during backward exploration pruning so that the original may be extended. When necessary, the original path set is extended through image computations, $PS.S^{i+1} = Img(PS.S^{i})$.

5.2.5 Ensemble Witness Extraction

For control-dependent exploration, iterative calls to the witness extraction algorithm shown in figure 5.7 are made until a complete ensemble is obtained. Figure 5.19 illustrates how a complete ensemble is extracted. After a single

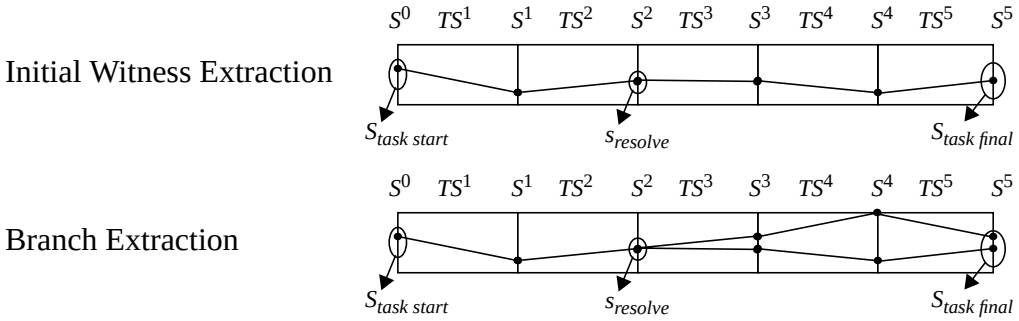


Figure 5.19 Conceptual view of ensemble witness extraction

application of the witness extraction algorithm to a *copy* of *PS*, a single schedule for precisely one control case is extracted. States labeled *resolve* are reached at various time-step state sets in this initial schedule. Whenever a $s_{resolve}$ state is reached, a random choice is made as to which control case to proceed on. When a subsequent iterative call to the witness extraction algorithm is made, it begins in the *original* path set with this $s_{resolve}$ state and at the appropriate time-step state set. When the image of $s_{resolve}$ is computed, it is restricted by intersection to include control cases other than what has already been completed. This is never empty as validation has guaranteed that the image of any $s_{resolve}$ state includes states covering all control cases. These subsequent iterative calls serve to extract branches for all control cases. Iterative calls to the witness extraction algorithm continue until all control cases are covered. The original witness schedule and all

branches are included in a final path set encapsulating a complete ensemble schedule.

Witness extraction is not an essential step in exploration. It is useful primarily when a single schedule or ensemble schedule must be explicitly examined. As a backward pruned path set contains all minimum latency schedules, an implicit direct path from this to a dynamic finite state machine controller is preferred.

5.2.6 Prioritized Exploration Overview

The problem with unprioritized exploration is that only the longest latency control case is optimized. It may be that other control cases have valid paths that complete in even fewer time-steps. Furthermore, other control cases may be optimized first at the expense of the longest latency control case. For instance, in a RISC processor, some instructions occur more often than others. Execution sequences for these high probability instructions may be optimized at the expense of other infrequent, though still necessary, instructions. Figure 5.20 shows a

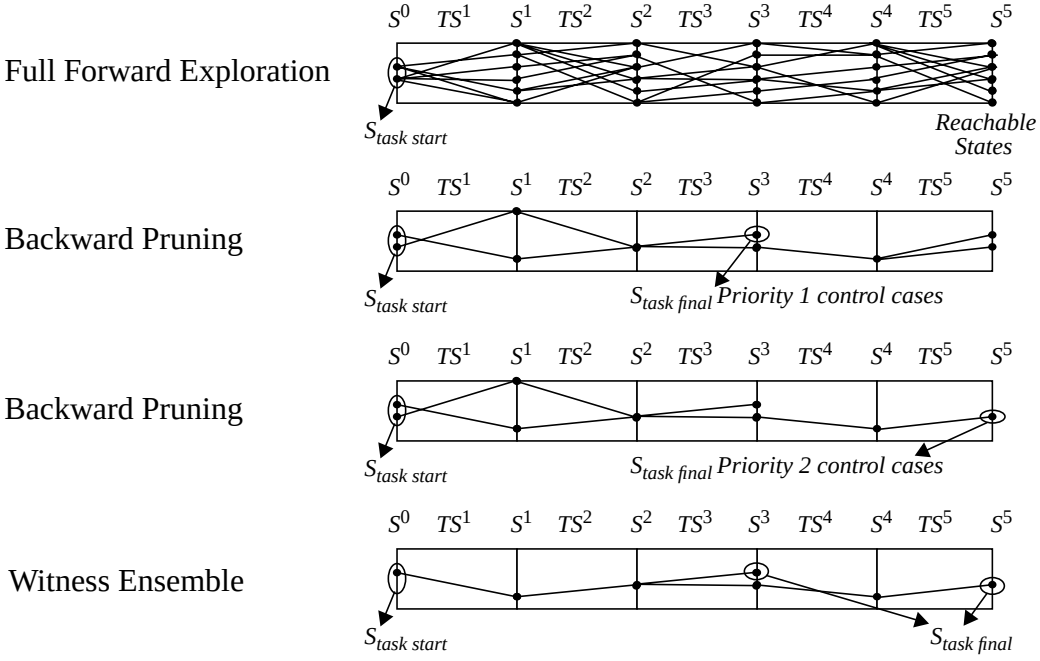


Figure 5.20 Conceptual view of control-dependent prioritized exploration

conceptual view of control-dependent prioritized exploration steps. First, forward exploration creates a path set that adds time-step state sets until *all* states are reached. This is desired because prioritizing one control case may delay termination of another control case. For instance, speculation in favor of a prioritized control case will consequently delay other control cases. By computing all reachable time-step sets, all possible delays are accounted for. Next, an **earliest termination** algorithm finds the first time-step set where states in $S_{task\ final}$ are reached that cover the highest priority control case(s). Future activity for terminated control cases is removed from the path set. Backward pruning then acts on the entire path set. If successful, the same steps are applied for the next highest priority control case. If unsuccessful, the control cases under current consideration are allowed to terminate two time-steps later and backward pruning is attempted again. This continues until all control cases have been prioritized. Finally, a witness ensemble may be extracted.

To facilitate control-case prioritization, a **priority list** is created. This is an ordered list of control-case terms, *cp*. Each control-case term is a present state expression of multivalue **MA** values. A control-case term may be a sum of products and hence represent multiple control cases. For a **CMA** with numerous control cases, it may be impractical or unneeded to prioritize each individual control case separately. Often a set of control cases, represented by one control-case term, is prioritized. In this case, the latency of the worst control case within this control-case term will be minimized. The latency for all other control cases is only guaranteed less than or equal to this worst-case latency. Even if better latency is possible for these other control cases, it will not be guaranteed unless these paths are specifically prioritized.

Control-case prioritization is a greedy algorithm. Even if a certain control case should only be favored slightly over another, it will be favored as much as possible

and at any expense to lower priority control cases. For this reason, a hierarchy of control-case terms is often used. At first, several control cases are prioritized jointly. After this, individual control cases may be prioritized. In an ideal control-case prioritization algorithm, control cases would be prioritized such that expected latency is minimized. The expected latency is the weighted sum of all control-case latencies where each latency is weighted by its execution probability.

5.2.7 Full Forward Exploration

Control-case prioritization requires a path set that includes all reached states. Full forward exploration, or complete reachability, is required. This allows prioritization of one control case to delay other control cases, yet these delayed control cases are still contained in a path set. A new termination condition of no new states reached is added to previously described forward exploration algorithms. Figure 5.21 describes the full forward exploration algorithm.

$i = 1$

$PS.S^i = S_{task\ start}$

$rs = Img(PS.S^0)$

`while(` $rs \neq PS.S^{i-1}$ `){`

  $PS.S^i = rs$

  $rs = Img(PS.S^i)$

  $i = i+1$

`}`

Figure 5.21 Full forward exploration algorithm

5.2.8 Control-Case Termination and Future Exclusion

Minimum latency prioritized schedules are not found by applying backward pruning directly to a full path set. Rather, a full path set must be pruned to include earliest terminations of the control case undergoing prioritization. Earliest

termination is a $PS.S^i$ such that states in $S_{task\ final}$ are reached which cover the currently considered control-case term, *cp*. Figure 5.22 describes the algorithm to

```
i = 0
while( i < |PS| ){
   if ( apc(PS.S^i ∩ S_task final) == cp) )
      break
   i=i+1
}
ets = i
```

Figure 5.22 Earliest termination algorithm for control cases *cp*

find such a $PS.S^i$. Beginning with the first time-step set in a path set, each time-step set is checked for intersection with $S_{task\ final}$ and containment of *cp*. If so, the algorithm breaks and *ets* indicates the earliest time-step where task final states are reached for *cp*.

Even though *cp* terminates in $PS.S^{ets}$, time-step state sets past this are still required as paths for other control resolutions, required for a valid ensemble, may terminate later. Still, for the control cases under consideration, no future terminations may be allowed as they would imply schedules with latencies longer than *ets.* To meet these two conditions, all path set time-step sets are kept but all time-step sets $PS.S^j$ where $j > ets$ are intersected with $\overline{cp}$ to exclude any states for the terminated control-case term.

5.2.9 Backward Pruning for Control-Case Prioritization

Earliest termination and future exclusion of terminated control-cases is performed on a copy of the original path set. As before, this allows for failure during backward pruning and additional attempts with incrementally extended path sets. Backward pruning for control case prioritization builds on what is

described in figure 5.18 and is detailed in figure 5.23. One difference is the

$i = |PS| - 1$

$PS.S^i = (PS.S^i \cap S_{task\ final})$

`while(` $i > 0$ `){`

$PS.S^{i-1} = (PS.S^{i-1} \cap ValImg^{-1}(PS.S^i))$

$i = i - 1$

$Terminated = (PS.S^{i-1} \cap S_{task\ final})$

$PS.S^{i-1} = (PS.S^{i-1} \cap Img^{-1}(PS.S^i))$

$PS.S^{i-1} = (PS.S^{i-1} \cup Terminated)$

$i = i - 1$

`}`

Figure 5.23 Backward exploration with preservation of terminated states

preservation of $S_{task\ final}$ states at each preimage computation of Δ^{data}.[5] This is required as backward pruning starts at the last time-step set in *PS*. This time-step state set may have some control cases excluded due to future exclusion of terminated control cases. Consequently, these previously terminating control cases will never appear in a series of preimage computations starting with the last time-step state set in *PS*. Hence, their termination states must be preserved when they appear in a time-step state set. Another difference is the removal of the break condition. Backward pruning now fails only if $PS.S^0$ is an empty set rather than when any time-step set is empty. This allows some late time-step sets in a full path set to be reduced to empty by validation while the real schedule exists in early time-step sets.

The entire prioritized exploration flow requires successive iterations of early control case termination, future termination exclusion and backward pruning for

5. This occurs only for preimage computations of Δ^{data} and not $\Delta^{control}$ as termination occurs at a clock-tick boundary and not at every time-step boundary.

all control cases, *cp*, in a priority list. Once all prioritization has completed, a witness ensemble may be extracted as described in section 5.2.5.

5.3 Cyclic Data-Flow Exploration

With a cyclic **CMA**, shortest path searches are also used to determine minimum iteration latency. However, unlike acyclic **CMA**, shortest paths from $S_{task\ start}$ to $S_{task\ final}$ are not sufficient. Rather, a shortest *repeating* path is sought. Such a repeating path, called a **repeating kernel**, represents a minimum iteration latency steady-state repeating behavior. This section describes how minimum iteration latency repeating kernels are determined for cyclic data-flow **CMA**.

This section presents several new concepts and algorithms required to find repeating kernels in a **CMA**. First, an overview of repeating kernels is presented. To determine repeating kernels, a behavior cut is made to determine a potential starting state set as neither $S_{task\ start}$ or $S_{task\ final}$ may even appear in minimum iteration latency repeating kernels. Then, in similar fashion to forward exploration, a path set, which contains all time-step state sets for candidate repeating kernels, is created. These candidate loop kernels are pruned so that only complete and closed repeating kernels remain. Finally, a single witness repeating kernel may be extracted.

5.3.1 Repeating Kernels

Consider at an abstract level how execution for some repeating data-flow behavior may appear. As shown in figure 5.24, execution begins with a sequence of

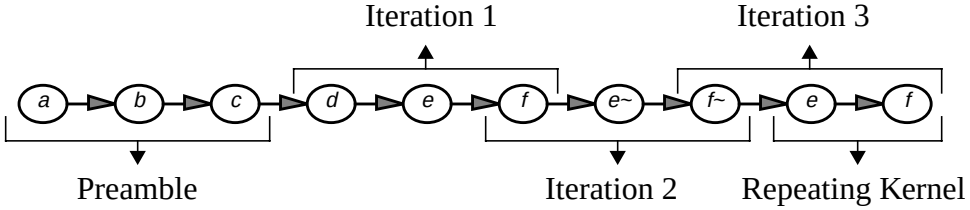


Figure 5.24 Repeating execution unrolled to reveal repeating kernel

preamble states and continues with multiple iterations of the behavior. After enough iterations have passed, a steady-state behavior may become apparent. In the figure, the state sequence *e*,*f* begins to repeat. Such a repeating state sequence is a repeating kernel. In this example, **iteration delay**, or time required for one complete iteration, is three time-steps, while **iteration latency**, or time between successive iterations, is two time-steps.

Two repeating kernels, shown as dotted transitions, are also seen in the explicit **CMA** for the example introduced in section 2.5 and duplicated here in figure 5.25.

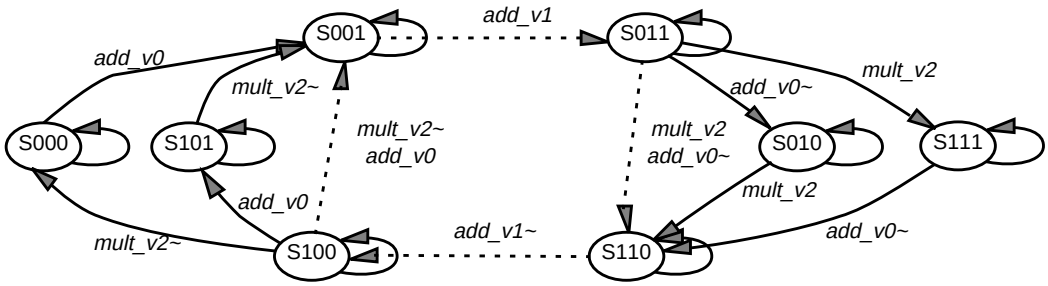


Figure 5.25 An explicit **CMA** for a cyclic data-flow scheduling problem

Because of the symmetric even/odd construction of cyclic **MA** and **CMA**, two iteration execution **legs**, even and odd, represent two potentially overlapping execution iterations of the repeating behavior. Both legs are identical except for sense (duals) and imply a complete closed repeating kernel consisting of a single leg. Such a repeating kernel represents a finite state machine controller with number of control steps equal to the iteration latency.

It may be that a complete closed repeating kernel may not consist of a single leg as in this example but rather an ensemble of legs. This may be necessary for complex repeating behaviors where the average iteration latency is optimized. For example, some behavior may require 2 time-steps for even iteration legs yet 3 time-steps for odd iteration legs. This leads to an average iteration latency of 2.5 time-steps. Even more complex cases are possible. This implies a finite state

machine controller with number of control steps greater than the average latency as each leg may have a different execution sequence. ABSS extracts a complete closed repeating kernel which contains such best average iteration latency schedules. In the current algorithm for witness extraction, a repeating kernel with exactly one leg is found.

Conventional scheduling techniques do not always guarantee optimal steady-state repeating behavior. Instead, search typically starts from a natural system starting state, proceeds through the initial iteration, and then enters a steady-state behavior based on this initial iteration. Basing the steady-state behavior on this initial iteration may lead to suboptimal solutions. On the other hand, determining optimal steady-state behavior is challenging as no obvious end or beginning exists. The questions of where and how much iteration overlap should exist in an optimal steady-state solution usually leads to circular reasoning. In this respect, the technique presented here cleanly finds and guarantees an optimal (indeed *every* optimal) steady-state repeating kernel since *all* possible executions are considered.

5.3.2 Overview of Repeating Kernel Algorithms

A repeating kernel is found for figure 5.25 by example. This serves as an overview of the upcoming specific discussion of repeating kernel algorithms. The first step is choosing an activity to construct a behavior cut *a priori*. It is not known what state(s) occur in a repeating kernel. It is known that some activity must occur during every iteration of the scheduling problem. Consequently, any single activity that is known to be necessary in every iteration, can be chosen as the behavior cut activity. All immediate next states of transitions exhibiting the behavior cut activity form a behavior cut state set, S_{bc}. In the example, if the activity `add_v0` is chosen as the behavior cut, then S_{bc} is as identified in figure 5.26. By symmetry, there is also the dual $S_{bc\sim}$.

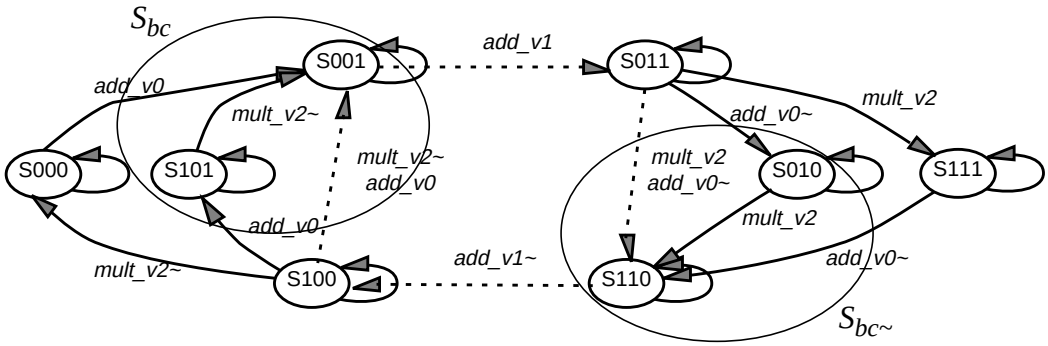


Figure 5.26 Cyclic **CMA** with behavior cut state sets S_{bc} and $S_{bc\sim}$

The next step is to find shortest paths from S_{bc} to $S_{bc\sim}$. This is just forward exploration and creates the path set, *PS*={{S101,S001}^{0}, {S101,S001,S011}^{1}, {S101,S001,S001,S010,S110}^{2}}. Then, a backward pruning step preserves only states belonging to shortest paths, *PS*={{S001}^{0},{S011}^{1},{S010,S110}^{2}}. This path set represents all iteration legs with latency of two time-steps.

Some of the iteration legs contained in a backward pruned *PS* are not repeatable within the same iteration latency. For example, if leg {{S001}^{0},{S011}^{1},{S010}^{2}} is used, then it may not execute a subsequent iteration in two time-steps or less. This can be determined by examining the last time-step state, S010. Its dual, S101 is not present in the first time-step of *PS*. Hence, there is no iteration leg to attach to this iteration leg which completes the next iteration in two or less time-steps. This argument may be made as a cyclic **CMA** is symmetric by construction and therefore a dual *PS*~ exists. A closure algorithm prunes a *PS* such that the first and last time-step states are exactly equal as duals. After closure, the closed pruned *PS*={{S001}^{0},{S011}^{1},{S110}^{2}}. This path set represents all minimum iteration latency repeating kernels and a witness extraction algorithm may choose a single repeating kernel.

5.3.3 Behavior Cuts and Tags

A valid *cut activity* is any atomic activity of some **MA** in the composition that occurs once in every complete iteration. Let $\Delta_{cut\ activity}$ represent the set of all even-sense transitions for which the *cut activity* is *true.* An even behavior cut state set is all next-states of these transitions and is computed as,

$$S_{bc} = \exists_{(X,\_)}\Delta_{cut\ activity}. \tag{5.4}$$

The dual $S_{bc\sim}$, may be found in similar fashion. Because of the encoding choice used in all cyclic **MA**, duals may also be found by bitwise complementation. This is the preferred method and may be implemented with the standard ROBDD *Compose*() function.

Scheduling problems may consist of multiple independent behaviors sharing one resource set. In this case, if a cut activity is selected from independent behavior *g1*, S_{bc} will contain all possible states in both even and odd senses for behavior *g2* by fact that it is independent. This implies that paths exist from S_{bc} to $S_{bc\sim}$ which schedule the first independent behavior but stall in both senses for the other independent behavior and consequently never execute a single task from *g2*. This is illustrated in figure 5.27. Here, a state is divided into two portions

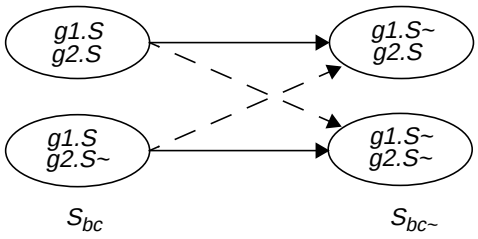


Figure 5.27 False closure

representing states contributed by both independent behaviors. The cut set is determined with an **MA** from *g1*. Hence it may have a state in only on sense in S_{bc} or $S_{bc\sim}$. On the other hand, all possible permutations for *g2* exist. Consequently, it

is possible to have states in both senses present for this independent behavior. Closed and pruned paths are shown that reach a valid $S_{bc\sim}$ yet no state change has occurred for *g2*. Closure mistakenly assumes that the dotted paths exist but in reality the solid paths exist. In reality, no execution activity for *g2* occurs.

To circumvent this scenario, additional ROBDD variables are added to S_{bc} and $S_{bc\sim}$ which 'tag' the sense of each *external* output operand identified from the scheduling problem's T_{output}. These do not change during traversal and hence provide a record of the initial sense of an output operand. Consequently, only paths producing *all* output operands at least once may be identified. In the example from section 2.5 and used in section 5.3.2, the only external output is `rv2` produced by task *v2*. The correctly tagged S_{bc} is $\{S0_001,S1_101\}$ where subscript '1' tags states labeled *rv2 known* and '0' tags states labeled *rv2~ known*. Likewise, the correctly tagged $S_{bc\sim}$ is $\{S1_010,S0_110\}$ where '1' tags states labeled *rv2~ known* (inverted from S_{bc}) and '0' tags states labeled *rv2 known*.

In the ABSS implementation, a triple of ROBDD variables is assigned for each state bit. This triple consists of (*tag*, *ps*, *ns*) where *ps* and *ns* are present and next state variable respectively. Applying an initial tag as required by S_{bc} requires intersecting the term $\overline{tag \oplus ps}$ for every state variable that belongs to a task **MA** producing an externally visible operand. Hence, the tag state is identical to the present state for these task in S_{bc}. Likewise, a tag as required by $S_{bc\sim}$ requires intersecting the term $tag \oplus ps$ for every state variable that belongs to a task **MA** producing an externally visible operand. Hence, the tag state is opposite from the present state for these task in $S_{bc\sim}$. Consequently, any path from S_{bc} to $S_{bc\sim}$ must toggle present state variable value relative to these tagged **MA** and can not simply idle. Finally, when computing a dual, tag state bits are not complemented.

5.3.4 Forward Exploration and Backward Pruning

The states sets S_{bc} and $S_{bc\sim}$ are analogous to $S_{task\ start}$ and $S_{task\ final}$ yet apply to legs in a repeating kernel. Forward exploration, as described in section 5.1.2 is used to create a path set, *PS*, containing shortest paths from S_{bc} to $S_{bc\sim}$. Also, backward pruning, as described in section 5.1.2, is used to prune away all states not on some shortest path from S_{bc} to $S_{bc\sim}$. The remaining sequences represent candidate minimum iteration latency legs for repeating kernels. As the next step, closure, may fail, a *copy* of *PS* is used during backward pruning. The original *PS* may then be extended by a single time-step and backward pruning attempted again. Furthermore, backward pruning may be applied to a *PS* with early terminations. These terminations are preserved and added back to the preimage computation as in figure 5.23. Finally, closure requires that all terminations, at whatever time-step they are reached, be considered. Therefore, the set $RS_{bc\sim}$ accumulates all states in $S_{bc\sim}$ that are reached at any time-step set. Figure 5.28

$i = |PS| - 1$
$PS.S^i = (PS.S^i \cap S_{bc\sim})$
$RS_{bc\sim} = PS.S^i$
`while(` $i > 0$ `){`
  $Terminated = (PS.S^{i-1} \cap S_{bc\sim})$
  $RS_{bc\sim} = (RS_{bc\sim} \cup Terminated)$
  $PS.S^{i-1} = (PS.S^{i-1} \cap Img^{-1}(PS.S^i))$
  $PS.S^{i-1} = (PS.S^{i-1} \cup Terminated)$ `}`

Figure 5.28 Backward exploration pruning with termination accumulation

describes this modified backward pruning. As shown, the *Terminated* set preserves any early terminating states. Also, the set $RS_{bc\sim}$ accumulates termination states reached at any time-step set in the path set.

5.3.5 Closure

As illustrated in section 5.3.2, paths exist in a backward pruned path set which are not closed --there is no way to continue their execution within the current minimum iteration latency bounds. A fixed-point algorithm prunes the path set so that only *closed* shortest paths remain. Backward pruning is applied to a path set until $PS.S^0$ equals all terminations, $RS_{bc\sim}$, as duals. Hence, by symmetry argument, only repeatable iteration legs of at most length n remain.

The algorithm in figure 5.29 describes the **closure fixed-point**. Within this fixed-point, the sets $PS.S^0$ and $RS_{bc\sim}$ are set equal as duals. As long as $PS.S^0$ and $RS_{bc\sim}$ are not equal as duals, a backward pruning is applied. Backward pruning computes a new $RS_{bc\sim}$. Only when $PS.S^0$ equals $RS_{bc\sim}$ exactly as duals does the fixed-point terminate. This guarantees that *all* paths originating in $PS.S^0 \subseteq S_{bc}$ always reach state(s) in $RS_{bc\sim} \subseteq S_{bc\sim}$ in at most n time-steps. Furthermore, for any path terminating at some state $s\sim \in RS_{bc\sim}$, there exists a state $s \in PS.S^0$ such that s always initiates path(s) reaching $RS_{bc\sim}$ with length $\leq n$. This is true by way of a **CMA**'s symmetry and the closure fixed-point.

```
while( PS.S^0 ≠ dual(RS_bc~) ){
    PS.S^0 = PS.S^0 ∩ dual(RS_bc~)
    RS_bc~ = dual(PS.S^0)
    PS.S^n = PS.S^n ∩ RS_bc~
    PS = BackwardPrune(PS)
    if( PS =∅ ) break
}
```

Figure 5.29 Closure fixed-point

Figure 5.30 illustrates a closed path set abstractly. The sets $PS.S^0$ and $RS_{bc\sim}$ are equal although in opposite senses. There are two paths (iteration legs), a to $a\sim$ and d to $d\sim$, directly repeatable after five time-steps. (By symmetry, paths also exists

from $a\sim$ to a and from $d\sim$ to d requiring five time-steps.) These paths represent schedules with iteration latency of five and requiring five control steps for the steady-state repeating kernel. A schedule which favors minimizing iteration latency at the expense of control depth is represented by the path from b to $c\sim$ and by symmetry from $c\sim$ back to b. This steady-state schedule has an average iteration latency of four but requires eight control steps. This path, where $c\sim$ was first reached at $PS.S^3$, is remembered as $c\sim$ is accumulated in $RS_{bc\sim}$. Note that if the best possible average minimum iteration latency schedule is desired, the initial path set must extend to a $PS.S^n$ such that all reachable states are included. In this way, all repeatable paths are represented and the best average combination is guaranteed.

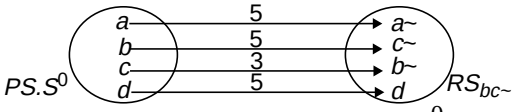


Figure 5.30 Closed paths from $PS.S^0$ to $RS_{bc\sim}$

If backward pruning returns an empty set during closure, it indicates that iteration leg(s) with iteration latency of n were found (i.e. all output operands were produced once) but no compatible iteration legs with iteration latency $\leq n$ exist to sustain repeating execution. A preserved copy of the forward exploration path set is extended by one time-step and backward pruning and closure are attempted again.

As constructed, a **closed path set** of cardinality n is the set of all iteration legs with iteration latency $\leq n$ such that any of these iteration legs may be used as part of a repeating kernel in which all other iteration legs (if required) also belong to this closed path set.

5.3.6 A Witness Schedule

Although a closed path set contains many execution sequences to choose from, it is often desirable to find a path from some state s directly to its dual $s\sim$ as it represents a FSM controller with number of control steps equal to the minimum iteration latency. This is a joint iteration latency/control depth minimization objective since schedules meeting other objectives such as average latency also exist in a **CMA**. This may be done by adding enough additional information (tags) to every state $s \in PS.S^0$ such that the identity of a parent state s may still be determined for children states in $RS_{bc\sim}$. In this way, any child state $s\sim \in RS_{bc\sim}$ with tag encoding equaling state encoding as duals identifies a path from some state s directly to its dual $s\sim$. Unfortunately, adding this much additional tag information at once is costly. Instead, this idea is implemented iteratively. First, a small number of state bits (5-10) for all states $s \in PS.S^0$ are tagged to record their initial value. Next, a closure fixed-point leaves only repeatable paths from parent states in $PS.S^0$ to children states in $RS_{bc\sim}$ in which parent and children states match as duals for the tagged portion of the state vector. These two steps are repeated until all state bits have been tagged. This results in a **directly closed path set**.

Given a **directly closed path set**, an arbitrary state s from it's $PS.S^0$ is picked as a witness. A closure fixed-point is applied with only s used as a new $PS.S^0$ and $s\sim$ as a new $PS.S^n$. This produces a set containing all valid executions from s to $s\sim$. A single witness repeating kernel may be extracted as described in figure 5.7. This single schedule is for steady-state repeating behavior and may be directly translated to FSM control steps.

Loop entry and exit sequences must still be determined as this witness repeating kernel represents only the steady-state execution. A straight-forward way to determine entry/exit sequences is to simply ignore tasks from previous/next

iterations in the witness schedule during loop entry or exit. This produces adequate although not necessarily minimum length entry/exit sequences. Alternatively, minimum length entry/exit sequences may be determined by finding shortest paths from $S_{task\ start}$ and $S_{task\ final}$ to any state in the witness schedule.

5.4 Cyclic Control-Dependent Exploration

Cyclic control-dependent exploration requires no fundamentally new exploration concepts. It does require combining techniques seen in acyclic control-dependent exploration as well as cyclic data-flow exploration. Consequently, cyclic control-dependent exploration is the most complex. This section reiterates the steps for cyclic data-flow exploration but includes ideas necessary for control-dependent behavior. First, behavior cuts and behavior cut state sets are described. This is similar to section 5.3.3 but always uses an operand sense **MA** transition as the cut activity. Next, forward exploration proceeds in similar fashion to acyclic control-dependent forward exploration. After this, a backward pruning step with validation is applied. As with cyclic data-flow exploration, closure of a backward pruned path set is required. Finally, control cases may be prioritized.

Exploration of a RISC processor **CMA** requires cyclic control-dependent exploration. Minimum latency executions sequences for high probability instructions are found. Still, valid executions sequences must exist for all instructions. Furthermore, any current instruction sequence must be able to lead into another execution sequence for any instruction supported by the RISC processor.

5.4.1 Behavior Cut

A cyclic control-dependent **CMA** typically contains several iterates. Each iterate has a sense operand **MA** as described in section 4.4.3. A sense operand has

the attractive traits that it occurs once during each iteration and is never bypassed. This makes a sense operand **MA** an ideal choice for a behavior cut. The sense operand **MA** for one iterate in the composition is arbitrarily picked as the cut **MA**. The transition $(\overline{mso.S_{sense\ known}}, mso.S_{sense\ known})$ where the even sense operand becomes known is used as the cut activity. S_{bc} is determined as shown in equation 5.4.

Consider what sets S_{bc} and $S_{bc\sim}$ look like if a composition contains only a single iterate. Remember that a sense operand **MA** is only allowed to change sense when all internal **MA** have completed or reached their $S_{task\ final}$. Consequently, S_{bc} is equal to the iterate's even sense $S_{task\ start}$ and $S_{bc\sim}$ is equal to the iterate's even sense $S_{task\ final}$. In this case, exploration reduces to acyclic control-dependent exploration as paths from $S_{task\ start}$ to $S_{task\ final}$ determine shortest legs in a repeating kernel. Even though both senses are present in an iterate, no iteration overlap is allowed within the iterate and hence this is equivalent to exploring an acyclic model. When several iterates are included in a composition, which is typically the case, exploration does not reduce to the acyclic case as iteration overlap is allowed among the iterates.

For independent iterates, the possibility exists that some iterate may never execute but stall indefinitely in paths from S_{bc} to $S_{bc\sim}$. This is the case as an independent iterate would have *all* of its states --no matter what the sense-- in both S_{bc} and $S_{bc\sim}$. Backward pruning and closure would still permit idling paths. This problem was addressed in cyclic data-flow exploration with tags. Section 5.3.3 described how **MA** producing externally visible operands were tagged to identify their original state in S_{bc}. Only states where change had occurred were allowed in $S_{bc\sim}$. A similar approach is taken for cyclic control-dependent exploration but may be greatly simplified by an iterate's sense operand **MA**. As complete execution of

an iterate is represented by its sense operand **MA**, only this **MA** need be tagged to guarantee execution of independent iterates.

5.4.2 Forward Exploration

Forward exploration for cyclic control-dependent models is similar to acyclic control-dependent forward exploration as described in sections 5.2.3 and 5.2.7. Instead of finding shortest paths from $S_{task\ start}$ to $S_{task\ final}$, shortest paths from S_{bc} to $S_{bc\sim}$ are found and used to create a path set. In full forward exploration, which is preferred[6], the initial path set contains paths to all reachable states. Note that with cyclic models, this full path set represents two iteration execution sequences for all iterates.

5.4.3 Control-Case Termination and Future Exclusion

The remaining discussion describes how minimum iteration latency paths are found and prioritized in a cyclic control-dependent exploration. This closely follows what was previously described for acyclic control-dependent exploration in sections 5.2.6 through 5.2.9. From a high-level, the procedure requires iterative application of a series of steps for each control-case term in the priority list. These steps are earliest control-case termination, backward pruning with validation and closure.

Earliest control-case termination takes a full path set, *PS*, and finds the earliest time-step set for which states in $S_{bc\sim}$ are reached for a particular control-case term, *cp*. The algorithm described in figure 5.22 is used with cyclic control-dependent models provided $S_{task\ final}$ is replaced with $S_{bc\sim}$. Also, longer latency terminations for *cp* must be prohibited. This restricts the search to the best-possible latency for

6. Full forward exploration is preferred as a path set containing all reachable states allows for control-case prioritization.

cp. As in section 5.2.8, all *PS* time-step sets are kept but all time-step sets $PS.S^j$ where $j > ets$ are intersected with $\overline{cp}$ to exclude any future termination states for *cp*.

5.4.4 Backward Pruning with Validation

Backward pruning for cyclic control-dependent exploration requires validation as well as preservation and accumulation of terminated states. This updated backward pruning is shown in figure 5.31. The salient difference from what was

$i = |\, PS \,| - 1$
$PS.S^i = (PS.S^i \cap S_{bc\sim})$
$RS_{bc\sim} = PS.S^i$
`while(` $i > 0$`){`
    $PS.S^{i-1} = (PS.S^{i-1} \cap ValImg^{-1}(PS.S^i))$
    $i = i-1$
    $Terminated = (PS.S^{i-1} \cap S_{bc\sim})$
    $RS_{bc\sim} = (RS_{bc\sim} \cup Terminated)$
    $PS.S^{i-1} = (PS.S^{i-1} \cap Img^{-1}(PS.S^i))$
    $PS.S^{i-1} = (PS.S^{i-1} \cup Terminated)$
    $i = i-1$
`}`

Figure 5.31 Backward pruning with validation & termination accumulation

previously seen in figure 5.23 is the accumulation set, $RS_{bc\sim}$, for reached termination states. Although there are two operand senses in cyclic control-dependent models, validation still works as before. Whenever a *resolve*-labeled state is reached, regardless of the iteration sense, validation still guarantees that next-state transitions exist for every possible resolution value. In the validation line from figure 5.17, $V^{j+1} = \forall_{m \in M_{multivalue}}((\forall_{(\_, m.X_{value})} V^j_{m.resolve}) + V^j_{\overline{m.resolve}})$,

both senses of resolve are considered together. In other words, the partition $V_{m.resolve}$ includes any state that is labeled *resolve* or *resolve~* for *m*.

5.4.5 Closure and Prioritized Control Cases

As with cyclic data-flow models, cyclic control-dependent models require a closure step. This insures that all legs in a backward pruned path set, *PS*, may be used in repeating kernels with sustainable iteration latency less than or equal to $|PS| - 1$. Unlike cyclic data-flow models, cyclic control-dependent models require that a valid *ensemble* of legs, which cover all control cases, exists. Still, the closure algorithm for cyclic data-flow models as described in section 5.3.5 is directly applicable for cyclic control-dependent models.

If backward pruning or closure fail to find valid solutions, then the earliest termination state for the current *cp* is delayed by one time-step and backward pruning and closure are attempted again. After backward pruning and closure has occurred for all control case terms in the priority list, a closed valid path set results.

5.4.6 Closed Valid Path-Set

The discussion so far has focused on determining a closed valid path set for a cyclic control-dependent **CMA**. It is helpful to discuss what a closed valid path set is. Consider a closed valid path set, *PS*, with cardinality *n*. Let $S_{bc} = PS.S^0$ and $S_{bc\sim} = \text{dual}(S_{bc})$. Let *s* be an arbitrary state selected from $PS.S^0$. There exists a path beginning with *s* that reaches some state $u\sim \in S_{bc\sim}$ within *PS*. This is true by successful closure. This path from *s* to *u~* is a single sequence of states and hence represents one arbitrarily chosen control case. At some state(s) in this path, control operands resolve and deterministic behavior may bifurcate. Let *t* be an arbitrary state in this path where some control operand **MA** is in its local *resolve*-labeled state. For every possible control operand value resolution, there exists a path from *t* that reaches some state $v\sim \in S_{bc\sim}$. This is true by successful validation. As *s* is

arbitrary, a path exists for every state in S_{bc} that eventually reaches $S_{bc\sim}$. Furthermore, as t is arbitrary, every such path has causal branches for every possible control resolution that also eventually reach $S_{bc\sim}$. Finally, as S_{bc} equals $S_{bc\sim}$ as duals and a cyclic **CMA** is symmetric by construction, every such path has an path-end state which is also a path-start state (as duals) of some other path in *PS*. Hence, all paths are infinitely and causally sustainable within a closed valid path set.

A closed valid path set is an interesting collection of execution sequences. Imagine choosing some state $s \in PS.S^0$. It is possible to wander endlessly through paths in *PS*. At each state in this journey, a new next-state out of a set of possible next-states may be arbitrarily chosen. As soon as a state $u\sim \in S_{bc\sim}$ is reached, this state is considered as $u \in S_{bc}=PS.S^0$ and the journey may continue. Each path from S_{bc} to $S_{bc\sim}$ represents a valid single execution of some control case through all iterates in a **CMA**. At every point a control-operand resolves, execution may continue for every possible resolution value.

As with a closed path set for a cyclic data-flow **CMA**, a closed valid path set may contain better average iteration latency solutions. This becomes even more complicated as some iteration leg may be preferred for one control resolution history while another iteration leg may be preferred for another control resolution history. For instance, suppose there is one control point, *true/false* in an iterate and two iterates in a **CMA**. Thus, there are four possible control cases, *true-true*, *true-false*, *false-true* and *false-false*. It may be that after several successive iterations of *true-true* execution, iteration sequence *leg1* is preferred. Now suppose execution shifts to *false-true*. Immediately after this shift, the best iteration sequence may be leg2 yet after *false-true* execution continues for several successive iterations, leg3 may be best. Every possible optimal iteration sequence leg, for every possible control pattern sequence or shift in control pattern sequence is contained in a

closed valid path set. Hence, all optimal *dynamic* finite-state machine controllers are encapsulated. For this reason, determining a witness schedule as done before is not very meaningful. Rather, research is required to synthesize an optimal deterministic dynamic finite-state machine directly from this representation.

5.5 Summary

This chapter described how all minimum latency execution sequences of a **CMA** may be represented as a path set. This serves two purposes. First, it provides a performance metric for the subsystem modeled by a **CMA**. Second, it provides a route to latency-optimal finite state machine synthesis. Determining minimum latency execution sequences is equivalent to finding shortest paths in a **CMA**. A series of algorithmic steps implement variations of Dijkstra's shortest path algorithm. Forward exploration builds an implicit unrolled network view of a **CMA** portion called a path set. Backward pruning restricts the path set to states and transitions only in shortest paths. Witness extraction arbitrarily selects a single deterministic schedule for examination or synthesis. Cyclic models require a fixed-point closure step to guarantee that executions are infinitely sustainable. Control-dependent models require a validation step during backward pruning to insure that only causal speculation occurs. Finally, control cases may be latency optimized as ranked in a priority list.

Chapter 6

Applications

This chapter presents results and discussion for various ABSS applications and is organized in four sections. Data-flow and control-dependent applications drawn from academia and industry are presented in the first two sections. A RISC processor model and results are presented in the last two sections.

ABSS is implemented in Python[118] and uses the Colorado University Decision Diagram package[123][53]. ABSS is freely available on the web[54]. All examples and applications presented in this chapter, except those from industry, are also available on the web[54].

6.1 Data-Flow Applications

In this section, experimental results are reported for several traditionally referenced acyclic and cyclic DFG benchmarks. A case study shows how a designer can use ABSS in a practical setting. Complexity issues are discussed. Results for five large synthetic benchmarks demonstrate scalability. Finally, an industrial example of meaningful scale and complexity illustrates practical application.

6.1.1 EWF Case Study

The elliptic wave filter, EWF, a common cyclic DFG benchmark[43][114], is used as a case study to demonstrate how a designer might interact with ABSS. An EWF composite task requires 26 addition tasks and 8 multiplication tasks. Suppose a designer needs to implement EWF using a particular standard cell and IP-block library. Given the nature of EWF, the designer decides to explore reuse of the IP block shown in figure 6.1. Internally, this IP block contains an optimized 3-stage pipelined floating-point multiplier, a single time-step floating-point ALU, a small coefficient ROM and one multiplexer. The timing of the multiplier's third stage and the ALU is such that they may be chained in one clock cycle. The output of the ROM is hardwired to one input of the multiplier. The multiplexer allows one external input to bypass the multiplier and directly feed the ALU. Depending on the control settings of the bypasses, this IP block may implement three functions: multiply by coefficient, multiply by coefficient and accumulate, and add.

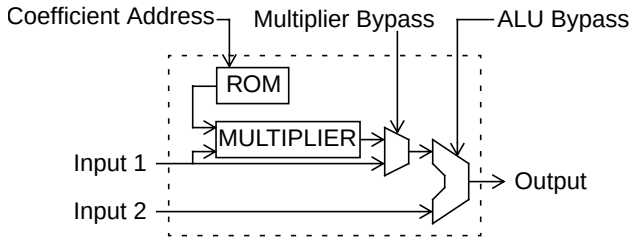


Figure 6.1 IP block for reuse

The designer codes the EWF composition task at an abstract level (< 100 lines) and specifies several appropriate **MA** (again < 100 lines). These **MA** specify the various executions sequences expected when using the IP block in figure 6.1. Table 6.1 summarizes results for this exploration while varying available IP blocks. At this point, the designer has the freedom to explore other IP options and configurations if he wishes. Suppose he decides that a configuration with one IP block and one additional adder provides acceptable performance with a small

resource contingent as the iteration latency, 18, is equivalent to using two IP blocks.

Table 6.1: Constrained IP-block results

IP Blocks	Iteration Latency	CPU Seconds
1	30	2.8
2	18	1.9
3	16	1.7
4	16	1.6

Figure 6.2 shows what type of local storage and interconnect the designer has in mind. A bank of registers stores intermediate results. Any of these registers connects to a function block input or output through a limited number of busses. The single IO port, which feeds bus structure 2, permits communication to and from the function blocks via the register bank.

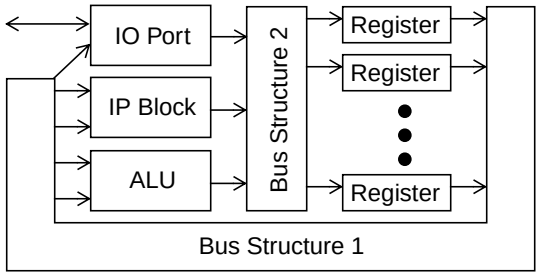


Figure 6.2 Target high-level architecture

After editing the EWF description and model files, (~20 edited lines), the designer now experiments with various register and bus constraints. Several fast iterations of ABSS provide the data shown in table 6.2. Given the existing 1 IP-block and 1 ALU constraints, execution of EWF is impossible with less than 9 registers and no improvement occurs for more than 9. Varying available busses does vary iteration latency. The designer has a trade-off decision and opts to reduce interconnect at the expense of iteration latency by choosing the 3/2 bus solution shown in bold. (Even with only 3 busses, both function blocks may simultaneously begin execution as a single operand may broadcast on one bus to multiple

function-block inputs.) Once the designer decides on a final constraint configuration, ABSS provides an optimal loop-pipelined witness schedule (control sequence) which may be directly synthesized into a FSM. Although the final selected solution has an iteration latency slightly greater than what is commonly reported as optimal for EWF, it incorporates practical and important interconnect, memory and IO-protocol constraints necessary for a realistic design.

Table 6.2: Results with Constrained Registers and Busses

Bus 1	Bus 2	Registers	IO Ports	Iteration Latency	CPU Seconds
-	-	8	1	Impossible	1.4
-	-	9	1	18	2.0
-	-	10	1	18	2.2
-	1	9	1	30	2.6
-	2	9	1	18	2.0
-	3	9	1	18	2.0
2	2	9	1	29	2.4
3	**2**	**9**	**1**	**20**	**2.0**
4	2	9	1	18	2.0
5	2	9	1	18	2.0

6.1.2 EWF Benchmarks

The next few sections summarize results for several academic cyclic benchmarks. All results in these sections were produced on an Intel-donated 866 MHz Xeon PC running Linux. As required computation resources are often a concern with symbolic and exact techniques, required CPU seconds, the imposed ROBDD memory model and peak ROBDD node usage are reported. Reported time and memory use includes all ABSS steps from parsing the scheduling problem, model construction, composition, refinement and exploration, to finally printing a single witness schedule. Peak ROBDD nodes indicates the maximum number of ROBDD nodes (1 node requires 16 bytes) kept by the manager at any point during symbolic scheduling. The constraint configuration column lists

resource bounds such as 1 two time-step pipelined multiplier (1 2-ts piped mult) imposed on the scheduling problem. Finally, when better *average* iteration latency solutions exist in the **CMA**, a lower iteration latency bound is reported in parenthesis.

Table 6.3: Elliptic Wave Filter Results

Benchmark	Iteration Latency	Constraint Configuration	CPU Seconds	Memory Model	Peak ROBDD Nodes
EWF1_IP	20	3 bus1, 2 bus2, 9 reg, 1 IO port, 1 IP block, 1 1-ts alu	2.0	64 MB	251,412
EWF1_a	17	2 1-ts ALU, 1 2-ts piped mult	1.0	64 MB	107,310
EWF1_b	16	3 1-ts ALU, 1 2-ts piped mult	1.0	64 MB	98,112
EWF2	16	3 1-ts ALU, 1 2-ts piped mult	4.2	64 MB	370,986
EWF1x1	9 (>8)	3 1-ts ALU, 2 2-ts piped mult, 1 IO Port	137	64 MB	2,466,086

The first row in table 6.3, EWF1_IP, shows results for a highly constrained single EWF loop using an IP block from the previous case study. The next two rows, EWF1_a and EWF1_b, duplicate previously reported optimal results[114] for a single pipelined EWF. The run times are fast enough that ABSS can be used in an iterative design environment.

As mentioned in chapter 4, allowing only one **MA** per task in a scheduling problem limits solutions by permitting only one instance of any particular operand to exist at any time-step. By unrolling a DFG, this complexity bound may be directly controlled. EWF2 is one unrolling of EWF so that two EWF iterates are in a composition. These iterates are not independent as they contain data dependencies between them. Together, they represent one execution of EWF yet with two **MA** per task so as to potentially overcome any capacity constraint impact. Although CPU run times increase by a factor of three to four, latency does not improve indicating that a single iterate EWF composition is not impacted by capacity constraints.

EWF contains several long inter-iteration data dependencies which prevent much benefit from loop pipelining. (Iteration latencies for optimal pipelined solutions are typically only one time-step improved over non-pipelined solutions.) Still, hardware resources may be under-utilized. By scheduling two independent iterates of EWF under one set of resources, additional resource utilization may be realized. This effectively models two independent EWF streams executing on a single hardware subsystem. The EWF1x1 row presents results for this experiment. A resource set of 3 single time-step adders and 2 two time-step pipelined multipliers is used. This resource set is fairly ideal for a *single* EWF loop as reducing available resources negatively impacts scheduling results but increasing available resources does not improve scheduling results. Furthermore, an IO protocol that limits the system to one external IO transaction per time-step and orders IO transactions between EWF copies is imposed. Although the iteration latency for a single EWF loop with this resource set is 16 time-steps, iteration latency for two parallel copies improves significantly to only 9 time-steps. As there is considerable added freedom (both copies are independent except for IO ordering), ABSS finds a way to make better use of the available hardware resources. A literature search revealed no other scheduling results reported for parallel EWF configurations.

6.1.3 FDCT Benchmarks

Table 6.4 presents results for a fast discrete cosine transform, FDCT [114][134]. The FDCT benchmark is challenging for three reasons. First, it contains two independent loops. For cyclic behavior, this independence leads to a substantial expansion of the solution space as solutions for every resource-compatible permutation of loop *a* with *b* over all time-steps may be represented. FDCT1_a (acyclic) and FDCT1_c (cyclic) differ only in acyclic versus cyclic

modeling and highlight this solution space expansion. Second, FDCT contains no

Table 6.4: Fast Discrete Cosine Transform Results

Benchmark	Iteration Latency	Constraint Configuration	CPU Seconds	Memory Model	Peak ROBDD Nodes
FDCT1_a	19	1 1-ts add, 1 1-ts sub, 1 2-ts piped mult, 4 bus*, 8 reg, partial IO order	3.3	128 MB	409,822
FDCT1_b	2	-	2.7	64 MB	240,170
FDCT1_c	16	1 1-ts add, 1 1-ts sub, 1 2-ts piped mult, 4 bus*, 9 reg, partial IO order	2375	256 MB	9,911,356
FDCT1_d	17 (>15)	1 1-ts add, 1 1-ts sub, 1 2-ts piped mult., 4 bus*, 7 reg, partial IO order	84.7	128 MB	4,813,620
FDCT1_e	13	1 1-ts add, 1 1-ts sub, 2 2-ts piped mult., 8 bus*, 7 reg, partial IO order	300	128 MB	5,001,668
FDCT1_f	20	2 1-ts ALU, 1 1-ts piped mult., 7 bus, 6 reg, partial IO order	33	256 MB	829,864
FDCT1_g	17 (>16)	2 1-ts ALU, 1 1-ts piped mult., 7 bus, 7 reg, partial IO order	78	256 MB	4,992,470
FDCT1_h	17 (>16)	2 1-ts ALU, 1 1-ts piped mult., 7 bus, 7 reg, partial IO order, 2 IO port (unbuffered reads and writes)	65	256 MB	2,970,954
FDCT1_i	17	2 1-ts ALU, 1 1-ts piped mult., 7 bus, 8 reg, partial IO order, IO protocol, 2 IO port (unbuffered reads and writes)	144	256 MB	5,788,608
FDCT1_j	20 (>18)	2 1-ts ALU, 1 1-ts piped mult., 7 bus, 11 reg, partial IO order, IO protocol, 1 IO port (buff. reads, unbuff. writes)	438	256 MB	9,966,544
FDCT1_k	19 (>16)	2 1-ts ALU, 1 1-ts piped mult., 7 bus, 14 reg, strict IO order, 1 IO port (buffered reads, unbuffered writes)	861	256 MB	9,769,298
*Assumes busses can be reconfigured from input to output in the same time-step as historically done[60][84][114].					

inter-iteration dependencies. This freedom permits considerable pipelining but further expands the solution space. Results for FDCT1_b, which contains no resource constraints, exhibits this freedom. Iteration latency is only 2 and is constrained only by the one live operand instance modeling state bound. (Since no resource constraints or inter-iteration dependencies exist, two copies of a final witness schedule may be directly translated to a single FSM with iteration latency

of 1.) Finally, FDCT is highly symmetric. Each path through the DFG is similar in length and operation sequence to every other path. This too enlarges the solution space and hence instance representation cost. Due to the high symmetry of this problem, a partial IO ordering is imposed to eliminate representation of structurally symmetric solutions.

FDCT1_b illustrates a general scheduling complexity concept. Although this result is for a challenging benchmark, very limited computational resources are required. On the other hand, the same benchmark scheduled with resource constraints, FDCT1_c, requires considerably more computational resources. This is expected as a scheduling problem with no resource contention such as FDCT1_b reduces to topological ordering. Hence, in the absence of resource contention, a straight-forward as-soon-as-possible list scheduler will always find optimal solutions and require no search. Even so, symbolic scheduling of FDCT1_b still requires some computational resources as *all* schedules, even those observing resource constraints, are encapsulated. In general, contention for resources makes scheduling hard. For ABSS, the most challenging cases occur when resource constraints tend to balance dependency constraints. When resource constraints either dominate or do not exist, ABSS is facile.

As FDCT1_c exhibits some of the greatest complexity, it is used as an example for a discussion on representation complexity and growth. Chapters 3 and 4 described how a **CMA** is constructed. This includes all **MA** construction as well as a **CMA's** composition, dependency/capacity constraint pruning, viability pruning, and resource constraint refinements. Surprisingly, all this accounts for an insignificant use of computational resources. For FDCT1_c, only 4.9 seconds and 1,145,662 ROBDD nodes are needed to create a **CMA** with 553,670 ROBDD nodes in the transition relation, Δ. The largest growth occurs during resource constraint refinement when Δ grows from 58,768 to 553,670 nodes. Note that at

this point, Δ contains all valid resource constrained executions of the scheduling problem. After 123.6 seconds of ROBDD sift reordering, the size of Δ is reduced to 341,338 nodes. Most of the representation growth occurs during the exploration steps described in chapter 5. For FDCT1_c, finding a path set requires 200 seconds while total ROBDD node usage peaks to 9,911,356. The largest ROBDD in the

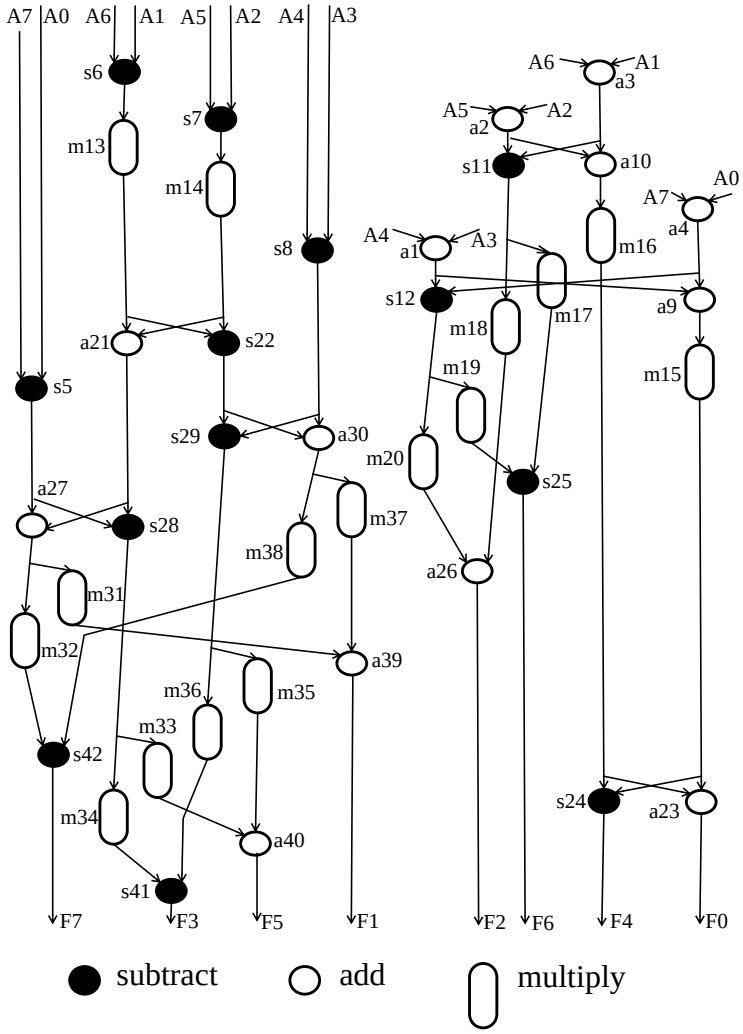


Figure 6.3 FDCT data-flow graph

path set is 753,831 nodes. Determining a closed repeating path set is not as costly in terms of node usage but does require time. This fixed-point requires 476 seconds. The largest ROBDD in this set is only 36,109 nodes and peak node usage

remains at 9,911,356 as no garbage collection is performed. Finally, finding all schedules with control steps equal to iteration latency requires 1570.5 seconds. The largest ROBDD in this set is 4,439 nodes and peak node usage does not increase. As a reward for this hard work, ABSS finds a 16 time-step solution for FDCT1_c, bettering the best previously reported result [134] of 17 time-steps for this benchmark with the same arithmetic resources.

Figure 6.3 shows the FDCT data-flow graph while table 6.5 shows a 16 time-step iteration latency solution for FDCT1_c. Each time-step lists the tasks that begin during that time-step as well as the result operands in local storage and available at the *beginning* of that time-step. For example, task m36 begins in time-

Table 6.5: Witness Schedule for FDCT1_c

Time-step	Task	In Storage
1	m31 a30 s29	m16 s11 m19 a27 s8 s28 s22
2	a23 s24 m37	m15 m16 s11 m19 a27 a30 s29 s28
3	m17 a4~ s5~	s11 m19 a27 m31 a30 s29 s28
4	a39 m36	m19 a4~ s5~ a27 m31 m37 a30 s29 s28
5	s25 m34	m17 m19 a4~ s5~ a27 a30 s29 s28
6	a3~ m32 s6~	a4~ s5~ a27 a30 s29 m36 s28
7	m38 s41	a3~ a4~ s5~ a30 s29 m36 m34 s28 s6~
8	a2~ m35 s7~	a3~ a4~ s5~ m32 s29 s28 s6~
9	a10~ s11~ m14~	a2~ a3~ a4~ s5~ m32 m38 s28 s7~ s6~
10	s42 m13~	a10~ s11~ a4~ s5~ m32 m38 m35 s28 s6~
11	a1~ s8~ m33	a10~ s11~ a4~ s5~ m35 s28 m14~
12	a9~ m16~ s12~	a1~ a10~ s11~ a4~ s5~ s8~ m35 m13~ m14~
13	m20~ a40	a9~ s11~ s12~ s5~ s8~ m35 m33 m13~ m14~
14	m18~ a21~ s22~	a9~ m16~ s11~ s12~ s5~ s8~ m13~ m14~
15	m19~ a27~ s28~	a9~ m16~ s11~ m20~ s12~ s5~ s8~ a21~ s22~
16	m15~ a26~	a9~ m16~ s11~ m18~ m20~ a27~ s8~ s28~ s22~

step 4. Since all multipliers are two time-step pipelined, it produces a result at the end of time-step 5. This is latched into local storage and ready to be used at the beginning of time-step 6. Tasks and result operands are shown in both the even and odd, '~', iteration senses. Iteration sense toggles when looping back from time-

step 16 to time-step 1. For instance, the result of s11~ is in storage at time-step 16 but appears as s11 in time-step 1. Although iteration latency is 16 time-steps, delay for one complete FDCT execution is 27 time-steps. This is the delay from the first odd tasks, a4~ and s5~ in time-step 3, through flipping sense when looping from time-step 16 to time-step 1, to the final now even task, a40, in time-step 13+16.

Result rows FDCT1_f through FDCT1_k from table 6.4 show more practical and useful configurations of FDCT than are typically reported. Two ALU resources rather than separate add and subtract resources are used. Also, one set of busses, where each bus is occupied in one way for the duration of a clock cycle, is used. FDCT1_g shows results for a minimal register contingent. FDCT1_h adds a two bidirectional IO port constraint. Reads and writes through this IO port are assumed unbuffered. FDCT1_i adds the protocol constraint described in figure 3.15. Even with this forced alternation between reads and writes, schedules with excellent iteration latencies are found. FDCT1_j restricts IO ports to one. Now, IO reads must be buffered and are modeled with several additional operand **MA**. All optimal solutions are readily found even with this tight set of constraints. FDCT1_k forces a strict ordering of input and output operands communicated though one IO port. This models what might happen in a real design where samples can not be reordered but must be accepted and produced in sequence. Although more registers are required, iteration latencies similar to loosely constrained configurations are still achieved. As these examples demonstrate, ABSS handles real-world design constraints, produces optimal results yet requires acceptable computational resources.

6.1.4 Miscellaneous Academic Benchmark Results

Table 6.6 presents miscellaneous academic benchmark results. These are smaller cyclic DFGs reported in the literature[25][134]. As would be expected,

their smaller size requires less computational resources for scheduling. These benchmarks, as well as all other benchmarks in section 6.1 with the exception of fdct1_a and the upcoming industrial example, are cyclic. ABSS has been applied to acyclic versions of EWF and FDCT[50]. These simpler acyclic solutions are a subset to what is presented here.

Table 6.6: Miscellaneous Academic Benchmark Results

Benchmark	Iteration Latency	Constraint Configuration	CPU Seconds	Memory Model	Peak ROBDD Nodes
DIFFEQ1	6	1 1-ts add, 1 2-ts piped mult, 4 reg	0.2	64 MB	3,066
FIR16P1_a	8 (>6)	2 1-ts add, 1 2-ts piped mult, 3 reg	0.8	64 MB	73,584
FIR16P1_b	15 (>14)	2 1-ts add, 1 2-ts piped mult, 2 reg	0.7	64 MB	29,638

6.1.5 Comparison to Other Work

Table 6.7 compares ABSS, with existing work. SST[114] is a symbolic

Table 6.7: Academic Benchmark Comparisons

	EWF[a]	EWF[b]	FDCT[c]	FDCT[d]
ABSS	17 (9r)	16 (10r)	16 (9r)	13 (7r)
SST[114]	17	16	19[e] (9r)	14[e] (11r)
Theda.Fold[60]	17	16	17	13
RS[25]	17	16	NA	NA
TCLP[120]	17 (10r)	16 (10r)	16[f] (12r)	NA
MARS[134]	17	16	17	13

a. 3 1-ts ALU, 1 2-ts piped mult
b. 3 1-ts ALU, 1 2-ts piped mult
c. 1 1-ts add, 1 1-ts sub, 1 2-ts piped mult, 4 bus
d. 1 1-ts add, 1 1-ts sub, 2 2-ts piped mult, 8 bus
e. Not loop pipelined
f. Assumes 1 1-ts mult

scheduler while the others are heuristic. Where possible, register use is reported in parenthesis. As can be seen, existing techniques produce optimal or near optimal results for these well-studied examples. For the heuristics, required computational

time is typically a few seconds. For ease of comparison, only limited additional constraints are applied. In fact, only TCLP bounds register usage and only SST and Theda.Fold bound bus usage.

6.1.6 Synthetic Benchmarks

Five larger synthetic benchmarks demonstrate the scalability of ABSS for cyclic data-flow problems. A set of guidelines generated realistically shaped synthetic benchmarks. These guidelines ensure that the synthesized DFG is fairly connected, contains several inter-iteration dependencies, yet is reasonably random. All synthetic benchmarks contain 100 tasks assigned to one of two resource classes. Resource class A consists of 2 single time-step units and resource class B consists of 1 two time-step pipelined unit. Each unit accepts two input operands and produces one output operand as would be the case for an ALU and pipelined multiplier. A large number of synthetic benchmarks were produced and from this pool, five finalists meeting the following two criteria were selected. First, a minimum iteration latency schedule requires loop winding. (i.e. a non-pipelined schedule cannot be an optimum throughput schedule.) Second, both dependencies and resource bounds must impact schedule solutions. Resource bounds are not made meaningless by tight dependency constraints and vice versa. Table 6.8 shows results for the five final synthetic benchmarks. A 128 MB memory model is used for all cases and computation times range from 15 to 154 seconds. These

Table 6.8: Synthetic benchmark results

Benchmark	Iteration Latency	Res. A	Res. B	CPU Seconds	Peak Nodes
216	43	79	21	15.6	1,101,716
229	37 (>35)	73	27	154.3	4,806,466
278	44 (>43)	72	28	41.7	3,859,072
282	40	74	26	14.2	1,670,970
288	37	67	33	26.6	3,244,850

benchmarks, all academic benchmarks, as well as the ABSS source code are available on the web[54] for comparison with other scheduling techniques.

6.1.7 An Industrial Example

ABSS was applied to a substantial industrial example. The assembly code for computing $f=x^y$ on Intel's new Itanium architecture was statically scheduled. This 127 task acyclic data-flow example is interesting because of the real-world peculiarities of the Itanium VLIW architecture. The Itanium contains 6 pipelined processors (m0, m1, i0, i1, f0 and f1), each of which may compute some subset of Itanium instructions. Depending on the type and complexity of the instruction, some instructions may only be assigned to one of processors m0, m1, i0 or i1, some may only be assigned to f0 or f1, some may only be assigned to i0 or i1 and so on. In the worst case, some instructions must be assigned to two processors, i0 and f0, concurrently.

A hierarchical resource bound (section 4.5.2) was constructed to enforce these complex constraints. Figure 6.4 illustrates the six hierarchical resource bounds applied to model these constraints. They are hierarchical as an instruction which must be assigned i0 and f0 concurrently must belong to five resource constraint groupings: ((i0), (f0), (i0, i1), (f0, f1), (m0, m1, i0, i1)). On the other hand, an instruction which may be assigned to i0, i1, m0 or m1 only need belong to one resource constraint grouping: (m0, m1, i0, i1). Through use of these six hierarchical constraints, all Itanium resource bounds were correctly modeled.

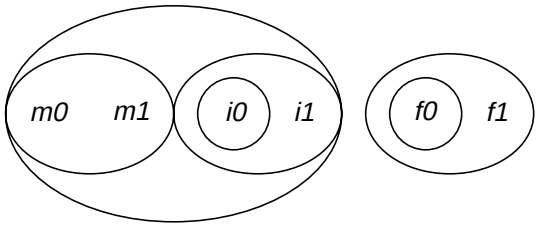


Figure 6.4 Hierarchical resource bounds

For this Itanium example, it was also necessary to model a communication penalty. This penalty varied depending on *which* child processor consumes a result operand. For example, processor *a* may compute a result which may be communicated to processor *b* after a delay of ≥5 but may only be communicated to processor *c* after a delay of ≥9. This type of sequential constraint was naturally modeled during specification of all **MA**. For instance, figure 6.5 shows an **MA** that requires one time-step to reach states labeled *early known* yet requires two time-steps to reach states labeled *late known*. Depending on the required communication delay, a child's input would be enabled by either *early known* or *late known* labeled states. This approach to modeling both computation of a result and communication delays within one **MA** was generalized to pipelined delays of up to 9 time-steps.

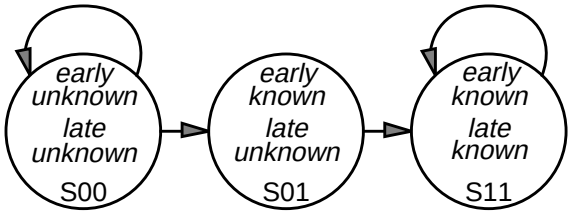


Figure 6.5 Acyclic MA with communication delay

An ABSS speed-up technique, **over-estimation**, was devised and used with this example. As described, ABSS creates a path set from a completely constrained transition relation, Δ. Forward exploration to create this path set is often the most computationally expensive step in ABSS --especially with certain resource concurrency constraints applied. In over-estimation, ABSS creates a path set with a *partially* constrained Δ. Some or all resource concurrency constraints are left out and hence an over-estimation of the true solution space results. Once backward pruning has reduced the size of the path set, a completely constrained Δ is used to determine if true solutions actually exist. Although over-estimation requires iterative refinements of a path set, it avoids costly completely-constrained forward

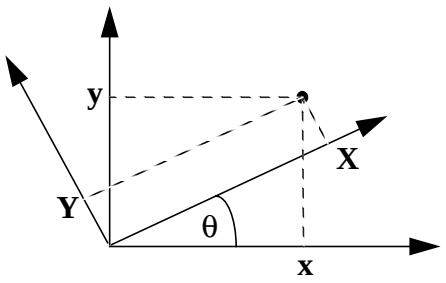
exploration. After all is said and done, all schedules plus an optimal witness schedule with latency of 81 time-steps required 131 seconds and 9,910,334 nodes (256 MB memory model) for computation.

6.2 Control-Dependent Applications

This section presents several academic control-dependent examples as well as an industrial control-dependent application. First, two academic examples illustrate iterate use for pipelining, control prioritization, average latencies and local registers constraints. Finally, an acyclic industrial control-dependent example is discussed. A new partitioning strategy, *time-zone partitioning*, reduces computational requirements for this industrial example.

6.2.1 ROTOR Benchmark

ROTOR, as shown in figure 6.6, was introduced by Radivojević [114] ROTOR performs a rotation of coordinate axes by angle θ and is used in applications ranging from graphics processing to positional control systems. As seen in figure 6.6, ROTOR requires computation of trigonometric functions. High-performance systems often precompute these values and store them in a look-up table. As a compromise between numerical accuracy and storage requirements, values for only one quadrant are stored. It is possible to compute trigonometric values for all quadrants based on a single quadrant table. ROTOR assumes that just a single look-up table is available for sine values in the first quadrant. Consequently, four control cases, covering correct look-up table modifications for each quadrant, occur in ROTOR.



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

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

```
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 6.6 ROTOR example

If only a single ROTOR iterate is used in a composition, then ABSS reduces to finding acyclic (non-overlapping) ROTOR execution sequences. Such results are directly comparable to SST[114] and are shown in table 6.9. Four sequential behaviors are defined for four classes of tasks: ALU (addition, subtraction, negation), table look-up, multiplication, and compare. Consequently, a ROTOR

composition task contains 28 internal tasks. The schedule latencies are equivalent to, yet the computation times are 5-10 times improved[1] over those for SST[114]. For comparison purposes, no control-case prioritization is applied in these results and consequently only the worst control case is optimized.

Table 6.9: Acyclic ROTOR Results

Function Units				Schedule Latency	CPU Seconds
ALU[a]	Multiply[b]	Table[c]	Compare[d]		
1	-	1	1	12	1.0
2	-	1	1	7	1.0
1	2	1	1	10	1.1
2	2	1	1	8	1.1

a. ALU is single time-step
b. Multiply is two time-step pipelined. '-' implies ALU does multiplication.
c. Table is single time-step
d. Compare is single time-step

Two ROTOR iterates may be included in a composition to explore pipelined solutions. Table 6.10 shows results for this depth of pipelining. With two iterates, there are 16 care control cases as both iterates may independently process a θ in one of four possible quadrants. A priority list with 21 control-case terms was used

Table 6.10: Results for Cyclic ROTOR with Two Iterates

Function Units[a]				Average Iteration Latency	CPU Seconds
ALU	Multiply	Table	Compare		
2	1	1	1	4.8	15
2	2	1	1	4.4	113
2	3	1	1	4.4	117
3	2	1	1	3.95	9.35
Unlimited	Unlimited	Unlimited	Unlimited	3.4	5

a. Function unit sequential behavior as in table 6.9.

to produce the reported average iteration latencies. First, all control cases were prioritized together so as to optimize the worst control case. Then, four groupings

1. Comparison is adjusted for differences in computation resources.

prioritized the quadrant combinations with the most tasks (combinations of quadrants 4 and 3) down to quadrant combinations with the fewest tasks (combinations of quadrants 1 and 2). The last 16 control case terms prioritized individually all possible control cases starting with the most task intensive down to the least task intensive.

The results in table 6.10 show that ROTOR benefits from pipelining. The row 2 result (113 seconds, 4.4 average iteration latency) has the exact resource constraints as the row 4 result from table 6.9 (8 time-step latency). Although the table 6.9 result is for the worst control case, the sustained pipelined iteration latency for this worst control case is 4.5 time-steps. Pipelining almost doubles the performance. This worst control case, quadrant 4 followed by quadrant 4, truly does have a non-integer average latency of 4.5 time-steps. ABSS finds a control

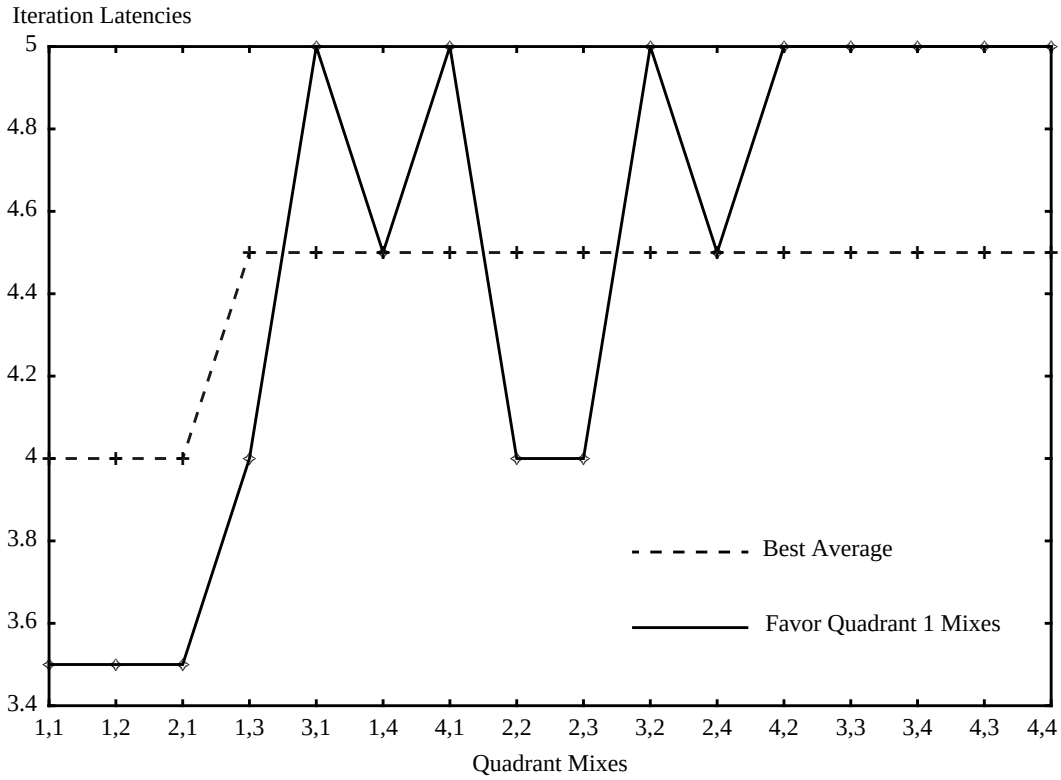


Figure 6.7 Results from two contrasting priority lists

sequence that adjusts even when two identical computations are in the pipe such that the best average is found. In this specific case, a quadrant 4 calculation taking 4 time-steps may only be followed by another quadrant 4 calculation requiring 5 time-steps.

The control-case priority list used in table 6.10 was carefully crafted to average iteration latency equally among all control cases. The driving assumption was that θ may lie in any quadrant with equal probability. This is not the case in all control-dependent behavior as there are often control cases which occur with considerably higher probability than others. It is possible to reverse the control-case priority list used in table 6.10 so that quadrants with the smallest number of tasks are most favored. This optimizes an expected ROTOR use where θ almost always lies in the first quadrant. Figure 6.7 contrasts iteration latencies for all 16 possible quadrant mixes given these two different priority lists. As can be seen, the reversed priority list heavily favors quadrant 1 mixes. ABSS is capable of favoring particular control case(s) if directed by simulation data or design intent.

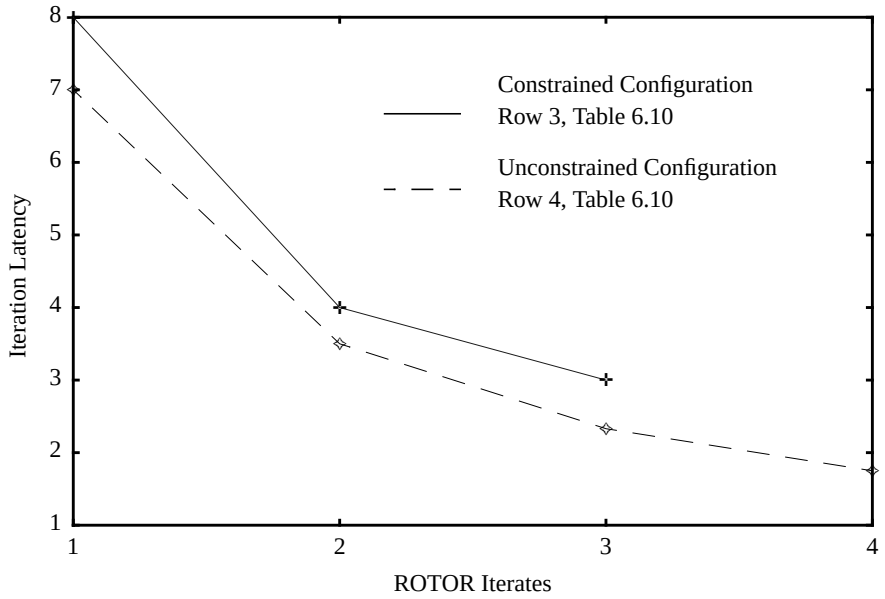


Figure 6.8 Iteration latency versus ROTOR iterates

Row 3 and 4 configurations from table 6.10 were extended to three ROTOR iterates in one composition. Figure 6.8 plots the iteration latency improvement achieved by doing this. The required computation resources for the constrained three iterate solution was 7 minutes on a 733 MHz Pentium III with a 400 MB memory model. Four iterates were attempted for the constrained configuration but computation exceeded an imposed two hour limit.

Complexity dramatically increases with additional *independent* iterates. The only dependencies and capacities imposed between ROTOR iterates is an ordering of operand sense **MA**. This allows all solutions, some potentially absurd, to be represented. Imagine a solution where iterate 1 begins and executes just one task and then nondeterministically stalls. Iterates 2 and eventually higher may begin, and even finish, as all prior iterates have begun. Hence, every dependency-allowed meshings of iterates is represented at often considerable cost. This may be avoided by adding additional constraints. IO protocols and orderings as well as local storage bounds are real constraints that may help prune the solution space. IO protocols and orderings eliminate some of the impractical meshings by defining more precisely when events in various iterates may take place. Local storage constraints group communicating tasks so that other impractical meshings, which typically require some operand to remain in local storage for an extended time, are pruned. Finally, artificial dependencies or groupings may be imposed to avoid impractical meshings. Time-zone partitioning, presented in section 6.2.3, is an example of this approach.

6.2.2 S2R Benchmark

S2R, as shown in figure 6.9, was also introduced by Radivojević [114]. S2R translates spherical coordinates [R, Θ, Φ] into the Cartesian (rectangular) coordinate values [X, Y, Z]. Both cosine and sine must be computed for Θ and Φ.

As with ROTOR, a single look-up table with trigonometric values for only the first quadrant is available. Consequently, four control-dependent behaviors are required to compute trigonometric values for both Θ and Φ in all four quadrants (2 parallel ROTORs). An S2R composition task contains 48 internal tasks, 16 care control cases, and requires similar resources as ROTOR.

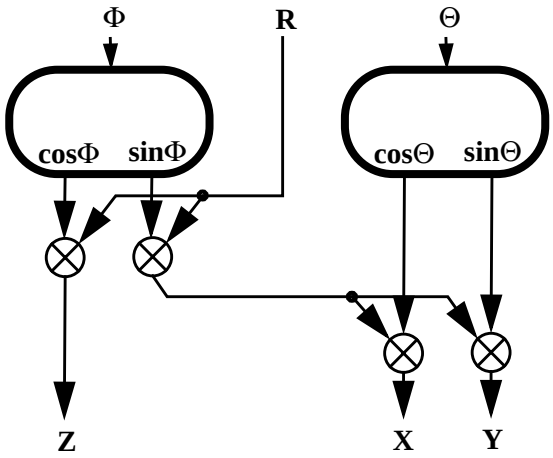


Figure 6.9 S2R example

Table 6.11 presents results for a single iterate of S2R. Of particular interest is the effect of local register bounds[2]. Both rows 1 and 3 achieve the same schedule latency yet row 3 constrains local storage to 4 registers. In fact, row 3 takes less computation resources even though it is more constrained than row 1. In some cases, register constraints tend to temporally group communicating tasks. This serves to beneficially prune the solution space. This is not always the case as seen when comparing rows 5 and 7. In fact, if a register constraint is slightly over what is minimally necessary, required computation resources often grow dramatically. Imagine now that there is always one (or more) extra registers at each time-step. ABSS searches all solutions and hence propagates all possible though non-

2. In this particular configuration, local register bounds apply to any internal operand storage. External input and output operands are unbuffered and must be either supplied on demand or stored separately.

productive uses of extra registers. Consequently, stringent register constraints are applied first and then iteratively relaxed. Alternatively, a register constraint hierarchy (section 4.5.2) may help reduce computation resource requirements.

Table 6.11: Acyclic S2R Results

Function Units				Registers	Schedule Latency	CPU Seconds
ALU[a]	Multiply[b]	Table[c]	Compare[d]			
2	1	1	1	-	10	7
2	1	1	1	3	11	5
2	1	1	1	4	10	7
2	1	1	1	5	10	12
3	2	1	1	-	9	5
3	2	1	1	3	10	6
3	2	1	1	4	9	7
3	2	1	1	5	9	11
3	2	1	-	-	8	4.3
-	-	-	-	-	8	2

a. ALU is single time-step
b. Multiply is two time-step pipelined. ‘-’ implies ALU does multiplication.
c. Table is single time-step
d. Compare is single time-step

A literature search identified only one heuristic[121] and SST[114] as the only other scheduling techniques to report results for acyclic instances of ROTOR and S2R. SST produces exact results for acyclic ROTOR and S2R instances. The heuristic[121] can achieve exact acyclic results although it is unclear what computation resources are required. An average search time plus a density of optimal solutions is reported. Consequently, estimated times for finding, but not guaranteeing, optimal solutions range from 1 to 20 seconds. ABSS determines all schedules, including minimum latency for individual control cases, in 1 to 12 seconds. Guaranteed exact minimum latency schedules are provided in comparable time to a well-built heuristic.

Table 6.12: Cyclic S2R Results

Function Units[a]				Registers	Worst Iteration Latency	CPU Seconds
ALU	Multiply	Table	Compare			
3	2	1	1	3	5	330
3	2	1	1	4	5	414
3	2	1	1	5	5	716
3	2	1	1	-	5	822

a. Function unit sequential behavior as in table 6.11.

Table 6.12 presents results for two S2R iterates. As shown, it is possible to achieve worst control-case iteration latency of five time-steps. This is almost double the performance achieved in row 7 of table 6.11. Interestingly, fewer registers are needed when pipelining S2R. The five time-step iteration latency is found even with only three local storage registers. No improvement occurs with additional registers. In fact, only computational complexity increases with additional register freedom. A configuration with two registers was attempted but failed to find any valid solutions.

6.2.3 Industrial Example

ABSS was applied to an industrial example with control-dependent behavior. This acyclic example, abstractly shown in figure 6.10, is a specialized graphics processing task containing 132 internal tasks and 6 care control cases. Furthermore, the behavior is such that it may be partitioned into 3 distinct blocks, where each block contains two control-dependent behaviors. Consequently, depending on control value resolution, either the first block only, the first and second block, or all blocks are executed. Finally, an IO protocol defines when external input and output operand transactions occur.

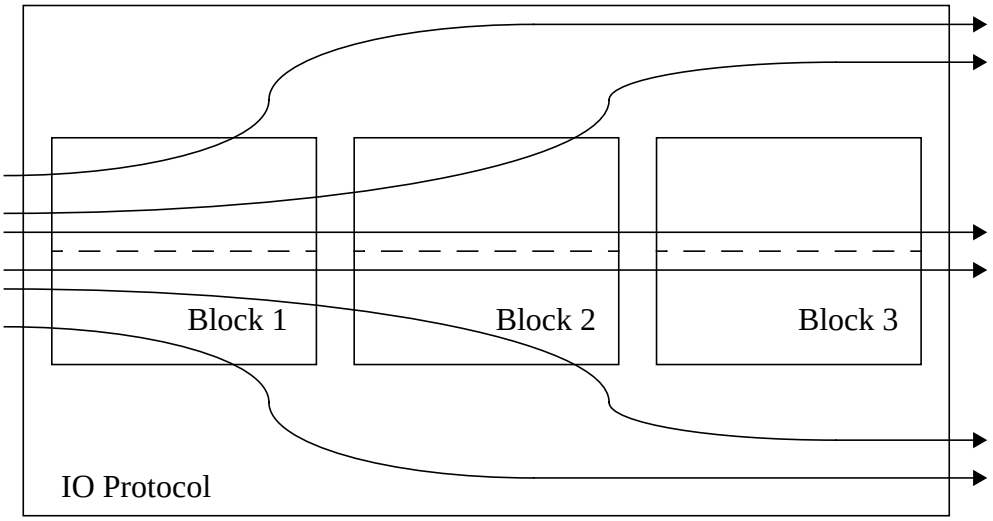


Figure 6.10 High-level behavior of industrial control-dependent example

Table 6.13 presents results for the control-dependent industrial example. These results were produced on a 400 MHz PII Linux machine with 512 MB of memory. Latencies are reported for all 6 care control cases. Both register file ports and local storage constraints are included since the behavior accesses a large number of

Table 6.13: Control-Dependent Industrial Example Results

	Configuration 1	Configuration 2	Configuration 3
ALU Units	1 1-ts	1 2-ts piped	1 3-ts piped
Multiply Units	2 1-ts	2 3-ts piped	1 3-ts piped
XOR Units	1 1-ts	1 1-ts	1 1-ts
MAC Units	-	-	1 4-ts piped
Local Registers	9	4 Single, 3 Double	5
Register File Ports	3 Read, 1 Write	2 Read, 1 Write	3 Read, 1 Write
IO Protocol	Yes	Yes	Yes
Interconnect Guide	-	Yes	-
Schedule Latencies	8, 11, 19, 27, 27, 37	12, 17, 26, 34, 36, 43	12, 18, 35, 36, 44, 46
CPU Seconds	29	865	11

coefficients stored in a register file as well as maintains a small number of intermediate results in local registers.

Configurations 2 and 3 in table 6.13 employ **time-zone partitioning** to reduce computational complexity. Without time-zone partitioning, tasks from the three behavior blocks have ample freedom to be scheduled over a wide range of time-steps as illustrated in the top of figure 6.11. This freedom causes excessive

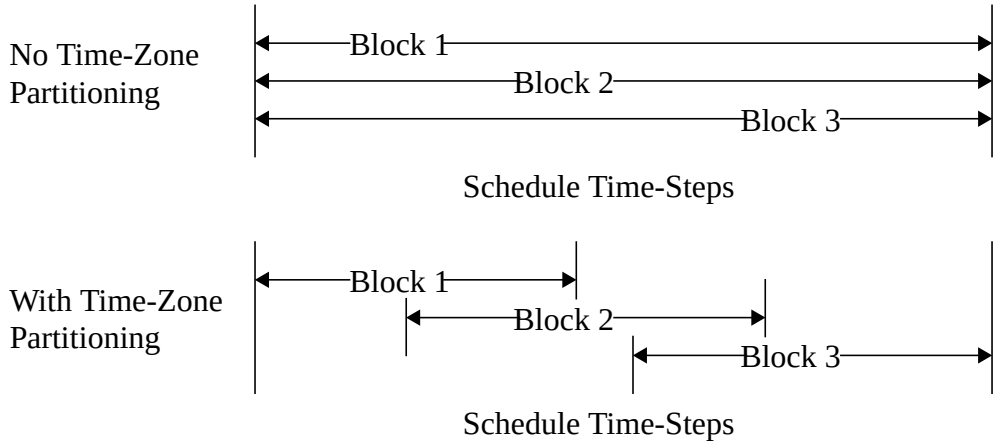


Figure 6.11 Time-zone partitioning

representation growth during exploration. With time-zone partitioning, tasks from the three behavior blocks may be scheduled only in one of three time-step ranges. These zones, which may overlap, allow a designer to restrict the scheduling freedom according to high-level design intent. This reduces representation growth during exploration at the expense of not guaranteeing optimal solutions.

Configuration 2 employs interconnect guides to produce bindable schedules which require less interconnect. Figure 6.12 shows where interconnect is required in this example. With a single pool of local registers, a result operand from any of the three function units may be stored in any register. Consequently, input operands for any of the three function units may exist in any register. Hence, full connectivity in interconnect blocks A and B is assumed. On the other hand,

suppose that results produced by the ALU and accepted by the multiplier may only be stored in a restricted subset of registers. Furthermore, suppose that analysis of communications in the original composite task allows all such communications to be 'routed' through a subset of registers[3]. If this is done, it is possible to exclude some connectivity in interconnect blocks A and B. These types of register concurrency constraints are imposed in a hierarchical fashion as described in section 4.5.2. Along with bus concurrency constraints, interconnect guides halve the upper bound for multiplexing requirements in this industrial example.

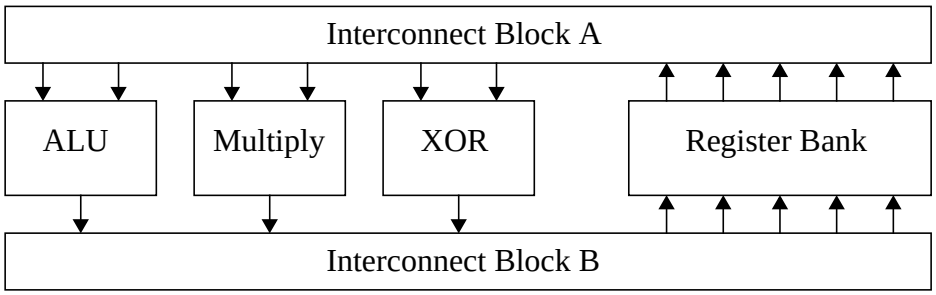


Figure 6.12 Abstract view of interconnect in industrial example

Configuration 3 replaces numerous individual multiply and add tasks with a MAC IP-block aggregate task. A use of this MAC is modeled by an aggregate **MA** as described in section 3.1.4. This reduces the task count to 58 Multiply, 7 ALU, 4 XOR and 21 MAC for a total of 80 tasks in the composition. By using a MAC with a constrained set of sequential behaviors, a more restricted set of datapath uses and hence interconnect is assumed. Finally, less state is needed to model the composition and consequently computation requirements are significantly reduced.

3. The number of registers in the register bank may be increased to make such subsets disjoint.

6.3 A RISC Processor Model

A RISC processor model, implementing the SimpleScalar[20] instruction set architecture, ISA, was constructed. The SimpleScalar ISA is a *superset* of the MIPS IV ISA. The primary reason for choosing this ISA is that the open-source SimpleScalar tool suite provides easily modifiable simulation and tracing tools. With this simulation capability, benchmarks from the MediaBench[70] suite of representative embedded applications may be profiled. These profiles are used in both prioritization and evaluation of the resulting schedules. This is critical since no processor can guarantee high throughput for every control case when contention for resources exists. Finally, the floating point subsystem is not modeled. However, all other instructions, including all control, integer and load/ store instructions are modeled.

6.3.1 An Instruction Task

RISC processor behavior is modeled at two levels of abstraction. At the higher level of abstraction, tasks represent execution of one entire instruction and are called *instruction* tasks. Figure 6.13 shows an instruction task graphically. Given

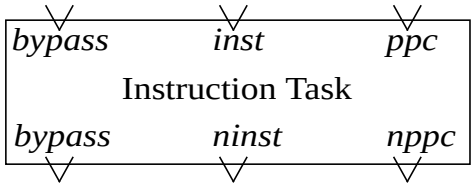


Figure 6.13 Abstracted instruction task

an instruction operand, *inst*, an instruction task sequences through some subset of behaviors depending on the decoded instruction. Internal to an instruction task are memory and register access tasks which are often sequentially constrained. Each instruction task represents a single, complete instruction execution. An instruction task computes the next preincremented program counter value *nppc*, as well as prefetching the next instruction, *ninst*, at the correct next program counter *ppc*

address. Both *nppc* and *ninst* are heavily control-dependent on the currently decoded instruction and on the processor state. Finally, to avoid dynamic data hazards between values that have yet to be updated and future instructions that will read the same values, a *bypass* operand is provided. Bypass allows a single operand to be forwarded to the next instruction as well as being written to the register file. Bypass can also cause processor stall behavior in cases of multiple data hazards.

This particular abstraction lends itself to ordered instruction fetch and ordered commit. Note that more sophistication may be added to instruction tasks. For instance, additional input operands, representing conditional register file commitment, may be added. These would prove useful for enabling limited out-of-order execution. Also, instruction prefetch could be separated from the instruction task, and a separate instruction fetch task, providing a model of out-of-order instruction fetch, would provide instruction operands to instruction tasks.

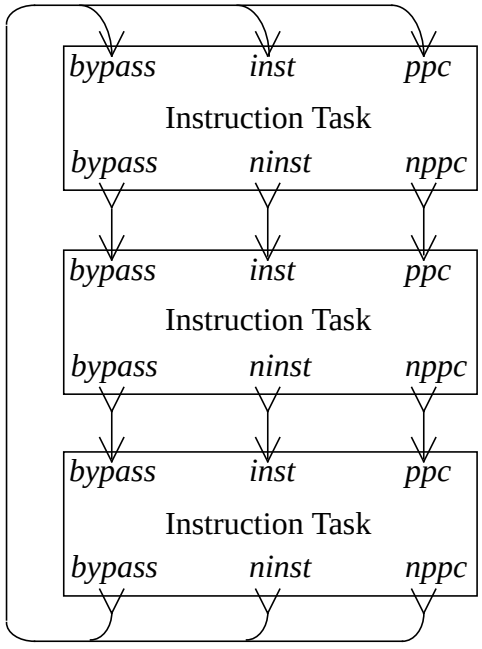


Figure 6.14 A processor composition of three instruction iterates

6.3.2 A Processor Composition

Each instruction task is atomic. Several instruction task iterates are composed to represent the behavior of several simultaneously executing instructions. For example, figure 6.14 composes three instruction task instances to represent behavior of three instructions. Resource concurrency constraints as well as other sequential constraints are applied globally in a processor composition. Thus, the resulting **CMA** describes all execution sequences for three 'in-flight' instructions on a target hardware. Clearly, if higher performance is desired, a composition consisting of *n* instruction iterates may be built with the trade-off of greater storage, resource, and control complexity. Finally, this technique models behavior independently of any anticipated structural hardware pipeline. The **CMA** exploration step reveals an appropriate hardware implementation --which very well may resemble a structural hardware pipeline.

6.3.3 Modeling an Instruction Task

Table 6.14: Low-Level Tasks

INTUS((op1, op2, cop), (op))	Simple integer computation
INTUC((op1, op2, cop), (ophi, oplow))	Complex integer computation
RFRD((regi), (op))	Integer register file read
RFWR((regi, op), ())	Integer register file write
MEMRD((addr), (op))	Memory read
MEMWR((addr, op), ())	Memory write
DMEMRD((addr), (op1, op2))	Double word memory read
DMEMWR((addr, op1, op2), ())	Double word memory write
RDHL((cop), (op))	Read architecture hi or lo registers
WRHL((op, cop), ())	Write architecture hi or lo registers
DECODE((inst), (class, subclass, cop))	Instruction decode

An instruction task is composed of smaller low-level tasks such as register file access, memory access, integer computations, instruction decoding and other activities. The complete set of low-level tasks used in the model is shown in

table 6.14. Small **MA** specify the target sequential behaviors of each of these low-level tasks. A low-level task's input operands are enclosed in the first set or parenthesis while its output operands are enclosed in the second set.

Table 6.14 and upcoming descriptions use generic operand names such as *op1*, *op2* and *addr*. These are name place holders and should not be confused with physical registers or global composition operand names. The high-level intent of an operand is revealed by its generic name. Table 6.15 summarizes the generic operand names used in an instruction task.

Table 6.15: Generic Operand Names used in an Instruction Task

op, op1, op2, ophi, oplow	Generic integer operands
addr	Generic address operand
regi	Generic register index operand
pc, ppc, npc, nppc	Instruction address operands
inst, ninst	Instruction operands
rs, rt, rd	Register index portions of inst (source, transfer, destination)
imm	Some immediate portion of the instruction
cop	A control operand produced by the decode task
bop	A branch control operand produced by comparison

Instructions are grouped into three sequentially distinct instruction classes: Integer, Load/Store and Control. Each class is further broken down into subclasses that describe additional variations of behavior. All instructions in a particular instruction class::subclass require the same organization of low-level tasks and hence exhibit the same expected sequential behavior. The class and subclass of an instruction is determined by the instruction decode task. Before the instruction decode is completed, every instruction class::subclass is possible and can be speculatively executed. For example, a register file read task is typically speculated simultaneously or prior to decode despite the specification order.

6.3.4 Integer and Load/Store Behavior Subclasses

Instructions in the Integer and Load/Store classes will exhibit one of the behaviors described in table 6.16. Most of these instructions represent simple behaviors which are largely distinguished by the operand fetch behaviors. The

Table 6.16: Integer Arithmetic and Load/Store Instruction Class

Subclass	SimpleScalar Instructions	Necessary Tasks
rr1	add, addu, sub, subu, and, or, xor, nor, sll, srl, sra, slt, sltu	RFRD((rs), (op1)) RFRD((rt), (op2)) INTUS((op1, op2, cop), (op)) RFWR((rd, op), ())
rr2	mult, multu, div, divu	RFRD((rs), (op1)) RFRD((rt), (op2)) INTUC((op1, op2, cop), (ophi, oplow)) WRHL((cop, ophi), ()) WRHL((cop, oplow), ())
ri	addi, addiu, andi, ori, xori, slti, sltiu	RFRD((rs), (op1)) INTUS((op1, imm, cop), (op)) RFWR((rt, op), ())
tohilo	mthi, mtlo	RFRD((rs), (op)) WRHL((op, cop))
fromhilo	mfhi, mflo	RDHL((cop), (op)) RFWR((rd, op), ())
lds	lb, lbu, lh, lhu, lw	RFRD((rs), (op1)) INTUS((op1, imm, cop), (addr)) MEMRD((addr), (op)) RFWR((rt, op), ())
ldd	dlw	RFRD((rs), (op1)) INTUS((op1, imm, cop), (addr)) DMEMRD((addr), (op1, op2)) RFWR((rt, op1), ()) INTUS((rt, 1, cop), (op)) RFWR((op, op2), ())
strs	sb, sbu, sh, shu, sw	RFRD((rs), (op1)) INTUS((op1, imm, cop), (addr)) RFRD((rt), (op)) MEMWR((addr, op), ())
strd	dsw	RFRD((rs), (op1)) INTUS((op1, imm, cop), (addr)) INTUS((rt, 1, cop), (op)) RFRD((rt), (op1)) RFRD((op), (op2)) DMEMWR((addr, op1, op2), ())

exceptions are `mult` and `div` instructions which use a special integer task INTUC. This task can be customized to represent the practical issues of making a scalar multiply/divide pipeline including potentially complex sequential constraints. Other exceptions are the memory access tasks, MEMRD, DMEMRD, MEMWR and DMEMWR, used in load/store instructions. These too may require more complex sequential constraints representing memory subsystem access protocols and/or delays. Additionally, memory access tasks may have nondeterministic control to model cache hits and misses. Since these sequential constraints are represented by automata models, they may be more flexible and realistic than interval based constraints and hence lead to potential improvements in scheduling efficiency.

6.3.5 Control Behavior Subclasses

The control class consists of those instructions primarily involved with update and management of the program counter, and hence correct instruction prefetch. Their behavior is dependent on whether the current instruction is a jump or branch and, if a branch, on whether the branch is taken or not. Next-pc calculation and instruction prefetch are often bottlenecks in processor architecture because the correct next pc and consequently the correct instruction prefetch location are not known until relatively late in an instruction's execution. In manual designs, speculative pc increment and instruction prefetch are used to improve performance for the most common cases. Although fork-type control behaviors are implicitly speculated in ABSS, join-types of behaviors (operand resolution) are not. To allow speculative pc increment and instruction prefetch in at least the most common cases, *multiple* pc increment and instruction prefetch tasks must be modeled through task-splitting.

```
if (branch taken) {
   INSTUS( (pc, 4, cop), (op1) )
   INSTUS( (op1, imm, cop), (npc_b) ) }
elseif (jump taken) {
   INSTUS( (pc, imm, cop), (npc_j) ) }
elseif (jump register taken) {
   RDRF( (rs), (npc_jr) ) }
elseif (default) {
   INSTUS( (pc, 4, cop), (npc_def) ) }

MEMRD( (npc_??), (ninst) )
```

Figure 6.15 Possibilities for next-pc calculation and instruction prefetch

Figure 6.15 shows a pseudo-code if statement representing all possible ways a next pc may be calculated. This pseudo-code calculates the next pc, *npc*, based on the decoded instruction. The correct *npc* is used to prefetch the instruction executed by the next instruction iterate in the processor composition. Since there is only a single instruction prefetch task modeled, the current instruction must be decoded to distinguish and correctly resolve the *npc* used in the prefetch.

Additional speculation freedom is added to the model by duplicating the instruction prefetch before the *npc* resolution point. Figure 6.16 shows this same pseudo-code but now with two instruction prefetches: speculative and nonspeculative. The speculative instruction fetch occurs only under default cases. Although the current instruction must be decoded to resolve the correct prefetched next instruction, *ninst*, the speculatively prefetched instruction, $ninst_{spec}$, may often be used *immediately* if the current instruction decodes to the default case. The dependency from current decoded instruction to instruction prefetch is removed for the default case by task splitting (section 4.5.3) an instruction prefetch within the default case.

```
if (branch) {
   INSTUS( (pc, 4, cop), (op1) )
   INSTUS( (op1, imm, cop), (npc_b) ) }
elseif (jump taken) {
   INSTUS( (pc, imm, cop), (npc_j) ) }
elseif (jump register taken) {
   RDRF( (rs), (npc_jr) ) }
elseif (default) {
   INSTUS( (pc, 4, cop), (npc_def) )
   MEMRD( (npc_def), (ninst_spec) ) }

if (not default) {
   MEMRD( (npc_??), (ninst_nospec) ) }
```

Figure 6.16 Speculative instruction prefetch in the default case

Figure 6.16 still requires that *pc* be incremented in the default case. Further optimization of the default case is achieved by speculatively preincrementing *pc* as shown in figure 6.17. Notice that both the branch and default cases now use a

```
if (branch taken) {
   INSTUS( (ppc, imm, cop), (npc_b) ) }
elseif (jump taken) {
   INSTUS( (pc, imm, cop), (npc_j) ) }
elseif (jump register taken) {
   RDRF( (rs), (npc_jr) ) }
elsif (default) {
   MEMRD( (ppc), (ninst_spec) )
   INSTUS( (ppc, 4, cop), (nppc_spec) ) }

if (not default) {
   MEMRD( (npc_??), (ninst_nospec) )
   INSTUS( (npc_??, 4, cop), (nppc_nospec) ) }
```

Figure 6.17 Speculative pc preincrement in the default case

preincremented *pc*, called *ppc*. Furthermore, the next preincremented *pc*, *nppc*, is computed speculatively in the default case and nonspeculatively in other cases. As with the speculatively prefetched instruction, $nppc_{spec}$ may be correctly resolved

and used *immediately* when the current instruction decodes to the default case. The dependency on which *pc* to increment in the default followed by default instruction case is removed by addition of a speculatively preincremented *pc* in the default case.

With these speculative pc preincrement and instruction prefetch goals in mind, it is possible to specify control class and subclass behaviors. Table 6.17 describes control subclasses and necessary tasks. Of particular note is the branch, b, subclass. This subclass is necessary if the instruction decodes to a branch and represents a branch's comparison behavior. A new control operand, *bop*, is produced in sync with this comparison. For branch instructions, *bop*, determines whether branch not taken, bnt, or branch taken, bt, subclass behaviors are valid.

Table 6.17: Control Class

Subclass	SimpleScalar Instructions	Necessary Tasks
default or bnt	all except j, jal, jr, jalr	(No task - ppc is used as npc) INTUS((npc, 4, cop), (nppc)) MEMRD((npc), (ninst))
ji	j	INTUS((pc, imm, cop), (npc)) INTUS((npc, 4, cop), (nppc)) MEMRD((npc), (ninst))
jil	jal	INTUS((pc, imm, cop), (npc)) INTUS((npc, 4, cop), (nppc)) RFWR((31), (ppc)) MEMRD((npc), (ninst))
jr	jr	RFRD((rs), (npc)) INTUS((npc, 4, cop), (nppc)) MEMRD((npc), (ninst))
jrl	jalr	RFRD((rs), (npc)) INTUS((npc, 4, cop), (nppc)) RFWR((rd), (ppc)) MEMRD((npc), (ninst))
b	beq, bne, blez, bgtz, bltz, bgez	RFRD((rs), (op1)) RFRD((rt), (op2)) INTUS((op1, op2, cop), (bop))
bt (only if branch taken)	beq, bne, blez, bgtz, bltz, bgez	INTUS((ppc, imm, cop), (npc)) INTUS((npc, 4, cop), (nppc)) MEMRD((npc), (ninst))

6.3.6 Data Hazards

An instruction task has bypass output and input operands to resolve data hazards. Incorporating data hazard detection and bypass freedom in an instruction task requires three additions. First, when an instruction is decoded, an additional control operand, *hazardop*, is produced which distinguishes between data hazard or no data hazard. Second, bypassed values must be passed and available from the previous instruction task. This bypass operand is an operand resolution of *every* potential register-file write-back operand from the previous instruction task. Finally, an operand resolution point is added whenever a register file operand is required in the current instruction task. At this operand resolution point, *hazardop* selects either the register file operand or the bypassed operand from the previous instruction-level task. For example, instructions in the rr1 class require the register file entry at *rs*. It may be that the last instruction iterate is computing a new operand that has yet to be written to this register file location. Instead of always reading a value at location *rs*, an operand resolution point is added to one input of rr1's INTUS task. Depending on the value of *hazardop*, either the register file read result or the bypassed value is accepted.

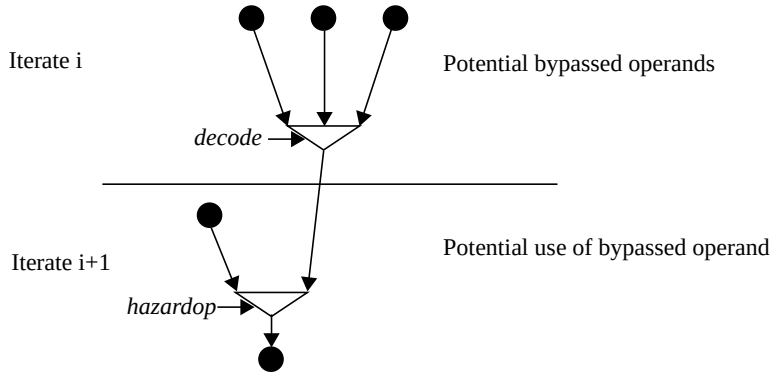


Figure 6.18 Single bypass modeling

In the worst case, three register file reads may be required by a single instruction belonging to the store double, strd, subclass. It is also possible that data

hazards exist on at most two of these register file reads as instructions belonging to the load double, ldd, subclass perform two register file write-backs. A designer may choose to model bypasses for as many situations as desired. This comes at a cost as each additional bypass value implies additional or expanded operand resolution points as well as another *hazardop* operand to indicate when this new bypass value is required. Because of this added complexity, often only a single operand is bypassed from one instruction iterate to the next. The production source of a bypassed operand is a resolution point of *every* potential register-file write-back from the previous instruction iterate. A single *hazardop* control operand distinguishes between three situations: no hazard, *rs* read hazard or *rt* read hazard.

When a single bypass is modeled, it is still necessary to correctly handle scenarios where multiple data hazards exist. Since no bypass route exists for additional data hazards, the dependent instruction task must stall until the previous instruction task has finished all register file (or other state) write-backs. This may be modeled with an additional control operand *stallop* within each instruction task. The operand *stallop* indicates whether or not the *next* instruction iterate should stall. If *stallop* resolves *true*, then the prefetched instruction is deemed to have too many data hazards and is not made available to the next instruction iterate. Hence, the next instruction iterate is forced to stall. Only when every register file write-back has occurred is the prefetched instruction made available. This implies an operand resolution for the passed prefetched instruction. When *stallop* resolves *true*, this operand resolution point requires the correct prefetched instruction AND all write-backs to be *known*. When *stallop* resolves *false*, this operand resolution point only requires that the correct prefetched instruction is *known*. Finally, *stallop* is used to stall dependent instructions for other data hazards, such as hazards on architecture registers hi and low, as well.

The control operands *hazardop* and *stallop* are both non-deterministic. This is because without actual data values, it is impossible for this model to determine precisely when data hazards occur. What *hazardop* and *stallop* do provide is distinguishable variations in behavior for data hazard and bypass scenarios. This allows probabilistic data, gathered from profiling actual code, to guide schedule prioritization during exploration.

6.3.7 Operand Resolution Points

As discussed in section 4.3.1, ABSS limits state growth and guarantees finite state representation in control-dependent behavior through use of operand resolution points. To better understand the freedoms and limitations of the SimpleScalar ABSS model, it is helpful to summarize all such resolution points. The most important resolutions are for instruction prefetches, $ninst_{spec}$ and $ninst_{nospec}$, and for program counter preincrements, $nppc_{spec}$ and $nppc_{nospec}$. These may be correctly resolved once the current instruction is decoded. If the current instruction decodes to a branch, correct resolution is delayed until the control operand *bop* is *known*. Other important resolution points relate to data hazards. If data hazards are modeled, then hazard-prone data fetches and mated bypasses must be correctly resolved. These too may occur once the current instruction is decoded and presence or absence of data hazards is determined. Finally, other operand resolutions occur in the presented SimpleScalar model which are hidden by covering dependencies. For instance, the integer write-back task occurs for three of the five Integer subclass behaviors. Operand resolution is required to determine which subclass' result should be the write-back value. For each of these three subclasses, an earlier task already depends on *cop* from the instruction decode task. Hence, instruction decode will have had to occurred *before* the operand resolution point is reached. Thus, some operand resolution points may be hidden and do not impact model freedom.

6.4 RISC Processor Results

As described in section 6.3 and shown in figure 6.19, an instruction iterate contains 25 low-level tasks[4] and 37 care control cases. Only 25 low-level tasks are required as many instruction subclasses share common tasks. For example, several subclass behaviors require an *rs* register fetch, yet only a single *rs* register fetch task is instantiated and all tasks requiring *rs* depend on this task.

When three instruction iterates form a processor composition, a cyclic scheduling problem with 75 tasks and 50,653 care control cases is created. This processor composition represents three ‘in-flight’ instructions and does not require that these instructions be initially pipelined as is traditionally done. This section presents results for scheduling a three in-flight instruction processor model. All control cases and cyclic considerations are handled automatically and exactly by ABSS. Although the behavioral specification does not demand pipelining but only ordered instruction fetches, what results is a pipelined implementation suitable for embedded applications with performance equivalent to what one expects from a well-done manual implementation.

The MediaBench[70] suite of benchmarks serve as a measurement point and exploration guide for RISC processor scheduling results. The MediaBench suite consists of 11 representative embedded applications. For the present purposes, 4 applications were eliminated due to substantial floating-point content or inability to compile correctly for the SimpleScalar tool suite. The remaining 7 consisted of three speech compression/decompression applications: adpcm, gsm and g721, three graphics applications: jpeg compression and decompression, mpeg compression and ghostscript postscript rendering, as well as one encryption/ decryption application: pegwit.

4. The figure shows more than 25 tasks as synced multivalue **MA** for control are included.

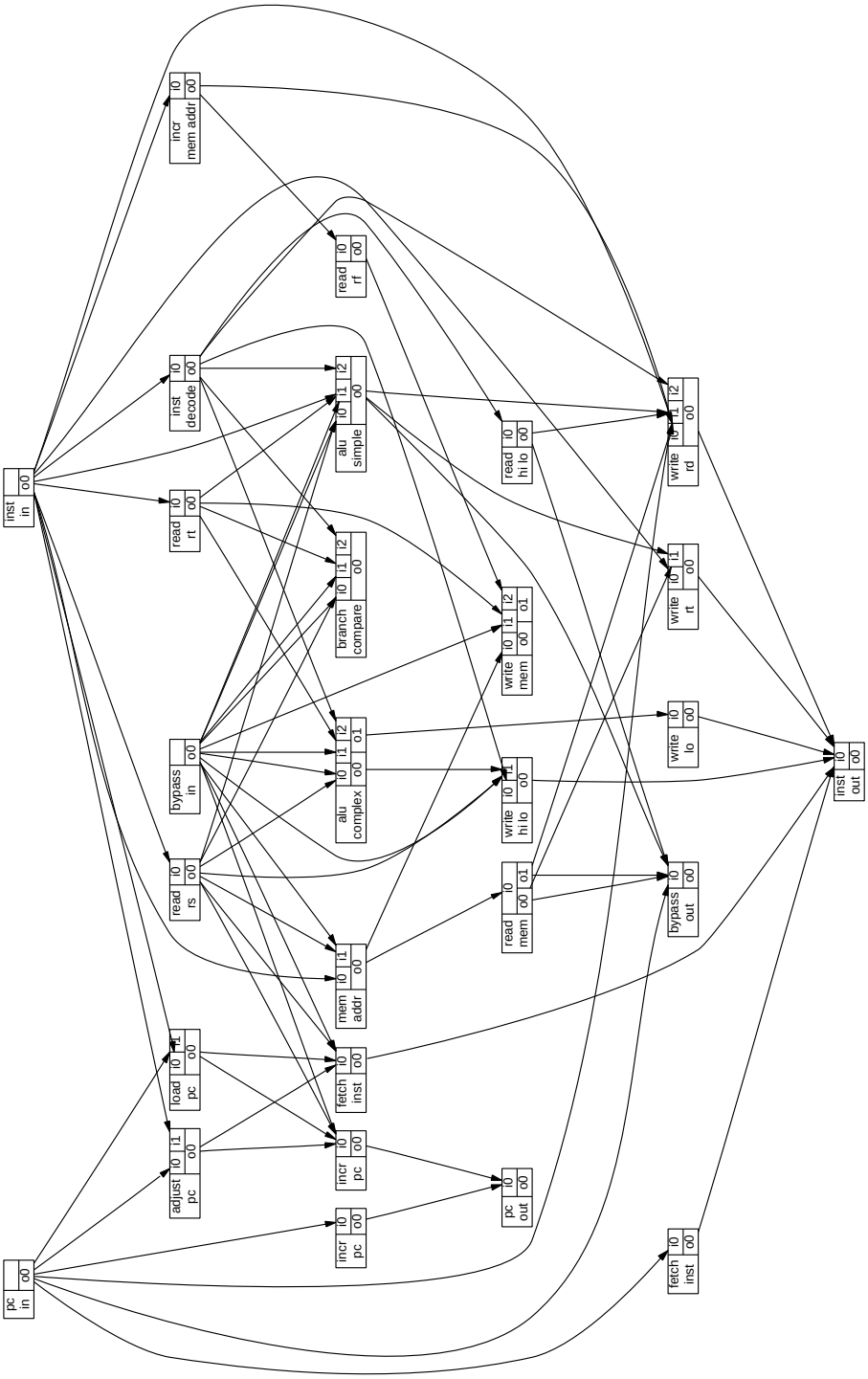


Figure 6.19 RISC processor instruction iterate

The SimpleScalar tool suite was modified to produce simulation statistics relevant to the processor model. Instruction class mix, correlation between two instruction classes, single and blocking hazard probabilities, and branch taken/not taken statistics were generated and analyzed. All 7 benchmarks exhibited similar behavior and hence the following generalizations were made. The majority of executed instructions, ~88% average for all benchmarks, were in the register-to-register simple integer (rr1), register-to-immediate simple integer (ri), branch taken (bt) or load single word (lds) subclasses. Within these four majority subclasses, rr1 and rri were most common, then bt and finally lds. Correlation among these four majority subclasses was also high at 72% on average for all benchmarks. Branches were taken about 80% of the time. Hazards which could be eliminated with a single bypass occurred for roughly 35% of all executed instructions. Hazards which stall successive instructions occurred for only approximately 1% of all executed instructions.

6.4.1 Priority Mix Set 1

With this application character in mind, two exploration control prioritization mix sets were devised to help evaluate and direct the expected performance of a FSM controller synthesized from this symbolic processor model. Priority mix set 1, as shown in table 6.18, gives highest priority to register-to-register, register-to-immediate and branch taken mixes. Other mixes, each with slightly less priority, are prioritized with load single word, branch not taken mixes being the last specifically prioritized mix. After this, all remaining instruction mixes are prioritized together in a final step. Likewise, mix set 2 also prioritized six instruction mixes but favors load single word, branch mixes over register-to-register, register-to-immediate mixes. A particular instruction mix in a mix set includes all permutations of three instructions with exactly one branch if a branch is present in a mix. Similarly, if no branch is present in a mix, but only register-to-

register, register-to-immediate and load single word instructions, then exactly one load single word instruction is present in every mix permutation. For example, the mix ‘rr1,ri,lds’ includes all permutations (rr1,rr1,lds), (rr1,ri,lds), (ri,ri,lds), (ri,rr1,lds), (rr1,lds,rr1), etc. with exactly one lds.

Two sequential constraint scenarios, A and B, were modeled. Within a scenario, constraints were added in two phases: sequential and concurrency. Scenario A assumes all tasks complete in a single time-step except for double word memory reads and writes and complex integer computations. A double word memory read returns the first word after a single time-step and then the final word in the immediate next time-step. Likewise, a double word memory write requires the first write word during the first time-step and requires the second write word during the second time-step. A complex integer computation returns the hi result after a single time-step and then the lo result in the second time-step. Scenario B extends scenario A such that all memory reads, whether instruction fetch or load instruction, require two time-steps. The read value is only available after two time-steps. These scenarios model a probable embedded processor core with two types of non-cached embedded system memory: single time-step word access and two time-step word access.

Three resource concurrency constraint configurations, none, moderate and tight, were applied to sequential constraint scenarios A and B. No concurrency constraints were applied for the none configuration and hence only sequential and dependency constraints apply. The moderate configuration assumes two memory ports for all memory read/write and instruction fetch tasks. Furthermore, the register file has 3 ports and supports at most 2 register file reads and 1 register file write concurrently during any single time-step. A single integer function unit is available for all instruction implementation computations. An additional integer function unit is available for all program counter updates and calculations. Only a

single instruction decode unit as well as single access to architecture hi/lo registers are allowed. The tight configuration is a restricted version of the moderate configuration in which only one memory port is available. Also, the register file has 2 ports and supports at most 2 register file reads or 1 register file read/1 register file write concurrently during any single time-step. Finally, just a single integer function unit is available for all program counter and instruction implementation computations.

Table 6.18: Expected CPI Given Mix Set 1

Prioritized Mix	Resource Configurations					
	A[a] none	A moderate	A tight	B[b] none[c]	B moderate[d]	B tight[e]
rr1,ri,bt	1.67	1.67	2.67	2.67	2.67	3.00
rr1,ri,bnt	1.33	1.33	2.00	2.00	2.00	2.00
rr1,ri,lds	1.33	1.33	2.00	2.33	2.33	2.67
lds,lds,bt	2.33	2.33	2.67	3.33	3.33	4.33/4.00
lds,lds,bnt	2.00	2.00	2.00	2.67	2.67	3.33
no stall	3.00	3.00	3.33	4.00	4.00	4.67
stall	5.00	5.00	5.00	5.00	6.00	6.00
best	1.00	1.00	1.33/1.00	2.00	2.00	2.00
CPU Seconds	180	848	1648	156	717	1610

a. All tasks 1 ts except double word read, write, multiply and divide which are 2 ts piped.
b. All tasks 1 ts except read, write, instruction fetches which are 2 ts, multiply and divide which are 2 ts piped.
c. No hardware utility constraints.
d. 1 datapath ALU, 1 pc ALU, 1 3-port register file (at most 2 reads, 1 write), 2 memory ports, 1 decode unit, 1 port to architecture hi/lo registers.
e. 1 datapath/pc ALU, 1 2-port register file (at most 2 reads or 1 read/1 write), 1memory port, 1 decode unit, 1 port to architecture hi/lo registers.

Table 6.18 shows expected cycles per instruction, CPI, data for a synthesized FSM controller which prioritizes mix set 1. In some cases, two CPI numbers are reported. The larger one is the expected CPI for entering this instruction mix from any arbitrary instruction and the smaller one is the expected CPI for sustaining this instruction mix. If a single CPI is reported, than the expected CPI is the same no matter what earlier instructions were executed. In all prioritized cases, the

moderate resource configuration achieves the same performance as the none configuration. ABSS is able to find optimal schedules for these prioritized control cases as well as all 50,653 care control cases for three instructions in-flight. The no stall (no multiple data hazards) and stall (multiple data hazards) rows indicates CPI for worst control cases while the best row indicates CPI for the best control cases.

Many of the 50,653 control paths do achieve single cycle throughput even though the expected FSM machine has been optimized for a particular subset of instruction mixes. For example, the mixes ‘rr1,rr1,rr1’ and ‘ri,tohi,rr1’ achieve single cycle throughput and are included in the best row. Of particular interest is the ‘rr1,rr1,rr1’ mix with an A tight resource configuration. Due to the restricted register file access, this mix only achieves single-cycle throughput *if hazards exist*. Because of the comprehensive exploration, a correct optimal sequence for this particular control case which utilizes the speed-up of a bypass is found. Although a single-issue MIPS pipelined processor is a well understood structure with expected high performance for any manually implemented design, ABSS meets this same level of expected performance yet does so automatically.

6.4.2 Priority Mix Set 2

Table 6.19presents expected CPI data for a synthesized FSM controller which prioritizes mix set 2. There is not much difference when compared to table 6.18. This may be attributed to the original RISC philosophy of few, simple and similar instructions. Four differences do occur for the B tight configuration and are shown in bold. Since the mix ‘lds,lds,bt’ is at a higher priority in table 6.19, a solution with CPI of 4.00 rather than 4.33 is found. By selecting this 4.00 CPI solution, other control cases of less priority are impacted. The ‘lds,lds,bnt’ mix is now actually slightly worse, 3.67 versus 3.33 CPI. Also, the ‘rr1,ri,bnt’ is slightly

worse, 2.33 versus 2.00 CPI. Yet, some of the worst case control paths benefit

Table 6.19: Expected CPI Given Mix Set 2

Prioritized Mix	Resource Configurations[a]					
	A none	A moderate	A tight	B none	B moderate	B tight
lds,lds,bt	2.33	2.33	2.67	3.33	3.33	**4.00**
lds,lds,bnt	2.00	2.00	2.00	2.67	2.67	**3.67**
rr1,ri,lds	1.33	1.33	2.00	2.33/**2.00**	2.33/**2.00**	2.67
rr1,ri,bt	1.67	1.67	2.67	2.67	2.67	3.00
rr1,ri,bnt	1.33	1.33	2.00	2.00	2.00	**2.33**
no stall	3.00	3.00	3.33	4.00	4.00	**4.00**
stall	5.00	5.00	5.00	5.00	6.00	6.00
best	1.00	1.00	1.33/1.00	2.00	2.00	2.00
CPU Seconds	184	867	1522	160	737	1795

a. Same as table 6.18.

from the choice to heavily optimize the ‘lds,lds,bt’ mix. Their CPI improves to 4.00 from 4.67. Even with tight resource constraints, ABSS helps find particular design trade-offs and best-possible control sequences.

Required CPU times range from 3 minutes to 30 minutes. All results in tables 6.18 and 6.19 were produced on a 866 MHz Xeon PIII processor with 2 GB of memory and running under Linux. It was a choice, not a necessity, to use a 2 GB memory model. As the memory is available, it is preferred to trade memory use for time. The most computational resource demanding example, B tight with mix set 2, is solvable on a 733 MHz PIII machine with 128 MB of memory and requires 80 minutes CPU time.

6.4.3 Representation Growth

Peak exploration path set time-step state set sizes, typically the largest, as well as transition relation sizes are shown for several configurations in figure 6.20. Configurations with resource concurrency constraints are most costly to solve as

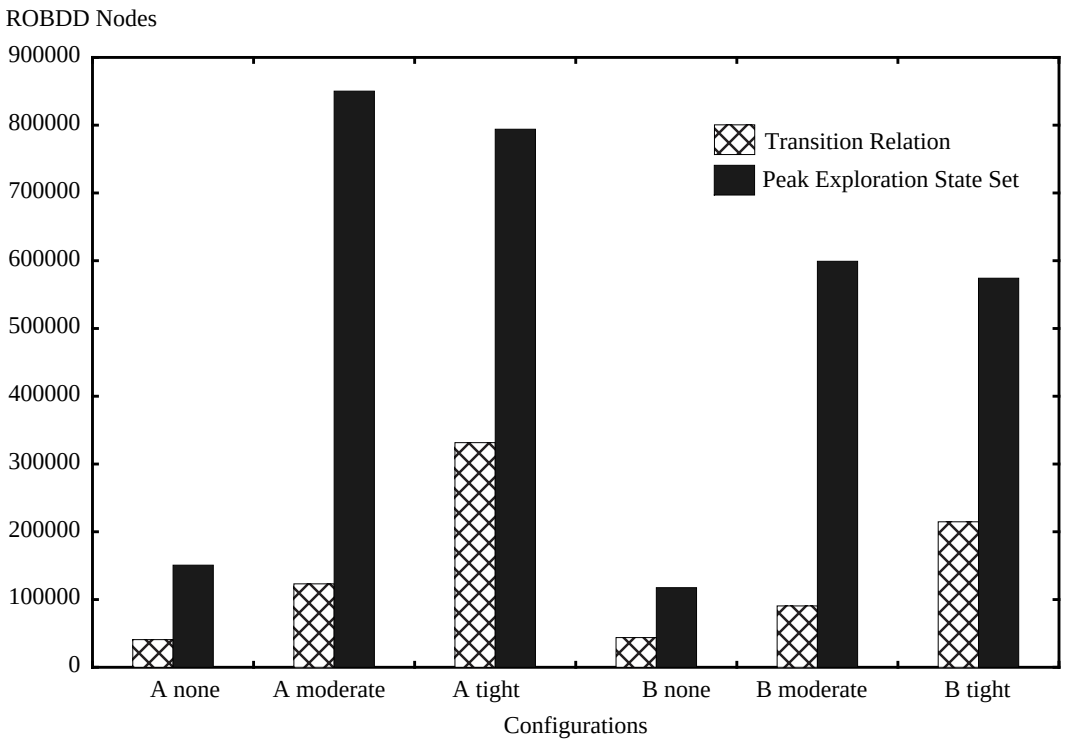


Figure 6.20 ROBDD node usage

resource concurrency constraints make scheduling intractable. Consider a scheduling problem with no resource concurrency constraints. An as-soon-as-possible list scheduler will never have to choose execution of one task over another as every task with satisfied dependencies may always begin execution. On the other hand, contention for resources during scheduling force choices as to which task executes and consequently makes this problem intractable. Still, scheduling without resource concurrency constraints with this technique takes some computational resources as *all* schedules of *every* latency are represented.

6.4.4 A Cache Hit/Miss Model

Configurations A and B model relatively simple sequential memory access. Configuration A assumes single time-step access while configuration B assumes two time-step access. Configuration A was modified to include two behaviors for each memory read. The cache hit behavior defaults to single time-step access

while the cache miss behavior requires three time-steps to complete access. Furthermore, if a cache miss occurs, the memory subsystem is pipelined and may continue processing other memory requests, regardless of whether they produce a cache hit or miss. Nondeterministic control distinguishes between a cache hit or miss. Three memory access tasks: speculative instruction fetch, nonspeculative instruction fetch and memory read, were subject to cache hit or miss and individually required nondeterministic control values. With these additional control points, the number of distinct care control cases in a three in-flight instruction composition increases to 474,552.

Table 6.20: Expected CPI For Model with Cache Hit/Miss

Prioritized Mix	Resource Configurations		
	none	moderate	tight
rr1,ri,bt	2.67	2.67	2.67
rr1,ri,bnt	2.00	2.00	2.00
rr1,ri,lds	1.67	1.67	1.67
lds,lds,bt	3.33	3.33	3.33
lds,lds,bnt	2.67	2.67	2.67
worst (no stall)	4.00	4.00	4.00
worst (stall)	6.00	7.00	7.00
best	1.67	1.67	1.67
CPU Seconds	719	7291	13,369

Table 6.20 summarizes results given priority mix 1. Because of cache miss penalties (one cache miss is assumed in each mix), all resource configurations achieve the same CPI for the prioritized instruction mixes. This indicates that a tight resource configuration is appropriate when cache miss probabilities are 20-30%. Finally, because of the additional control cases, required computation time is greater.

6.4.5 2, 3 and 4 Iterates

Tests cases with 2, 3 and 4 iterates for resource configuration A were solved. Solutions for the 2 iterate composition were impacted by capacity constraints. Solution iteration latencies for the 4 iterate composition did not improve when compared to the 3 iterate composition. For this particular model and configuration, a 3 iterate composition contains sufficient state to represent most problem freedoms. Computation times range from 44 to 806 to 19,435 seconds for 2, 3 and 4 iterates respectively. Complexity increases by a factor of roughly 20 for each additional iterate.

6.4.6 Summary

This chapter presented applications of ABSS. First, data-flow examples, drawn from academia and industry, were scheduled. These included the elliptic wave filter, the fast discrete cosine transform, and a math library function for Intel's Itanium processor. Growth and practicality issues were discussed in this setting. Next, control-dependent examples, again drawn from academia and industry, were scheduled. These included a rotation of coordinates algorithm, spherical to rectangular coordinate conversion and a graphics processing subsystem. Local register use, additional iterates and control case prioritization were discussed. Finally, a RISC processor model, based on the MIPS architecture, was introduced. A suite of common embedded benchmarks was used to guide and interpret results. Although no pipeline direction existed in the original specification, ABSS found such pipelined execution sequences for three 'in-flight' instructions that are comparable to high-quality manual implementations.

Chapter 7

Discussion

This chapter discusses the novelty and limitations of ABSS. Also, future ABSS research directions are outlined. Finally, a brief conclusion summarizes the main concepts presented in this dissertation.

7.1 ABSS Novelty

Chapter 1 expanded the scope of the scheduling problem to encompass operand dependence, control dependence, sequential requirements, hardware resource requirements, repetition and pipelining. Although prior work in scheduling has addressed all these issues to some extent, ABSS makes substantial advances regarding sequential constraints as well as repetition and pipelining with control. Furthermore, ABSS accommodates all scheduling problem scope constraints in concert while systematically determining high-quality practicable solutions for problems of useful scale. The ability to represent and schedule sequential models of repeating behavior with hundreds of tasks and over 500,000 control paths substantially raises the bar as to what is believed possible for exact scheduling models.

7.1.1 Sequential Representation

The fundamental and pervasive structures in ABSS are nondeterministic finite automata, NFA. These naturally represent any sort of sequential constraints or requirements in a finite digital system design. This was most clearly demonstrated with sequentially constrained IP blocks and external IO protocols used in the EWF case study (section 6.1.1), FDCT configurations (section 6.1.3), the control-dependent industrial example (section 6.2.3), as well as the RISC example with cache hit/miss protocols (section 6.4.4). ABSS lays a foundation for exact scheduling of such sequential models. Future work in ABSS will build on this foundation to represent and schedule models with even greater sequential complexity.

Notable prior work on scheduling automaton models is attributed to Yen[140][141] and Yang[138]. The work by Yen used an explicit FSM model and did not produce exact results. Furthermore, Yen's technique had difficulties with practical design constraints and problem scale. Yang's technique was symbolic and implicit but experienced severe difficulties with problem scale and did not guarantee correct solutions for control-dependent problems. ABSS is unique in that it represents and exactly schedules automaton models of practical scale given hardware resource requirements, repeating behavior and control dependence.

7.1.2 Repeating Behavior with Control

Scheduling repeating behavior with control in the absence of sequential constraints is itself a difficult problem as evidenced by the scarcity of prior work. Lakshminarayana's *Wavesched* [68] is perhaps the best recent attempt. This heuristic targets control-flow intensive (limited control case) loop behaviors. ABSS provides a theoretical advance because it represents and exactly schedules repeating behavior with control. Moreover, as demonstrated by the RISC example

(section 6.1.1), ABSS can accommodate close to 500,000 control cases! This is unparalleled in the literature. Furthermore, ABSS still supports some types of speculation as demonstrated by the ROTOR examples (section 6.2.1). Finally, ABSS does all this exactly --also unparalleled in the literature.

7.1.3 Quality

Perhaps the most novel aspect of ABSS is the quality of solutions. Quality may be measured in terms of how completely all necessary design constraints are accommodated. In this respect, ABSS is able to accommodate all design constraints, except final explicit binding, *in concert*. Some demonstrations of this are the FDCT configurations in section 6.1.3 that incorporate function unit and register bounds as well as internal and external sequential constraints. Quality may also be measured as exactness. ABSS exhaustively encapsulates and explores the entire solution space and hence can guarantee exact solutions. Furthermore, since all solutions are represented, ABSS provides a complete set of solutions that cover potential odd-ball yet necessary constrol cases and constraints. The RISC examples with bypass and potential stall for multiple hazards (section 6.3.6) demonstrate this completeness. The constructive automata approach to scheduling produces solutions with quality and completeness unmatched by any prior scheduling work.

7.1.4 Useful Scale

Although ABSS is exact and exhaustive, it still solves problems of useful scale. This is evidenced by the two industrial examples (sections 6.1.7 and 6.2.3). Compared to exact schedulers, the largest problems presented (100, 127 and 132 tasks) substantially raise expectations regarding problem scale. Compared to heuristic schedulers, the largest problems ABSS solves often exceed or equal the scale of typically reported problems with resource contention. If control scale is

considered, the ABSS RISC examples (section 6.3) with over 50,000 control cases or even the pipelined S2R example (section 6.2.2) with 256 control cases are unrivaled. Finally, ABSS is able to solve problems of useful scale on reasonable hardware and in reasonable time. The problems presented in this dissertation were often solved on a modern PC (733 MHz) with 512 MB of memory or less.

ABSS does not overcome the intractable nature of resource-constrained scheduling. ABSS does provide an exact and workable technique for many problems. With respect to problems of very large scale, a novel aspect of ABSS is its formulation forethought regarding abstraction. A composite task and associated **CMA** are fundamentally the same as a task and its associated **MA**. Hence, a mechanism exists to create a hierarchy of refinement, as described in section 7.3.4, that addresses such large problems.

7.2 Limitations and Complexity of ABSS

Scheduling with resource contention is an NP-complete problem[41][103]. Still, implicit ROBDD-based techniques have dramatically raised expectations regarding what variations and scale of NP-complete problems are solvable. Although ROBDD-based techniques are time-and-space exponential in the worst and often typical case, they are often practicable for many problems of useful scale. Through careful formulation, ABSS exploits this potential ROBDD speedup. Unfortunately, this potential speedup makes it difficult to precisely quantify the complexity of ABSS. This section examines where complexity occurs in ABSS and identifies when problem limits are reached with conventional computational resources.

7.2.1 Finite State

ABSS complexity is directly proportional to the number of state bits needed to construct a **CMA**. To model a composite task, an **MA** is assigned to each task within the composition. Hence, the amount of state in a **CMA** is directly proportional to the number of tasks, or **MA**, in a composition. Furthermore, if the low-level tasks within a composition exhibit complex sequential behavior, additional state is required for each **MA** which again leads to state growth in a **CMA**. The largest examples solved in chapter 6 are summarized in table 7.1. From

Table 7.1: Select Largest ABSS Examples

Example	Tasks	State Bits	CPU Seconds
288 from section 6.1.6	100	133	26.6
Itanium from section 6.2.3	127	339	131
Configuration 2 from section 6.2.3	132	271	865
4 Iterate RISC from section 6.4.5	100	159	19,435

these experimental results, one may conclude that, given a current computer[1], present ABSS limits are roughly 100-135 tasks represented with 130-340 **CMA** state bits. Required computation times range from a few minutes to a few hours.

An **MA** represents production and existence of system operands. As an **MA** contains finite state, the set of operands it represents is also finite. This bound is particularly important for operands that are produced iteratively in loops and pipelines. In fact, only one instance of an operand per cyclic **MA** is allowed as described in section 3.2. This bound may be relaxed in a controlled way, at the expense of additional state, by adding iterates, section 4.4, or operand buffers, section 4.5.4, to a composition. Finally, a finite set of operands is also maintained

1. A current computer is a 733 MHz Pentium II with 512 MB of memory.

at operand resolution points. Since only one postresolution **MA** exists, only one set of operands may be produced with one set of preresolution input operands. This state-related limit is fully described in section 4.3.1 and may be relaxed in a controlled manner at the expense of additional state with task splitting, section 4.5.3.

7.2.2 Composition Character

State requirement is not the only factor contributing to ABSS complexity. The overall structure and number of operand dependencies in a composition, as well as the imposed hardware concurrency bounds, significantly influences complexity. For example, a fast discrete cosine transform contains highly symmetric operand dependencies and two independent behavior subgraphs. Computational requirements to solve this relatively ‘loose’ behavior are significantly higher than for the ‘tight’ behavior of the elliptic wave filter (section 6.1). In general, less complexity is encountered if a composition contains many freedom-constraining operand dependencies, orders **MA** in a ROBDD to minimize operand dependency length and overlap (section 4.6), and imposes either no or extremely constraining hardware concurrency bounds. On the other hand, greater complexity is encountered if a composition contains few operand dependencies organized to allow significant freedom, orders **MA** in a ROBDD to maximize distance between operand producers and accepters, and imposes hardware concurrency constraints that interact with operand dependency constraints in a balanced way. In fact, the ability for these techniques to solve very highly constrained problems offers the complement to conventional techniques which typically are more efficient for weakly constrained problems.

7.2.3 Exploration Complexity

The **CMA** exploration steps presented in chapter 5 are the greatest algorithmic contributors to ABSS complexity. The discussion in section 6.1.3 indicates that exploration often accounts for 80-90% of the computation CPU seconds. In particular, exploration's fixed-point algorithms are the prime sources of such complexity.

To facilitate a summary of exploration complexity, two broad definitions of fixed-point algorithms are defined. A **greatest-fixed point**, *GFP*(), begins with a *larger* state set S^0, and uses a series of *n* preimage, image, existential quantification, and/or universal quantification steps to reach a *smaller* S^n such that $S^n=S^{n+1}$ and $S^n \subseteq S^0$. Likewise, A **least-fixed point**, *LFP*(), begins with a *smaller* state set S^0, and uses a series of *n* preimage, image, existential quantification, or universal quantification steps to reach a *larger* S^n such that $S^n=S^{n+1}$ and $S^0 \subseteq S^n$.

Table 7.2 summarizes exploration algorithms for the four classes of scheduling problems: acyclic data-flow, acyclic control-dependent, cyclic data-flow and cyclic control-dependent. From this summary, complexity conclusions may be made. The

Table 7.2: Summary of Exploration Algorithms

	Data-Flow		Control-Dependent	
	Acyclic	Cyclic	Acyclic	Cyclic
Forward Exploration	*LFP*()	*LFP*()	*LFP*()	*LFP*()
Backward Pruning	*LFP*()			
Closure with Backward Pruning		*GFP*(*LFP*())		
Backward Pruning with Validation			*LFP*(*GFP*())	
Closure with Backward Pruning and Validation				*GFP*(*LFP*(*GFP*()))

least complex exploration, requiring just two least-fixed points, is for acyclic data-flow scheduling problems. Closure adds a greatest fixed-point computation to cyclic data-flow exploration, *GFP*(*LFP*()). When written nested, each step of the *GFP*() requires a complete *LFP*() computation. Validation adds an *LFP*() computation for control-dependent exploration. Finally, exploration for cyclic control-dependent problems requires the greatest complexity with three nested fixed points. Complexity growth during exploration is clearly where ABSS encounters limits.

Still, ABSS exploration complexity is justified. Chapter 6 demonstrates that ABSS is practicable for problems of useful scale. Furthermore, no other known techniques exist that produce exact results for cyclic control-dependent scheduling problems.

7.3 Future ABSS Directions

ABSS lays a foundation for new approaches to scheduling and automated design. As such, there are several directions to pursue. These include specification, encoding, re-encoding, partitioning, hierarchy, binding, heuristics as well as direct dynamic FSM and datapath synthesis.

7.3.1 Specification

Chapter 3 describes how low-level **MA** are manually specified using a handful of states and transitions. Techniques and tools exist that simplify specification of sequential behavior. In particular, Synopsys® Protocol Compiler [122] uses a hierarchical specification of sequential behavior similar to regular expression semantics that has proven useful and helpful for meaningful design. A technique similar to this may facilitate specification of ABSS sequential behavior and interaction for both low-level **MA** and composite **MA**.

7.3.2 Encoding

ABSS uses a predominantly ‘one-hot’ encoding to represent operand existence in a system. This encoding works fairly well. On the other hand, experiments with an alternative encoding[51], which models physical function units and binds logarithmically encoded operands to these units, performed relatively poor. Even so, the ABSS ‘one-hot’ encoding perhaps errs on the side of sparsity. An improved encoding might find a better balance between one-hot sparsity and pure logarithmic state compactness when considering efficient ROBDD representation.

ABSS uses a static allocation of state to represent operand existence. In other words, once a state bit is assigned to a particular task, it always models that particular task. A dynamic interpretation of state encoding, where the interpretation of some state depends on other state, may benefit ABSS. This is one conceptual route to a more compact state encoding. Furthermore, the notion of dynamic state may enable increased freedom in an ABSS model without the expense of additional state. For example, task splitting (section 4.5.3), iterates (section 4.4.5) and operand buffers (section 4.5.4) are all ways to increase model freedom by representing more operand instances at the expense of additional state. In fact, these correspond to transformation of the original behavior graph or composite task. As another example, algebraic transformations, where the associative, distributive and commutative laws are used to generate equivalent expressions, may also be expressed as graph transformations. A dynamic encoding and use of nondeterminism may achieve some beneficial graph transformation without the expense of additional state.

7.3.3 Partitioning

As demonstrated with time-zone partitioning (section 6.2.3), partitioning techniques can benefit ABSS. Time is not the only way to partition an ABSS

problem. A composition task may be partitioned according to spatial placement of function units [129]. In this way, specific binding considerations are accommodated while potentially reducing computational complexity. Other techniques, which partition ABSS problems in various ways, are almost certainly beneficial, especially when faced with extremely large problems.

7.3.4 Hierarchy of Refinement

ABSS was carefully formulated so that an **MA** and a **CMA** are interchangeable. They both exhibit sequential behaviors of accepting and producing operands. This interchangeability provides the mechanism for a hierarchy of refinement as shown in figure 7.1. NFA models are the vehicles for

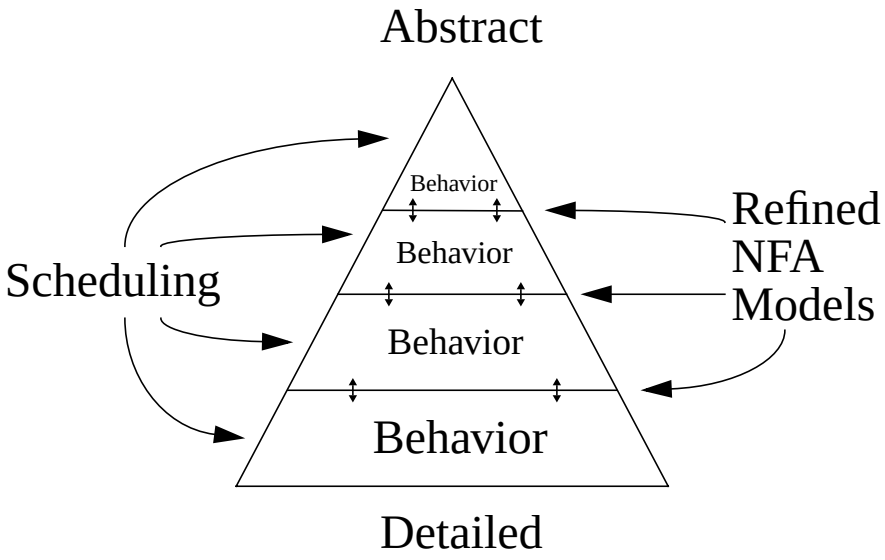


Figure 7.1 Hierarchy of Refinement

refinement between hierarchy levels. ABSS performs the refinement process within each hierarchy level. For example, suppose a behavior is described sequentially at the highest level. This description need not be precise but may be relatively general. Even so, this top-level description is a collection of co-executing tasks. Each of these tasks may also be described as a collection of tasks and so on until base behaviors are reached. Thus, a hierarchy is created. Different

expected base sequential behaviors may be assigned to tasks within compositions. Resulting composite tasks throughout the hierarchy may be scheduled until constraints from the bottom filter up to the top. Likewise, top sequential behaviors, such as protocols, may be imposed and propagated downward to influence base behavior and implementation. Consequently, several such refinements of this hierarchy produce specific and valid executions sequences and implementations at each level.

To be truly beneficial for extremely large designs, complexity from lower levels must be hidden from higher levels by abstraction. Suppose that a **CMA** from a lower level encapsulates one million valid sequences representing all possible implementations. It may be that only one hundred distinguishably different sequences exist when only considering *externally* required and produced operand events. At a higher level, there is often no need to know exactly how a lower-level task *internally* processes information, but only a need to know how a lower-level task may be interfaced to. Consequently, before being passed up in the hierarchy, a **CMA** may be re-encoded to only represent the fewer externally distinguishable sequences. In this way, complexity may be abstracted through an appropriate, though as yet unknown, **CMA** re-encoding step.

7.3.5 Heuristic Exploration

As presented in chapter 5, ABSS performs exact exploration of a **CMA**. This is the most computationally expensive step in ABSS. It may not be necessary to determine all exact solutions but rather some solutions of acceptable quality. For such cases, ABSS may benefit from new heuristic exploration techniques. Symbolic traversal and reachability are well-studied problems[21][90] and techniques exist for approximate solutions[95]. These techniques, developed

primarily for symbolic model checking, may be directly applicable to new, more efficient, ABSS exploration.

7.3.6 Synthesis

As presented, ABSS may return a witness schedule that can be used for FSM synthesis. This approach is rudimentary as it does not exploit the full *dynamic* nature of a **CMA**'s path sets. Work is needed to directly synthesize a FSM from ABSS structures. Preferably, the synthesized FSM need not be minimum state but best implementation quality (performance, area, etc.). It could potentially encapsulate numerous execution sequences and implement a dynamic control scheme. As shown in section 6.1.3, ABSS encapsulates schedules with best *average* latencies. For example, loop iterations legs with latencies of 2-3-2-3… yield an average latency of 2.5. Work to find such best average latency schedules[56] may lead to direct synthesis of FSMs from ABSS structures.

ABSS does not perform precise binding, although interconnect and binding guide constraints may be formulated (section 6.2.3). Methods are needed to precisely constrain or interpret *which* function unit a task is physically implemented on. This will most likely require additional state information as a single witness schedule currently represents numerous possible bindings. One attractive approach under investigation adds additional state by duplicating tasks and operands according to spatial bindings yet does so in an efficient partitioned manner[129]. With the ability to produce precise bindings, ABSS may be used to synthesize not only FSM controllers but datapaths.

7.4 Conclusions

A set of techniques for representing the high-level behavior of a digital subsystem as a collection of nondeterministic finite automata, NFA, were

presented. Sequential behaviors for base units, such as ALUs, multipliers, register-file ports, etc. are represented as small NFA called modeling automata, **MA**. Tasks from a behavioral graph specification are each assigned an **MA** that models appropriate sequential behaviors. All such base **MA** are composed by a Cartesian product step into a larger composite **MA** or **CMA**. Behavior operand dependence, control dependence and hardware resource requirements prune a **CMA** until it only encapsulates all valid execution sequences of the digital subsystem. Implicit symbolic ROBDD-based techniques find shortest paths in a **CMA**. These shortest paths correspond to minimum latency schedules for the digital system. This provides a very general, systematic mechanism to perform exact high-level synthesis for cyclic, control-dominated behaviors constrained by arbitrary sequential constraints. Viability and scalability of this technique is demonstrated by constructing and then performing exact scheduling on problems of practical sizes and complexities drawn from both academic and industrial sources. In particular, a substantial RISC model that supports pipelining, data hazards, bypass, stall and cache hit/miss protocols was scheduled exactly.

Bibliography

1. A. Aho, *et al*., *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, *et al.*, "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*, pp.509-516, June 1978.
5. P. Ashar and M. Cheong, "Efficient Breadth-First Manipulation of Binary Decision Diagrams", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.622-627, San Jose, USA, Nov. 1994.
6. J. R. Allen, *et al.*, "Conversion of Control Dependence to Data Dependence", *Proc. 10th Ann. ACM Symp. Principles of Programming Languages*, pp. 177-189, Jan. 1983.
7. J. Babb, *et al.*, "Parallelizing Applications into Silicon", *Proc. Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, pp. 70-80, April 1999.
8. R. I. Bahar, *et al.*, "Algebraic Decision Diagrams and their Applications", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.188-191, San Jose, USA, Nov. 1993.
9. R. A. Bergamaschi, *et al.*, "Allocation Algorithms Based on Path Analysis", *Integration, the VLSI Journal*, vol.13, no.3, pp. 283-299, Sept. 1992.
10. A. Biere, *et al*., "Symbolic Model Checking Using SAT procedures instead of BDDs", *Proc. 36th ACM/IEEE Design Automation Conf.*, pp.317-320, New Orleans, USA, June 1999.
11. 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.
12. G. Borriello, *et al.*, "Interface Synthesis: A Vertical Slice from Digital Logic to Software Components", *IEEE Int. Conf. Computer-Aided Design*, pp. 693-695, 1998.

13. G. Borriello, *et al.*, "Embedded System Co-Design: Towards Portability and Rapid Integration," *Hardware/Software Co-Design*, M.G. Sami and G. De Micheli, EDs., Kluwer Aacademic Publishers, 1995.

14. K. Brace, R. Rudell, and R. Bryant, "Efficient Implementation of a BDD package", *Proc. 27th ACM/IEEE Design Automation Conf.*, pp.40-45, Orlando, USA, June 1990.

15. F. Brewer and D. Gajski "Chippe: A System for Constraint Driven Behavioral Synthesis" *IEEE Trans. CAD/ICAS*, pp.681-95, July 1990

16. R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. Computers*, pp. 677-691, Aug. 1986.

17. R. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams", *ACM Computing Surveys*, pp. 293-318, Sep. 1992.

18. R. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification", *Proc. Int. Conf. Computer-Aided Design*, pp. 236-243, San Jose, USA, Nov. 1995.

19. J. R. Burch, *et al.*, "Symbolic Model Checking for Sequential Circuit Verification", *IEEE Trans. CAD/ICAS*, pp. 401-424, April 1994.

20. D. Burger, *et al*., "Evaluating Future Microprocessors: The SimpleScalar Tool Set", *Technical Report CS-TR-96-1308*, University of Wisconsin-Madison, July 1996. (http://www.simplescalar.org, accessed Oct. 2000.)

21. G. Cabodi, *et al.*, "Improving the Efficiency of BDD-Based Operators by Means of Partitioning", *IEEE Trans. CAD/ICAS*, pp. 545-556, May 1999.

22. R. Camposano, "Path-Based Scheduling for Synthesis", *IEEE Trans. CAD/ICAS*, pp. 85-93, Jan. 1991.

23. R. Camposano, *et al.*, "The IBM High-Level Synthesis System", *High-Level VLSI Synthesis*, R. Camposano and W. Wolf, eds., Kluwer, 1991.

24. R. Camposano and W. Rosenstiel, "A Design Environment for the Synthesis of Integrated Circuits", *11th Symp. Microprocessing and Microprogramming EUROMICRO '85*, Brussles, Belgium, pp. 211-215, Sept. 1985.

25. L.-F. Chao, *et al*., "Rotation Scheduling: A Loop Pipelining Algorithm", *Proc. 30th Design Automation Conf.*, pp. 566-572, 1993.

26. H. D. Cheng and C. Xia, "High-Level Synthesis: Current Status and Future Prospects", *Cicuits Systems Signal Process*, pp. 351-400, 1995.

27. H. Cho, *et al.*, "Algorithms for Approximate FSM Traversal", *Proc. 30st ACM/IEEE Design Automation Conf.*, pp. 25-30, Dallas, USA, June 1993.

28. P. Chou, *et al*., "Scheduling Issues in the Co-Synthesis of Reactive Real-

Time Systems," *IEEE Micro*, pp. 37-47, August 1994.

29. 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, Orlando, USA, June 1990.

30. 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, San Jose, USA, Nov. 1994.

31. O. Coudert, *et al*., "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.

32. 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, San Jose, USA, Nov. 1990.

33. S. Davidson, *et al.*, "Some Experiments in Local Microcode Compaction for Horizontal Machines", *IEEE Trans. Computers*, pp. 460-477, July 1981.

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

35. J. Elliot, *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*, Kluwer Academic Publishers, 1999.

36. J. Ferrante, *et al*., "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. Fisher, *Global Code Generation for Instruction-Level Parallelism: Trace scheduling-2*, Hewlett Packard Laboratories Technical Report HPL-93-43, June 1993.

38. D. Gajski, *et al.*, *SpecC Specification Language and Methodology*, Kluwer Academic Publishers, 2000.

39. D. Gajski, *et al.*, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

40. D. Gajski and L. Ramachandran, "Introduction to High-Level Synthesis", *IEEE Design & Test of Computers*, pp.44-54, Winter 1994.

41. M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1983.

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

43. C. Gebotys, “Throughput Optimized Architectural Synthesis”, *IEEE Trans. VLSI Systems*, vol. 1, no. 3, pp. 254-261, Sep. 1993.

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

45. G. Goosens, *et al*., “Loop Optimization in Register-Transfer Scheduling for DSP-systems”, *Proc. ACM/IEEE Design Automation Conf.*, pp. 826-831, 1989.

46. T. Granlund and R. Kenner, “Eliminating Branches using a Superoptimizer and the GNU C Compiler”, *Proc. of the ACM SIGPLAN’92 Conference on Programming Language Design and Implementation (PLDI)*, pp. 341-352, San Francisco, USA, 1992.

47. D. Greve, “Symbolic simulation of the JEM1 microprocessor”, *Proc. Formal Methods in Computer-Aided Design*, FMCAD ‘98, Palo Alto, CA, pp. 321-333, Nov. 1998.

48. G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, 1996.

49. B. Haroun and M. Elmasry, “Architectural Synthesis for DSP Silicon Compiler”, *IEEE Trans. CAD/ICAS*, pp.431-47, April 1989.

50. S. Haynal and F. Brewer, “Efficient Encoding for Exact Symbolic Automata-Based Scheduling”, *IEEE Int. Conf. Computer-Aided Design*, pp. 477-481, 1998.

51. S. Haynal and F. Brewer, “Model for Scheduling Protocol-Constrained Components and Environments”, *Proc. of 36th ACM/IEEE Design Automation Conf.*, pp. 292-295, 1999.

52. S. Haynal and F. Brewer, “Automata-Based Scheduling for Looping DFGs”, *University of California, Santa Barbara Technical Report ECE99_14*, October 1999. To be published: *IEEE Transactions on Computers.*

53. S. Haynal, “PYCUDD: A Python Interface for CUDD”, http://bears.ece.ucsb.edu/Home.html, accessed Oct. 2000.

54. S. Haynal, “PYSCHED: A Python Implementation of Automata-Based Symbolic Scheduling”, http://bears.ece.ucsb.edu/Home.html, accessed Oct. 2000.

55. J. Hennessy and D. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

56. J. Hsu, Personal communication, Nov. 2000.

57. A. Hu, *et al.*, “Higher Level Specification and Verification with BDD’s” *Computer-Aided Verification: Fifth Int. Conference*, 93, Lecutre Notes in

Computer Science v.697, Springer-Verlag, 1993.

58. S. H. Huang, *et al.*. “A Tree-Based Scheduling Algorithm for Control Dominated Circuits”, *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 578-582, Dallas, USA, June 1993.

59. H. Hulgaard *et al.*, “An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems”, *IEEE Transactions on Computers*, vol. 44, no.11, pp. 1306-1317, Nov. 1995.

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

61. C.-T. Hwang, *et al.*, “A Formal Approach to the Scheduling Problem in High Level Synthesis”, *IEEE Trans. CAD/ICAS*, pp.464-475, Apr. 1991.

62. S.-W. Jeong and F. Somenzi, “A New Algorithm for the Binate Covering Problem and its Application to the Minimization of Boolean Relations”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.417-420, San Jose, USA, Nov. 1992.

63. 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.

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

65. 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.

66. D. Knapp, *Behavioral Synthesis,* Prentice Hall, 1996.

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

68. G. Lakshminarayana, K.S. Khouri and N.K. Jha, “Wavesched: A Novel Scheduling Technique for Control-Flow Intensive Designs”, *IEEE Trans. CAD/ICAS*, vol. 18, no. 5, pp. 505-523, May 1999.

69. L. Lavagno, E. Sentovich, “ECL: A specification environment for system-level design”, *Proc. 36th ACM/IEEE Design Automation Conference*, New Orleans, June 1999.

70. C. Lee, M. Potkonjak, and W.H. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, *Proc. 30th Annual International Symposium on Microarchitecture*, pp. 330-335, Dec. 1997. (http://www.cs.ucla.edu/~leec/mediabench/, accessed Oct. 2000.)

71. T.-F. Lee, *et al.*, “A Transformation-Based Method for Loop Folding”, *IEEE Trans. CAD/ICAS*, vol. 13, no. 4, pp. 439-450, April 1994.

72. T-.F. Lee, *et al.*, “An Effective Methodology for Functional Pipelining”, *IEEE Trans. CAD/ICAS*, vol. 13, no. 34, pp. 439-450, Apr. 1994.

73. C.E. Leiserson, *et al.*, “Optimizing Synchronous Circuits by Retiming”, *Proc. Third Conf. VLSI*, 1983.

74. R. Leupers and P. Marwedel, “Time Constrained Code Compaction for DSPs”, *IEEE Trans. on VLSI Systems*, pp. 112-122, 1997.

75. R. Leupers and P. Marwedel, “Retargetable Generation of Code Selectors from HDL Processor Model”, *Proc. of European Design & Test Conference*, pp. 140-144, Paris, France, March 1997.

76. R. Leupers and P. Marwedel, “Algorithms for Address Assignment in DSP Code Generation”, *Proc. Int. Conf. Computer-Aided Design*, pp.109-112, San Jose, USA, Nov. 1996.

77. R. Leupers and P. Marwedel, “A BDD-based Frontend for Retargetable Compilers”, *Proc. the European Design & Test Conference*, pp.239-243, Paris, France, March 1995.

78. S. Liao, *et al.*, “Code Optimization Techniques for Embedded DSP Microprocessors”, *Proc. 32nd ACM/IEEE Design Automation Conference Proc.*, pp.599-604, San Francisco, USA, June 1995.

79. H.-T. Liaw and C.-S. Lin, “On OBDD-Representation of General Boolean Functions”, *IEEE Trans. Computers*, pp. 661-664, June 1992.

80. B. Lin, *Synthesis of VLSI Designs with Symbolic Techniques*, Ph.D. Dissertation, memo. no. UCB/ERL M91/105, UC Berkeley, Nov. 1991.

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

82. T. Ly, *et al.*, “Scheduling using Behavioral Templates”, *Proc. 32th ACM/IEEE Design Automation Conf.*, pp. 101-106, 1995.

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

84. D. Mallon and P. Denyer, “A New Approach to Pipeline Optimisation”, *Proc. EDAC-90*, pp. 83-88, 1990.

85. P. Marwedel, “The Mimola Design System: Tools for the design of digital processors”, *Proc. 21st, ACM/IEEE Design Automation Conference*, Albuquerque, USA, 1984.

86. P. Marwedel and G. Goosens (eds.), *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

87. H. Massalin, "Superoptimizer -- A Look at the Smallest Problem" in *Proc. of the Second Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pp.122-126, 1987.

88. 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.

89. K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

90. C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design*, Springer, 1998.

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

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

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

94. C. Monahan and F. Brewer, "Symbolic Modeling and Evaluation of Data Paths", *Proc. 32nd ACM/IEEE Design Automation Conference Proc.*, pp.389-394, San Francisco, USA, June 1995.

95. C. Monahan and F. Brewer, "Scheduling and Binding Bounds for RT-Level Symbolic Execution", *Proc. IEEE Int. Conf. Computer-Aided-Design, pp. 230-235*, 1997.

96. I.-H. Moon, *et al*, "Least Fixpoint Approximations for Reachability Analysis", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.41-44, San Jose, USA, Nov. 1999.

97. 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.

98. T. Murata, "Petri Nets: Properties, Analysis and Applications", *Proc. of the IEEE*, pp. 541-576, April 1989.

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

100. 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, San Jose, USA, Nov. 1994.

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

102. B. M. Pangrle and D. D. Gajski, "Design Tools for Intelligent Silicon Compilation", *IEEE Trans. CAD/ICAS*, pp.1098-1112, Nov. 1987.

103. C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algroithms and Complexity, Prentice Hall*, 1982.

104. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, Wiley, 1999.

105. N. Park and A. C. Parker., "SEHWA: A Software Package for Synthesis of Pipelines from Behavioral Specifications", *IEEE Trans. CAD/ICAS*, pp.356-370, March 1988.

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

107. N. 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.

108. E. Pastor and J. Cortadella, "Efficient Encoding Schemes for the Symbolic Analysis of Petri Nets", *Proc. Design, Automation and Test in Europe*, pp. 790-795, 1998.

109. P. Paulin and J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Trans. CAD/ICAS*, pp.661-79, June, 1989.

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

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

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

113. 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.

114. I. Radivojević , *Symbolic Scheduling Techniques*, Ph.D. Dissertation, Univeristy of California, Santa Barbara, 1996.

115. V. K. Raj DAGAR: An automatic pipelined microarchitecture synthesis sys-

tem.", Proc. International Conference on Computer Design (ICCD'89), Boston MA Oct. 1989.

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

117. 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.

118. G. van Rossum, "Python: An Interpreted, Interactive, Object-Oriented Programming Language", http://www.python.org, accessed Oct. 2000.

119. R. Rudell, "Dynamic Variable Ordering for Binary Decision Diagrams", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.42-47, San Jose, USA, Nov. 1993.

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

121. L. Dos Santos, *et. al*., "A Code-Motion Pruning Technique for Global Scheduling", *ACM Trans. Design Automation of Electronic Systems*, vol. 5, no.3, pp. 1-33, Jan. 2000.

122. A. Seawright and F. Brewer, "Clairvoyant: A Synthesis System for Production-based Specification", *Proc. IEEE Trans. on VLSI Systems*, vol. 2, no. 2, pp. 172-185, June 1994.

123. F. Somenzi, "CUDD: Colorado University Decision Diagram Package", http://vlsi.Colorado.EDU/~fabio/CUDD/, accessed Oct. 2000.

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

125. 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.

126. A. Timmer, *From Design Space Exploration to Code Generation*, Ph.D. Thesis Eindhoven University of Technology, 1996.

127. 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.

128. 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.

129. M. Torres, Personal communication, Nov. 2000.

130. H. J. Touati, *et al.*, "Implicit State Enumeration of Finite State Machines using BDD's," *Proc. Int. Conf. Computer-Aided Design*, pp.130-133, San Jose, USA, Nov. 1990.

131. J. Van Praet, *et al.*, "A Graph Based Processor Model for Retargetable Code Generation", *Proc. of European Design and Test Conference*, pp.102-7, Paris, France, 1996

132. 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, Anaheim, USA, June 1992.

133. 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.

134. C.-Y. Wang and K. Parhi, "High-Level DSP Synthesis Using Concurrent Transformations, Scheduling, and Allocation", *IEEE Trans. CAD/ICAS*, vol. 14, no. 3, pp. 274-295, Mar. 1995.

135. N. Warter, *et al.*, "Enhanced Modulo Scheduling for Loop with Conditional Branches", *Proc. 25th Ann. Int. Symp. Microarchitecture*, pp. 170-179, 1992.

136. W. Wolf, *et al*., "The Princeton University Behavioral Synthesis System", *Proc. 29th ACM/IEEE Design Automation Conf.*, pp. 182-187, 1992.

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

138. 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*, p.166-83, Feb. 1996.

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

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

141. Ti-Yen Yen; Wolf, W. "An efficient graph algorithm for FSM scheduling", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol.4, (no.1), IEEE, March 1996. p.98-112.