Analysis of Conditional Resource Sharing using a Guard-based Control Representation *

Ivan P. Radivojević Forrest Brewer
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA, U.S.A.

Abstract

Optimization of hardware resources for conditional data-flow graph behavior is particularly important when conditional behavior occurs in cyclic loops and maximization of throughput is desired. In this paper, an exact and efficient conditional resource sharing analysis using a guard-based control representation is presented. The analysis is transparent to a scheduler implementation. The proposed technique systematically handles complex conditional resource sharing for cases when folded (software pipelined) loops include conditional behavior within the loop body.

1. Introduction

Resource constraints play a crucial role in high-level synthesis of digital systems. *Conditional resource sharing* enhances resource usage by enabling simultaneous operations on different control paths to share physical resources. For example, operations belonging to "then" and "else" branches of an *if-then-else* statement are mutually exclusive when the choice of branch to execute is made prior to the execution of the branches. However, it has been shown [1] that superior scheduling results are possible if operations belonging to branch arcs are executed before the branching decision is made (*speculative* execution). In this case, static exclusivity analysis (performed before scheduling) is not sufficient for optimal use of resources.

A recent approach [2] successfully detects a static pairwise mutual exclusiveness. In *tree scheduling* [3], a tree representation allows code motion and sub-trees induced by a branch can share resources. *CVLS* [1] uses *condition vectors* [4] to dynamically track exclusiveness of pre-executed operations. However, a case was reported wherein condition vector analysis can lead to an erroneous conclusion on mutual exclusion between the operations [5]. The representation from [5] handles nested *if-then-else* structures correctly, but is not applicable to some other forms of conditional behavior (e.g. *goto*). These representations are all restricted to *conditional tree* structures. Parallel trees are addressed in [1], but the trees are either scheduled sequentially or tree duplication is performed. Most importantly, none of these approaches discusses conditional resource sharing in cyclic control/data flow graphs (CDFGs) with loop pipelining.

Numerous techniques for cyclic data-flow graph (DFG) optimizations have been proposed, ranging from heuristics [6][7][8] to ILP methods [9][10]. However, none of them discusses cases in which conditional behavior occurs within the loop body. Semantics-preserving techniques and the BFSM-based approaches are applicable to cyclic CDFGs, but they either lack a formal treatment of conditional resource sharing [11][12] or introduce an excessive number of 0/1 variables to model resource and exclusivity constraints [13]. Recently, *rotation scheduling* [6] has been extended to pipelining of CDFGs [14]. This technique is based on a *condition flag* representation restricted to cases where execution conditions can be represented as a Boolean cube. Conditional resource sharing analysis is performed using *usage flags* assigned to individual functional units. Support for *node dividing* [1] is not discussed.

Guard-based control representation is a foundation for exact symbolic techniques for resource-constrained scheduling [15][16]. In many aspects, the guard-based model is similar to execution conditions from *path analysis* [17], where Boolean conditions are used in the hardware allocation phase (after AFAP scheduling). That research demonstrated that OBDDs (Ordered Binary Decision Diagrams [18]) efficiently represent control signals in large scale problems. Guard-like models are used in several recent exact techniques [19][20]. There, conditional branches are labeled by Boolean functions and a proper interpretation of mutual exclusion is guaranteed by construction. However, specification-level formalism restricts code motion beyond the code-block boundaries (e.g. speculative execution).

In this paper, we present an exact technique for conditional resource sharing analysis. We do not discuss a scheduler implementation. However, the analysis in this paper is implementation transparent. In Section 2 we present an overview of a guard-based control representation. In Section 3 this control model is directly linked to evaluation of resource constraints. First, we describe the simple case of acyclic CDFGs. The discussion is then extended to the more general case of pipelining of cyclic CDFGs. Finally, experimental results are presented in Section 4.

* Supported in part by a fellowship donation from Mentor Graphics Corp.

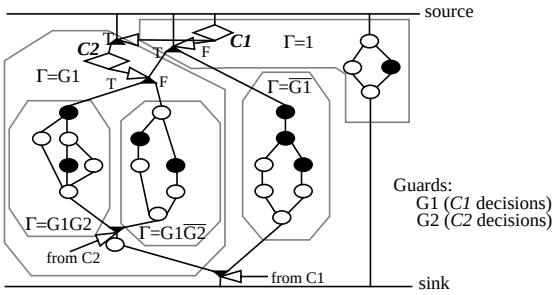


Figure 1. Kim's example

2. Control representation

We assume a CDFG specification describing both data-flow and control dependencies between operations (similar to the one used in [4]). Operation nodes are atomic actions potentially requiring the use of hardware resources (e.g. arithmetic/logical operations, read/write cycles). Conditional behavior is specified by fork and join nodes. An operation node generating a control signal for a fork/join pair is called a *conditional*. Directed arcs establish a link between the conditional and a related fork/join pair. Conditionals make decisions on the flow of control (whether *true* (T) or *false* (F) branches provide operands for successor operations).

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

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

In Fig.1 (Kim's example [21]), two guards (G_1, G_2) corresponding to the conditionals (*C1*, *C2*) encode the conditional behavior. There are three possible execution paths: $(G_1G_2, G_1\overline{G_2}, \overline{G_1})$. Indicated blocks correspond to operations that share the same guard function Γ $(1, G_1, G_1G_2, G_1\overline{G_2}, \overline{G_1})$. We note that Γ's are not restricted to product terms (thus, they can handle constructs such as:

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

Figure 2. Pseudo-code fragment

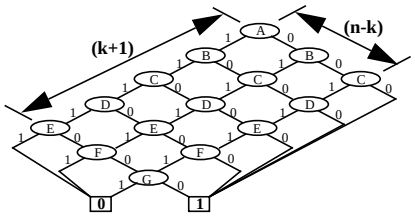


Figure 3. *At-most-k-of-n* constraint (k=4, n=7)

goto, *exit*, *case*). In the pseudo-code fragment shown in Fig.2, the execution condition for statement *c* is described as: $\Gamma_c = G_{C_1} + \overline{G_{C_1}} G_{C_2} = G_{C_1} + G_{C_2}$. Guard-based representation also applies to parallel or correlated control structures. If two copies of Fig.1 are executed in parallel, only two more guard variables are introduced, while the number of control combinations (9) grows much faster. The number of guards is not proportional to the number of control paths, but is determined by the number of conditionals.

3. Conditional resource sharing analysis

First we briefly review the OBDD form of the *At-most-k-of-n* constraint [15]. We refer to an *At-most-k-of-n* constraint as $B_{k,n}$. The Boolean equation form of $B_{k,n}$ is:

$$B_{k,n}(f_1, f_2, \ldots, f_n) = \sum_{1 \le (l_p \ne l_q) \le n} \overline{f_{l_1}}\,\overline{f_{l_2}} \cdots \overline{f_{l_{(n-k)}}} \qquad (1)$$

where f_i are *Boolean* functions. Fig.3 shows the OBDD form of $B_{k,n}$ in which f_i's are simple Boolean variables. The number of product terms in the $B_{k,n}$ is $\binom{n}{k}$. However, the OBDD form is compact and can be built efficiently using *ite* (if-then-else) calls. The vertices in this symmetric template need not be restricted to variables -- arbitrary Boolean functions f_i can be inserted into the template shown in Fig.3 (e.g. bus/register scheduling constraints described in [15]).

3.1 Acyclic CDFGs

Guard functions may be used to perform conditional resource sharing analysis for an *arbitrary* number of CDFG operations. We illustrate the idea using a CDFG fragment shown in Fig.4. Assume that the scheduling has been completed for *step_1* and that operations *1* and *2* have been scheduled in the *step_2*. We want to analyze scheduling operation *3* in *step_2* assuming that only one "white" resource is available. Evaluating an $B_{1,3}$ using guard functions Γ_i (i = 1, 2, 3) as arguments we obtain:

$$B_{1,3}(\Gamma_1, \Gamma_2, \Gamma_3) = \overline{\Gamma_1}\,\overline{\Gamma_2} + \overline{\Gamma_1}\,\overline{\Gamma_3} + \overline{\Gamma_2}\,\overline{\Gamma_3} = 0 \qquad (2)$$

Since the constraint evaluates to '0', we conclude that the schedule is infeasible on all paths. If two resources are

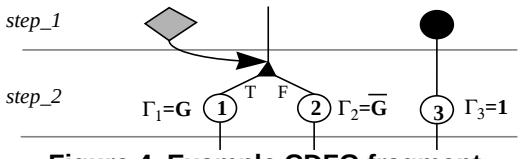


Figure 4. Example CDFG fragment

available, the constraint $B_{2,3}$ evaluates to '1':

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

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

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

$$B_{1,2}(\Gamma_2, \Gamma_3) = \overline{\Gamma_2} + \overline{\Gamma_3} = G \quad (4)$$

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

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

The proof is straightforward since every individual control path is represented as a product of guard variables. We can evaluate $B_{k,n}(\Gamma_1, \Gamma_2, \ldots, \Gamma_n)$ for every possible combination (minterm) of guard variables and obtain '1' (if the minterm is covered by at most k Γ_i functions) or '0' (if the minterm is covered by more than k functions). Note that although the conceptual complexity of the test is very high, it can be performed efficiently since Γ_i functions are represented by OBDDs -- the computation amounts to insertion of guard functions into the template $B_{k,n}$ (Fig.3).

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

$$S_j = \Gamma_j B_{k,n}(\Gamma_1, \Gamma_2, \ldots, \Gamma_j, \ldots, \Gamma_n) \quad (5)$$

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

$$\Gamma_3 \setminus S_3 = \Gamma_3 \overline{S_3} = \overline{G} \quad (6)$$

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

To support code motion across the basic code blocks, Γ_i functions may have to be modified during the scheduling. For example, if operation *1* (Fig.4) is executed speculatively in *step_1*, variable G has to be factored out from Γ_1 (i.e. Γ_1 becomes '1'), since the corresponding conditional (shaded comparator) is unknown at that time. This reflects the fact that during *step_1*, paths G and $\overline{G}$ are indistinguishable.

1. This reduces to a pair-wise mutual exclusion test ($\Gamma_i\Gamma_j$=0) as a previously observed special case (e.g.[2][15][17]).
2. The scheduler, however, has to ensure that node dividing is done in a causal manner (e.g. not to allow dividing of nodes with respect to a conditional whose value is still unknown at a particular time step).

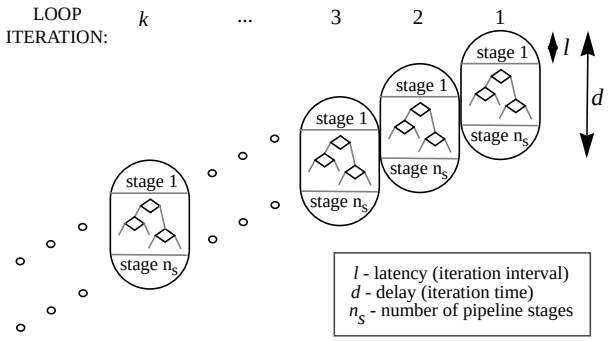


Figure 5. Overlapping of loop iterations

The proposed approach is computationally efficient. We observe that the number of operations in a typical CDFG is much larger than the number of potentially distinct guard functions. Only one pointer to a guard function need be stored for each operation instance during the scheduling process. Furthermore, memory overhead for storing guard functions is expected to be very low due to the sharing property of the OBDD data structure [18]. Compared to the method proposed here, condition vectors [1] are less efficient and have smaller expressive power since in that approach: (*i*) control paths are "one-hot" encoded, (*ii*) no sharing is possible between the vectors, and (*iii*) execution order of conditionals is pre-specified.

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

3.2 Pipelining of cyclic CDFGs

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

Fig.5 shows an example of overlapped execution of a loop using n_s pipeline stages. Assume that the loop body exhibits n_p distinct control paths. In Fig.5, the number of paths may grow as $(n_p)^{\lfloor (step-1)/l \rfloor + 1)}$. For example, for time steps $l < step \le 2l$, the number of paths is potentially $(n_p)^2$,

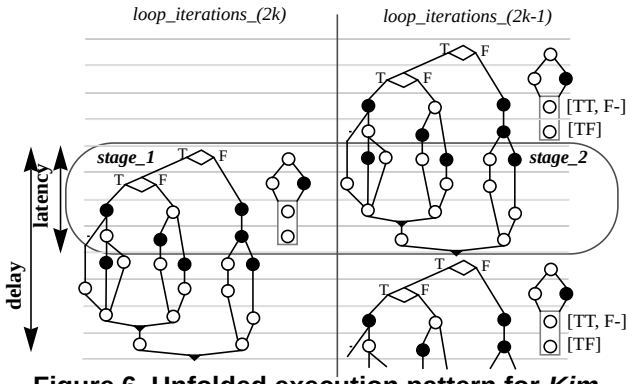


Figure 6. Unfolded execution pattern for *Kim*

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

The unfolded execution of a functionally pipelined version of *Kim* (Fig.1) is shown in Fig.6. We assume two adders ("white" operation), one subtracter ("black" operation) and one comparator (single-cycle units assumed). The example requires 8 cycles on these resources if loop pipelining is not performed [4]. With loop pipelining, a schedule using 2 stages and having latency of 4 (using the same resources) can be found as indicated in the Fig.6 (delay remains 8 cycles). One operation is divided as indicated by the values of the guards corresponding to conditionals (*C1*, *C2*). The indicated block in the middle of the figure shows a pipelined loop pattern. Although there are nine control paths, the control is simple since the schedules for the two iterations are independent. In general, this need not be the case: superior schedules may be achieved when a control correlation is introduced among the overlapping iterations.

We now extend conditional resource sharing analysis to the more general case of pipelining of cyclic CDFGs. Consider the CDFG shown in Fig.7. Assuming that only one single-cycle resource of each type (comparator, "white", "black") is available, the CDFG from Fig.7 can be scheduled in 4 time steps without loop pipelining. However, latency can be reduced to 2 time steps using three pipeline stages. For simplicity, we assume that the CDFG has to be executed an infinite number of times and that no inter-iteration data dependencies exist. These assumptions do not affect the generality of the approach[5].

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

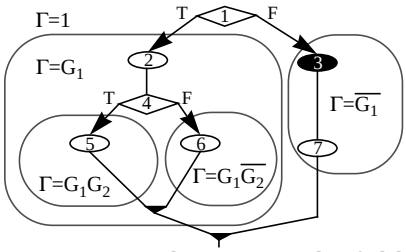


Figure 7. Example CDFG to be folded

The overlapping iterations are treated as parallel threads of computation, leading to the following resource analysis procedure [6]:

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

2. Compute $\Gamma_{j,1}$ by substituting $G_{k,1}$ for each G_k in Γ_j. Resource constraints are evaluated as described for a CDFG without loop folding (Section 3.1).

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

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

4. Repeat steps 3.a and 3.b for each time step and each pipeline stage. Conditional resource availability is computed as described in Section 3.1.

Steps 3.a and 3.b preserve all inter-iteration and intra-iteration control dependencies. They reflect the fact that

3. Increasing the amount of state available for control generation may improve a schedule, but is likely to lead to more complex controllers.
4. Assuming no speculative execution, 8-cycle schedule can be found even using only one single-cycle adder.
5. In the general case, a loop test must be explicit in the CDFG specification and the scheduler has to enforce inter-iteration precedences.
6. Some schedulers first generate a feasible pipelined schedule (in terms of dataflow dependencies) and subsequently resolve resource violations by incremental partial rescheduling [7][11]. Alternatively, the initial non-pipelined schedule can be free of resource violations and the latency is then reduced through incremental operation rotation [6][14].

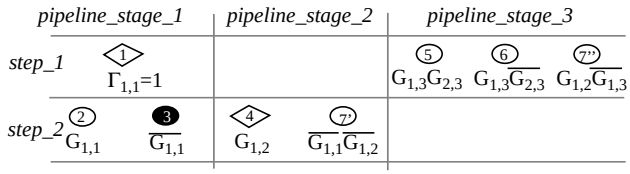


Figure 8. Folded CDFG from Fig.7

overlapping loop iterations flow through the pipeline stages in a synchronous fashion.

We now apply the procedure to the CDFG in Fig.7. A feasible schedule using three pipeline stages, achieving latency of 2 is shown in Fig.8. Assume $n_i = n_s = 3$ and that stage 1 has been scheduled as shown in Fig.8. Since operation 4 is pushed from the first pipeline stage into the second pipeline stage, its new guard function becomes: $\Gamma_{4,2} = G_{1,2}$. If the $B_{1,2}$ constraint at step 1 is evaluated using $\Gamma_{1,1}$ and $\Gamma_{4,2}$ we obtain:

$$B_{1,2}(\Gamma_{1,1}, \Gamma_{4,2}) = \bar{1} + \overline{G_{1,2}} = \overline{G_{1,2}} \quad (7)$$

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

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

$$B_{1,2}(\Gamma_{2,1}, \Gamma_{7,2}) = \overline{G_{1,1}} + \overline{(\overline{G_{1,2}})} = \overline{G_{1,1}} + G_{1,2} \quad (8)$$

indicating the paths free of resource violations. Since:

$$S_7 = \Gamma_{7,2} B_{1,2}(\Gamma_{2,1}, \Gamma_{7,2}) = \overline{G_{1,1}}\,\overline{G_{1,2}} \quad (9)$$

operation 7 cannot be scheduled at step 2 if the 'T' path is simultaneously taken in the CDFG being executed in the first pipeline stage ($G_{1,1}\overline{G_{1,2}}$). However, operation 7 can be split (see Fig.8). The guard function of 7' can be set to:

$$\Gamma_{7',2} = S_7 = \overline{G_{1,1}}\,\overline{G_{1,2}} \quad (10)$$

Operation 7 has yet to be scheduled on paths:

$$\Gamma_{7'',2} = \Gamma_{7,2} \backslash \Gamma_{7',2} = G_{1,1}\overline{G_{1,2}} \quad (11)$$

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

$$\Gamma_{7'',3} = G_{1,2}\overline{G_{1,3}} \quad (12)$$

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

$$B_{1,3}(\Gamma_{5,3}, \Gamma_{6,3}, \Gamma_{7'',3}) = 1 \quad (13)$$

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

3.3 Probabilistic interpretation

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

$$OnSetSize(1) = 2^{n_c} \quad (14)$$

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

$$\frac{OnSetSize(\Gamma_j)}{OnSetSize(1)} = P(j) \quad (15)$$

where *P(j)* indicates the probability that operation *j* will be conditionally executed. Probability *P(j)* or its variations are frequently used in resource-constrained schedulers to define heuristic priority functions (e.g. [4]). We observe that the computation of *OnSetSize(f)* amounts to a simple one-pass traversal of an OBDD representation of *f*. When the probability of a conditional's outcome is not uniform, behavioral description analysis/simulation can be performed to determine probability values. In such cases, the OBDD traversal algorithm for *OnSetSize*(*f*) can be easily modified to take into account individual probabilities $P(G_c)$ [7].

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

$$\frac{OnSetSize(\overline{B_{k,n}})}{OnSetSize(1)} \quad (16)$$

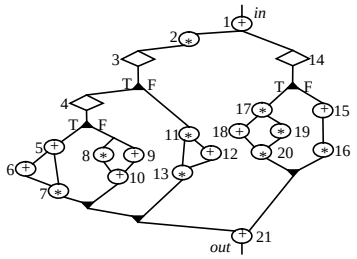
This ratio indicates the probability of a violation occurrence. Such information is useful for schedulers that resolve resource violations through partial rescheduling.

4. Experimental results

Two types of experiments are performed. First, we wish to investigate the benefits of exploiting conditional resource sharing. Table 1 we summarizes results for three examples: the *VerySmall* (Fig.7), *Kim* (Fig.1), and *SC* (Fig.9, [14]). As assumed in the previous sections, *VerySmall* uses 1 resource of each type (add, subtract, compare) and *Kim* uses 2 adders, 1 subtracter and 1 comparator. *SC* schedule (using 1 multiplier and 2 ALUs) is shown in Fig.9. Cycles 3 through 8 form a repetitive pattern that can be pipelined [8].

For all examples, we present three results for the same resource bounds. *Original* corresponds to CDFGs without unrolling and pipelining. *Unrolled* corresponds to the unrolled versions of a CDFG (no pipelining), while loop pipe-

7. We still assume these probabilities correspond to independent events.
8. Solution in [14] has latency of 6 as well, but uses 4 pipeline stages.



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

Figure 9. SC example and its schedule

lining is allowed in *pipelined*. The *original* and *unrolled* results are obtained using exact symbolic techniques [15][16]. The *pipelined* results were generated in a semi-automated fashion [9]. Table 1 shows that systematic treatment of resource sharing can increase throughput even in cases when the loop body exhibits conditional behavior.

To investigate the computational and storage overhead of the approach, *"pipelined"* results were verified for potential resource constraint violations. The overhead due to guard variables and functions is very small: 14 OBDD nodes (*Kim*, Fig.6), 18 nodes (*VerySmall*, Fig.8), and 24 nodes (*SC*, Fig.9). In all examples, verification of the resource bounds took less than 0.03 CPU seconds.

5. Summary

An approach to conditional resource sharing analysis using a guard-based control representation was described. To our knowledge, this is the first method to systematically handle conditional resource sharing for pipelined loops that exhibit conditional behavior within the loop body. In the future, we plan to implement a scheduler based on the presented concepts. This requires that several additional issues be addressed (e.g. node unification, incremental recalculation of Γ functions when conditionals are rescheduled, timing model for operation chaining, etc.).

References:

1. K. Wakabayashi and H. Tanaka, "Global Scheduling Independent of Control Dependencies Based on Condition Vectors", *Proc. 29th DAC*, 1992.
2. H.-P. Juan, V. Chaiyakul, and D.D. Gajski, "Condition Graphs for High-Quality Behavioral Synthesis", *Proc. ICCAD*, 1994.

9. Symbolic techniques can be extended to solve a relaxed version of the conditional pipelining by adding some necessary conditions for existence of a repetitive pattern in a schedule of the unrolled loop. All of the presented results were generated using such an approach, but they were manually verified for potential inter-iteration dependency violations.

Table 1. Throughput comparisons

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

3. S.H. Huang *et al.* "A Tree-Based Scheduling Algorithm for Control Dominated Circuits", *Proc. 30th DAC*, 1993.
4. K. Wakabayashi and T. Yoshimura, "A Resource Sharing and Control Synthesis Method for Conditional Branches", *Proc. 26th DAC*, 1989.
5. M. Rim and R. Jain, "Representing Conditional Branches for High-Level Synthesis Applications", *Proc. 29th DAC*, 1992.
6. L.-F. Chao, A. LaPaugh, and E.H.-M. Sha, "Rotation Scheduling: A Loop Pipelining Algorithm", *Proc. 30th DAC*, 1993.
7. T.-F. Lee *et al.*, "An Effective Methodology for Functional Pipelining", *IEEE Trans. CAD/ICAS*, vol.13, no.34, Apr. 1994.
8. M. Potkonjak and J. Rabaey, "Optimizing Resource Utilization Using Transformations", *IEEE Trans. CAD/ICAS*, vol.13, no.3, March 1994.
9. C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis", *IEEE Trans. CAD/ICAS*, vol.10, no.4, Apr. 1991.
10. C.H. Gebotys, "Throughput Optimized Architectural Synthesis", *IEEE Trans. VLSI Systems*, vol.1, no.3, Sep. 1993.
11. R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation Based Synthesis", *Proc. 27th DAC*, 1990.
12. T.-Y. Yen and W. Wolf, "Optimal Scheduling of Finite-State Machines", *Proc. ICCD*, 1993.
13. A. Takach and W. Wolf, "Scheduling Constraint Generation for Communicating Processes", *IEEE Trans. VLSI Systems*, vol.3, no.2, June 1995.
14. J. Siddhiwala and L.-F. Chao, "Scheduling Conditional Data-Flow Graphs with Resource Sharing", *Proc. 5th Great Lakes Symp. VLSI*, 1995.
15. I. Radivojević and F. Brewer, "Symbolic Techniques for Optimal Scheduling", *Proc. 4th SASIMI Workshop*, 1993.
16. I. Radivojević and F. Brewer, "Incorporating Speculative Execution in Exact Control-Dependent Scheduling", *Proc. 31st DAC*, 1994.
17. R.A. Bergamaschi, R. Camposano, and M. Payer, "Allocation Algorithms Based on Path Analysis", *Integration, the VLSI Journal*, vol.13, no.3, Sept. 1992.
18. R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. Computers*, vol.C-35, no.8, Aug. 1986.
19. J. Yang, G. De Micheli, and M. Damiani, "Scheduling with Environmental Constraints based on Automata Representations", *Proc. EDAC*, 1994.
20. C.N. Coelho Jr. and G. De Micheli, "Dynamic Scheduling and Synchronization Synthesis of Concurrent Digital Systems under System-Level Constraints", *Proc. ICCAD*, 1994.
21. T. Kim, J.W.S. Liu, and C. L. Liu, "A Scheduling Algorithm for Conditional Resource Sharing", *Proc. ICCAD*, 1991.